Network services and tool support for cloud environments

ANDREAS HEDENLIND
and
PÄR TJÄRNBERG

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2011

TRITA-ICT-EX-2011:206

KTH Computer Science
and Communication

Network services and tool support for cloud environments

Master of science thesis at Ericsson Research
Stockholm 2011

ANDREAS HEDENLIND
PÄR TJÄRNBERG

Supervisors:
Jan-Erik Mångs - Research engineer
Bob Melander - Research engineer
Examiner:
Vladimir Vlassov
Associate Professor, PhD

Abstract

Virtualization of servers and network infrastructure is an effective way to reduce hardware costs as well as power consumption. Today cloud systems are often used to handle virtualization of servers but lack the ability of deploy and configure network equipment.

Facilitating network equipment configuration within a cloud environment, would make it possible to create complete virtual networks along with well known and proven features of cloud computing today.

Our solution provides users with a graphical tool for easy and quick configuration of not only virtual machines but also virtual network equipment within a cloud environment. This makes it possible for a user to create advanced network topologies. Creating complete virtual networks like this using a single tool, will speed up configuration and minimize the errors that can occur when manually configuring multiple instances.

The implemented software solution consists of two major parts, a graphical user interface (GUI) and a back-end server. The back-end server is responsible for handling communication between the user application and an underlying cloud platform, in this case OpenStack.

The graphical user interface gives the user the possibility to draw networks and launch virtual machines using simple drag-and-drop features. It also monitors all the running virtual instances and physical machines, and alerts the user if a problem occurs.

This project is the first step towards supporting global virtual networks spanning across multiple data centers. It shows that it is possible to create virtual networks using a cloud environment as a base.

Referat

Virtualisering av servrar och nätverksutrustning är ett effektivt sätt att minska kostnaderna för hårdvara samt att minska energianvändningen. Idag används ofta molnmiljöer för att virtualisera servrar men det finns inte samma städ när det gäller vanlig nätverksutrustning. Kan man använda den beprövade tekniken hos molnplattformar för att även sköta virtualisering av nätverksutrustning, bör det leda till att man kan skapa helt virtualiserade nätverk.

Vår lösning ger en användare möjlighet att med hjälp av ett grafiskt verktyg konfigurera virtuella servrar och virtuell nätverksutrustning som exekverar i en molnplattform. Med hjälp av enbart detta verktyg kan en användare skapa avancerade virtuella nätverk på ett snabbt och säkert sätt, utan att behöva manuellt konfigurera varje enskild maskin.

Den implementerade mjukvaran består av två huvuddelar. Dels ett grafiskt användargränssnitt samt en back-end server som sköter kommunikationen mellan det grafiska gränssnittet och en molnplattform. Den plattform som har använts under detta projekt är OpenStack.

Det grafiska gränssnittet ger användaren möjlighet att rita upp virtuella nätverk och placera ut servrar genom att använda ett enkelt drag-and-drop system. Systemet övervakar sedan alla exekverande instanser, inklusive hårdvarumaskiner, och kan uppmärksamma användaren om fel uppstår.

Det här projektet är ett första steg mot att erbjuda globala virtuella nätverk som sträcker sig över flera datacenter. Det visar att det är möjligt att skapa virtuella nätverk med en molnplattform som bas.

Acknowledgements

We would like to start by thanking our supervisors, Bob Melander and Jan-Erik Mångs for the opportunity to do our Master's thesis at Ericsson Research.

This has given us a lot of new knowledge in the latest technologies regarding our academic focus, and it has been a good experience to be involved in the work at a research department.

We would also like to thank Hareesh Puthalath for his technical advices during this thesis project.

Finally we would like to thank our examiner, Vladimir Vlassov, for giving us the academic feedback needed throughout this project.

Contents

Contents

Chapter 1

Introduction

1.1 Background

Virtualization of individual servers and also entire networks are getting more and more common among enterprises and network operators. This is partly due to the fact that it reduces the operational and equipment costs. Furthermore, the utilization of the underlying hardware gets more efficient. One common solution for virtualization of servers is by placing them into a cloud environment. These environments embrace and fulfills the requirements of robustness and reliability of platforms and services running within, in addition to its major advantage: scalability. Cloud customers can let their IT infrastructure grow, letting the cloud environment handle issues concerning hardware upgrade, load balancing etc.

The cloud fills an important role where companies can put separate services, platforms and parts of its IT infrastructure to be scalable for increased future demand. However, what clouds do not provide is the ability to manage entire IT infrastructures where virtual network equipment is included. If clouds would support this it would introduce the possibility for companies to keep all their IT infrastructure in a cloud environment, with the benefit of one single spot of management.

Virtualized networks in a cloud environment has so far only been utilized to separate customers from each other within the cloud, not giving them access to each others network traffic. Considering stand-alone virtual machines running with-in the cloud, reachable via global addresses or placed on virtual private networks, this type of network setup is sufficient. But consider the following scenario:

> *Alice has developed a new protocol to be used on a content delivery network (CDN). Before deploying it on the real network she wants to test the protocol in an authentic network environment. In addition Bob wants to test a promising network capable application and needs a couple of servers, routers and switches to do so.*
>
> *Alice and Bob could collect some off-the-shelf hardware products to setup their environments separately and conduct their experiments. It is time*

consuming though and gets even worse if they want to alter their setup afterwards.

Another approach would be setting up the servers as virtual machines on just a few physical servers. Moreover, they could share the same physical network separated using i.e. VLAN (802.1q). This is a more neat setup, but still requires lots of manual configuration.

Extended network capabilities in a cloud environment where the physical network could be abstracted, making users capable of creating whatever virtual network needed would solve many of the issues described in the scenario above. The extension would allow the use of network equipment such as routers and switches connecting servers and services in a virtual world, very similar to the regular approach of setting up physical networks.

A cloud environment with the network capabilities described above would be beneficial for laboratory environments when researchers are performing experiments as in the scenario presented. But such a solution is not only constrained to this type of applications. Already existing legacy systems with special requirements on network setup could be placed in this type of environment introducing easier future extensions of it.

The architecture of cloud platforms today covers one single server cluster on which the issues introduced by an extension of network capabilities would need to be handled. The full strength of such a solution, however, would only be reached if the concept of cloud platforms would be extended to also cover several server clusters working in collaboration. This would be a concept called something like a "distributed cloud". It would then be possible to create and deploy network topologies spanning across the globe without knowledge of the hardware equipment, at least from a user perspective.

Given that a cloud environment would already exist, including the extended network capabilities discussed above, the solution would benefit a user-friendly interface to manage the virtual network as well as monitoring the system as a whole. This could apply on customers as well as cloud administrators and should preferably be realized using a graphical user interface (GUI). It would gather all configuration of networks, virtual machines etc to one single spot. Which in turn would facilitate future extensions of entire IT infrastructures.

1.1.1 Previously presented solution - VIBox

Prior to this master thesis project is another project, called VIBox[4]. It was an initial attempt to realize virtual network management in a cloud environment. The project was performed as a "proof of concept". The resulting prototype included a Java based GUI where users were able to drag-and-drop virtual network equipment in addition to virtual machines and connect them together. A back-end server handled the requests from the GUI to set up whatever virtual machines necessary

including network configuration. The virtual machines were hosted by a hypervisor called Xen [27].

VIBox was not built on top of an already existing cloud platform, but instead attempted to create a new type of cloud environment with less constraints on network settings. When the platform was developed from scratch its focus skewed towards solving issues concerning e.g. image repositories and creation and deletion of virtual machines, instead of network capabilities. As a side effect the final prototype had undesired dependencies between the physical and the virtual world. The user must know and specify the exact physical topology for the system to work properly. Moreover, the architecture of VIBox does not specify a standard communication protocol between client and server. In fact VIBox does not even have a documented architecture at all and many of its features are ad-hoc solutions where distribution of responsibility is unclear. Some procedures that should be placed in the back-end server are instead handled by the client application.

Nevertheless the prototype proved it would be feasible to develop a user friendly management tool, including network topologies, and raised interesting issues to be solved in future development.

1.2 Objectives

This project aims at providing an administrator with a tool for easy management of cloud environments, focusing on network configuration.

Here is a list of objectives produced in collaboration with our supervisors to reach the goals of this thesis. A more extensive discussion on how to reach these objectives is provided under the heading Expected Results in section 1.3.

1.2.1 Selection of underlying platform

Select a platform for VM management, either to use the VIBox prototype and extend it or looking at some other platforms like OpenNebula or OpenStack. The platform must be an open source solution registered under a licence that allows us to do modifications if necessary. Features included in this management platform would preferably be:

- Storage solution for VM images and profiles/roles.
- Life cycle control of virtual machines
- Storage of userâ€™s deployed services and their current status
- Advanced network capabilities, in extension to a common cloud platform that only focuses on separating users, we should be able to configure internal networks within specific projects.

1.2.2 Cloud platform with network configurations enabled

One important objective of this thesis is to provide a platform that gives the possibility to create virtual networks. However, the platform should also be equipped with tools to simplify the daily work of end-users, deploying virtual machines and networks in the cloud. This platform will be created as a back-end server and should be as independent as possible concerning user applications.

The following four requirements are what to consider during development.

Virtual Server Templates (VSTs)

The system should be able to handle multiple types of server templates. These could be various operating systems with preinstalled software. One way of enabling this is by using a base Linux distribution and then add functionality in form of software packages and configurations. This makes it possible to create several types of roles using minimum amount of resources. As different roles can use the same base functionality, it would be beneficial to have some kind of inheritance. By introducing role inheritance, the user can faster create new roles that has similar functionality to already existing ones.

Capability to create and store network templates

Capability to create and store network templates including arbitrary amount of virtual network equipment. This would enable administrators/developers to define preconfigured networks, including servers and other network equipment as well as what services to run on each one of them. This would result in shorter setup time, after definition of the templates.

Storage of Network Topologies on Back-End Server

Previously deployed network topologies should be stored on Back-End Server, initially the virtual network, and they should be delivered to GUI client on demand. This is an essential part if a user want to change an already existing setup that is running within the cloud.

Virtual networks spanning over multiple server clusters

A user should be able to setup virtual networks that is spread over more than one server cluster. It should be possible to see several clusters as one cloud platform where you can run your projects.

There is an ongoing project at Ericsson, in parallel to this one, which expected result will expose a web service interface providing inter-connections between distributed server clusters. The project is called MESON. The resulting prototype of this project would benefit on this inter-connection. It would make it possible to create virtual networks spanning over multiple server clusters.

1.2.3 Graphical User Interface (GUI)

A Graphical User Interface where the user can use drag-and-drop actions to plan for virtual machines in their cloud. It should also be possible to connect these machines to each other, making a virtual network including servers, routers and switches.

All logic and control should be handled by a back-end server, making the GUI a representational view of the current state of the back-end server. As a consequence, all system information should be presented/viewed on demand in the GUI and delivered by the back-end server.

Communication channel to the running instance

By having a command line interface to the running VMs, e.g. using a ssh connection, the user can manage its own applications and protocols on each machine. There might be some configuration that must be done inside the VM, and it would be easier if this could be done using the same graphical interface as you use during the setup. A command line interface to the VM could preferably also be used to pull statistics data.

An application might save information in a log file that can be fetched from the command line. An extension to the fetching of statistics can be a template file for statistics that the GUI can show graphically. If the application places its log file which is structured like the template, at a prespecified location, it can be automatically shown to the user.

Automated graphical representation of physical network topology

The physical network topology should be discovered and delivered to the user application. By having a graphical representation of the physical network in the GUI, it can be used for monitoring purposes. Functionality that can use this representation might be:

- Detection if the load is not evenly balanced
- Alarms from specific hardware machines, if faults have occurred
- Manual placement of virtual machines

1.2.4 Communication protocol

The Graphical User Interface (GUI) must be able to communicate with the Back-End Server in a standardized way. Preferably this is done using an XML pattern or JavaScript Object Notation (JSON). This will make the GUI independent of the back-end architecture, making it easier to extend the GUI with new features.

1.2.5 User Manual

Document a user manual how to run, configure and extend the developed software. This will include how to set up the environment with hardware devices and the software platform. This objective is of very big importance if others want to continue with and extend the project later.

1.3 Expected result

From the list of objectives, a selection has been made of what should be feasible with-in the boundaries of a master thesis project.

What separates this project from others in the area of cloud platforms concerns configurations of virtual networks, independently of the physical network setup. To become a useful administration tool as well as contributing to the cloud platform community, the extended network capabilities has to work smoothly.

The network capability mentioned above could be placed as an extension of the resulting prototype of the preceding master thesis project, VIBox. It could also be an extension of existing open-source cloud platforms. Hence developing a cloud platform is not in the scope of this master thesis project, why any of the suggested paths just mentioned should be chosen. Three platforms will be briefly evaluated; OpenNebula, OpenStack and previously developed platform called VIBox, and one of them will be chosen in which to extend the network capabilities.

This is not an evaluation project, why each platform will not be thoroughly evaluated but presented in short and evaluated in more detail in terms of current available functionality. The comparison will be conducted on paper.

It is important to pass the knowledge achieved throughout this project and make it available for following projects to proceed the development. Due to this, emphasizes will be put on well structured and informative documentation of all code written as well as a user manual of how to set up and use the developed software and tools.

1.3.1 Resulting prototype

This project has two major objectives, the ability to configure networks in a cloud environment and to provide users with a GUI to configure their virtual networks and machines. These two objectives are closely related and the GUI will partly be used to demonstrate how the virtual networking can be done. Below follows a description of the components that will need to be developed to reach the expected results.

Back-End Server

To be able to control the underlying cloud platform from a GUI, we need a back-end controller which will work as an intermediate layer. This controller will receive

messages from the GUI and translate them into calls against the platform.

This layer is needed if there are extensions to the platform that has no easy way of getting accessed through the original API and it also makes it easier to extend the GUI with functionality outside the scope of the cloud controller platform. These GUI extensions could for instance be monitoring of virtual machines, or even monitoring of applications within a VM. It could be a way of accessing a terminal of a VM running inside the cloud, making the user control the whole system through only one graphical interface.

Virtual server templates The system must be able to handle different types of virtual machines (different roles). A user can select different roles when launching a machine and the system should also provide the user with the possibility to create new roles with different settings. A role is equivalent with a virtual server template, which is an operating system image file together with defined hardware properties.

Graphical User Interface

To be able to demonstrate network configurations in an easy way there is a need for a Graphical User Interface (GUI). There should be an easy way to deploy several Virtual Machines on a cluster of nodes without the need for entering each VM and do configurations.

User hierarchy This GUI should have two modes, one user mode and one administrator mode. As a user you would log in, receive your current projects and then be able to edit the current configuration. To make this possible, the GUI must receive the user state from the back-end controller which must save it in a persistent storage.

As an administrator, you would like to see the status of all running virtual machines in the system, independently of which project they belong to. Also to see where the virtual machines are deployed, i.e. on which physical node the VM is running. The latter is necessary so the administrator can observe if there is a need for load balancing in the system etc.

Virtual networks spanning over multiple server clusters

To deliver a prototype for creating virtual networks spanning across cloud platforms spread over various locations requires several steps before fulfillment. First of all, a complete definition of the concept "distributed cloud" and what functionality to expect from it is required. The term *distributed cloud* is from a user's perspective a single cloud platform but it is spread across multiple data centers. Why a complete definition of a distributed cloud is needed, is that this type of architecture does not exist yet. Secondly, how the collaboration between the different cloud platforms should be done also needs to be defined. In addition, the inter-connection needs

to be developed and/or introduced into the solution and if networking should be enabled across this inter-connection even further issues needs to be considered.

What is listed above is all feasible, but not with-in the time frame of this project. Because of this the creation of virtual networks spanning over multiple server clusters will not be considered for the resulting prototype. However, the ideas of how to solve the issues concerning virtual networking will be discussed further in this report.

1.3.2 Evaluation procedure

The developed prototype, as a result of the project, should be evaluated according to certain criteria. These should be mapped to what functionality to expect from the prototype as well as what requirements it should meet. Further, a literature study will be performed covering earlier approaches to solve similar issues. This will be used as a comparison with the resulting prototype of this project.

Comparison

The time frame of this master thesis project does not allow installation of several different software suites for testing their respectively functionality. Due to the time constraint, the comparison will be conducted using existing documents. This requires well documented solutions of what functionality each solution has to offer, whilst they will otherwise be assumed not to support the specific functionality.

As one of the main objectives of this thesis covers network control this is what will be mainly considered when comparing to related solutions. A list of subjects to compare are:

- Ability to separate user networks on the physical level
- Ability to create virtual network topologies
- Existence of graphical tools
  - If such a tool exists, how easy is it to use?
  - What functionality is provided?
  - Extendability

Verification using user stories

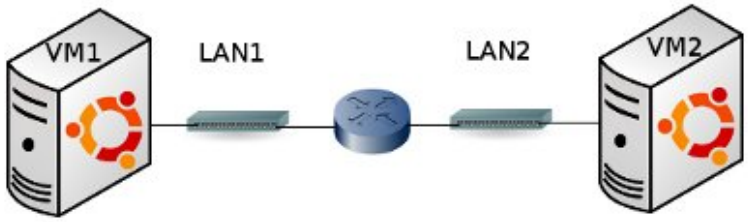
To be able to test and verify that the implemented system behaves as it is supposed to, we will use predefined user stories and evaluate the result of them. These user stories should be created to test, e.g. if virtual machines located on different hardware nodes can be configured to be located in the same virtual network, if the communication between different projects are rejected etc. Here is a list of possible network scenarios that the system should be able to handle:

- Two virtual machines on different hardware hosts connected to the same virtual bridge. This scenario is a very simple network topology but it is demanding that the two machines are connected to the same virtual bridge, meaning that they are located in the same broadcast domain.

Figure 1.1. Two virtual machines connected to the same virtual bridge

- Two virtual machines connected to two separate bridges that communicates via a router. This is the simplest possible scenario where two virtual machines are located in separated LANs. All traffic from VM1 to VM2 must now go through bridge number one, then through the router and finally through bridge number two.

Figure 1.2. Two virtual machines connected to two separate virtual bridges



Manage the system state Not only should the system be able to handle different network topologies, it should also be able to manage the state of a users project. An example of this is that if the client is requesting a network map for a specified project, the system must be able to control if all instances represented in the network map is in fact running. If not, the user must be notified in some way.

1.4 Thesis outline

The rest of this thesis is presented as follows:

The second chapter, 2, describes the main concepts of cloud computing including lower level technology and tools.

In chapter 3 there is a description of network concepts that are used inside cloud environments.

Presentation of related work is located in chapter 4, where an overview of cloud platforms can be found.

Chapter 5 describes the architecture of the solution presented in this project, and the implementation of the solution is described in chapter 6.

The evaluation of the implemented software is located in chapter 7.

Results and Conclusions are presented in chapter **??** and 8 and is presenting a discussion about the results combined with a conclusion. A future work section is also provided where possible extensions to the solution is presented together with objectives that are not yet implemented.

Chapter 2

Concepts of Cloud Computing

2.1 Introduction to Cloud Computing

The concept of Cloud Computing is quite broad and this section will clarify the most common parts of a cloud today. A Cloud in its simplest form is a cluster of servers where users can deploy data, services or even complete virtual machines.

2.1.1 Different cloud delivery models

The three most common types of cloud delivery models are Software as a Service(SaaS), Platform as a Service(PaaS) and Infrastructure as a Service(IaaS). The three different types are described as follows:

Software as a Service (SaaS)

Software as a Service is when users are provided with applications that runs over a network. Instead of having to install the necessary software on each user's computer the installation and execution is done inside the cloud making it easier to maintain the delivered software.

Platform as a Service (PaaS)

Platform as a Service is a solution where complete platforms are placed in the cloud and users can develop applications that runs on these platforms. Instead of owning their own IT infrastructure, users develop applications using cloud specific APIs and uploads their applications to the cloud.

Infrastructure as a Service (IaaS)

Infrastructure as a Service gives the users even more possibilities than the other two types of cloud systems. Now a user can be provided with a complete computer infrastructure in a virtualized way. Instead of buying computer hardware, the user is provided with computing power through virtual machines.

In this way a user can upload complete virtual machines and install whatever software they think is needed.

The solution for this project will be using the last named type, Infrastructure as a Service, which means that the system will handle virtual machines in the cloud. To be able to manage the life cycle of these virtual machines, each physical node need to be equipped with a manager called hypervisor.

2.2 Hypervisors

A hypervisor or virtual machine monitor is a technique for running multiple virtual machines (guests) on the same hardware platform (host). The hypervisor provides a platform for each virtual machine and can then monitor the execution of the virtual machine. The guest operating system sees the platform it is running on as it would be regular hardware, meaning that the original features of an OS can be used without, or with small configuration changes.

Examples of hypervisors are Xen, KVM, VMWare, VirtualBox and Microsoft Hyper-V, where Xen, KVM and VirtualBox are open source.

This project aims at using open source solutions to build on-top. Therefore, even if VMWare and Microsoft Hyper-V are widely spread and used as of today, they are not suitable because of the fact that they are not open source. The remaining hypervisors, Xen, KVM and VirtualBox, are on the other hand open source and therefore of interest.

Xen were used as a base in the VIBox project and since this project is the succeeding of VIBox, Xen are thoroughly discussed in this section. KVM is a hypervisor solution for Linux and many open source cloud platforms on the market has support for KVM hypervisor as well as Xen. Even though VirtualBox is open source, there is no widely spread support for it in open source cloud platforms as of today, hence VirtualBox will not be considered during this project.

There are two major types of virtualization techniques provided by hypervisors. One is called full virtualization and the hypervisor benefits virtualization provided by hardware and that virtual machines running on top efficiently can utilize the underlying hardware. The other technique is called paravirtualization were virtual machines does not have direct access to the underlying hardware, but instead the hypervisor emulates the hardware. Using this technique, the virtualized operating systems need to be aware of being virtualized and thereby requires slightly modified operating systems, while the former virtualization technique allows unmodified operating systems to execute as if on a physical server.

2.2.1 Xen

Xen is an Open-source platform which allows virtualization of multiple operating systems on a single machine. The platform enable full virtualization, or Hardware Virtual Machines (HVMs), which requires hardware support and para-virtualization. This section aims on clarifying the main components provided by Xen and also some

third-party services accepted by the community for management and control of Xen and the virtualized systems within. [27]

Components

A computer running Xen hypervisor contains three components: First of all Xen Hypervisor which runs directly on computer hardware beneath the operating system. This layer allows multiple guest operating systems to run concurrently on top of the hardware. The second component is Domain 0 (Dom0), which is a privileged guest operating system with direct hardware access and guest management responsibilities. The third and final component is a set of Unprivileged Guest Domains (DomU). They have no direct hardware access and are managed by Dom0. The components just mentioned will be presented more thoroughly under separate headings below, respectively.

Xen Hypervisor

Xen Hypervisor is the base abstraction layer of Xen and sits directly on top of the hardware below any operating system. It it responsible and are aware of CPU scheduling and memory partitioning between virtual machines running above. However, it has no knowledge of networking, external storage, video or other I/O devices.

Dom0

Domain 0 (Dom0) is a unique virtual machine running on top of Xen Hypervisor. It has privileged access to I/O devices as well as interact with less privileged virtual machines (DomU) running on Xen Hypervisor. Two drivers are included in Dom0 to support network and disk requests from DomU Paravirtualized (PV) guests; the Network Back-end Driver and the Block Back-end Driver

DomU

In contrast to Dom0, DomU guests have no access to I/0 devices, in the normal case. Therefore they have to, via the Xen Hypervisor, communicate with Dom0 to get access to disk or network devices. Further, there exists two approaches of virtualization in Xen; one were the virtualized system is unaware of being virtualized called full virtualization or HVM guest and the second approach is called Paravirtualized guest or PV where the virtualized machine is a modified operating system aware of being virtualized. To get hardware access Domain U PV have two main drivers included; the Network Driver and the Block Driver, which they use to get access needed. A Domain U HVM Guest does not have the PV drivers located within the virtual machine; instead a special daemon is started for each HVM Guest in Domain 0, Qemu-dm. Qemu-dm supports the Domain U HVM Guest for networking and disk access requests. Further, Domain U HVM must initialize as

if on a physical machine, why a software called Xen virtual firmware is added to simulate the BIOS an operating system would expect on startup.

Management and Control

There are several services that supports the overall management and control of the virtual machines running on Xen Hypervisor. They all exists within Dom0. Some of the existing tools that are provided by Xen for control and management are the following:

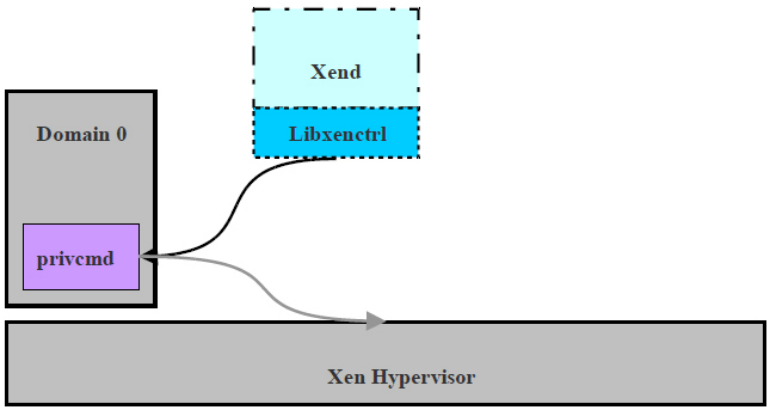
- Libxenctrl - Libxenctrl is a C library that provides the ability to talk with the Xen hypervisor via Domain 0. A special driver within Domain 0, privcmd delivers the request to the hypervisor.

- Xend - The Xend daemon is a python application that is considered the system manager for the Xen environment. It leverages the libxenctrl library to make requests of the Xen hypervisor. All requests processed by the Xend are delivered to it via an XML RPC interface by the Xm (see below) tool. * Xm The command line tool that takes user input and passes to Xend via XML RPC.

- Xenstored - The Xenstored daemon maintains a registry of information including memory and event channel links between Domain 0 and all other Domain U Guests. The Domain 0 virtual machine leverages this registry to setup device channels with other virtual machines on the system.

To clarify the dependencies among above mentioned services we present an illustration provided by Xen.

Figure 2.1. Control flow of Xen management and control of DomU guests



2.2.2 Kernel-based Virtual Machine (KVM)

Kernel-based Virtual Machine (KVM) is a solution for running virtual machines on a physical host using the Linux kernel as the base. As a hypervisor is a special type of operating system, KVM can make use of already existing functionality of the Linux kernel. Each VM running on a KVM system is treated as a regular process, which means that the VMs can easily be scheduled and also swapped if necessary.

KVM was designed after the hardware manufacturers started to provide hardware virtualization and does not need para-virtualization, like Xen. Instead KVM requires and benefits on hardware virtualization support such as AMD-V or Intel VT-X. This given, the guest operating systems are unaware that they are running on a KVM hypervisor.

Moreover, KVM uses the underlying architecture of Linux to provide the hypervisor with necessary components to control virtual machines, such as memory manager, process scheduler, device drivers, network stack etc. This is feasible as KVM were designed only to provide virtualization supported by hardware.[24]

Due to the design choices made by KVM development team, there is little information about what KVM supports. Instead, generally speaking, it can be stated that what Linux supports, also KVM supports.

2.3 Hypervisor management tools and APIs

There are a lot of tools for controlling the hypervisors on the physical nodes. Many tools are command line based and are using common APIs to access the hypervisors. Here are two APIs described that are widely deployed and has become de facto standards for cloud computing management tools.

2.3.1 Amazon Elastic Compute Cloud (EC2) interface

Amazon Elastic Compute Cloud (EC2) is a web service API, developed by Amazon, where users can configure compute capacity in the cloud environment. By using the EC2 interface, a user is provided with functions like life cycle control of virtual machines, image management and security group management etc.[1]

Amazon is one of the world leading suppliers of cloud services and their EC2 service has been accepted as a standard by the open-source community. Because of this, all large open-source cloud platforms are providing an implementation of it.

Systems that uses the EC2 specification often include Amazon's Simple Storage Solution (S3)[2]. This is a simple online storage which is accessed through web service calls. When moving systems to a cloud environment, users often need plenty of space to store their data so EC2 and S3 services are often strongly coupled. Many open-source solutions also have their own implementation of the S3 storage solution.

Tools using EC2

Euca2ools[8]: command-line tool that was originally developed by the Eucalyptus[9] project, described in section 4.1.4, and can be used to control virtual machines on any cloud environment implementing the EC2 API.

2.3.2 Libvirt

Libvirt is a free API for interacting with virtual machines and hypervisors. It provides functionality of creating and configuring virtual machines in several types of hypervisors, such as Xen, KVM, Virtual box and VMWare. The base library of Libvirt is written in C but there are also several bindings for other common languages, such as java, python and C#.

In Libvirt a Node is the physical machine that runs the virtualization software. In each node there is a hypervisor controlling the above domains that represents running operating systems. Libvirt is made to be used as lower level building block where other applications can be built upon. There are several applications today that use the Libvirt API, both graphical user interfaces and command line based tools.[11]

XML format

Libvirt uses XML as the main solution for handling configuration of domains. The configuration files contains information about hardware resources, external devices and how the domain should be started.

A virtual machine is described in an XML file with the root element called domain. This domain element holds information about what hypervisor will be used and the id of the virtual machine.[10]

Listing 2.1 shows an example of a Libvirt XML that specifies a device for a virtual machine:

Listing 2.1. Libvirt devices example

```
<devices>
        <disk type='file'>
        <driver type='qcow2'/>
        <source file='/var/lib/nova/instances/instance
            -00000091/disk'/>
        <target dev='vda' bus='virtio'/>
        </disk>
</devices>
```

Tools using Libvirt

Libvirt is widely deployed and used in many cloud platforms, and there are many tools that are built on top of the Libvirt API. Here are some common tools:

- virsh - command line program for interacting with hypervisors
- virt-viewer - a tool for accessing the console that is associated with a virtual machine

Chapter 3

Networking in Cloud environments

To better understand why problems occur when enabling networking within a cloud environment, this section present some of the most common problems and techniques to handle them. Problems arises on different layers of the OSI-stack, mostly link layer (layer 2) and network layer (layer 3). Two problems are to separate running instances, and that network configuration in the virtual world should not be dependent on the configuration of the physical world.

3.1 Virtual network introduction

The concept of virtual networks could possibly lead to some confusion, why this introduction aims at clarifying what this imply considering a cloud environment.

The virtual network in the context of a cloud is an abstraction of the underlying network in the sense that virtual equivalents are used instead of the actual hardware. Because, even if the underlying hardware is abstracted and does not interfere with the virtual network, virtual equivalents of the hardware exists were all the rules of normal networking still holds and needs to be considered. However, the virtual network topology could possibly be totally different from that of the physical one.

3.1.1 Virtual network equipment

Mapping of hardware network equipment into its virtual equivalents means that physical servers, switches and routers need to be conceptualized in a virtual manner. The physical server is represented by a virtual machine. A router could also be a virtual machine with routing capabilities enabled.

A switch is somewhat different. When representing a physical network switch in a cloud environment it is in the form of a bridge which is a software representation of a physical switch. Several can exist on the very same physical machine even if only one interface is present. The bridge could either be a Linux bridge or an OpenVSwitch bridge, discussed in section 3.5.1. Further on in the report, whenever a bridge is mentioned, the virtual representation of a physical switch is what it

refers to. A bridge is constrained to the same networking rules and capabilities as a physical switch and supports for example VLAN, discussed in section 3.4.1.

3.2 Separation of virtual machines inside a cloud

One important part in a cloud environment is to be able to separate virtual machines from each other. First, instances belonging to different users should not be able to communicate like they were connected to the same LAN. There must not be a possibility for a user to manipulate virtual machines that belongs to another user. Also machines belonging to the same user could be separated to simulate that they are connected to separate networks.

To separate instances that run on the same hardware, different techniques can be used. One is to separate machines by putting them on different broadcast domains. This can be done by using different VLANs for different instances. By putting two machines in different VLANs they can not find each other through regular ARP requests. More information about VLAN can be found in section 3.4.1.

Another approach is to filter all traffic that is received by a machine. If a machine is receiving a packet that comes from an instance that is not allowed, the packet will be rejected. To filter traffic on instance level either ebtables or iptables, described in section 3.4.2, can be used.

3.3 Network configuration in the virtual world

To provide the possibility for a user to create virtual networks in a cloud there must be some way to use common network concepts. A virtual machine can have an IP address, netmask and a gateway just like a physical machine. Two machines that are hosted on the same physical hardware and are located in different virtual LANs must communicate via a configured virtual router. If the default gateway for an instance is a third machine, the traffic must now go through that instance and not through any internal path on the physical host. To be able to create internal networks it is of big importance that machines located on different virtual LANs, cannot find a direct path between each other.

3.4 Separated broadcast domains

This section describes how a separation of broadcast domains can be created. The two approaches described are VLANs and filtering using ebtables or iptables.

3.4.1 VLAN

Virtual Local Area Network (VLAN) is a network technology that has been developed to structure networks as they grow in size and complexity. A VLAN is, in its simplest form, a collection of network nodes grouped together on a single broadcast

domain, meaning that they are separated from other network nodes on that switch. Communication between VLANs requires the use of a router. Several VLANs can be created on one single switch and one VLAN can span over several switches. For VLANs on several switches to be able to communicate via a single link you must use a process called trunking. To create a VLAN you need to login to the switch and specify specific parameters; name, domain and ports. After creation, any network segments connected to the assigned ports will become part of that VLAN.

VLAN (802.1Q)

To specify that a packet is associated with a specific VLAN, there is a special tag. The tag is inserted in the Ethernet frame and makes the frame 4 octets longer than a regular one. This is done so the original content would not be changed.

Every port on a 802.1Q compliant device, e.g. a switch, can be used for tagging or untagging. If a port has the tagging property set, it will set all VLAN information on each packet that flows in or out of the port. If the packet already has the VLAN information set, it will just forward the packet without editing the VLAN information. A port that is set to untagging, will do the opposite of the tagging port. It will remove the VLAN information from the packet and make it possible to forward the packet to a device that is not 802.1Q compliant.[7]

It is very important to configure switches that should be used in a VLAN enabled network. Each port on the switch should be configured to enable or disable packets from certain VLANs

3.4.2 Host based filtering using ebtables/iptables

To control the incoming requests on each host one can use the Linux ebtables filtering. This can be used to separate broadcast domains if the filters are configured properly. When an ARP-request has reached the virtual machine the ebtables filters should make sure that only requests from the same LAN will be forwarded, otherwise the request should be rejected. There is also the possibility to use iptables which will work similar to ebtables but focusing on the IP-packets instead of the ethernet frames.

3.5 Ongoing projects

There are several ongoing projects on how to simulate hardware network devices in a virtual environment. This section describes some technologies and protocols that are used to control virtual network setups.

3.5.1 OpenFlow

OpenFlow is an open switch specification created for researchers to be able to run experiments inside a production network. Most switches and routers contain inter-

nal flow tables but each vendor has its own table and set of functions. OpenFlow provides an interface to configuring the flow tables on these hardware devices, using a common set of functionality that these devices have.

In regular switches and routers, the data forwarding and the control function(routing) is embedded in the same device. In OpenFlow, there is a separation of these functions and the routing decisions are moved to a separate controller while the packet forwarding still use the regular switch. There is communication between the OpenFlow enabled switch and the controller part, and that is done using the OpenFlow protocol.

If an OpenFlow switch receives a packet that it has never seen before, i.e. if no packet fields in its flow table matches the received packetâ€™s header, it relies on the controller. It forwards the packet to the controller which will make a decision, if it should be dropped or forwarded. If the packet is supposed to be forwarded, the controller adds a rule in the flow table of the switch so it knows what to do with similar packets in the future.[13]

OpenVSwitch

OpenVSwitch is a multilayer virtual switch that can operate both as a soft switch within a hypervisor and as control software to act as a hardware switch. It can run inside any Linux-based virtualization environment and is the default switch in the Xen Cloud Platform (XCP), see section 4.1.4. Further it has been adopted as the default switch, and thereby replaced the Linux bridge, in the OpenNebula project and is also in the pipe to be adopted by the OpenStack project. More about these projects, how they relate to this project and their respectively benefits can be found in chapter 4.

OpenVSwitch has support for the OpenFlow protocol, in addition to switching standards, including standard 802.1Q VLAN model with trunking. Further, OpenVSwitch was designed to support distribution across multiple physical servers which makes it suitable in cloud environments.[22]

3.5.2 NOX - Operating system for networks

NOX is an open platform which can be seen as an operating system of Open Flow enabled switches. On top of NOX, management applications can be developed which controls all the switches in the network. Thereby one can control the traffic in the entire network. As NOX has adopted OpenFlow switch abstraction, a controller application can specify what action should be taken on a packet matching the specified header. Examples of actions are: forward as default (as if NOX were not present), forward on pre-specified interface, deny, forward to a controller process or modify various packet header fields (i.e. VLAN tags, source and destination IP address and port number).

NOX in combination with OpenVSwitch would introduce possibilities to develop sophisticated applications to control network traffic in a virtualized environment,

such as a cloud. Though, as this project aims on supporting physical switches available in data-centers today and off the shelf OpenVSwitch products are not available yet, NOX is not suitable for managing the physical network. However, NOX could be useful for controlling the virtual networks in a cloud environment were OpenVSwitch is running as a soft switch within the hypervisors. By abstracting the physical switches, only focusing on the soft switches one could benefit NOX to create separate virtual networks, perform experiments and do measurements of network traffic within each network without the risk of interference from other virtual networks running on the same physical network.[12]

3.6 Virtual networks spanning over multiple data centers

The final goal, where this project plays an important role, if looking at the big picture is to be able to enable virtual networks spanning over multiple data centers. It is not in the scope of this project to solve this issue. Instead, this project has focused on solving the virtual networking problem at one single data center. However, this section is intended as a generalized blueprint of how to grasp the problem and from what angle this project look at it. This to finally be able to solve the issue of virtual networks spanning over multiple data centers. To further clarify, do not read this section as if a solution to the problem. It is more of guidelines of how it could possibly be solved.

3.6.1 Network as a Service (NaaS)

Network as a Service (Naas) is a new term describing an extension to IaaS where also network equipment is considered. The term NaaS is not adopted by the community but used for instance in the OpenStack project.

Amazon has in a beta stage a service called Virtual Private Cloud (VPC) which makes it possible to create private networks in the cloud. The service has several restrictions such as it is only available in two regions and the most essential is that the VPC must be within a single availability zone. [3]

Other platforms such as OpenStack, described more in section 4.1.2, has NaaS as ongoing work. OpenStack has the focus on a single data center, and their blueprints in the area do not consider multiple data centers. Based on the two above services, NaaS is as of today not covering virtual networks spanning over multiple data centers.

3.6.2 Extending the concept

Extending today's NaaS architecture would be to move from covering one data center to multiple data centers. This would introduce the possibility to spread virtual machines over multiple data centers and let the communication between them only depend on a defined virtual network topology.

Figure 3.1. Virtual network spanning over multiple server clusters

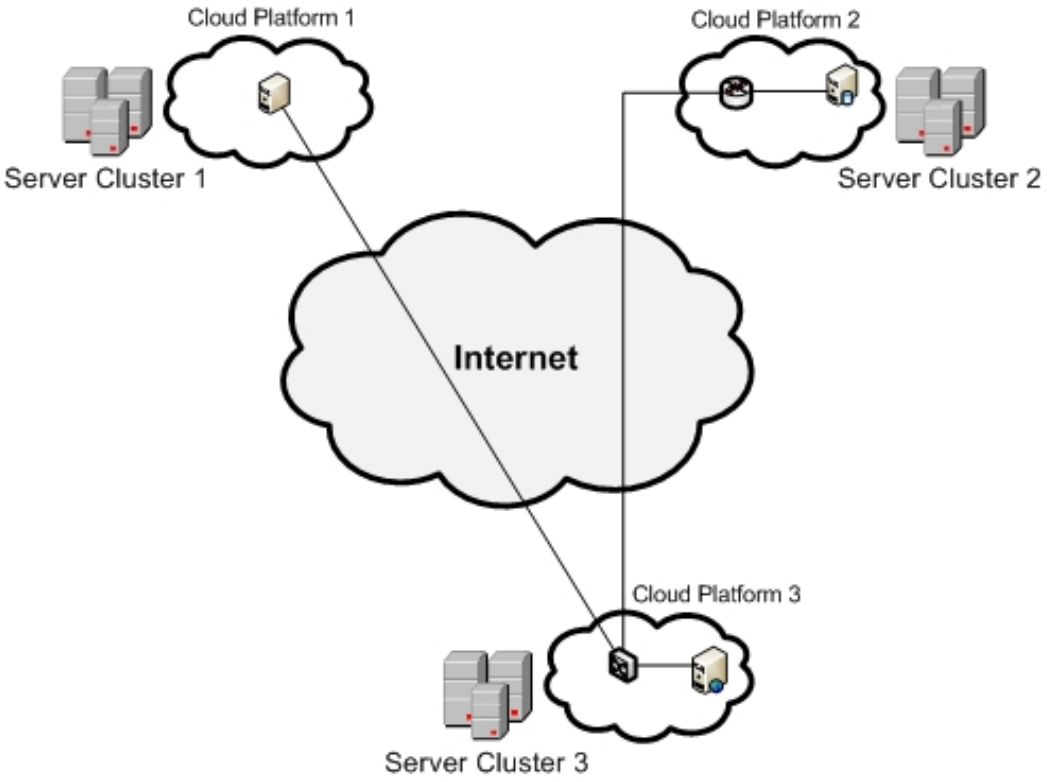


Figure 3.1 illustrates a virtual network setup that would be possible in such a solution. Each data center hosts a distinct set of virtual network equipment. They can be connected over a virtual network, which can span only locally or over several long distance located data centers. The connections spanning across the Internet should preferably not be on dedicated links, but instead on dynamic link paths setup via already existing equipment. This network traffic should by default not be public. Instead a user of the system should be able to set the ingoing and outgoing points of the virtual network towards the public Internet.

Suppose that each data center separates virtual networks using VLAN, as described in section 3.4.1. These VLANs will not be able to propagate across the Internet unaltered. One idea might be to let each data center be equipped with a translator module. This would be responsible for translating a local VLAN to Internet link traffic and finally from Internet link traffic back to a local VLAN.

MESON

The MESON project provides a layer 2 link between two points making it look like a single cable. This link is setup on request via a web service call. With this link setup, two machines located in different data centers should be able to communicate

with each other on layer 2. The machines will appear as if they are connected to the same LAN. Applying this to what is described above means that MESON would be able to setup the links spanning across the Internet in figure 3.1.

Chapter 4

Related work

This section contains description of related technologies, where the focus is on entire platforms for managing cloud environments. Last there is a discussion of what technologies, platforms and tools that will be used during this thesis project.

4.1 Existing cloud platform solutions

There are several commercial cloud services available, like the famous Amazon cloud solution. However as this project might require extension of the core of the cloud platform, only open-source platforms will be considered.

This section describes a collection of these open-source platforms that are used in a large scale today together with the VIBox project. Among these platforms there might be interesting ones that can be used as a base for this thesis project.

4.1.1 Virtual Internets in a Box - VIBox

The VIBox solution is the base of this thesis project and they have solved many problems on the way of creating a full cloud platform. Image handling, configuration of virtual machines and life cycle control are concepts that are partly already implemented. Concepts that are missing in this platform are for instance users, projects and persistent storage. The approach when creating VIBox was not to create a full user friendly cloud platform but a solution to be used within a test environment.

Architecture

The VIBox platform uses Xen as hypervisor and communicates with each hypervisor using the Libvirt API. The Back-end core is implemented as a pool of threads receiving simple calls using regular strings. There are no documented API how to communicate with the platform and each function like e.g. starting an instance is consisting of several calls to the platform.

Each call to the platform will be received, evaluated, reconfigured and then be forwarded to the hypervisor that is responsible for executing the command. The platform itself does not keep any records of what calls have been made or which hypervisors has received what command. As no state is kept there is no way for a user to continue on an ongoing project.

Networking

The networking in VIBox is very free as the platform allows almost any (or can be extended to support any) type of call.

The platform can receive and execute some OpenFlow rules for the bridges in the virtual network. All bridges are replaced with OpenVSwitch An example of a OpenFlow rule is that the system could control the path where packets are sent between multiple physical switches.

4.1.2 OpenStack

OpenStack is a collection of open source tools for managing private and public clouds. It consists of three main development components, OpenStack Compute (Nova), OpenStack Object Storage (Swift) and OpenStack imaging service (Glance). The OpenStack software was founded by NASA and Rackspace Hosting and is now a large software community with over 50 companies supporting the project.

OpenStack provides, like many other platforms, an implementation of the Amazon EC2 web service interface. The current instructions how to setup virtual machines in the cloud is through the EC2 API.

Compute (code-named Nova)

The back-end server in OpenStack, responsible for orchestrating the cloud platform is called Compute, code-named Nova. OpenStack Compute introduces the concept of projects, each equipped with resources, separated from other projects. In our view, a project can be viewed as a customer of the cloud platform or a research team working in the laboratory using the internal cloud platform. Each OpenStack project consists of separate networks, volumes, instances, images, keys, and users.

OpenStack compute can work with several types of hypervisors including the Xen and KVM described earlier. The default hypervisor is KVM as the presented installation guides uses an Ubuntu release as operating system.

Besides the Compute project, OpenStack also provides an Object storage and an imaging service. These two are code named Swift and Glance and are supposed to be used later on instead of Compute's built-in object storage and imaging service. For smaller projects and test networks, the embedded nova components are enough. If you are attempting to create a cloud environment consisting of thousands of nodes, you probably want to use Swift and Glance which is created to be very scalable. As Nova has an implementation of the Amazon EC2 web service interface, the storage solutions for OpenStack have modules for interacting with the Amazon S3 service.

An example of an architecture of servers using OpenStack could be like the following, consisting of a single Controller node and several worker nodes:

Cloud Controller:
The Cloud controller node in a Nova architecture holds all the Nova services including a database that is compatible with SQLAlchemy, i.e. MYSQL or Postgresql. This node is responsible for managing the cloud, including network, API server and the scheduling of instances.

Compute-nodes:
A Compute node in Nova is a node that runs the nova-compute service but refers to the database on the cloud controller node. The compute node hosts the Virtual machines but do not control their setup.

A regular cluster setup consists of one cloud controller and multiple compute nodes, but in really large clusters the cloud controller services may be spread across several hosts for load balancing purpose.

Networking

The network controller in Compute provides the possibility for computing nodes to communicate with each other and with the public network. There are three different types of networks supported in Compute, all implemented as different managers. These are: Flat network, Flat DHCP network and VLAN network.

In the Flat network, all instances are attached to the same bridge which is configured by the network administrator. Each instance receives a fixed IP address from a pool of addresses which is defined by a subnet. This address is injected to the image file during the launch of the virtual machine.

For simplicity in the base setup the physical nodes (hosts) and the virtual machines (guests) are connected to the same bridge.

In a Flat DHCP network, there is a DHCP server that passes out the IP addresses from the specified subnet. In both Flat network an Flat DHCP networks, each node has a public IP address.

The third type of network, the VLAN network, is the default network mode in OpenStack. The system creates a specific VLAN and a bridge for each project to be able to enclose the virtual network. The IP addresses for the instances running in the project is given by a DHCP server. These addresses are private, belonging to the same subnet and they are only reachable from inside the same VLAN. For a user to be able to reach the virtual machines from outside the project, there is a need for a special VPN connection called cloudpipe.[21]

Using any of the above mentioned network managers, the physical nodes have iptables and ebtables rules that are created for each project. These rules are to protect against e.g. IP address spoofing and ARP poisoning. The rules are making it impossible to send an IP packet or MAC frame with a source address different from the real source.

4.1.3 OpenNebula

OpenNebula is a community driven open source project used to build and manage IaaS clouds. It supports various hypervisors, including Xen, VMWare and KVM. To access and remotely manage the virtual environment OpenNebula provides three different interfaces. Firstly an implementation of Amazon EC2, which is described in section 2.3.1, and secondly an implementation of OCCI [15]. These two are both accessed via RESTful web service APIs. The third one is accessed through a web interface and is called OpenNebula Sunstone. All three access mechanisms can be used in parallel.[18]

The architecture of an OpenNebula system is a cluster like architecture with a front-end and one or several nodes, running the hypervisors. The front-end is running the OpenNebula software and has a connection to an image repository that will be shared with the cluster nodes. The image repository will be exported to the nodes as a shared file system, meaning that the nodes will have access to the image files needed to start the virtual machines. The size of the repository must be large enough to hold not only the operating system images at the start stage, but also the copies needed when the virtual machines are being launched.[17]

The basic components of OpenNebula are the following:

- OpenNebula daemon - a daemon to orchestrate the system and control VMs life-cycle.
- drivers - the connection to different cluster systems like storage or hypervisor.
- scheduler - The scheduler is responsible for the assignment between virtual machines and the cluster nodes.

The OpenNebula daemon, listed above, is the core of OpenNebula. This consists, in turn, of a set of components to control virtual machines, virtual networks, storage and hosts. This is realized by the following managers:

- Request Manager - handles client requests through XML/RPC interface
- Virtual Machine Manager - manages and monitor running VM instances
- Transfer Manager - is in charge for file transfer needed on VM deployment
- Virtual Network Manager - is responsible for handing out IP and MAC addresses and their association with VMs and attached physical bridges
- Host Manager - manager and monitor physical resources
- Database - persistent storage to keep current state of OpenNebula core

Networking

In OpenNebula several virtual networks will share the same physical network. To isolate the different virtual networks from each other it uses filtering on the ethernet level, making it impossible for machines in different virtual networks to communicate. The filtering is made locally on the cluster node running the virtual machine, using ebtables. To be able to configure the ebtables, the OpenNebula administrator need to be in the suedoers file on each cluster node with no password protection.

The documentation regarding isolation of virtual network on the homepage of OpenNebula is not complete in the sense of what type of virtual machines are allowed to execute on that type of network. However, OpenNebula explicitly explains a scenario were a DHCP server is placed in a virtual network to handle the VM's IP addresses.

As explained above, regarding virtual networks, the filtering of network traffic is done on each cluster node, individually. Thus, any physical switch in the cloud environment / data center is seen as a patch panel connecting cables together. This network setup could potentially be erroneous as separate virtual networks are allowed to, through for example broadcast, cause congestion on the shared physical network.[16]

4.1.4 More solutions

As described earlier there are numerous open source solutions for cloud management. Except OpenStack and OpenNebula, here are a few other widely deployed solutions.

Convirt

Convirt is a management tool for handling virtual machines. It works with Xen hypervisor and KVM and It comes in two different versions. One open source solution and one enterprise edition covering even more functionality than the open source. The Open source solution of Convirt doesn't have a command line interface or a programmatic API, which makes it very hard to extend the solution with company specific demands. As there are no good ways of communicating with Convirt Open Source, this is not a good solution for our open source project.[6][5]

Eucalyptus

Eucalyptus is an infrastructure for implementing cloud computing. It is provided in two editions, one enterprise edition and one open-source edition. The enterprise edition is not considered but the open-source edition is a potential starting point. Eucalyptus uses Libvirt to perform calls to hypervisors and it is also compatible with Amazon EC2 and S3, just as OpenStack and OpenNebula. Also eucalyptus is widely used and it is the core element of the Ubuntu Enterprise Cloud (UEC).[9]

Xen Cloud Platform (XCP)

Xen Cloud Platform (XCP) is an open source cloud platform, delivering the Xen hypervisor to host guest operating system. XCP provides an API, called XAPI, for control and management of the Xen hypervisors within the cloud and the virtual machines (guest) executing on top of these.

The platform addresses cloud providers who wants to support Xen hypervisors. By developing on top XAPI cloud providers automatically benefits security, storage and network virtualization technologies enabled by the XCP platform beneath.

The platform is derived from Citrix XenServer[28], but is in contrast licensed under GNU General Public License (GPL2) and fully open source.

When XCP was designed to be a software layer as part of a larger surrounding cloud platform, introducing specific support for Xen hypervisors, it is not to be considered a stand-alone complete solution for cloud providers. Instead the platform delivers support for Xen hypervisors and its technologies and tools, more thoroughly discussed in section 2.2.1. On top of the hypervisors is the XAPI, responsible for the orchestration of physical hosts, virtual machines and their configuration and connectivity.[26]

4.2 Graphical management tools

There are many technologies that can be used when it comes to user interfaces and here are some that will be considered.

VIBox

The VIBox project used java for creating their GUI. The main component used for creating the drag and drop ability is a tool called JGraph, which is an open-source project since 2001 and it is still frequently updated. The coding was made in NetBeans so the easiest way to extend the code is to use NetBeans as development platform.

NetBeans platform

The new way of creating graphical applications using NetBeans, is to use their own solution called NetBeans platform. This platform is very good at separating code by placing independent pieces of code in separate modules. Each module can only access files within the same module unless you specify in a module that it can be accessed from others. This separation makes it easier to develop separate pieces in parallel as there is no risk that the code you refine or remove is used by another developer.

When building applications in NetBeans platform, common window components are easily added as modules within the application. These window components can be specified to be placed in very common positions, like e.g. editor, which is placed

in the center of the main window and output, which is placed in the bottom. By having these predefined templates for how the application should look like makes it very user friendly. This as the user can recognize the look and feel of the application from earlier experiences with similar software. Further, the platform delivers a complete window manager. This minimizes the work for developers not needing to handle window movement and so this, the platform creates it for you. The result is an application where the user can customize the placement of window components which suits their needs.

The NetBeans platform have some built-in functionality of drag-and-drop. It also have support for creation of components that are often used in drag-and-drop environment, e.g. a palette holding the tools that the user has access to.

Web-based solution

Another approach is to use a web based user interface making it accessible from multiple types of devices, not only from a PC. This approach has many technologies to use as a base but it makes it harder to reuse the code from the previous project VIBox, which was developed using Java.

4.3 Survey of relevant technologies

Earlier sections have described related technologies and their advantages. Some of the stated technologies will be used as building blocks for the solution created in this project.

4.3.1 Platform selection

To fulfill the objectives of this thesis there is a need for a good starting point of our development. The first thing that must be selected is the cloud management platform to extend and work upon. Because, it is not in the scope of this project to develop an entirely new platform. There are three platforms to consider, VIBox (section 4.1.1), OpenStack (section 4.1.2) and OpenNebula (section 4.1.3).

At the moment VIBox is a platform that can handle image creation, deletion and configuration but it uses its own back-end server that only communicates using simple strings. There are no API to use when calling this platform and it won't keep any states at the moment. As it does not keep states, the client will not be able to receive its ongoing work when reconnecting to the server.

The hardware platform must be known in advance and must be recreated in the GUI to function properly. This includes the network setup. Also on the back-end server there must be a configuration file containing all hardware nodes that are running the hypervisors. Further, the platform is only a prototype.

Due to the issues discussed above, concerning VIBox platform, it is not a suitable platform for this project.

OpenNebula has been around for a couple of years and there is ongoing development of the project. It is an older project than OpenStack, so the platform is more mature and are already used in large scale systems.

OpenStack on the other hand is a very new proposed standard but has the advantage of being supported by a large number of big companies and organizations. This platform seems to be very attractive at the moment as more and more companies are joining their community.

At the moment, internal projects at Ericsson Research are using OpenStack as a part of their cloud environment. If this project uses the same platform it will be easier to integrate it with the other ongoing internal cloud projects.

Compared to VIBox, OpenStack provides more common cloud functionality and they try to be as compatible as possible with Amazon's EC2 which is already widely used. They keep states of multiple projects and it is kept in a SQL database located at the controlling hardware node. If our solution can communicate using EC2 or directly to the database, it will be possible to reuse a lot of functions for regular VM management.

A disadvantage of OpenStack is that it does not provide any good solution for creating networks within networks, at the moment of writing. Their idea of a network is very tied together with a project, meaning that each project receives one single network to run VMs on. To be able to create internal networks within a project, there is a need for extending their network managers source code or using a simple configuration and then add support for these virtual networks. This disadvantage holds also for OpenNebula.

What justifies OpenStack as a starting point, even as an immature platform, is the fact that it is being designed from now on with ideas of network capabilities somewhat equivalent to the objectives of this project. One network architecture that they are presenting is called Network as a Service (NaaS), and is presented as a blueprint[19] by them.

Finally, and probably the most essential advantage on behalf of OpenStack is that many of the worlds leading companies are getting involved, giving the project a momentum not apparent in the case of OpenNebula. A selection of companies involved are Rackspace, Cisco, NASA, Citrix, Canonical etc.

Conclusion

OpenStack and OpenNebula have lots of similarities on paper. OpenNebula is more mature, as when many of OpenStack's features still only exist on blueprints. The usage of OpenStack in related projects at Ericsson is considered to be the main advantage over OpenNebula. Also their highly interesting proposed network functionality in the future, gives it an advantage. Because of this and the reasons stated above, the selected platform to build upon is OpenStack.

The decision to select OpenStack as a platform implies that the underlying operating system that it will execute on will be Ubuntu. The reason for this is that OpenStack is thoroughly tested on Ubuntu. Moreover, tutorials on how to

install OpenStack assume the use of Ubuntu why it is preferred to avoid future setup complications. Ubuntu is equipped with KVM as default hypervisor and the majority of available tutorials address KVM related solutions for various issues, and because of this KVM has been decided as the hypervisor throughout this project.

4.3.2 Selection of graphical management tool approach

The VIBox project delivered a graphical tool to launch virtual machines and create networks between them. A user were able to drag and drop various kinds of virtual machines in a graph. Building networks of virtual machines was not a simple task though as you as a user was required to draw the physical network and adjust the network of virtual machines to suit the physical one. The configuration steps of virtual machines were also quite complicated.

With the issues discussed above put aside, the idea of having a graph with attached palette as in the VIBox application is interesting also for this project. The GUI application of VIBox were developed using java and with the source code available this project can use those ideas, refactor and extend when needed. Further, the NetBeans platform is an extensive java based framework which, with its concept of modules, introduces weak dependencies and thereby makes it easier to implement "good practise" code. By using NetBeans platform as a framework would also imply a more robust GUI application which would still work on future java releases.

Due to the reasons above this project will use java on top of NetBeans platform to develop the GUI application to manage the underlying cloud platform and the back-end server.

Chapter 5

System Architecture

This section describes the architecture of the solution created during this master thesis project. It gives a detailed description of the underlying platform as well as the modules that has been created.

The system is divided into two parts, a back-end server together with a cloud platform and a graphical user interface. Both are described in this section together with how these parts communicate with each other. The implementation details and configurations are described in chapter 6.

5.1 Back-End server

As described earlier in section 4.3.1, we have chosen to use OpenStack as a cloud platform making it the base for the back-end server architecture. The idea is to leave the underlying cloud platform unaltered, making use of the virtual machine life cycle management, image service, the ability to scale up and down, access control etc that is provided by OpenStack. The platform, in turn, will be managed through the developed back-end server. Which, in a conceptual matter, is placed as a layer above the underlying OpenStack cloud platform.

The back-end server controls the cloud programmatically through various APIs provided by OpenStack. The back-end server introduce capability to store network topologies (both deployed and as templates). It defines virtual server templates (VST) which uses the underlying image service, provided by OpenStack, with the addition of extended startup scripts, description, mapped graphical representation (image) etc. When the back-end server is requested to deploy an entire network topology, it is responsible for making decisions in which order machines will be launched, based on predefined priorities. The actual startup commands on the underlying cloud platform are done through provided APIs.

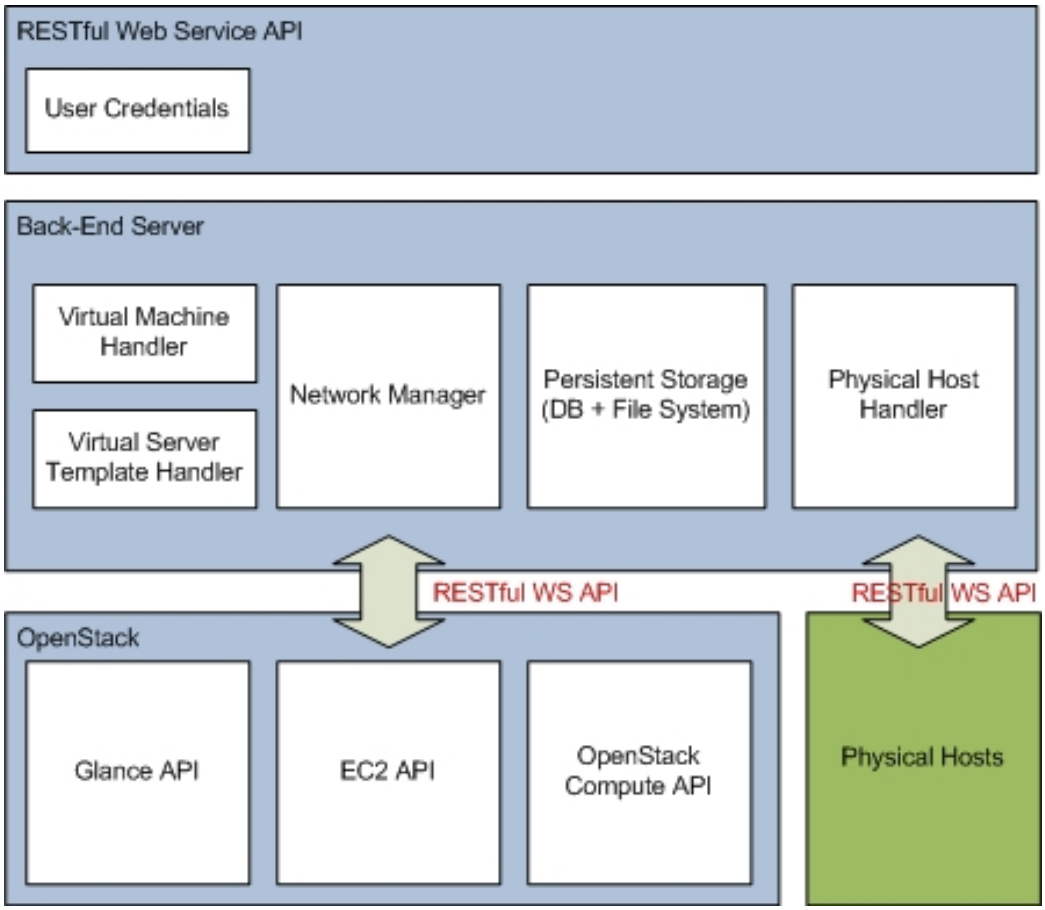
After deployment, the back-end server monitors the virtual machines and can deliver current system state on demand. Not only should it monitor virtual machines, but also the physical servers that hosts them. This to introduce the possibility for an administrator to take actions in case of load balance issues and server faults etc.

Additional monitoring services could be implemented ontop of this, such as network traffic measurements.

Another reason for placing the back-end server as an abstraction layer above the cloud platform is to be able to use the system in a more distributed manner. If several server clusters are used as underlying platform, the back-end server will still act like one interface for the user application. This is one of the future plans in this thesis project.

The developed back-end server architecture is illustrated in figure 5.1. The parts are more thoroughly discussed in the sections below.

Figure 5.1. Back-End Server Architecture, clarifying its modularity and collaboration with OpenStack as an underlying cloud environment



5.1.1 RESTful Web Service API

The back-end server has a RESTful web service interface for the GUI to access. The GUI can use this web service to do the following calls to the back-end

- **networkmap:**
  GET(project): return the network map associated with the project.
  POST: deploy the network map on the OpenStack platform.
- **rolesandcategories:**
  GET: return all the server templates that are defined. This also include network equipment such as routers and switches.
- **project:**
  GET: return all projects that are registered on the back-end server.
- **user:**
  GET(project): return all users associated with the project or all users if administrator.
- **physicalhosts:**
  GET: return data from all the physical hosts in the system. This request is only for administrators.
- **vmsfromhost:**
  GET: return all virtual machines that are running on the specified host.
- **imagefile:**
  GET: return a specified image file (.jpg or .png)
- **authenticate:**
  GET: return a value depending on the authentication of the user. Username and password are sent as parameter to authenticate the user against the Nova database.

Other less common commands are: e.g. rebootinstance(POST), availableips(GET), forbiddenips(GET), virtualservertemplate(POST) etc.

5.1.2 Core Architecture

The core architecture of the back-end server is represented as four main components. A virtual machine handler, a network manager, a physical host handler and a virtual server template handler. The four components are used together with persistent storage in the form of a database and file system.

Virtual Machine Handler

The virtual machine handler is responsible for monitoring the virtual machines in the system. It uses databases to update states of the running instances so the user can get the current states of the machines. This state information is then used to detect any errors in the system, e.g. if a virtual machine has been shutdown or is rebooting etc.

Network Manager

The network manager is responsible for mapping the introduced concept of network topologies into action commands towards the underlying cloud platform. The network manager is responsible for comparing a received network map against already running virtual machines, and determine if machines should be launched, deleted or left untouched. It will also provide OpenStack with enough information to know how the virtual machines should be launched, if they should be equipped with multiple network interfaces and if so how they should be configured. This information must be stored in a database for the OpenStack platform to be able to access it.

Physical Host Handler

The physical host handler is responsible for monitoring all the physical servers included in the cloud infrastructure. Each physical host in the system is responsible for serving the right information about its configuration and its states. In this solution, each physical server hosts a web service that is accessible from the back-end server. This web service delivers JSON encoded messages that tells the system about its current configuration and load. This message is saved on the back-end system and then provided to the graphical user interface on demand. By keeping a copy of the data at the back-end server, the communication to the GUI will be faster as the data from the servers are already retrieved.

Virtual server template handler

All virtual machines that can be launched in this project need to be defined before launch time. The system holds a set of virtual server templates(VSTs) that describes a type of virtual machine, or what role it has. It contains information about what VM-image file it has, what resources it should have and what image file (.jpg or .png) it should be represented with in the GUI. It also holds description of the template together with a category. Each template could be a network switch, a router or a server where switch is not represented with a running virtual machine. The differences between a router and a server are small, e.g. a router is equipped with software for routing decisions.

Persistent Storage

The back-end server is intended to be stateful and to realize this, persistent storage is used in the form of a database and file system storage.

- **Database** - OpenStack provides a database that holds all necessary information about instances, projects and users. For this project to work properly there is a need for even more data storage,

  To store network topologies and virtual server templates etc, there is an additional database that holds this master thesis specific data. The database consists of the following tables:

  - **network_map** - holds the network topology created for each project.
  - **virtual_server_templates** - contains all predefined virtual server templates that can be used in the GUI.
  - **multi_nic/nics** - together holds interface information for each of the virtual machines. This information is used to determine how many network interfaces each machine is launched with.

- **File System** - The system holds images to be used by the graphical user interface and system configuration files in the regular file system. Also instance specific launch configurations are stored as separate files, to easily be added as parameters.

5.1.3 Cloud Platform APIs

OpenStack provides multiple APIs, not only for Compute but also for the image service and the object storage. To work with OpenStack Compute, the EC2 API or Nova Compute API can be used. To work with the image service glance, there is a separate API.

Nova Compute API

Nova Compute API is the base for handling OpenStack Compute. This API is an extension to an existing one from Rackspace and is supposed to contain all the latest features of OpenStack. The API have client code written in python and can be used directly in the back-end system code.

Glance API

To control the image service glance, a special web service API is used. Also this API have client code for python, making it a lot easier to manage the system's virtual machine images.

EC2 API

OpenStack also provides an implementation of Amazon's EC2 API, so earlier developed client applications can be used with OpenStack. This API does not provide any additional features than the OpenStack API, and it is not used in the solution.

5.1.4 Network configurations

The network configuration of the platform must allow the user to launch virtual machines with preconfigured network settings. The default solution for OpenStack networking is to use a DHCP server that takes addresses from a specified subnet.

The configuration is done so that the first available IP address in the database is injected into the image at startup. To manually manipulate the database and place

the next chosen IP address at the highest position will force the network manager to inject the intended address.

The ability to choose whatever IP address you like makes it harder to have multiple ongoing projects in the cloud. OpenStack provides each project with a subnet of addresses, and if the subnet is big enough it should be sufficient. An example of this subnet could be address space 10.1.170.0/24, where the user has 256 addresses to use. It could be divided in smaller internal networks using different netmask than the proposed 255.255.255.0 that represents the /24.

Changing IP addresses on running instances is a feature that is not supported by OpenStack. This is not so strange as with this enabled a user can manually steal IP addresses from other projects making the network separation mechanism fail. An instance receives its IP address from the pool of addresses and will then live with it until termination. To be able to change an IP address, the system must then destroy the VM and start an identical one with another valid IP address from the Pool of addresses. As there is no support for dynamically changing the network settings in OpenStack, this is not used in this solution.

Network configurations

The VLAN-manager in OpenStack provides each project with its own VLAN, Linux bridge, DHCP server and subnet. These automated configurations are really good when you want to create several projects and spin up some virtual machines without caring about network configurations. There is a problem when you want to configure IP addresses, netmasks etc as the DHCP server is pulling this configuration from the database.

The original idea for OpenStack is that each virtual machine is given an IP address on launch time, which is predefined in the database. The solution in this master thesis project is based on the ability to launch an instance with multiple network interfaces. To be able to do so the OpenStack database and code for launching instances need to be rewritten to fit into this new architecture.

Distributed Switches

Every physical node in an OpenStack setup is connected to each other via switches making it possible to communicate all over the cluster. Each virtual node is connected to a bridge that is dedicated to a project. By having all virtual machines connected to the same bridge on each host, the bridge works as a distributed switch. The problem that now occur is that when all instances within the same project have contact via the bridge they can also send ARP requests to each other. If a user wants to separate virtual machines with e.g. a router, some configuration must be done so the VMs cannot detect each other via ARP requests. A solution to this is to be able to chose IP addresses from different subnets on the different VMs, forcing the machines to use routing tables to communicate.

Separate broadcast domains

Projects in OpenStack are separated using different VLANs. There must now be a mechanism for separating internal LANs within each of the projects.

The solution to this problem is to provide a new set of VLANs and network bridges for each subnet created within a project. This means that if the project consists of three switches, representing three LANs, the final solution will use three bridges and three VLANs to create the different LANs. This is done to guarantee that the instances belonging to different LANs cannot communicate using ARP messages.

Management network

The OpenStack system is using preconfigured virtual machine images that needs special network configurations to work properly. A problem with the ability to provide multiple network interfaces with different subnet configurations is that the instances can have different default gateways. If the instance do not have a direct path to its host machine, problems with fetching instance specific meta-data will occur. The virtual machine image is preconfigured to fetch this meta-data on the IP address 169.254.169.254 and needs configuration on the cloud controller in order to be able to retrieve the data. There is a need of a forwarding rule from 169.254.169.254 to the IP address of the OpenStack API server. By using this rule, every instance is able to access the OpenStack API server when launching.

To not disturb the way meta-data is fetched from the API server, this project's solution will keep the original configuration retrieved from OpenStack as a management network. The interfaces connected to this network will only be used for meta-data serving and for the user to be able to ssh connect to the running instances. All other communication will be over the other interfaces that the user has defined. Using this approach, all instances will now have at least two network interfaces so the underlying platform must provide multiple interfaces.

5.1.5 User Management

OpenStackâ€™s rights management system employs the RBAC(Role-based access control) model and currently supports the following five roles[20]:

- Cloud Administrator. (admin) Users of this class enjoy complete system access.
- IT Security. (itsec) This role is limited to IT security personnel. It permits role holders to quarantine instances.
- Project Manager. (projectmanager)The default for project owners, this role affords users the ability to add other users to a project, interact with project images, and launch and terminate instances.

- Network Administrator. (netadmin) Users with this role are permitted to allocate and assign publicly accessible IP addresses as well as create and modify firewall rules.

- Developer. This is a general purpose role that is assigned to users by default.

Each user in the system is mapped to one of the roles mentioned above. Each project has an administrator and zero or more other types of users. The complete OpenStack cloud also has a system administrator. This user is created to be able to manage the projects, networks, databases and other users of the system.

User Credentials

To be able to separate users there must be some kind of credentials that can be used to authorize users. In OpenStack, two types of credentials are used depending on what API will be accessed. The Nova API uses a username and an authentication key, and the EC2 API needs a specific key-pair. This EC2 key-pair consists of the keys EC2_ACCESS_KEY and EC2_SECRET_KEY and are needed in each call using the EC2 API. This credential solution is the same as Amazon is using and this to provide the possibility for using the EC2 and S3 web service interfaces.

The Nova username and password are used to access the Nova API. The first call to OpenStack is to an authentication service that needs the username and password. This service returns an authentication token that will be passed to further calls to the platform.

When accessing an EC2 or Nova web service the different user credentials must be passed as a parameter, alternatively they can be set as environment variables. When the web service receives a request it checks the credentials and then knows which projects or instances that can be accessed by the current user.

These credentials must be sent by the user interface to a back-end module acting like a login service. When the user has sent the credentials for the first time to the back-end, the back-end system can now save the credentials for later use. An example of this could be when a user wants to launch a complete network map consisting of multiple virtual machines, it will only be one call to the back-end server. The server will then use EC2 or OpenStack API towards the OpenStack platform for launching each of the virtual machines with their predefined configurations.

5.1.6 Instance configurations

Each instance that the user want to create must be configurable in some way. The user might want to change the resources allocated by the instance, if the server cluster in where the system will run have different hardware capacities. There is also a need for network configuration such as IP addresses, subnets and number of interfaces for an instance.

Resource allocation

One major configuration that the user might want to do is to specify how much resources the instance is allowed to consume. In this configuration the standard parameters are memory, hard disk and number of virtual CPUs. These settings could be manually set by the user before running the instance or taken from predefined templates. OpenStack provides a set of predefined templates, called instance types, that can also be found in other cloud platforms like Amazon EC2. Examples of these templates are. e.g. m1.tiny, small, medium and large. where m1.small has larger amount of memory and disk space than m1.tiny.

As a user you should be able to specify exactly what amount of memory, disk space and virtual CPUs the instances will have. The back-end will receive these parameters and transform them into an appropriate instance type before launching the instance.

Network interface configuration

Each instance is configured with at least two network interface cards. All interfaces except for the management interface will be configured with IP address, netmask and default gateway, and they should be set by the user.

5.2 Graphical User Interface

The graphical user interface (GUI) is the end-user representation of the entire system. It should present the specific states of the back-end server implementation as well as details about each virtual machine, individually.

The predecessor of this project, called VIBox, used a Java based GUI. The choice of language was made to provide an application with multi-platform interoperability. This is also important in this project, why Java is what will be used. Further on, the GUI of VIBox has attempted to simplify the configuration process by introducing a palette with preconfigured virtual machines. This idea will be kept and developed further in this project.

5.2.1 Requirements

While designing the architecture of the GUI component of this project there are some requirements to consider.

- **Stateless** - The application should be a representation of the current state of the back-end server. It should therefore not hold any local state, but instead request information from the back-end server when needed.

- **Intuitive** - The application should have a familiar look-and-feel, which is easy to understand for an end-user.

- **Modular** - The design of the application should be based on a modular approach. This is to support easier development further on as well as concurrent development.
- **Loose coupling** - There should be as few dependencies as possible between the different components of the application. This must be considered continuously throughout implementation via refactoring. The goal is to produce written code easy to understand for other developers.
- **Communication with back-end server** - The application should be able to modify the state of the back-end server. This should be realized by introducing a standard communication protocol between back-end server and the application.

5.2.2 NetBeans Platform

When developing a graphical tool, it is important that it is easy to use and that it leaves little room for misinterpretation for the user. Features provided by the tool should be intuitive to find and use, only then the tool will get powerful. One key aspect is to create an application with a look-and-feel familiar to the user.

The above is one of the reasons why NetBeans Platform[23] has been chosen. It provides a look-and-feel corresponding to the operating system that the application is executing on. Also it delivers a window manager where the user can drag, drop, undock, resize, use tabs etc. Exactly the way modern applications should work. From a developer point of view, utilization of these features is done with very few lines of code, if any. By using the platform, focus can be put on developing internal application logic instead of spending hours on graphical development. This is instead delivered by NetBeans Platform.

5.2.3 Application Components

The developed client architecture is illustrated in figure 5.2. The architecture consists of four (4) logical levels.

Back-End Server Interface

The lowest level, level 0, is illustrated at the bottom and is the interface towards the back-end server. The communication is performed via web service request, described in section 5.3. Furthermore it is an intermediate layer as it translates the state of the back-end server into data structures understood by modules on architectural higher levels.

Separate worker threads are executing at this level. They are periodically checking the state of their responsibility from the back-end server. Such a check is directly followed by an update of the corresponding data structure.

This layer reveals an API towards the higher levels through which it gives access to the data structures mentioned above.

Windows

Level 1 consists of window components, which are visible for the end user. They are all representations of the data structures on the level below, and correspondingly a representation of the current state of the back-end server. The following window components are available

- **Server Explorer** - Monitoring of physical servers within the cloud
- **Project Explorer** - Monitoring of projects within the cloud
- **Network Map** - A map on which the user can draw network topologies as well as monitoring virtual machines
- **Network Palette** - Provides components which can be made visible on the network map

Common Tools

Level 2 consists of tool available from all window components as wells as from the menu. This include wizards, configuration, properties and buttons to deploy network topology etc.

Window Manager

Level 3 is the highest level of the architecture. This is where the window manager resides. The level is responsible for orchestrating the components of the levels below. This includes window movement, component selection, button activation etc.

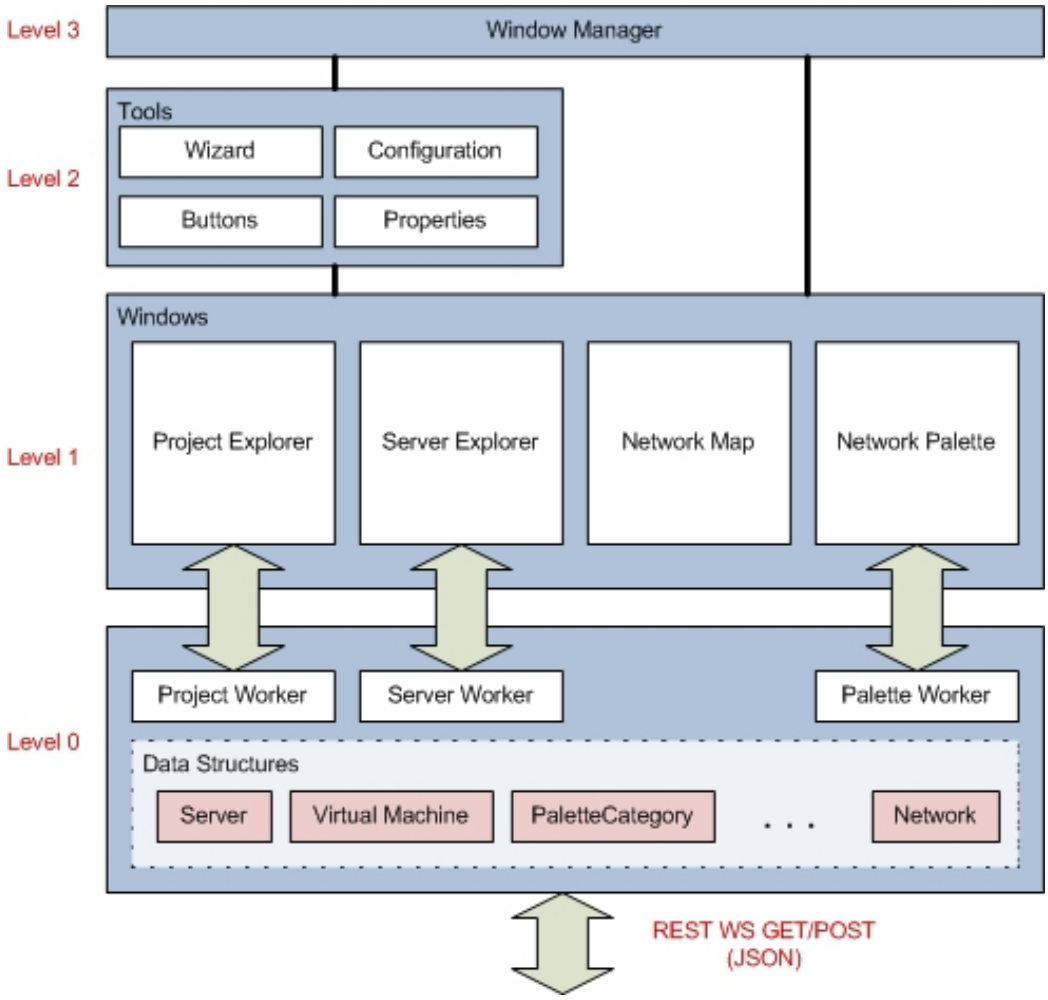
5.2.4 User/Administrator mode

The application is divided in two different views, user mode and administrator mode, where window components are disabled or enabled depending on the users rights. A regular user (could be seen as a customer) is only allowed to see its own projects and virtual machines. An administrator of the system might want to see all running instances and also the status of the physical machines in the system. This is required if the administrator should be able to monitor the health of the hosts and guests and find out if some reconfiguration must be done.

Concurrent modification using the GUI

The system provides a single login feature for both a regular user and an administrator. The system does not check if another user is logged in, even if it is the same account. This means that it can occur concurrent modifications in the back-end server and what will happen is that the last write will be the current state. To solve the problem with concurrent modifications is not in the scope of this thesis.

Figure 5.2. Client Architecture, using a stateless approach whereas workers consumes the state of the back-end server



5.3 Communication

For the application to work properly there must be a standardized way for the GUI and the Back-end server to communicate.

The GUI is responsible for retrieving the data that is necessary for it to run. The Back-end server is focusing on responding to incoming calls instead of keeping state of multiple user interfaces that are currently running.

The GUI will access data from the back-end server using a web service interface. This will make it easy to setup a service that can provide parallel access from multiple threads of the GUI. This is needed so the user interface can act responsive even if several calls are currently running in the background.

5.3.1 Communication protocols

The data that is transferred between the GUI and the Back-end is in the form of a JavaScript Object Notation (JSON)[14] string. The data is simply a string of characters that can be parsed by any JSON compatible parser making it easy to use earlier developed tools. By using an existing format such as JSON there are numerous tools in different programming languages that can handle the data. This makes it possible to use almost any modern programming language to handle the data in an easy way.

Example of a JSON String when the GUI is requesting different types of predefined instance types:

Listing 5.1. JSON Example get Instance types

```
1  [
2          {"flavorid": 3,
3           "name": "m1.medium",
4           "memory_mb": 4096,
5           "vcpus": 2,
6           "local_gb":40,
7           "id": 1},
8           {"flavorid": 1,
9            "name": "m1.tiny",
10            "memory_mb": 512,
11            ...
12            }
13          ...
14 ]
```

Chapter 6

Implementation

This section describes what is implemented during this master thesis project. It shows the different software modules with its most important classes and last there is a description of how the platform is configured to work with the created software.

6.1 Back-End Server

This section contains the implementation details of the Back-end server and its modules. Also the changes in the OpenStack source code needed for network configurations are described.

All of the source code for OpenStack is developed using Python as language. The back-end server which will communicate a lot with the OpenStack code should be implemented in a way that this communication can be done in a simple way. Because of this, the back-end server software is also implemented using Python.

During development multiple versions of OpenStack have been used for the cloud platform. To be able to use the latest features of OpenStack the developer edition, also known as trunk edition, has been used mostly. At some point of time, no more updates were done so changes in the OpenStack source code could be done. The last used version of OpenStack is: *2011.3-dev (2011.3-workspace:tarmac-20110516205108-5l1lmir6lxiy1kur)*, meaning that it is the developer edition from 2011-05-16.

To handle the virtual machine image files, OpenStack Glance was used with the 2011.3 version.

6.1.1 API

The user application communicates with the back-end server using a defined API. There is a web server that implements this API and all calls are request-response initiated by the client, i.e. the server only acts on requests.

RESTful web service

The API is a restful web service that receives "get" and "post" requests from the user application. The service is using an external module called Webpy [25], which makes it very easy to setup a service for the two verbs that need to be handled. Each URL is represented with a class e.g. "<nova api server ip>/physicalhosts" is represented by the class "physicalhosts". The different calls (get and post) are represented with methods within the class. The different calls that the user application can call are described in section 5.1.1.

6.1.2 Core modules

The back-end server architecture described in chapter 5 is mapped to these core modules. The modules are: virtual machine handler, network manager, virtual server template handler and physical host handler. The implementation of each module (or set of modules) are described below.

Virtual Machine Handler

Each call to the back-end server that asks for the complete network topology of the project will trigger an update call on the current state. This update call checks all the running machines of the project and sets the current configurations and states before returning the information to the user application. These update calls are performed periodically and the state that the user sees is delayed with, in worst case a couple of seconds.

Network Manager

When a user wants to deploy its network topology, a JSON file is created and sent to the back-end server. This file must then be translated into system specific data so the virtual machines described in the JSON file can be started, stopped, monitored etc. A parsing module is called as the first step in this translation and it returns python objects matching the JSON file received. By creating these objects as the first step, the other modules that are called later on can access the data in a much simpler way. When the objects have been created a comparer function is called which determines which machines should be launched or deleted and which should remain the same.

Now the system knows what machines to create and calls the OpenStack Compute API to launch these machines. The credentials needed to launch the instances has been sent from the client so the machines will be launched in the correct project without any further information.

When the new instances have been launched, OpenStack returns the IDs of the launched instances. These IDs are now replacing dummy values in the JSON file representing the network topology, so the system knows which machines are already running. A machine with a dummy value in the form *instance_X*, where the X is

an auto increment number, counts as an instance that is not running. If the ID is in the form of just an integer, the machine has got the ID from OpenStack and will be considered launched. This information is used next time the back-end server receives a network map from the client application, to know which machines are already launched.

Virtual Server Template Handler

The back-end system holds information about predefined virtual server templates, which are different kind of servers and routers. The information is stored in a database and is sent upon request from the user application. The templates can be used by the GUI to represent virtual machines, and the back-end server must know what template has been used to know how the instance should be launched. Therefor each virtual machine in the network map, that is sent to the back-end server, contains a role name that can be found in the database.

If the user does not find a suitable role to use for an instance, a new one can be created using the GUI. The user then specifies what kind of instance type it should have, or manually chose the amount of resources available, virtual machine image, graphical representation etc. This data is then sent to the back-end server and saved in the database for later use.

Physical Host Handler

Each physical host in the system is running a web service that will respond with a list of configurations and status for the host. This data contains e.g. CPU load, available memory, current network interface configurations etc. The web service is called from the back-end server using a *get* request to *<IP address>/systeminfo* on port 9000. This information is retrieved periodically and is delivered to the user on demand. The calls to each of the physical hosts are performed periodically and the information is cached by the back-end server. When the user wants the information, the back-end server can respond instantly. This is a faster solution than call all of the physical hosts on demand from the user.

6.1.3 Network configurations

A big problem with OpenStack as the cloud platform, is the lack of network configuration possibilities. The platform assumes that the user is satisfied with a virtual machine with one static IP address that can be used to access it. The idea of having multiple network interfaces is not supported so the solution must manipulate the OpenStack cloud platform to provide the possibility of several interfaces.

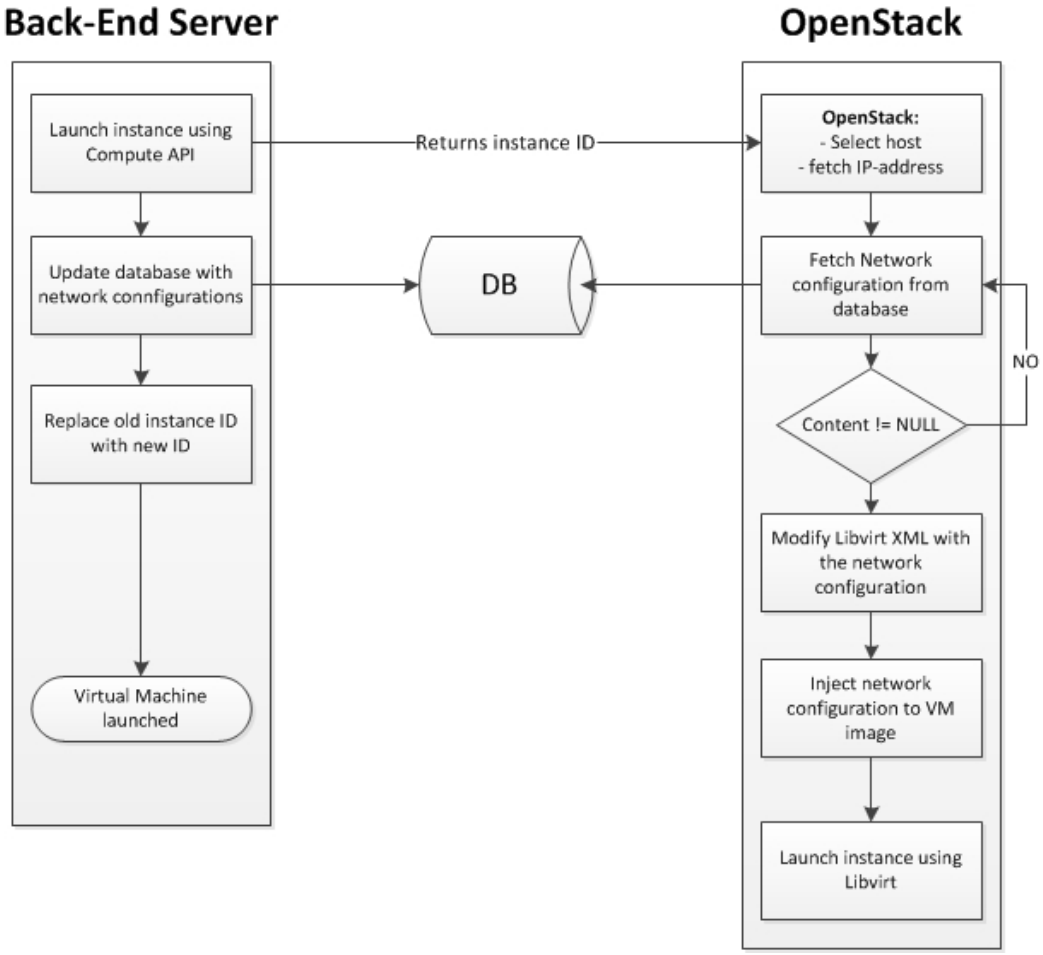
To solve the problem with multiple network interfaces, this solution sets entries in the database for each network interface that should be created when launching the instance. There is a mapping between the id of the instance with these interface entries so the system knows how many interfaces each instance should have.

When the database entries are set by the back-end server, the OpenStack platform calls the database to get the information needed to determine how many network interfaces should be configured on the instance. If the content in the database has not yet been set, the system will wait a few milliseconds and then try again. This database connection and instance configuration has been added to the OpenStack source code, to provide the possibility of having multiple network interfaces.

All the steps above are done as the final step before OpenStack is launching the instance using Libvirt. This is done at that point of time so OpenStack can finish its own work before the extended code is running. This reduces the number of dependencies between the original OpenStack solution and the extension created in this thesis project.

The flowchart in figure 6.1 shows how the system launches a new instance. It shows how the calls to the database are done and how the back-end server works in parallel with OpenStack Compute.

Figure 6.1. Launching an instance



Separated broadcast domains

To separate the broadcast domains within a project, different VLANs and network bridges are used for each LAN. The OpenStack code that is responsible for launching the instances has been modified to ensure that these VLANs and bridges are created before launch. The data fetched from the database does not only contain information about the IP addresses, netmasks etc needed for each interface but also information about which bridge the interface should be connected to. This information is used to create the bridge and VLAN on the host machine and is following this pattern: A project using the original bridge br_100 will get the VLAN 100 from the nova network manager. The extensions added are that each LAN gets its own bridge: br_100X together with VLAN 100X. If two LANs are created they will get bridge names br_1001 and br_1002 together with VLAN 1001 and 1002.

Management network

The first network interface at each instance is connected to a bridge that is provided by the Nova network manager. these network interfaces together with the bridge is used as a management network, making it possible to access the virtual machines from outside and makes it possible for the instances to retrieve meta-data from the Nova API server. The interface that is connected to this management network is not configurable by the user interface, and should not be used as a part of the network infrastructure that the user is creating.

Each virtual machine is preconfigured with a special route that tells the machine to forward packets with destination 169.254.169.254 out on the management network. This is the path to the meta-data service and needs to be set for the instance to start properly. When the packets are sent out on the management network they will be forwarded on the physical host to the Nova API server using the forwarding rule described in listing 6.1

Listing 6.1. Add route to meta-data service

```
iptables -t nat -A PREROUTING -d 169.254.169.254/32 -p tcp -
   m tcp --dport 80
 -j DNAT --to-destination \$NOVA_API_SERVER:8773
```

Configuration methods

The solution for setting all configurations before launching the machines is by modifying the instance's Libvirt XML file or by "injecting" data into the virtual machine image. The two methods are described below.

XML modification

The XML file for Libvirt created by OpenStack contains all information needed to launch the instance, including: CPU, memory, disk, interfaces, serial consoles

etc. The interesting part here is the *<devices.interface>* element which contains data how to create the network interface for the instance. The *<devices.interface>* element is cloned to be the base for the additional interfaces that is going to be created. The MAC address is removed completely and Libvirt will let the underlying hypervisor determine the new MAC address for the interface. Also the bridge where this interface should be connected to is described in this element and it will be replaced with the appropriate one. The <devices.interface> element is shown below in listing 6.2

Listing 6.2. XML file describing an interface

```
<interface type="bridge">
    <source bridge="br_174"/>
    <mac address="02:16:3e:76:b8:c2"/>
    <filterref filter="-000002bc-02163e76b8c2">
        <parameter name="IP" value="10.1.174.19"/>
        <parameter name="DHCPSERVER" value="10.1.174.7"/>
    </filterref>
</interface>
```

Configuration injection

OpenStack injects SSH configurations into the virtual machine image before launch time so the user can access the instance using a private SSH-key. The steps below describes how OpenStack injects data into a virtual machine image:

1. Mount the virtual machine image file
2. Create the directory hierarchy needed
3. Create the configuration files that should be injected
4. Unmount the image file

The most simple network manager in OpenStack, the FLAT manager, uses the same technique for injecting static network configurations into the images before launching, instead of using a DHCP server. This project's solution will modify, in a Debian based operating system, the file "/etc/network/interfaces" from containing the configuration shown in listing 6.3, to look like the listing 6.4. This is done for all of the network interface cards that the instance should be launched with. The previously described injection steps is used to do this modification. The OpenStack source code has been modified to call the injection method with the correct network information for all network interfaces.

The first entry in the "/etc/network/interfaces" is replaced with the address that the instance receives from Nova, making it work as the management interface for the instance.

Listing 6.3. DHCP configuration

```
auto eth0
iface eth0 inet dhcp
```

Listing 6.4. static IP configuration

```
auto eth0
iface eth0 inet static
        address X.X.X.X
        netmask X.X.X.X
        gateway X.X.X.X
```

The combination of the file injection together with the modification of the Libvirt XML file will let the instance boot with the correct settings for all network interfaces, on both the virtual machine and the hypervisor.

6.1.4 User Management

As described in section 5.1.5, Nova has multiple user roles with different rights. As a regular user you will only receive the instances related to your projects when a list command is sent to the OpenStack API server. If a system administrator performs the same call, the complete list of running virtual machines is returned to the user.

To keep the difference between the different types of users, the Back-end server uses the Nova API and will receive the correct instances on demand. Also different calls to this project's web service may filter the administrators from regular users and return more data. An example of this filtering is the authenticate request which is used to determine if the user application should provide certain modules or not. Another example is the *get* request to the URL */users* which will return all users in the system if the call is made by an administrator and only return project-specific users if a regular user is making the call.

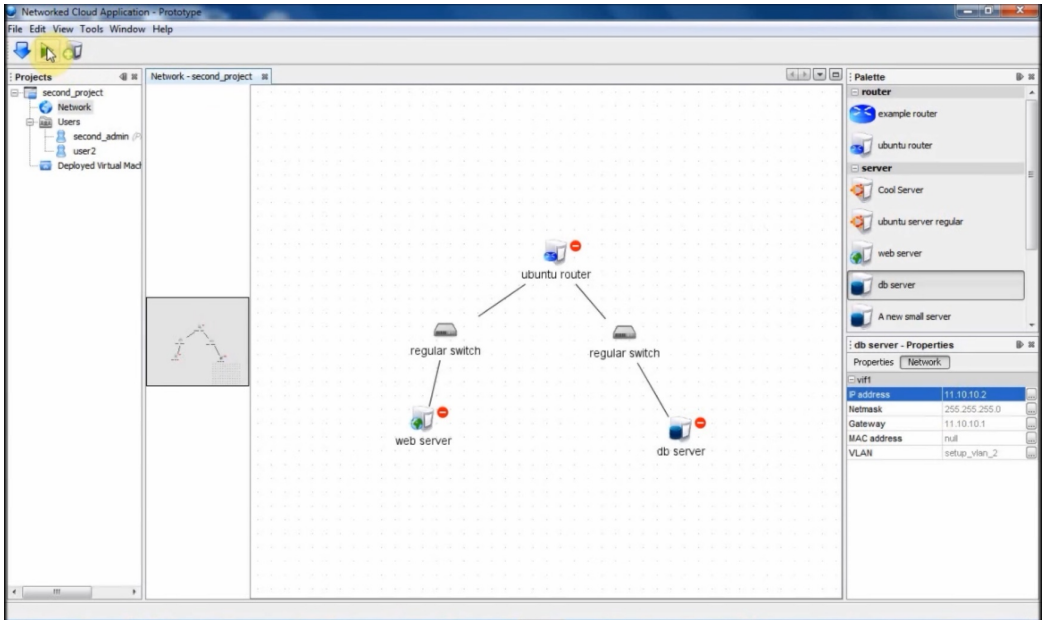
6.1.5 Instance Configurations

When the back-end server receives the network map with all the virtual machines, the instance types, described in section 5.1.6, are also specified. If a predefined instance type is selected, its data can be fetched from Nova's database. If the instance type is a custom created one, the system must create a new instance type with the parameters specified by the user. First the instance type is created in the Nova database and then the new id is used when the machine is launched. By doing this, the user can specify any type of instance type (As long as the numbers are valid, e.g. An instance type must have at least one virtual CPU).

6.2 Graphical User Interface

This section contains the implementation details of the Graphical User Interface, and all of its submodules. The illustration below, figure 6.2, shows the graphical user interface in action.

Figure 6.2. A screenshot of the graphical user interface



6.2.1 NetBeans Platform modules

The user interface is developed using NetBeans platform as a framework, on top of which several modules have been developed according to the system architecture described in section 5. The division and creation of modules have been made based on task responsibilities and can roughly be placed in two categories; non-graphical and graphical modules. The following sections describes the modules in more detail.

6.2.2 Back-End Server Interface

The GUI as an application consists of several graphical components that can be enabled or disabled for the user depending on the rights. The application also consists of several non-graphical modules that acts as data containers, communication abstractions etc, making it a more model-view-controller (MVC) like architecture. The most essential non-graphical modules are the BackEndService and BackEndServiceProvider modules and these two are described in the following sections.

BackEndService

The BackendService module is the core of the application. It has two major roles. First of all it acts as the container of data structures understood by the graphical modules layered above it. This can be seen as the abstract file system of the application. Although only persistent while running. Secondly this module benefits on NetBeans platform service concepts. This means it reveals a service interface to every other modules within the application delivering information of what data structures are stored in the application file system. Furthermore, the service reveals methods to call if modules needs to interact with the back-end server.

To further clarify what sort of data structures are placed inside this module, here are the most essential ones:

- **Server** - a representation of a physical machine within the cloud environment. Moreover the structure itself keeps track of hosted virtual machines.
- **Project** - a representation of the project concept of the back-end server. It maintains data structures of users, virtual machines and the network setup between them.
- **User** - a representation of a user belonging to a project.
- **NetworkManager** - this is the hard working thread of a project, with-in the client application. It is responsible for collecting the state of virtual machines from the back-end server, as well as triggers state changes as it delivers information to the back-end server on deployment of the network topology. Each project has its own NetworkManager.
- **GraphSceneCurrentState** - this is a listener who listens for changes in the graphical representation of the network topology. It delivers this state to the NetworkManager when a deploy request is being triggered.
- **PaletteCategory** - a representation of a category in the palette. Each PaletteCategory maintains a set of roles delivered by the back-end server, i.e. a switch or a Ubuntu 10.04 Server.
- **VirtualMachine** - a representation of a virtual machine running within the cloud.
- **Switch** - a switch is a concept in the client application which is intended to be intuitive for an end user. It is mapped with a Linux bridge coupled with a VLAN tag in the back-end server.

Some of the data structures located within this module interact with the back-end server when needed. As an example, a project requires information about its users, the network map coupled with the project and finally a list of virtual machines to be able to deliver relevant information to graphical components on request. Therefor

the data structure representing a project is responsible for gathering this information from the back-end server, all in separate threads. This way, when the graphical components are requesting information, the information can be delivered instantly without any delay caused by back-end server interaction.

BackEndServiceProvider

The BackEndServiceProvider module are implementing the service interface which the BackEndService are exposing for other components of the application. It facilitates the service concept of NetBeans platform. This means that it is loosely coupled with every other module of the application, but no strong dependencies are introduced. This means less error prone code as well as it is easier to do changes in the future. It is exposed as a service by creating a key folder named *META-INF.services*. This folder, in turn, stores one single file named *org.nca.backendservice.BackEndService*. This file contains one line: *org.nca.backendserviceprovider.BackEndServiceProvider*. This is the mapping between the interface and the implementation. NetBeans platform interprets this and delivers a service at runtime. The service is stateful during execution, why it is also suitable as a temporary storage for the running application.

The BackendServiceProvider is responsible for managing the content to and from the back-end server. This module consists of several threads that are periodically pulling data from the back-end server and updating variables that are accessible from other modules.

This module is also responsible for the user initiated communication to the back-end server. This could be the deploy functionality, creation of a new server template or updating to the last deployed network topology.

All communication is as described in chapter 5 System Architecture, done using JSON encoded messages. An external library for JSON parsing is used by this module to encode or decode the messages.

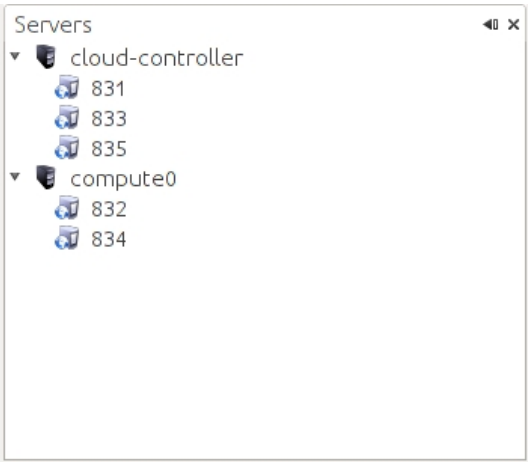
6.2.3 Window Explorer Components

To be able to manage the machines that runs in the cloud, explorer views have been created to graphically view both physical and virtual servers. There are two types of views, called Servers and Projects.

Servers

This explorer view, figure 6.3, shows all the physical hosts that make up the cloud environment. It is intended for administrators to use where they can monitor the load of the hardware nodes and where each virtual machine is actually located. This should not be accessible for regular users.

Figure 6.3. Servers explorer view, shows the physical hosts and the virtual machines running on them



Projects

The projects explorer view, figure 6.4, shows the graphical representation of the data structure, with the very same name, described in section 6.2.2. This includes project name, user coupled with it, virtual machines and access to a network map belonging to the project. Several projects can be explored at the same time. Which ones are visible depends on the user credentials used to login, on which the back-end server decides on what the user are allowed to view. The application only shows what is delivered to it and does not take decisions by its own.

The information that the project explorer view are using is also used to update the network map component. The state of the virtual machines can be viewed by a special text string which is dynamically changed and delivered to the user.

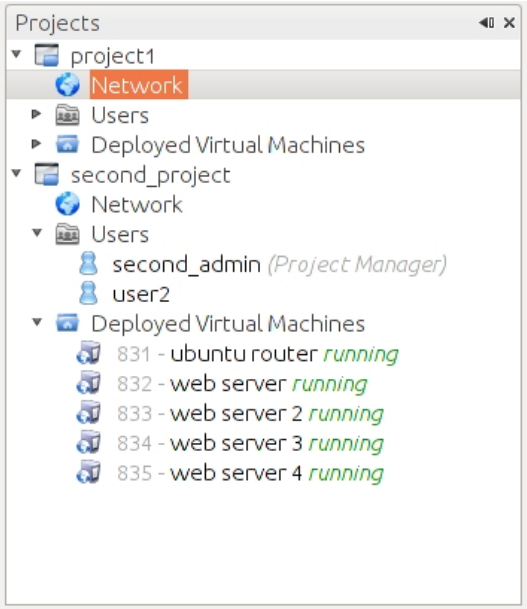
6.2.4 Network Map

In the GUI, the network map, figure 6.5, is the most essential part and describes the virtual network that will be launched on the back-end platform. The map holds virtual machines (servers, routers) and switches (representing virtual bridges) and they can be connected to each other forming a complete virtual network topology.

Before deploying the network being set up, a user is able to configure each virtual machine individually. This includes setting the number of CPUs, amount of RAM and disk space. Also, a user can configure the network interface settings. The switches can also be configured before deployment. However this does not affect the bridge created at the back-end server. A switch in the GUI can be configured as a DHCP server similar to a home networking scenario. This feature is enabled to simplify the setup of large networks decreasing the amount of manual configurations steps required by a user.

The state of the virtual machines can be viewed by a special icon on the virtual

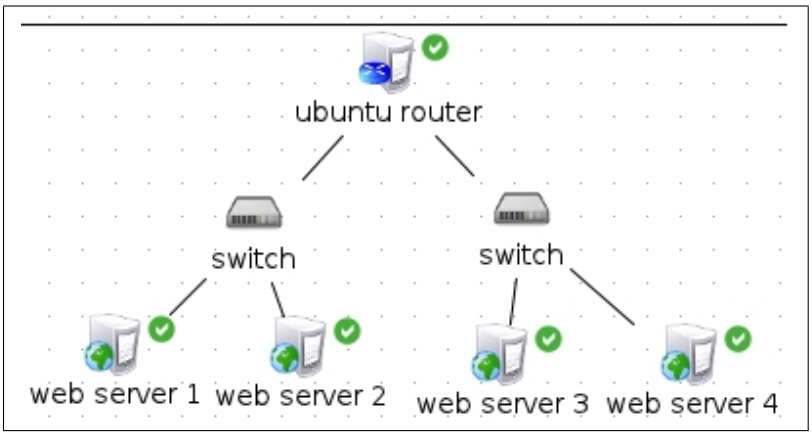
Figure 6.4. Projects explorer view



machine widget which is changed depending on the data retrieved from the back-end server.

The Network map will be transferred to the back-end server so the back-end knows what machines will be launched.

Figure 6.5. Network map, Shows the virtual network created with all instances and their edges

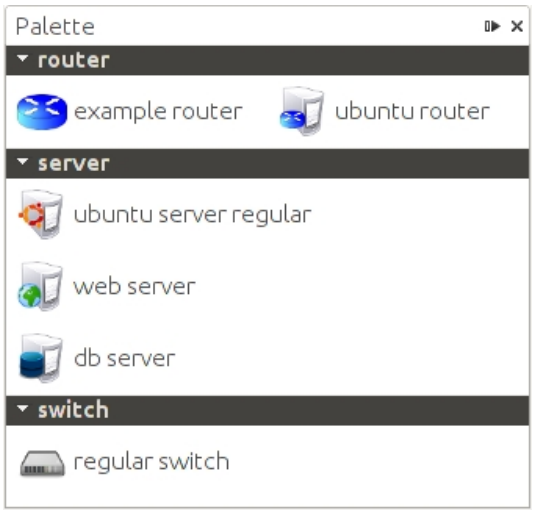


Network Palette

All virtual server templates, also called server roles, are shown to the user in a palette like component, figure 6.6. All predefined roles are shown here and they

will be used as building blocks in the network map component to create the virtual network topologies. Each template is shown to the user as a widget holding both the name of the template and a representing image. These images and underlying data containers are provided by the BackendServiceProvider module described in 6.2.2.

Figure 6.6. Network palette



Drag and Drop

A virtual server template (also described as server role) or other network equipment can be dropped on the network map from the palette. When adding these items to the network map they will be open for connections to other equipment in the network map. What happens when an item is dropped in the map is that a copy of the server template is added to the current map. This copy can now be configured by the user and also be copied multiple times to save configuration time. If many servers should be configured in a similar way, the user can configure one instance and then copy it, keeping all the configurations from the first one.

Connecting equipment with edges

If the network map contains more than one item, a virtual network can be created. To connect two instances, e.g. a virtual machine to a switch, the user simply put an edge between them. When this edge is created, the virtual machine will enable one network interface that can be configured to fit into the virtual network.

The user is not able to connect the different servers, routers and switches in all possible ways. There are some restrictions to make it easier for the back-end server to appropriately interpret the received data. The restrictions are; A virtual machine cannot be connected directly to another virtual machine, it has to go

through a switch. The same restriction is when adding routers which also must be connected to a switch, even if there is only one machine on one side of the router. Why there must be a switch between the machines is so the back-end server knows which virtual machines should be launched on the same virtual bridge. Each switch is representing a virtual bridge, so without the switch, the VM would not belong to any bridge at all.

The above is all a consequence of the network abstraction created in this project and is not caused by any limitations of the underlying cloud platform.

VM configuration

Each Virtual machine in the network map can be separately configured through a configuration wizard. This wizard enables the user to select predefined instance types or manually select the amount of memory, disk space and the number of virtual CPUs for the instance. The wizard also provides configuration of network interfaces with IP address, netmask, default gateway and MAC address. To make sure that the information is correct, the user will get an error if the inserted data does not meet the requirements. Two examples of this are the name of the instance can not be the empty string "" or that the virtual machine memory must be greater than 0.

To enter the configuration wizard, the user will right click on the instance which he or she wants to configure and select "configure".

6.2.5 Common Tools

The application has a toolbar enabled which is visible and reachable no matter of which of the window component, discussed in previous sections, is visible at the moment. The tools are the following:

- **Download network topology** - This tool is only enabled when viewing a network topology. When clicked, it triggers a request towards the back-end server and the last deployed network topology is then to be viewed in the application.

- **Deploy network topology** - This tool is only enabled when viewing a network topology. When clicked, it will trigger the application to transfer the current state of the viewed network topology to the back-end server. The back-end server will then alter its current state to match the one viewed in the GUI, i.e. if a user clears the map and clicks on the deploy button all virtual machines running will be terminated.

- **Create virtual server template** - This triggers a wizard for creating a new virtual server template. When configured, it is sent by the application to the back-end server where it is saved. From then on it is delivered and viewed as a new role in the network palette when working with the network topology.

- **Help** - Responsible for the help content of the application.

6.2.6 User/Administrator Mode

NetBeans platform provides an ability to enable or disable modules at runtime using its built-in update manager. By dividing as much as possible into separate modules one can develop an application with different features enabled, based on some predefined criteria. One of the objectives in this project was that an administrator should be able to monitor the physical servers load, at which virtual machines were hosted etc. Whereas a normal user should not have access to this kind of functionality. This functionality has been implemented using one specific module called Login Module which in turn uses the update manager to adjust the set of accessible modules for the application.

Login Module

The login module contains a NetBeans platform specific class of the type Installer. At startup, NetBeans platform guarantees such a class to be loaded before anything else is executed, which makes it suitable for login actions. When attempting to login the communication goes through the BackEndService, section 6.2.2, like normal communication behavior within the application. On successful login the module checks the returned user privileges and sends this information to the update manager to take actions.

Update Manager

The built in update manager of NetBeans platform is intended to be used for easy update of separate modules of an application. However, it can also be used to enable or disable entire modules. This is the case in this application. Even though the back-end server delivers information based on user credentials, i.e. not delivering information about a project the user is not privileged of viewing, the back-end server cannot itself remotely disable certain parts of the application.

Therefore, a user who login without administrator rights should not even be aware of the fact that there exists a module to monitor the physical servers and where virtual machines are hosted. Instead the application itself are able to disable the server explorer, see section 6.2.3, on demand by the back-end server response of logging in.

Chapter 7

Evaluation and results

The implemented software in this master thesis project has been evaluated according to criteria described in section 1.3.2.

This chapter shows how this solution is compared to similar projects and also how well the final implementation meets the expected results described in section 1.3. It describes scenarios tested to verify that; the network configurations of the virtual machines are valid and that the system can monitor the virtual machines in a proper way.

7.1 Comparison

The architectural design of cloud platforms, now only talking about IaaS, does not consider building network topologies between virtual machines. As a user you are not expected to use this kind of feature and a consequence no management tools are available on the market. The only prototype providing similar functionality to the one developed in this project is the resulting prototype of the preceding project, namely VIBox. Due to this, the comparison will be conducted against VIBox and will cover both back-end server solution and graphical user interface.

7.1.1 Back-end server

The back-end server solution of this project and the one of VIBox differs at several points. First of all, VIBox back-end server is directly managing several hypervisors and delivers an image service prototype. All built from scratch. Where, on the other hand the back-end server of this project is built on-top of an existing cloud platform. Additional concepts of virtual server templates, network topology etc. are placed in this back-end server when suitable. Building on-top of an existing platform, benefiting on the features provided creates a more robust solution where focus can be put on extended functionality. This is big advantage against VIBox. Secondly, the back-end server of VIBox does not keep any state of previously deployed network topology, it only starts the virtual machines as requested by the client application

and are not aware of the network configuration itself. The back-end server of this project is on the other hand fully aware of what has been deployed and how the virtual machines are connected. The state is stored in a database controlled by the back-end server, from which it can create and deliver the current system state on demand.

To sum it up, the back-end server of VIBox works as a relay server which propagates whatever command received, from a client application, to the hypervisors it is managing. This is done without knowledge of the consequences on the system state. The back-end server of this project can also be seen as a relay server, but on top of a cloud platform. It is a relay server in the sense that it translates general requests into platform specific API calls towards the underlying cloud platform. However, this back-end server is fully aware how the calls will affect the system state. Moreover, this back-end server can monitor and control the virtual machines and their life-cycle after being deployed and is also able to monitor the physical servers they are executing on.

7.1.2 Graphical user interface

Also the graphical user interface of VIBox and the one developed in this project differs quite a lot. One big difference is the fact that VIBox application is a java desktop application developed from scratch, whereas this project's client application is developed using NetBeans platform as framework. By using a framework this client application gets many features with no coding required. For example window management, drag-and-drop, easy update of already deployed applications and an interface which adjust the look-and-feel depending on what operating system it is executing on. Except drag-and-drop functionality, all of these are features that are missing in the VIBox client application.

The two client applications present network topologies in a similar fashion and they both provide drag-and-drop functionality of network equipment onto a map. VIBox also provides a view of the physical network topology where a user to some extent can manage physical switches. This is not present in the client application of this project, and this is because the virtual network topology is entirely independent of the physical topology. However the physical machines are represented in the user interface for monitoring purpose and to see location of virtual machines.

To configure the physical topology in the VIBox solution is a neat feature, however it is not delivered automatically to the user application. Instead the user must be aware of the entire network setup and draw this manually to be able to manage the equipment. This manual mapping is also required to be able to deploy any virtual machines at all in the VIBox solution. The drawback is a problem considering the responsibility distribution between the back-end server and the GUI. Almost all logic in the solution is located in the client application, instead of the back-end server.

The total impression of VIBox prototype and the one of this project is that the latter is more mature and benefits on clear division of what are the responsibilities

of the back-end server and the client application, respectively.

7.2 Verification of network configuration

To test that the network configuration is correct, some scenarios are tested to verify the network topologies. First there is a description of the evaluation setup, both hardware and software, and then followed by tested scenarios.

7.2.1 Platform setup

Hardware

The hardware used in this project for running the OpenStack software are two servers with the following configuration:

- **Name:** HP Proliant dl380 G6
- **CPU:** Intel Xeon E5540 @ 2.53GHz, 4 cores, 8 threads
- **RAM:** 12 GB
- **HDD:** 72 GB configured in RAID 1
- **OS:** Ubuntu 10.04 (Lucid Lynx)

Figure 7.1 shows an overview of the system setup of how the resulting prototype is connected to the underlying OpenStack cloud platform. The top two (the cloud controller and one compute host) of the OpenStack cloud servers shown in the figure are actually executing on one single machine and the other compute host is executing on one. Each on one of the machines listed above.

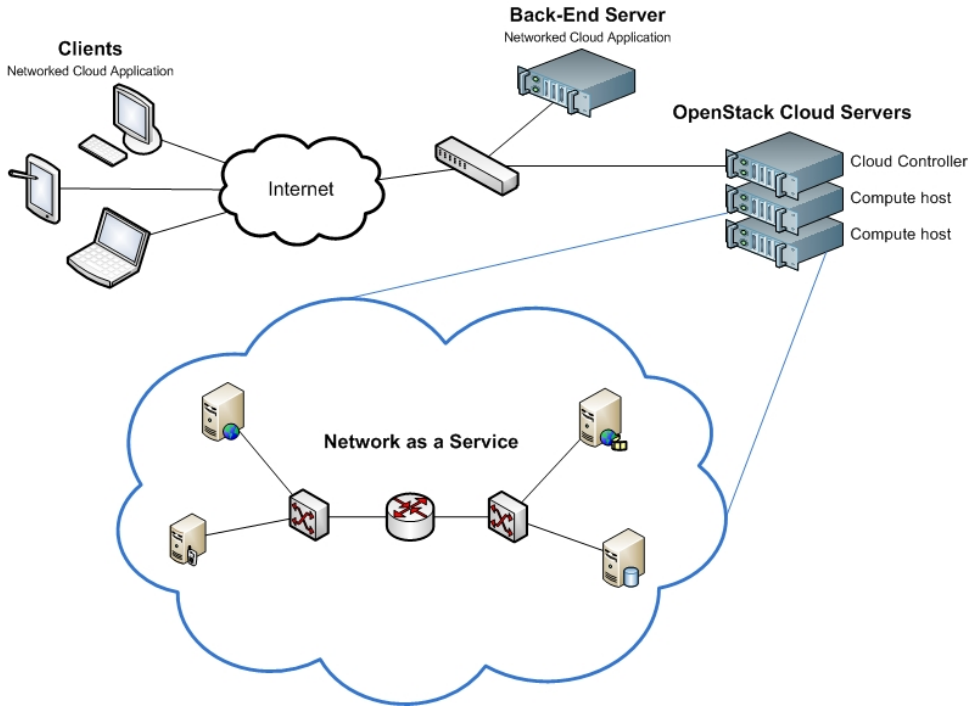
OpenStack setup

OpenStack is configured to run on the hardware described in the last section. It is configured to have one machine responsible for all Manager services: Network, Scheduler, ObjectStore and holds both the Nova database and the Nova API server. This machine is referred to as the Cloud-Controller and it also runs the Nova Compute service to be able to host virtual machines. All other hardware nodes are supposed to run the Nova Compute service only, and then referring to the Cloud-Controller for database access etc.

Virtual Machines

The tests are performed on the cloud platform using Ubuntu Server 10.04 (Lucid) as the operating system of the virtual machines. The router role is based on the same image but has multiple network interfaces. The router also provides IP-forwarding to guarantee that the packets be able to pass through the machine.

Figure 7.1. System setup showing OpenStack in collaboration with the resulting prototype



7.2.2 Communication within LAN

When launching two or more machines within the same project, and connected to the same switch in the GUI, the machines are supposed to find each other using ARP messages. This means that all the machines within the same switch (also referred to as bridge), should have entries in their ARP table containing the other machines connected to the same bridge. This should be the case even if the machines are located on different physical hosts as the bridges on the hosts are supposed to be distributed over all physical hosts running the Nova Compute service. There is also the requirement that two machines connected to different projects or different bridges cannot find each other using ARP messages.

To test that the ARP messages are sent correctly the following test setup is used:

- Virtual Machine A, IP address: 10.1.1.1/24, Physical host A
- Virtual Machine B, IP address: 10.1.1.2/24, Physical host B
- Virtual Machine C, IP address: 10.1.2.1/24, Physical host A

When logging in at any of the Virtual machines A or B one can see that the ARP table entries are showing the machines A and B but no entry for Virtual Machine

C. This is what is expected as Virtual machine C is located in a different subnet than Virtual machines A and B.

7.2.3 Communication over multiple LANs

To test how the routing is performed in the network, three or more machines are used to create multiple LANs. To connect these LANs a router is used, containing the routing entries to be able to forward traffic correctly. A router is in the virtual environment almost nothing more than a virtual server. The difference is that it has more network interface cards and that it must provide packet forwarding. To test that the traffic is router correctly the following setup is used:

- Two Local Area Networks A and B
- One Router with two network interface cards connected to LAN A and B
- One regular virtual machine with one network interface card connected to LAN A
- One regular virtual machine with one network interface card connected to LAN B

The configuration of the virtual machines are set as follows:

- VM 1 (Router)
  NIC 1: IP address: 10.1.1.1, netmask 255.255.255.0
  NIC 2: IP address: 10.1.2.1, netmask 255.255.255.0
- VM 2 (LAN A)
  NIC 1: IP address: 10.1.1.2, netmask 255.255.255.0, default gateway 10.1.1.1 (Router NIC 1)
- VM 3 (LAN B)
  NIC 1: IP address: 10.1.2.2, netmask 255.255.255.0, default gateway 10.1.2.1 (Router NIC 2)

The virtual machines VM1 and VM2 are located on the same physical machine, making it possible for the machines to communicate with each other if no additional setup is done. The router is located on another hardware node, forcing the traffic to go out on the physical network to pass through the router. If the injected network configuration is correct, the result should be that the traffic is passing through the router and it should be possible to capture the traffic when logged in to the router.

A *tcpdump* command on VM1 (the router) verifies that the traffic is passing the router when ICMP messages are sent between the VMs 2 and 3 that are located on different LANs.

7.3 Valid states of virtual machines

A machine that is previously deployed in the cloud should be monitored by the implemented software. When a user fetches a deployed network topology, the instances that are running in the cloud should be shown with their current state. This means that if an instance has been shut down, it should be shown in the GUI to the user.

7.3.1 Manual manipulation of running VMs

A machine is shown to the user in two ways. The first is as a widget in the network map and the second is like an icon in the explorer view. In both views, the machine is shown together with a status icon or text describing the current state of the virtual machine.

To verify that the system behaves correctly a manual manipulation of the system is done and the result can be viewed using the graphical user interface. The manual manipulation is done like follows:

1. A remote login to the Cloud-controller host machine using SSH connection
2. Reboot a virtual machine that is located in the current shown project in the GUI
3. Delete the same virtual machine that was rebooted.

These manual steps triggers changes in the virtual machine state, which is shown to the user. After step 2, the virtual machine name is in the explorer view followed by a "*rebooting*" text and in the network map it is now showed together with a new icon representing the reboot state.

After step 3, the machine is shown as "*shutdown*" and there is a warning icon tied with the VM widget to indicate that something is wrong.

Chapter 8

Conclusions and future work

8.1 Summary

This thesis describes the first steps towards supporting virtual networks over multiple data centers. The approach was to select a cloud platform as a base to build upon. The selection was done and OpenStack was decided as base due to its promising upcoming functionality as well as its current momentum on the market with several influential companies involved.

Trying to embrace multiple data centers at once seemed too ambitious and not possible within the time frame for this project. The project was decided to support virtual networks in one single data center as a first step. When no open source cloud platform supports this yet, this was an appropriate approach. Even though Amazon (not open source) supports virtual networks, their solution are restricted to a single data center. Because of this, this project has kept the multiple data center approach in mind throughout development and has also delivered some ideas of how this could be done.

While designing the system architecture a decision was made trying not to change the implementation of the underlying cloud platform, OpenStack. Instead a back-end server was logically placed as a layer above where to place the extended functionality. In addition to the back-end server an architecture was designed for a client application which was supposed to be used as management tool where a user should be able to create and deploy virtual networks as well as monitoring the running instances. Moreover administrators should be able to monitor the physical machines, in the same application, on which the instance were executing.

The final implemented prototype consists of a slightly modified OpenStack as an underlying cloud platform and a back-end server above which controls the cloud platform. A graphical user interface has also implemented as a client application. This client application communicates directly with the back-end server to which it, for example can deliver a predefined abstraction of a network topology. The back-end server translates this abstraction to calls against the underlying cloud platform to launch appropriate instances as well as manipulating the platform to be able to

create virtual networks. Several virtual networks can co-exist in the cloud platform without interfering with each others network traffic. The separation is done on layer 2 of the OSI-stack using VLANs. The virtual networks and the separation between them are tested and verified via predefined use cases.

8.2 Conclusions

This thesis is showing how network and common machine configurations can be achieved in a cloud environment. It gives a user friendly tool where a user can configure, launch and monitor virtual machines inside a cloud platform, in this case OpenStack.

The original architecture of a cloud platform does not consider the ability to create virtual networks. In this sense, this project is in conflict with the system architecture of OpenStack. However this project shows that with minor changes in the OpenStack source code, virtual networks within one data center can be achieved. This also implies that it is promising to take the solution further to also support virtual networks spanning across multiple data centers.

8.2.1 OpenStack as Platform - pros and cons

OpenStack provides a lot of functionality for handling virtual machines in the cloud, but as it is an ongoing project many blue-prints have not yet been implemented or tested properly. During this master thesis project, there has been, besides multiple developer versions, one big new release of OpenStack Compute (Nova). It went from Bexar release to Cactus release and introduced a lot of new features. The next big release of OpenStack Compute is planned after the end of this project so to be able to use the very latest implemented features, the trunk edition of Nova has been used mostly. This developer edition contains the latest implemented features but all has not been tested properly making the system sometimes unstable. To work with a platform that is under heavy development, makes it hard to see what has really been implemented and what is still just on paper. Therefor, more time than expected has been used for troubleshooting and source code searching of the platform and its setup.

Network configurations

The most problematic downsides with OpenStack is that it does not provide instances with multiple network interfaces. To be able to let the platform provide this, changes need to be done all over the source code as the original architecture is produced with the assumption that each instance is provided with a single interface.

Database The OpenStack database has very detailed information about each launched instance. This data is accessed many times during a launch of a virtual machine and the system is very sensitive to what data is stored. As the data is

accessed multiple times during launch, there are a lot of dependencies that needs to be considered if any changes are done in the database.

There are some cases where the original idea is not very suitable for this master thesis project. The most obvious case is that each instance is mapped to one MAC address. This is not a very smart solution if an instance is allowed to run with multiple network interfaces, which is not the case in a default installation of OpenStack. This instance to MAC address mapping made it necessary to use an additional database with network configurations stored.

Virtual server images

The number of image formats supported by OpenStack is at this point of time limited to a few. The most commonly used images are Amazon Machine Images (AMI) which can be downloaded from many web pages often zipped together with kernel and ramdisk images. There is ongoing work to be able to support more image types which would make it easier for users to migrate from other cloud platforms to OpenStack.

Snapshots One common solution for creating new virtual machine images is to take a snapshot of a running instance. What is done is that the image of the running instance is copied to a new image file and can be launched as a new instance. This means that the two instances, the original one and the snapshot image, are having the exactly same data. This is a very smart solution if a user wants to have an instance containing a lot of additional packages, there would be enough to just configure one machine and then take a snapshot of it.

OpenStack does not currently provide fully functional snapshots of running instances. Using the installed versions of OpenStack Compute and Glance did not provide any snapshot image files at all. There is an API call that is described as creating a new snapshot image, but it does not return any new image only meta-data. As the calls are possible to make and meta-data is created, one could think that fully functional snapshots are coming very soon. This feature would be of great interest for a user to be able to specify multiple types of roles in an easy way.

Virtual server configuration files Using EC2 API, a machine can use its meta-data service to fetch specific user-data, containing e.g. scripts that will be launched at startup. This feature is not currently supported by Nova API, but instead an injection of files into the virtual machine image. This makes it possible to configure your virtual machine before launching it.

Currently there is no support for injecting files using Libvirt, only Xen (through a special API, XenAPI) has this support.

8.3 Future work

There are several different paths to consider to move this prototype towards the next step. This section will first of all present the objectives that were excluded while presenting the expected results of this project, see section 1.3. Secondly, this section presents ideas of how the prototype should evolve to finally reach the desired functionality of virtual networks spanning over multiple data centers.

8.3.1 Extend virtual server templates

This feature is available in todays prototype. However, the concept should be enlarged and also include what packages should be installed, startup scripts to run and also support all common operating systems. Today only Linux distributions are supported.

The extension would involve additional development of both back-end server as well as the client application. In the latter a user needs to be able to configure the virtual server template, to be stored, in various ways. In the back-end server, several approaches should be considered. OpenStack are working on APIs where it would be possible to take snapshots of running virtual machines and also make it possible to add startup scripts to a call to launch an instance. These are all interesting and important features to consider when extending the concept of virtual server templates.

This solution has code for launching virtual machines with startup script using the EC2 interface that OpenStack provides. This means that a machine can be provided with instructions to download and install predefined packages, e.g. An Ubuntu distribution downloads packages from the APT repository. The original architecture of this project mentioned the EC2 interface as it would not be used in this solution, so the future goal is to provide something similar through OpenStack API.

Furthermore what should be considered are the different types of image's container formats that exists, i.e. ovf, bare, aki, ari and ami. They are all supported by Glance, the image service provided by OpenStack, but maybe needs to be handled and presented for the user in a more intuitive way.

8.3.2 Network templates

This objective has not been considered at all during the project. It would reduce the amount of work while setting up virtual networks using the graphical tool. A definition and architecture is needed of how to store and handle this type of template in the back-end server as well as how it should be delivered to the client application. A natural attempt would be to use the same approach how network topologies are stored and sent back and forth between the back-end server and the client application. The problems to solve then would only be of graphical matter,

how it should be presented in the network palette as well as actions to take when the component are dragged on top of the network map.

8.3.3 Communication channel to the running instance

By providing the ability to get a terminal to individual instances, directly in the client application, configuration of instances would be a lot easier. OpenStack provides an ajax terminal through its EC2 API, why this should be quite straight forward to implement. Moreover, NetBeans platform can easily be equipped with an internal Internet browser module, why presenting such a terminal in the client application should be as easy. This is not implemented in the prototype, client application or back-end server. One of the reasons for this was that the ajax terminal proxy provided by OpenStack kept throwing internal errors while trying to fetch such terminals. Most likely this error is corrected in the nearby future and making it possible to add this feature to the prototype.

8.3.4 Virtual networks spanning over multiple data centers

The final goal is to support virtual networks spanning over multiple data centers hosting cloud environments. Section 3.6 covers some ideas of how one could grasp the problem at hand. The path towards the solution could also imply that some alterations needs to be done on the prototype of this project. Examples of what that could be are presented in this section along with some ideas of how to merge OpenVSwitch, presented in section 3.5.1, into the solution.

Refinement of network abstraction

The network abstraction of building virtual networks in the prototype of this project is focusing on doing this in one data center. This might also be the correct approach while extending the solution. However, then some sort of translator would be needed to manipulate messages while being sent between different data centers. In any case, this needs to be researched further as well as what approach should be used to setup the long distance links between multiple data centers.

OpenVSwitch

People in the OpenStack community are working on a way to use OpenVSwitch as the default switch for Nova. The solution today is using the *brctl* binary for creating the bridges and OpenVSwitch has a compatibility layer that works with the *brctl* binary. Because of this there should be a way of replacing the original bridge with the OpenVSwitch to add more functionality and control over the network. The GUI created could benefit from the functionality of OpenVSwitch like, showing network traffic on the edges, give the user more control over how the traffic should be routed etc.

Bibliography

[1] Amazon. Amazon elastic compute cloud (amazon ec2). http://aws.amazon.com/ec2/, 2011.

[2] Amazon. Amazon simple storage service (amazon s3). http://aws.amazon.com/s3/, 2011.

[3] Amazon. Virtual private cloud. http://aws.amazon.com/vpc/, 2011.

[4] Hareesh Puthalath Victor Souza Martin Johansson Daniel Öhman Bob Melander, Jan-Erik Mångs. Vibox - virtualized internets-in-a-box: a tool for network planning experimentation. *IEEE*, 2010.

[5] Convirture. http://www.convirture.com/products_opensource.php, 2011.

[6] Convirture. Compare products. http://www.convirture.com/products_compare.php, 2011.

[7] D-Link. D-link tm dgs-3224 managed 24-port gigabit ethernet switch, 2006.

[8] Eucalyptus. Eucatools user guide. http://open.eucalyptus.com/wiki/Euca2oolsGuide_v1.1, 2011.

[9] Eucalyptus. Introducing eucalyptus. http://open.eucalyptus.com/wiki/IntroducingEucalyptus_v2.0, 2011.

[10] Libvirt. Domain xml format. http://www.libvirt.org/formatdomain.html, 2011.

[11] Libvirt. Terminology and goals. http://www.libvirt.org/goals.html, 2011.

[12] Justin Pettit Ben Pfaff Martín Casado Nick McKeown Scott Shenker Natasha Gude, Teemu Koponen. Nox: Towards an operating system for networks. *CCR online, the Computer Communication Review*, 2008.

[13] Tom Anderson Nick McKeown, Guru Parulkar. Openflow: Enabling innovation in campus networks, 2008.

[14] JavaScript Object Notation. Introducing json. http://json.org/, 2011.

[15] OCCI. Open Cloud Computing Interface. http://occi-wg.org/, 2011.

[16] OpenNebula. Network guide 2.0. http://www.opennebula.org/documentation:rel2.0:nm, 2011.

[17] OpenNebula. Opennebula 2.0 architecture. http://www.opennebula.org/documentation:archives:rel2.0:architecture, 2011.

[18] OpenNebula. Opennebula cloud. http://www.opennebula.org/cloud:cloud, 2011.

[19] OpenStack. Blueprint - network service. http://wiki.openstack.org/NetworkService, 3 2011.

[20] OpenStack. Managing users. http://nova.openstack.org/adminguide/managing.users.html, 2011.

[21] OpenStack. Networking options. http://docs.openstack.org/openstack-compute/admin/content/ch04s01.html, 2011.

[22] OpenVSwitch. Overview of functionality and components. http://openvswitch.org/?page_id=14, 2011.

[23] Oracle. Netbeans platform. http://netbeans.org/features/platform/, 2011.

[24] Red Hat. KVM - KERNEL BASED VIRTUAL MACHINE. http://www.redhat.com/f/pdf/rhev/DOC-KVM.pdf, 2009.

[25] Aaron Swartz. Webpy. http://webpy.org/, 2011.

[26] Citrix Systems. Xen cloud platform. http://www.xen.org/products/cloudxen.html, 2011.

[27] Citrix Systems. Xen overview. http://wiki.xensource.com/xenwiki/XenOverview, 2011.

[28] Citrix Systems. Xenserver. http://www.citrix.com/English/ps2/products/product.asp?contentID=683148, 2011.

Appendix A

Glossary

- VM - Virtual Machine
- NIC - Network Interface Card
- VLAN - Virtual Local Area Network, also referred to as 802.1Q
- GUI - Graphical User Interface
- SSH - Secure Shell, a network protocol for remote administration of Unix computers
- REST - Representational State Transfer, a software architecture where servers delivers its current state on request, i.e. GET or POST
- Host - Physical machine where virtual machines are launched
- Guest - Virtual machine instance running on Host
- JSON - JavaScript Object Notation
- ARP - Address Resolution Protocol
- KVM - Kernel-based Virtual Machine
- API - Application Programming Interface
- EC2 - Elastic Compute Cloud

Appendix B

User manual

B.1 Introduction

This manual is intended to be used as reference to be able to reproduce the system setup, including configuration, used when running the prototype developed during the "Network services and tool support for cloud environments" project.

Hardware equipment that has been used, including their basic configuration are listed first. Also the network switch and how it should be configured is listed.

How to install and configure OpenStack as used during the project is described after the hardware configuration. Finally, the two last sections covers details of how the back-end server solution as well as the client prototype can be configured and where different services are supposed to run.

B.2 Hardware configuration

B.2.1 Servers

The hardware used in this project for running the OpenStack software are two servers with the following configuration:

- **Name:** HP Proliant dl380 G6
- **CPU:** Intel Xeon E5540 @ 2.53GHz, 4 cores, 8 threads
- **RAM:** 12 GB
- **HDD:** 72 GB configured in RAID 1
- **OS:** Ubuntu 10.04 (Lucid Lynx) x64

Moreover, one of the above servers hosted the additional database required for the project.

B.2.2 Laptops

The back-end server software developed in the project, executed on a laptop with the following configuration:

- **Name:** HP ProBook 6550b
- **CPU:** Intel Core i5-M450 @ 2.4GHz, 4 cores
- **RAM:** 4 GB
- **HDD:** 320 GB
- **OS:** Ubuntu 10.10 (Maverick Meerkat) x64

The client application was written in Java and should be platform independent. Even if not tested on all available operating systems it was, in addition to a laptop with the same configuration as the back-end server, also run on a laptop with the following configuration:

- **Name:** HP 6530b
- **CPU:** Intel Core2 Duo P8400 @ 2.26GHz
- **RAM:** 2 GB
- **HDD:** 250 GB
- **OS:** Windows 7 Professional x64

B.2.3 Switch

One single switch has been used to connect the servers. It is a D-Link DGS-3048. As seen in the listing B.2, in section B.3, one can see the following line: *–vlan_interface=eth1*. This indicates that the two servers mentioned above uses the eth1 to communicate using VLAN tagged ethernet packets. To allow these packets to pass through the switch, this needs to be configured on the ports where eth1 from the servers are connected. These switch ports should preferable be configured for all possible VLANs expected in the cloud environment, i.e VLAN 1-4096 to maximize number of VLANs.

B.3 OpenStack

B.3.1 Version

The last used version of OpenStack is: *2011.3-dev (2011.3-workspace:tarmac-20110516205108-5l1lmir6lxiy1kur)*, meaning that it is the developer edition from 2011-05-16.

To handle the virtual machine image files, OpenStack Glance were used with the 2011.3 version.

B.3.2 Installation

The initial setup is using two physical servers, connected using a regular switch, to simulate a cloud environment. The first server acts both as a cloud controller and as a compute node for hosting virtual machines. The second server acts only as a compute node.

Cloud Controller

The cloud controller server can be installed on Ubuntu 10.04 LTS either through a script provided by OpenStack community or by adding Nova to the repository and then manually installing it using i.e. apt-get install. The scripted installation does in fact not differ from the manual but only ensures no necessary installation step are missed, providing a fully functional OpenStack Nova Cloud Controller installation.

The following commands triggers the scripted installation:

1. wget –no-check-certificate https://github.com/elasticdog/OpenStack-NOVA-Installer-Script/raw/master/nova-install

2. sudo chmod 755 nova-install

3. sudo bash nova-install -t cloud (<-t compute> for a compute host)

The following parameters will be set during the scripted installation

- Cloud Controller Host IP address.
- S3 IP, or use the default address as the current server's IP address.
- RabbitMQ Host IP. Again, default can be used.
- MySQL host IP address.
- MySQL root password and verification
- Network range for all projects in CIDR format.

Whatever installation approach are chosen, the configuration file for the installation can be found and altered in */etc/nova/nova.conf*. A lot of network configuration concerning the cloud environment can be made here.

More information on how the scripted installation works, or how to do the manual installation of OpenStack can be found here: *http://wiki.openstack.org/NovaInstall/MultipleServer*

Add more compute nodes

To be able to run more instances in the cloud you would need to add more compute nodes to the system. Each compute node can be started using the start-up script provided by OpenStack, which will setup the network configurations needed to communicate with the cloud controller node.

Each compute node might be set in the same network as the cloud controller node(s).

The */etc/nova/nova.conf* file must be copied from the cloud controller node so the compute node gets the correct paths to the nova modules, e.g. the EC2 API, scheduler, etc.

This copying can be done like the following:

- scp -r ericsson@$CLOUD_CONTROLLER_IP:/etc/nova/nova.conf /etc/nova/nova.conf

B.3.3 Configuration

Network

The networks connecting the OpenStack servers is realized using two networks, separated using VLANs. One of them is used as management network and the other (OpenStack network) is used for virtual machines to communicate. The configuration of each network, during this project, are presented below. The OpenStack servers were configured for management network on eth0 and for OpenStack instance networks on eth1.

Eth0:

- Management Network: 192.36.165.48/29
- Gateway: 192.36.165.49
- Netmask: 255.255.255.248

192.36.165.48 is the network address, 192.36.165.49 is the gateway and 192.36.165.55 is the broadcast address. This leads us to the following available addresses, 192.36.165.50-54, to use for servers and clients. The addresses have been distributed as follows:

- OpenStack Nova Cloud-Controller - 192.36.165.53
- OpenStack Nova Compute0 - 192.36.165.54
- Client0 - 192.36.165.51
- Client1 - 192.36.165.52

The clients does not necessarily need to be on the same network as the OpenStack servers, but has been placed this way for simple access during testing and development.

Nova configuration file (nova.conf)

The configuration file for OpenStack, the nova.conf, can be customized to fit the current environment. Important to mention is that this file does NOT ALLOW comments or empty lines which would render the nova server to crash. This is the only place where OpenStack configurations are supposed to be and in this project, this is how the file ended up to look like:

Listing B.1. /etc/nova/nova.conf

```
--dhcpbridge_flagfile=/etc/nova/nova.conf
--dhcpbridge=/usr/bin/nova-dhcpbridge
--logdir=/var/log/nova
--state_path=/var/lib/nova
--lock_path=/var/lock/nova
--networks_path=/var/lib/nova/networks
--scheduler_driver=nova.scheduler.simple.SimpleScheduler
--sql_connection=mysql://root:password@192.36.165.53/nova
--network_manager=nova.network.manager.VlanManager
--verbose
--s3_host=192.36.165.53
--rabbit_host=192.36.165.53
--cc_host=192.36.165.53
--ec2_url=http://192.36.165.53:8773/services/Cloud
--fixed_range=10.0.0.0/12
--network_size=5000
--FAKE_subdomain=ec2
--verbose
--libvirt_type=kvm
--vlan_interface=eth1
--glance_host=192.36.165.53
--image_service=nova.image.glance.GlanceImageService
--quota_instances=100
--quota_cores=200
--quota_volumes=100
--quota_gigabytes=1000
--quota_floating_ips=100
--quota_metadata_items=128
--max_cores=65
--ajax_console_proxy_url=http://192.36.165.53:8000
```

Enabling Access to virtual machines on the Cloud Controller

One of the most commonly missed configuration areas is not allowing the proper access to VMs. Use the 'euca-authorize' command to enable access. Below, you will

find the commands to allow 'ping' and 'ssh' to your VMs:

- euca-authorize -P icmp -t -1:-1 default
- euca-authorize -P tcp -p 22 default

Glance Imageing service

In the Cactus release of OpenStack, glance seems to be required to launch instances through the OpenStack API. It can be found using the repository in Ubuntu, command: *apt-get install glance*.

B.3.4 Using OpenStack Commandline tools

Upload images to the Glance service

The current Ubuntu repository version of Glance does not work well with the publishing tools outside the Glance project. (uec-publish-tarball)

The user has to use glance add <image> to upload the image to Glance. Both machine image and kernel image are needed, and are added separatly.

Launch an instance

There are different ways how to launch an instance in OpenStack. You can use the EC2 API with Euca2ools or you can use the OpenStack API with the nova-binary. The commands are shown below:

- **EC2:**
  euca-run-instance $image_id -t $flavor_id -k $ssh_key_pair
- **OpenStack API:**
  nova boot –image $image_id –flavor $flavor_id –key [path to]/$ssh_key_pair

The key_pairs used in these commands can be created using the Euca2ools command *euca-add-keypair <name> > <file>*. This key is used to access the instance through SSH without the need for username and password. To access a running instance: *ssh -i [path to]$ssh_key_pair <IP address>*.

B.3.5 Source code alteration

The prototype have been aiming at not modifying OpenStack source code. However, to support multiple network interfaces some alterations was needed. This has been done in the file, *nova/virt/libvirt_conn.py*, which is the last step before the Libvirt call is made to launch an instance.

B.4 Back-end server software

B.4.1 Python version

Python version: 2.6.6

B.4.2 Configuration

For the system to know which physical machines are used in the OpenStack setup, there is a configuration file with all IP addresses of the machines. It is located in vibox.configfiles with the name physical_servers. Each machine is represented on a single line with: <NAME,IP ADDRESS>

Database

In addition to the OpenStack database another one named vibox_db were used. This database was running on the OpenStack Cloud Controller host, with the same username and password as the OpenStack database. The configuration of the database is shown in figure B.1

B.4.3 OpenStack client APIs

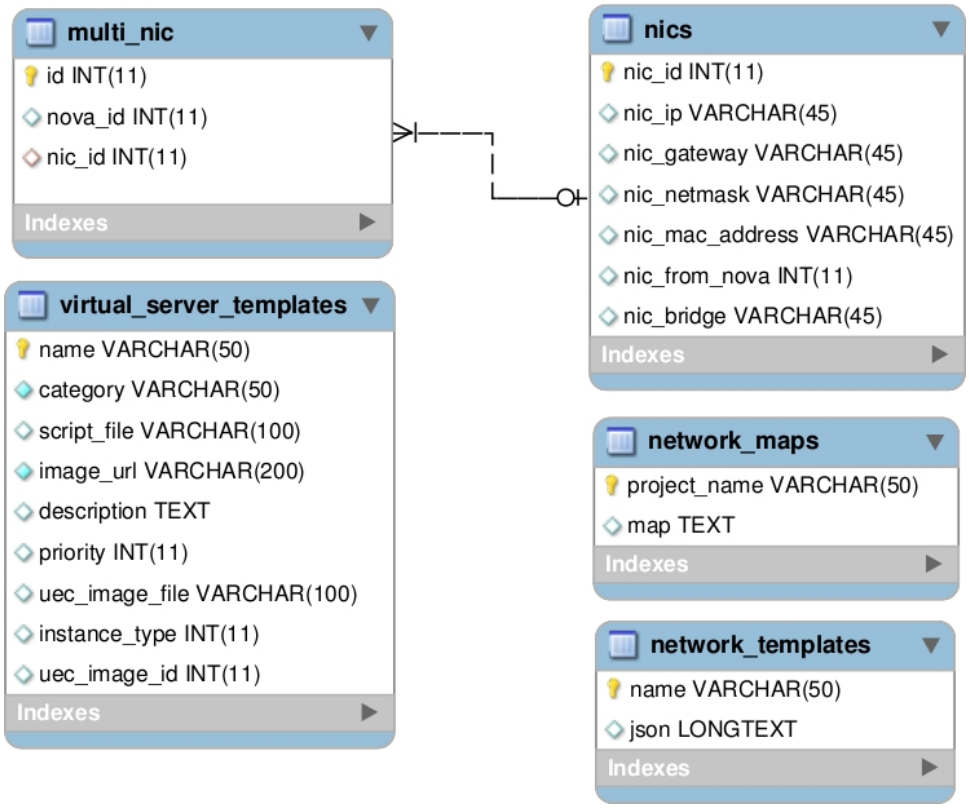
The different APIs for communicating with OpenStack has python clients available. the three different APIs are the following:

- **EC2 client:** The EC2 client used is from Amazon. It is called *Amazon EC2 Library for Query in Python* and can be downloaded from: http://aws.amazon.com/code/Python/552
- **Glance client:** The client for glance used in this project is the package python-glance from the Ubuntu repository.
- **Nova client:** The Nova client that has been used is downloaded from Ubuntu repository. It is the python-novaclient package that has been used.

B.4.4 Additional libraries

- **SimpleJson** - This is used for encoding and decoding JSON messages, more information about the package can be found at: http://pypi.python.org/pypi/simplejson/
  INSTALL: sudo easy_install simplejson
- **Netifaces** - This is used to poll information about active network interfaces on the physical servers. More information can be found at http://alastairs-place.net/netifaces.
  INSTALL: sudo apt-get install python-netifaces

Figure B.1. MySQL database model



- **Pynetinfo** - This is used to gather information from the routing table of each physical server. More information can be found at http://pypi.python.org/pypi/pynetinfo/0.2.3.
  INSTALL:

  1. git https://github.com/ico2/pynetinfo.git
  2. sudo python setup.py install

- **Psutil** - This is used to monitor the system state of the physcial servers. More information can be found at http://code.google.com/p/psutil.
  INSTALL: sudo easy_install psutil

- **Python-rest-client** - This client is used internally by the back-end server to gather current state from the RESTful interface exposed by each physical server. More information can be found at http://code.google.com/p/python-rest-client.
  INSTALL: sudo easy_install python-rest-client

- **Webpy** - This is the framework providing a RESTful interface for the server solution. It handles GET and POST requests and can determine response mime-type based on request header. More information can be found at http://webpy.org.
  INSTALL: sudo easy_install web.py

B.4.5 Interaction with back-end server via RESTful API

The back-end server exposes a RESTful API towards which client applications can perform requests. The following calls are available:

- **networkmap:**
  GET(project): return the network map associated with the project.
  POST: deploy the network map on the OpenStack platform.
- **rolesandcategories:**
  GET: return all the server templates that are defined. This also include network equipment such as routers and switches.
- **project:**
  GET: return all projects that are registered on the back-end server.
- **user:**
  GET(project): return all users associated with the project or all users if administrator.
- **physicalhosts:**
  GET: return data from all the physical hosts in the system. This request is only for administrators.
- **vmsfromhost:**
  GET: return all virtual machines that are running on the specified host.
- **imagefile:**
  GET: return a specified image file (.jpg or .png)
- **authenticate:**
  GET: return a value depending on the authentication of the user. Username and password are sent as parameter to authenticate the user against the Nova database.

Other less common commands are: e.g. rebootinstance(POST), availableips(GET), forbiddenips(GET), virtualservertemplate(POST) etc.

B.4.6 Monitor physical servers via RESTful API

The RESTful WS vibox-systeminfo is intended to be used to collect system information from physical servers using a simple RESTful API. Upon a request the server delivers its state described by a JSON string.

Location

The RESTful WS vibox-systeminfo should be placed at each OpenStack server. The following package/source code structure is needed:

- format
  - format.py
- server
  - disk
    - disk.py
  - network
    - network.py
  - server.py
- daemon.py
- README.txt - please read it before running the daemon
- **vibox-systeminfo.py**

are currently located at the OpenStack servers at */home/ericsson/vibox_python/ServerInfoWS/src/*

How to run

To run RESTful web service the following options are made available:

1. **As a daemon service** - sudo python vibox-systeminfo.py start|stop|restart
2. **Debug mode** - sudo python vibox-systeminfo.py debug

In all cases above the REST WS will answer on port 9000 (configurable in vibox-systeminfo.py source code). Requesting the system information is done via an URL formed like *http://[SERVER_IP]:9000/systeminfo*.

B.4.7 Network topology storage

OpenStack does not understand network topologies, why the back-end server itself stores the network topology abstraction in its database formed as a JSON string seen in the listing below. Using this network abstraction it can deploy virtual machines on the OpenStack platform with correct network configuration and also check the current state of deployed machines.

Listing B.2. Network topology stored as JSON string

```
{
"switches":
[
        {
        "subnet": "10.10.10.0",
        "description": null,
        "default_gateway": "10.10.10.1",
        "vlan": "br_1741",
        "y_position": 411,
        "netmask": "255.255.255.0",
        "role_name":
        "regular switch",
        "x_position": 286,
        "role_display_name": "regular switch",
        "image_url": null,
        "gui_vlan": "setup_vlan_2"
        }
],
"virtual_machines":
[
        {
        "is_router": true,
        "nr_of_vcpus": 1,
        "id": "990",
        "uec_image": "lucid",
        "start_priority": 0,
        "nics":
        [
                {
                "default_gateway": "0.0.0.0",
                "vlan": "br_1741",
                "netmask": "255.255.255.0",
                "mac_address": null,
                "interface": "vif0",
                "dhcp": false,
                "ip_address": "10.10.10.1"
                },
                {
                "default_gateway": "0.0.0.0",
                "vlan": "br_1742",
                "netmask": "255.255.255.0",
                "mac_address": null,
```

```
                "interface": "vif2",
                "dhcp": false,
                "ip_address": "10.10.11.1"
                }
        ],
        "gui_id": "instance_96",
        "memory": 256,
        "metadata":
        {
                "Key name": "user_2_key",
                "Image ID": "4",
                "Scheduled at": "2011-06-23 12:30:17",
                "Launched at": "2011-06-23 12:30:00",
                "Ramdisk ID": "",
                "Kernel ID": "3",
                "Launched on": "compute0"
        },
        "status": "running",
        "description": "Ubuntu server acting as router",
        "y_position": 265,
        "role_name": "ubuntu router",
        "x_position": 457,
        "host_id": null,
        "local_gb": 0,
        "name": null,
        "disks": [],
        "instance_type":
        "m1.verytiny",
        "image_url": null,
        "flavor_id": null,
        "role_display_name": "ubuntu router"
        }
]
}
```


B.4.8 How to run the service

The back-end server software is launched as a web server with the following command:

python webpy_server.py [OPTIONAL port number, DEFAULT: 8080]

B.5 Client application software

The client application was developed using NetBeans platform as a framework. In addition to using the module libraries this framework is equipped with out of the box, third party libraries have been used as well. The NetBeans platform configuration and additional packages are presented in this section.

B.5.1 Java version

The Java Runtime Environment used to run the application have been java version "1.6.0_18".

B.5.2 Configuration

All the configuration needed for the client application is located at org/nca/backendservice/configuration/RESTfulWS.java. Within this file a developer can configure the following:

- **Server URI** - The URI to the back-end server, i.e. "http://192.36.165.51"
- **Server Port** - The port that the back-end server is listening on, i.e. 8080
- **Server Base Path** - The base path that will be added to the URI, i.e. "/"
- **Polling Frequency** - A developer can set at what interval each worker should poll the corresponding state from the back-end server, i.e. the state of each virtual machine with-in a project
- **Available Server Calls** - In addition, this class also maintains all the possible calls that are available against the back-end server. The reason for this is to reduce the risk of spelling errors to one spot.

B.5.3 Netbeans platform

Version

6.9.1

APIs used

This section lists all the APIs used within the application that is available through the NetBeans SDK. The ones that are important to mention, due to how the API have been used when developing the application, have an additional short description.

- **Actions API** - The toolbar buttons, i.e. "deploy network map" is built on-top of this API.

- **Apache's Felix OSGi Implementation**
- **Auto Update Services/UI** - This API is used to enable and disable modules depending on user rights. The API makes this possible, even at runtime.
- **Bootstrap**
- **Common Palette** - Used to create the network palette visible when editing the network map.
- **Core Executing/UI/Windows**
- **Datasystems API**
- **Dialogs API** - Used as a wrapper when creating dialog pop-ups.
- **ETable and Outline**
- **Execution API**
- **Explorer & Property Sheet API** - Used for the explorer views of projects and regions, as well as the property mapping towards the property window component.
- **File System API**
- **General Queries API**
- **I/O API**
- **JavaHelp Integration**
- **JNA**
- **Keyring API**
- **Look & Feel Customization Library**
- **Lookup** - This API is required to be able to find instances of classes between modules with very little dependency introduced. Using this API results in producing "good practise" code.
- **MIME Lookup API**
- **Module System API**
- **Nodes API** - Every clickable component in the application is a node.
- **Options Dialog and SPI** - Pop-ups with user interaction enabled.
- **OSGi Specification**

- **Progress API/UI** - The progress bar giving users information of heavy tasks that the application is currently working on.
- **Quick Search API**
- **Settings API**
- **Startup API**
- **Tab Control**
- **Text API**
- **UI Utilities API**
- **Utilities API**
- **Window System API** - This API provides window management to the application. This makes the application behave as expected, i.e. undocking/docking window components and replacing them with-in the application frame. The API also introduces some statefulness to the application and remembers the layout of components used the last time the application was run.
- **Visual Library API** - The network map window component is built on-top of this API.

B.5.4 Additional libraries

This section contains third party APIs used when developing the client application.

- **rest-assured** - This API is used for interaction with the back-end server through the REST client interface provided. Furthermore, the API handles JSON parsing and modification. More information on the API can be found at http://code.google.com/p/rest-assured.

Appendix C

GUI screenshots

This chapter contains screenshots of the graphical user interface in action. First there is a figure, figure C.1, showing the system when logged in as a regular user. Secondly, figure C.2, shows the system when logged in as an administrator. Here you can see that there are more than one project available and that the physical machines are also present. The third illustration, figure C.3, shows the region window that is only visible when logged in as an administrator.

Figure C.1. Screenshot of the graphical user interface

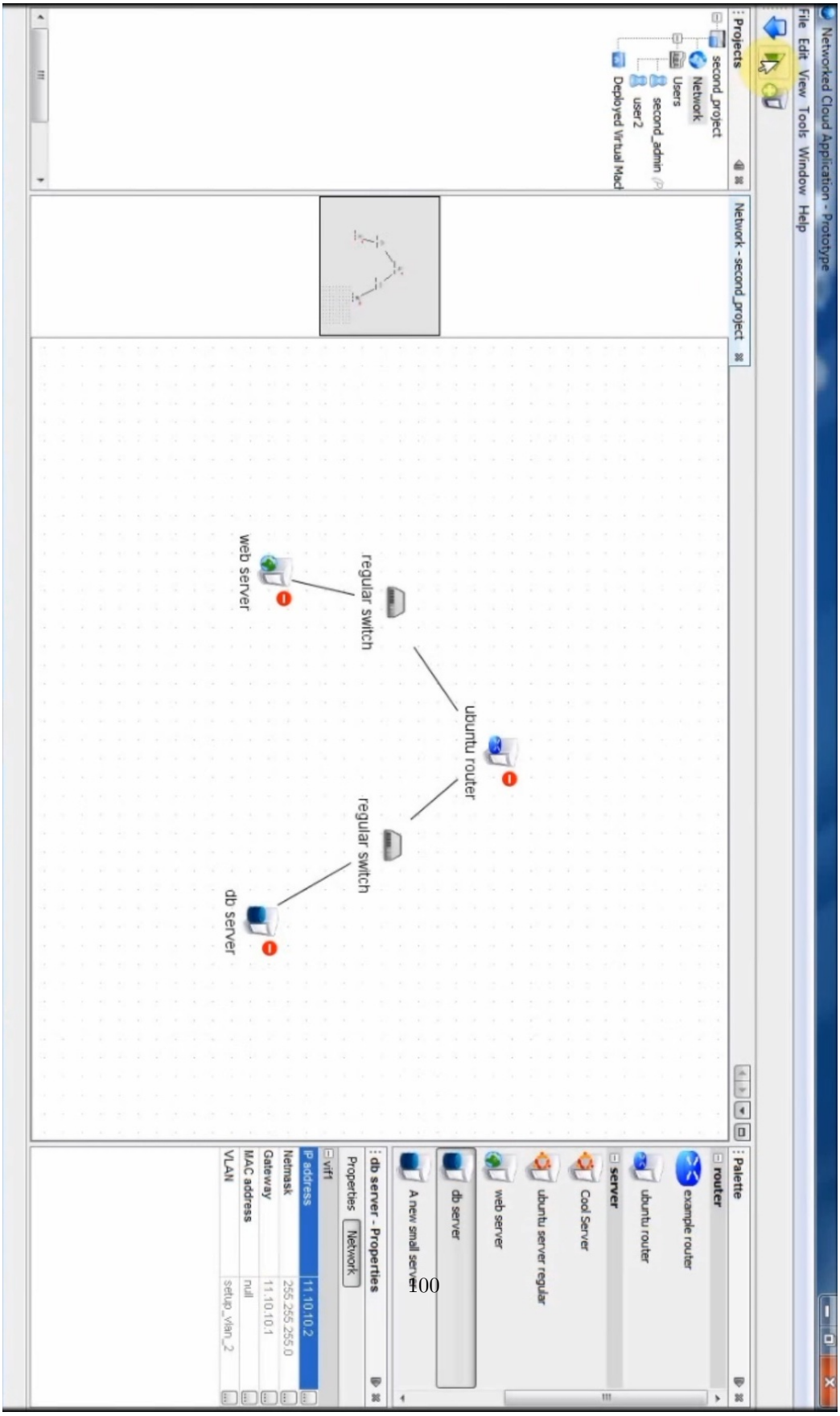


Figure C.2. Screenshot of the graphical user interface, logged in as an administrator

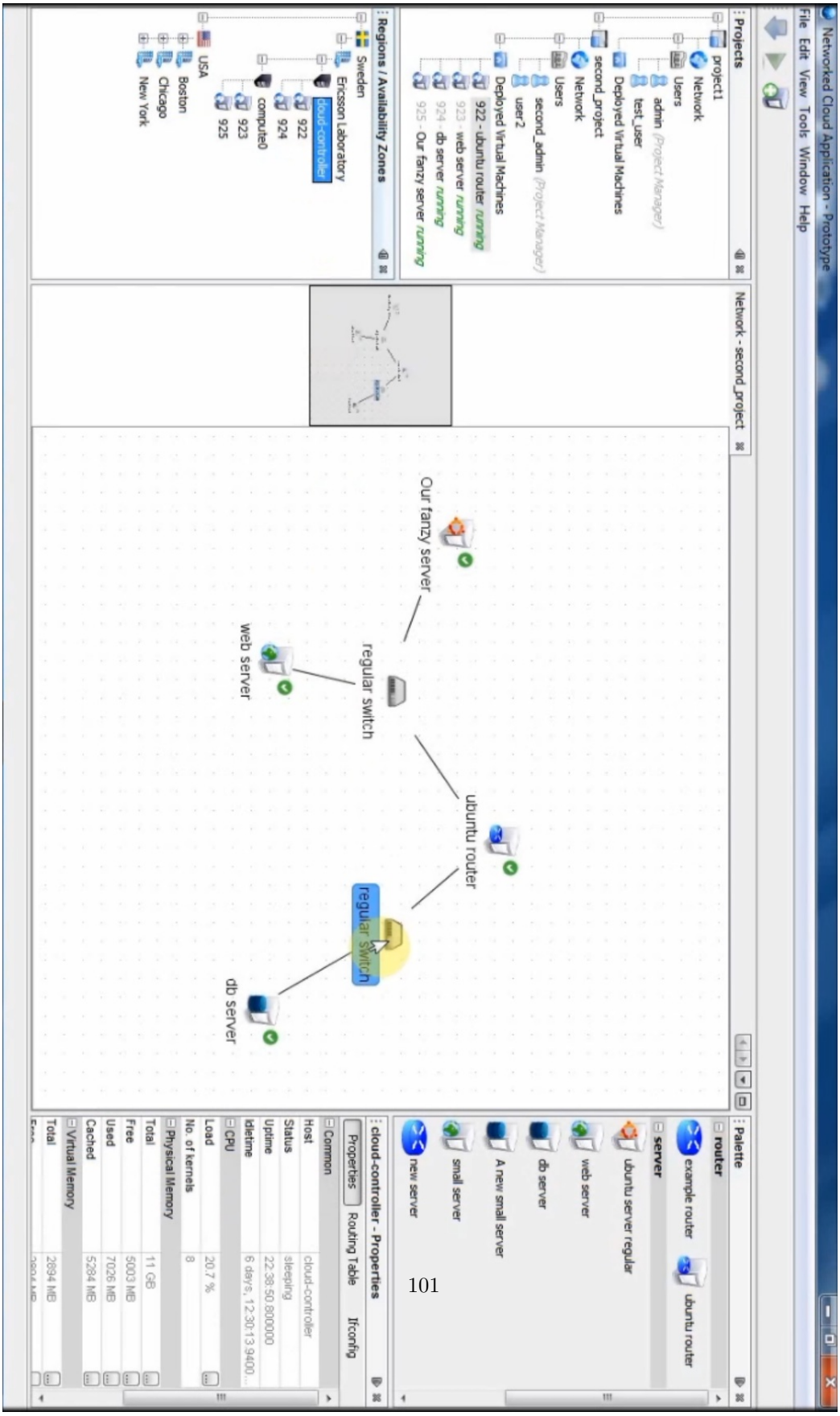


Figure C.3. Administrator region view

TRITA-ICT-EX-2011:206

www.kth.se