Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Kimmo Ahokas

Load balancing in LTE core network with OpenStack clouds:

Design and implementation

Master's Thesis
Espoo, Nov 9, 2015

Supervisor: Professor Antti Ylä-Jääski
Advisor: Docent Sakari Luukkainen

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Kimmo Ahokas

Title:
Load balancing in LTE core network with OpenStack clouds: Design and implementation

Date: Nov 9, 2015 **Pages:** 63

Major: Data Communication Software **Code:** T-110

Supervisor: Professor Antti Ylä-Jääski

Advisor: Docent Sakari Luukkainen

Cloud computing, software defined networking (SDN) and network functions virtualisation (NFV) are gaining popularity among the whole IT industry. The scalability, seemingly infinite amount of resources and flexible payment options make these technologies attractive for wide variety of use cases. As a new trend the industry is exploring the possibility of combining private and public clouds as a single hybrid cloud.

The network industry has been slow in adopting these new technologies, partly because the aging proprietary technologies are difficult to implement with current cloud technologies. However, new 4G networks are built using protocols commonly used in internet thus making it finally possible to implement the core network using virtualisation and cloud technologies.

This thesis explores the possibility of implementing LTE core network components as applications running on virtual machines provided by OpenStack cloud. Furthermore, we build a hybrid cloud consisting of two OpenStack instances and implement simple load balancer that automatically creates virtual machines to the most suitable cloud when requested by application.

We show that OpenStack as a cloud platform is suitable for building hybrid clouds. With powerful hardware and fast network it can be used for running LTE core network components. Furthermore, we show that the LTE network components could be implemented using hybrid clouds and that the technology could benefit network operators. The possible benefits include lower cost for running the network, increased resiliency and even increased performance for end users if the system is built correctly.

Our findings also show that more optimized implementations for network entities are needed for building large-scale LTE networks using hybrid cloud. Also the algorithm for placing virtual machines needs refinements.

Keywords: OpenStack, NFV, cloud, load balancing, LTE, vEPC

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Kimmo Ahokas		
Työn nimi:	Kuormantasaus LTE-ydinverkossa OpenStack-pilvien avulla: Suunnittelu ja toteutus		
Päiväys:	9. Marraskuuta2015	**Sivumäärä:**	63
Pääaine:	Tietoliikenneohjelmistot	**Koodi:**	T-110
Valvoja:	Professori Antti Ylä-Jääski		
Ohjaaja:	Dosentti Sakari Luukkainen		

Pilvilaskenta, ohjemlistomääritteinen tietoliikenne (SDN) ja verkkokomponenttien virtualisointi (NFV) kasvattavat jatkuvasti suosiotaan IT-alalla. Skaalautuvuus, näennäisesti rajattomat resurssit, joustavat hinnoittelumallit sekä alempien kerrosten abstraktointi tekevät näistä teknologioista houkettelevia useissa erilaisissa käyttötarkoituksissa. Uutena trendinä alalla nähdään julkisten ja yksityisten pilvijärjestelmien integrointi niinsanotuksi hybridipilveksi.

Verkkoalalla uusien teknologioiden käyttöönotto on ollut verrattaen hidasta, osittain siksi että vanhojen verkkoteknologioiden yhteensovittaminen pilvipalveluiden kanssa on haastavaa. Mutta koska uudet 4G-verkot on toteutettu internetissä yleisesti käytetyn IP-protokollan avulla, voidaan ydinverkko vihdoin toteuttaa käyttämällä erilaisia pilvipalveluita.

Tässä diplomityössä tutkitaan LTE-ydinverkon osien toteutusta ohjelmistokomponentteina, jotka suoritetaan OpenStack-pilvialustan tarjoamilla virtuaalikoneilla sekä -verkoilla. Tätä tarkoitusta varten rakennamme kahdesta erillisestä OpenStack-pilvestä koostuvan hybridipilven. Lisäksi toteutamme yksinkertaisen kuormantasauskomponentin, joka luo automaattisesti virtuaalikoneita tarpeen mukaan vähemmän kuormitettuun pilveen.

Osoitamme, että OpenStack-alusta soveltuu hybridipilvien toteuttamiseen ja riittävän tehokkaalla laitteistolla varustettuna myös LTE-ydinverkon komponenttien ajamiseen. Tuloksemme osoittavat myös, että LTE-ydinverkon toteuttaminen on mahdollista hybridipilvien avulla ja että tälläisestä järjestelystä on mahdollisesti hyötyä verkko-operaattoreille. Mahdollisiin hyötyihin lukeutuvat tehokkuuden ja virheidensietokyvyn parantuminen sekä kustannusten lasku.

Tuloksistamme myös nähdään, että paljon lisää tutkimus- ja kehitystyötä vaaditaan, ennen kuin nämä teknologiat ovat valmiita kaupalliseen käyttöön. LTE-komponentit vaativat huomattavan määrän optimointia toimiakseen tarpeeksi tehokkaasti ja lisäksi kuormantasauksen algoritmeja on hiottava.

Asiasanat:	OpenStack, NFV, pilvipalvelu, kuormantasaus, LTE, vEPC
Kieli:	Englanti

Acknowledgements

I would like to thank my instructor Sakari Luukkainen and supervisor Antti Ylä-Jääski for making this thesis possible. Big thanks to my co-workers, especially Antti Tolonen and Gopika Premsankar for the tremendous amount of help I got from both of them. Also the great people from Aalto Electrical engineering department, Jose Costa-Requena, Jesus Llorente Santos and Vicent Ferrer Guasch deserve my gratitude for their help. Finally I would like to express my gratitude for my girlfriend, family and friends for their support with this thesis.

Espoo, Nov 9, 2015

Kimmo Ahokas

This work has been performed in the framework of CELTIC-Plus project C2012/2-5 SIGMONA. The author would like to acknowledge the contributions of their colleagues, although the views expressed are those of the author and do not necessarily represent the project. This information reflects the consortium's view, but the consortium is not liable for any use that may be made of any of the information contained therein.

Abbreviations and Acronyms

3G	3rd Generation
3GPP	3rd Generation Partnership Project
4G	4th Generation
APN	Access Point Name
AWS	Amazon Web Services
CCFM	Cross-Cloud Federation Manager
COTs	Commercial of-the-self
CSS	Cascading Style Sheet
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
E-UTRAN	Evolved Universal Terrestrial Radio Access Network
eNB	Evolved Node B
EPC	Evolved Packet Core
ETSI	European Telecommunications Standards Institute
GRE	Generic Routing Encapsulation
GTP	GPRS Tunnelling Protocol
HSS	Home Subscriber Server
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
ICMP	Internet Control Message Protocol
IMSI	International Mobile Subscriber Identity
IP	Internet Protocol
IPsec	Internet Protocol Security
JSON	JavaScript Object Notation
KVM	Kernel-based Virtual Machine
LTE	Long Term Evolution
MME	Mobility Management Entity
MMU	Memory Management Unit
NAT	Network Address Translation

NFS	Network File System
NFV	Network Functions Virtualisation
NTP	Network Time Protocol
OS	Operating system
OVS	Open vSwitch
PaaS	Platform as a Service
PDN-GW	Packet Data Network Gateway
QoS	Quality of Service
REST	Representational State Transfer
RTT	Round Trip Time
S-GW	Serving Gateway
S3	Simple Storage Service
SaaS	Software as a Service
SDN	Software Defined Networking
SIM	Subscriber Identity Module
SMSC	Short Message Service Center
SSH	Secure Shell
UE	User Equipment
vEPC	Virtualised Evolved Packet Core
VLAN	Virtual Local Arean Network
VM	Virtual Machine
VPN	Virtual Private Network
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language

Contents

Abbreviations and Acronyms **5**

1 Introduction **9**
1.1 Motivation 9
1.2 Research Goals 11
1.3 Structure of the Thesis 12

2 Background **13**
2.1 Cloud Computing and Virtualisation 13
2.2 Load Balancing Among Multiple Clouds 14
2.3 OpenStack 15
2.3.1 OpenStack Nova Compute Service 15
2.3.2 OpenStack Neutron Networking Service 16
2.3.3 OpenStack Glance Image Service 21
2.4 Software Defined Networking and Network Functions Virtualisation 21
2.5 Long Term Evolution (LTE) 23
2.5.1 Distributed MME 25
2.6 Web Technologies 26
2.7 Related Work 28
2.7.1 Cloud federation and load balancing 28
2.7.2 Virtualization performance 29
2.7.3 Virtual Evolved Packet Core 29

3 System Design **30**
3.1 High-level Architecture 30
3.2 Internal Architecture 32

4 Testbed Architecture **35**
4.1 Physical Architecture 35
4.2 Software Architecture 37

4.3 Network Architecture 39

5 Evaluation **41**
5.1 OpenStack Testbed Network Performance 41
5.1.1 Network Latency 42
5.1.2 Network Throughput 43
5.2 MME Load Balancing 44
5.3 MME Performance on Multiple Clouds 46

6 Discussion **49**
6.1 Testbed Performance 49
6.2 Multi-cloud Load Balancing 50
6.3 Distributed MME on Multiple Clouds 51
6.4 Future Work 52

7 Conclusions **54**

A Commands for configuring SIGMONA project in OpenStack **60**

Chapter 1

Introduction

1.1 Motivation

Cloud computing has gained a lot of traction during last few years among IT industry. The virtually infinite resources and near real-time scalability offered by various cloud services are very attractive aspects in many use cases. Furthermore, the pricing models are often compelling and getting started with cloud platforms is easier than building and maintaining own infrastructure. Multiple cloud operators offer *virtual machines (VM)* with pay-as-you-go pricing model where customers pay only on the usage without any upfront costs. Combining this pricing model with scalable software components could offer substantial cost savings to service operators.

For larger companies maintaining their own physical infrastructure is still beneficial, as the price of computation in large environments can be lower than in cloud services. Furthermore, on-premise infrastructure can offer better performance than cloud service especially if the applications require high-speed connections between servers and end-users. To fully utilize their infrastructure some of these companies employ cloud-like management systems for their infrastructure. This enables teams or individual employees to reserve computation capacity from this *private cloud* on-demand without intervention from system administrators. However, with this approach companies lose the illusion of infinite resources and the cost savings related to public cloud.

To get the benefits of both private and public clouds companies are moving to so-called *hybrid cloud* [37], where companies own part of the infrastructure and offload part of the computation to public cloud. This way companies can fully utilize their private cloud and still have infinite amount of resources from public cloud.

To build hybrid cloud setups operators need tools to manage their resources across multiple isolated environments. Resources should be located to optimal location to gain cost savings and to offer best possible performance to end users. Thus the physical location as well as network connections and computation prices should be considered when deploying virtual machines. All these decisions should happen automatically in fraction of second when service load increases. Moreover, the tools should be integrated to applications and services so that the scaling decisions can be affected by most relevant factors determined by application developers.

Seamless integration of the public and private clouds is a must for hybrid clouds. Ideally resources are reserved through single interface and system automatically reserves resources from private or public cloud. The location of resources should not matter to the actual application, instead all resources across the whole hybrid cloud should offer similar performance. Furthermore, the reserving and releasing of resources should be possible via *Application Programming Interface (API)* so that it can be integrated easily to applications. This way applications can trigger the scaling operations automatically based on actual load, without administrator intervention.

The transition to cloud computing has led to a significant change in software architecture as well. To fully utilize the benefits of cloud computing software must be designed to be *horizontally scalable* [18]. Practically this means that running multiple copies of the same program should improve the performance for end-users. In best case application performance is linearly scalable, which means that doubling the number of running copies of the application doubles the performance.

Load balancing among multiple clouds could offer significant benefits for the whole IT industry from small business to global service providers. Seamless integration of resources from multiple vendors and locations can improve application performance and resilience while reducing costs. However, the benefits come with trade-offs so companies leveraging multiple clouds must choose which benefits to emphasize in their deployments.

At the same time new network technologies such as Software Defined *Networking (SDN)* and *Network Functions Virtualisation (NFV)* are gaining popularity among IT industry. These technologies enable network operators to abstract the logical network independently of the physical network. SDN and NFV aim for better error tolerance and higher utilization in networks thus enabling new opportunities for cost saving. Moreover, these technologies allow abstracting multiple physical networks as a single virtual network.

Traditional phone networks, including current 3G networks, are based on circuit-switched systems and built using highly customized, dedicated hardware and software solutions [35]. The high complexity in networks prevents

network operators from modifying their networks in accordance to changing capacity demand. Instead they must plan and install network equipment based on predicted future peak load so that the network can handle future load before new equipment is installed. As these dedicated devices are expensive it is not economical for operators to have unused network capacity. Instead scalable solutions that can react to changes in network load are needed.

New 4G networks such as *Long Term Evolution (LTE)* are based on common *Internet Protocol (IP)* and *packet switching* which makes it possible to use common hardware and simpler software to build the network. Especially the LTE core network, *Evolved Packet Core (EPC)*, is a good candidate for virtualisation. EPC contains multiple different components that are responsible for authenticating users, managing mobility, billing and data routing. In *Virtualised Evolved Packet Core (vEPC)* [20] the different network entities are implemented as software applications running inside virtual machines. This makes it possible to scale the core network capacity simply by adding or removing virtual machines. Using cloud technologies in EPC could possibly improve network performance and resilience and even decrease the cost of running the network.

At the moment OpenStack seems to emerge as one of the most popular cloud resource management platforms among telecommunication industry. It is open-source software that allows users to customize it to their needs thus making it good choice for novel applications. Furthermore, the platform is designed to be highly scalable and fault tolerant. For these reasons OpenStack is selected as the cloud platform to be studied in this thesis. However, the same principles should be applicable to any other cloud platform offering similar services as well.

1.2 Research Goals

This thesis studies the possibility of implementing vEPC using hybrid cloud setup. We begin by implementing a testbed consisting of two independent OpenStack clouds. Then, we build simple load balancer that can share the application load among the two clouds. Finally we evaluate the performance of our system in vEPC use case. Specifically we deploy an EPC entity called *Mobility Management Entity (MME)* on the platform and assess it's performance.

The detailed research goals are as follows:

1. Deploy two OpenStack installations as a hybrid cloud platform.

2. Implement prototype load balancing system for multiple OpenStack clouds.

3. Analyze the networking performance of the deployed platform.

4. Analyze the performance of distributed MME on the deployed platform.

5. Assess the viability of the developed solution in real use cases.

To validate our results we compare them to prior research. Especially the measurements done by Guasch [19] and Tolonen [39] are used for comparison.

The main contributions of this thesis is two-fold. First matter is the viability of hybrid OpenStack cloud and the performance of such setup. Second is the viability of hybrid clouds for vEPC use case.

1.3 Structure of the Thesis

This thesis is divided in seven chapters. Chapter 1 introduced the topic of this thesis and our research goals. Chapter 2 explains the related technologies and most interesting research on related topics. Chapter 3 details the built load balancer and it's implementation. Chapter 4 presents our testbed, detailing both physical architecture and software configuration. Chapter 5 explains our test methods and the results of measurements. Chapter 6 analyses the results obtained in previous chapter and discuses the reasons and implications of the results. Finally in Chapter 7 we present our conclusions and briefly suggest some topics for further research.

Chapter 2

Background

This chapter discusses the most relevant technologies used in our system and the related research on the topic. First we introduce the topic of cloud computing. Second, we discuss the benefits of leveraging multiple clouds simultaneously. Third section addresses the OpenStack platform and it's internal architecture in detail. The fourth section briefly introduces Software Defined Networking and Network Functions virtualisation. The fifth section discusses LTE networks and EPC. The sixth section explains the web technologies used for our load balancer implementation. Finally the seventh section gives overview of the most interesting related research.

2.1 Cloud Computing and Virtualisation

Cloud computing has become widely used paradigm in the cloud industry in a surprisingly short time scale. Armbrust et all. describes cloud computing as a "long-held dream of computing as a utility" [5]. According to them cloud computing has three significant advantages compared to traditional computing models:

- Appearance of infinite computing resources available on demand.
- Elimination of up-front hardware costs.
- Pay-as-you-go pricing allowing reserving resources only when needed.

In essence cloud computing allows companies to acquire computing resources on-demand and only pay based on actual usage. Thus companies do not need to reserve resources for peak usage. Although on-demand resources are usually more expensive than reserved resources, the cloud model can

lower total expenses because resources can be reserved and released based on momentary service load in matter of minutes.

To realize the full potential of cloud computing high-performance virtualisation solutions are needed. Virtualisation enables cloud operators to run multiple independent operating systems on a single physical machine at the same time. Thus the operating systems share the available physical resources. In ideal situation every operating system functions as if it was the only operating system on the machine, virtual machines should not have any access to each other and the performance would be equal to running on bare metal.

However, cloud computing requires that applications are designed to be vertically scalable. This means that application servers should be nearly stateless and adding new application servers should increase the applications performance or user capacity nearly linearly. This allows adjusting the system performance based on load just by adding or removing virtual machines to the system.

2.2 Load Balancing Among Multiple Clouds

Inter cloud load balancing means buying computation or network resources from multiple cloud platforms and distributing application load among the different clouds. This approach can offer multiple benefits including better performance, improved resilience and reduced cost. However, the benefits come with a trade-off of increased complexity and management overhead.

Traditionally companies must deploy servers to handle the peak load of their software. Usually this means that the servers are highly utilized during the day but during nights there are unused resources. Even single cloud helps improve the situation, as companies can buy only the resources they need. But multi-cloud setup can offer more benefits.

The pricing models in cloud services can be complicated, thus it is possible to achieve cost savings by intelligently deciding from where to buy resources. For example network data transfer price is often dependent on the physical location of the client. Transferring data to a client in USA from data center located in Europe can cost a lot and decrease the user experience as latencies are high. Thus locating servers close to users can offer multiple benefits. Multi-cloud load balancing allows companies to dynamically relocate their application servers to as close to clients as possible.

If the used resources are located to different clouds, failures in single cloud will not affect the whole service and all users. Furthermore, in case of a failure in single cloud service the needed resources can be temporarily

moved to other cloud services. Thus multi-cloud load balancing can be used to improve application resilience.

Distributing the network application to multiple clouds requires that the application nodes can communicate with each other. One option is to transfer traffic through public internet, but it is not often desired, especially if the application traffic is not encrypted. More secure way of connecting the networks in different clouds is using *Virtual Private Network (VPN)*, which is a secure tunnel between clients over insecure internet [38]. For example *Amazon Web Services (AWS)* public cloud supports *Internet Protocol Security (IPsec)* protocol suite for connecting to private networks securely [22].

2.3 OpenStack

OpenStack is an "open source software for creating private and public clouds"[1]. It was selected as a target platform for this thesis because it seems to be one of the most popular open-source cloud platforms. According to extensive analysis of open-source IaaS platform communities, OpenStack has the largest amount of active contributors and participants in discussions [21]. Due to it's fully open-source code and support for new standards OpenStack is seen as the most promising solution for implementing a cloud platform [36].

The OpenStack architecture is documented extensively in the project documentation[2]. Although the newest version of OpenStack is Kilo (released on April 2015), on this thesis we concentrate on the previous version, OpenStack Icehouse. OpenStack is a modular platform that consists of multiple parts that together form the complete IaaS platform. The main parts of OpenStack are Nova Compute service, Neutron Networking service and Glance Image service. In addition there are shared services, such as Keystone Identity service and Horizon Dashboard which provides web-based user interface for end users.

2.3.1 OpenStack Nova Compute Service

The OpenStack Nova Compute service manages virtual machines in an OpenStack cloud. It is responsible for scheduling VMs in a single cloud, connecting the instances to selected virtual network provided by Neutron networking, connecting the instances to virtual machine images from Glance and connecting to additional storage services.

[1] `http://http://www.openstack.org/`
[2] `http://docs.openstack.org/`

Nova consists of multiple services running on multiple hosts on the OpenStack cluster. The main components of Nova are API server, scheduler, messaging queue and virtualisation drivers [27]. As the name implies, API server implements the front-end API that end-users and other OpenStack components use to communicate with the computing service. Scheduler is responsible for allocating physical resources for virtual machines by selecting suitable compute node. Messaging queue is used for internal communication between different components of the compute service. Virtualisation drivers are used for communicating with the actual virtualisation hypervisor.

Nova supports multiple virtualisation hypervisors via different virtualisation drivers. The default hypervisor used by OpenStack Nova is qemu[3]/KVM[4] with libvirt[5]. Other supported hypervisors include but are not limited to Microsoft Hyper-V[6], VMware[7], XenServer[8] and Xen[9] with libvirt.

As mentioned, the default configuration of nova uses libvirt, qemu and KVM to run the actual virtual machines. Libvirt provides a stable API and programming language bindings for multiple hypervisors that are used by the nova driver. Qemu is a generic machine emulator that can run software made for one machine on another machine, for example ARM code on x86 platform. With KVM (Kernel-based Virtual Machine) it can be used as a high-performance virtualisation software for x86 architecture. KVM itself is a Linux kernel module that enables hardware-accelerated virtualisation on platforms using Intel and AMD processors. The main benefits of KVM are virtualised *memory management unit (MMU)*, I/O virtualisation, Linux integration and support for live migration [23]. The qemu/KVM hypervisor achieves near native performance, as most of the CPU instructions in quest operating system and software are run directly on the physical CPU. Only small subset of x86 commands must be intercepted, translated by qemu and then executed on the CPU.

2.3.2 OpenStack Neutron Networking Service

The OpenStack network architecture greatly influences the performance of our testbed. Thus, it is essential to understand how OpenStack connects the virtual machines to each other. OpenStack supports multiple different

[3] `http://www.qemu.org`
[4] `http://www.linux-kvm.org/`
[5] `http://libvirt.org/`
[6] `https://technet.microsoft.com/fi-FI/library/hh831531.aspx`
[7] `http://www.vmware.com/`
[8] `http://xenserver.org/`
[9] `http://www.xenproject.org/`

network configurations, including *Open vSwitch (OVS)* and Linux bridge networks. Our system is configure to use Open vSwitch as network back-end, so in this section we give overview of this architecture in OpenStack. The Open vSwitch networking is described in great detail in OpenStack networking guide [29].

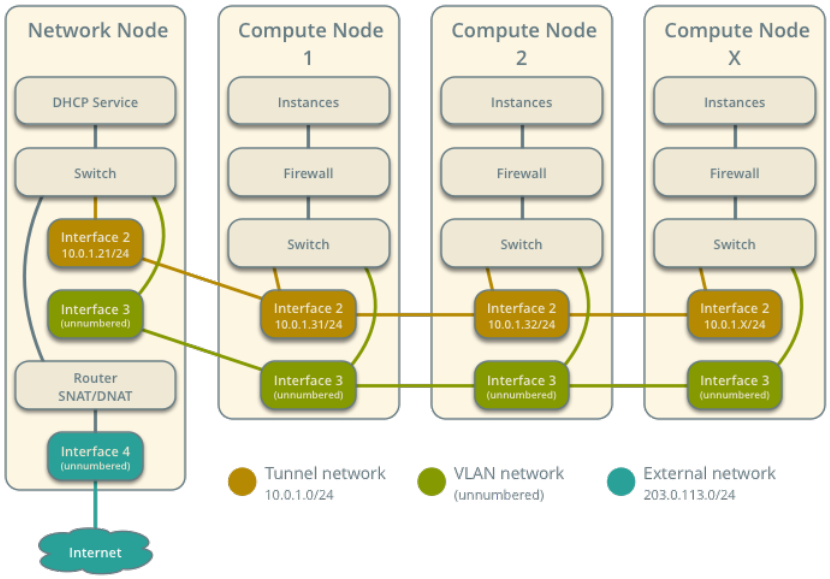

Figure 2.1: The general OpenStack network architecture with Open vSwitch configuration. [29]

Figure 2.1 introduces the high-level netowrk architecture in OpenStack. As can be seen, compute nodes run firewall and switch services that provide connectivity to virtual machines. The each virtual switch connects to switches in other compute nodes and network node. The switch in network node is connected also to virtual router which then connects to internet. The figure shows two tenant networks, one using *Generic Routing Encapsulation (GRE)* tunnels and one using *Virtual Local Area Network (VLAN)*. Each virtual network in OpenStack has either own VLAN tag or GRE tunnel tag. Essentially both VLAN and GRE allow creating multiple virtual networks which share the same physical network. It is not necessary to configure both of these networks, but the image illustrates that it is possible to use both techniques simultaneously.

Figure 2.2 details the networking components running inside the network node. There are Open vSwitch bridges for each physical network interface, in this example br-tun for GRE-tunneled tenant networks, br-vlan for VLAN

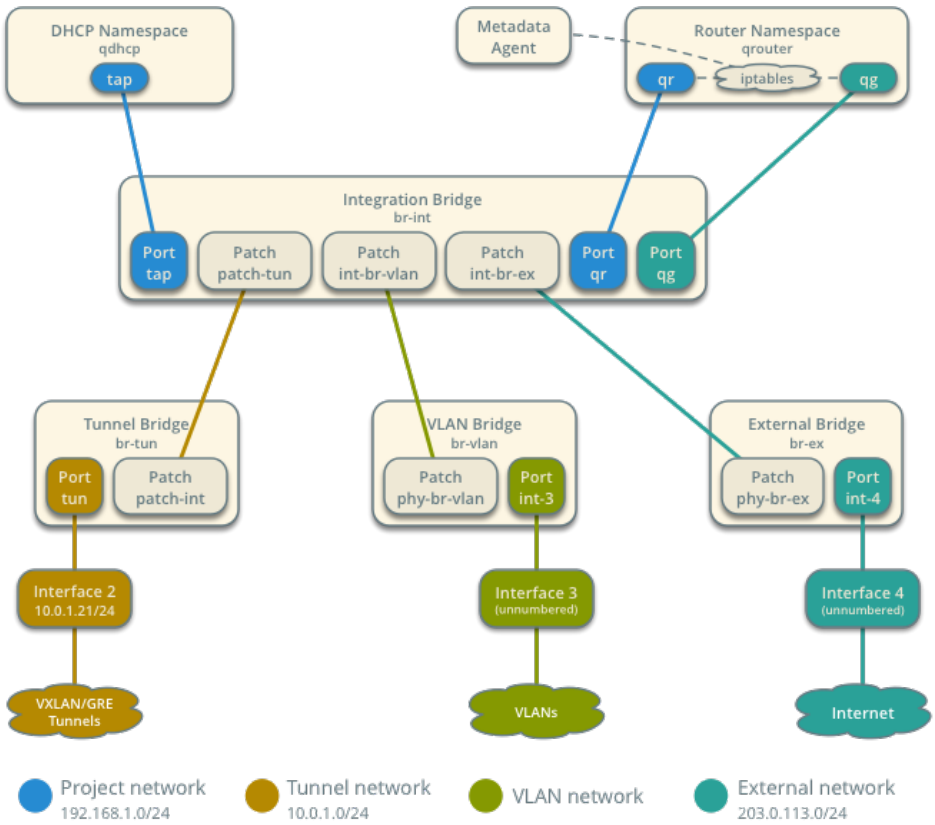

Figure 2.2: Overview of network components running inside network node in Open vSwitch configuration. [29]

tunneled tenant networks and br-ex for external network (internet). Integration bridge br-int is used for connecting tenant-specific network components to specific tenant network. For each tenant network there is dedicated *Dynamic Host Configuration Protocol (DHCP) server* running on the network node. For all virtual routers the network node has four virtual ports in total and a metadata agent. The routing is implemented using iptables filtering.

In addition to network bridges the OpenStack network node uses Linux network namespaces[10] extensively to separate tenant networks from each other. Network namespaces provide applications with their own copy of Linux network stack. Each network namespace has it's own routing rules, firewall rules and network devices. Each DHCP server has their own network namespace as does each virtual router. Thus networks belonging to different tenants can't access resources from other tenants.

[10] `http://man7.org/linux/man-pages/man8/ip-netns.8.html`

On conceptual level the network node contains multiple virtual routers. In most use cases each tenant has their own virtual router, which is connected to shared public network (internet) and to one ore more tenant-specific internal networks. Using the virtual router tenants can utilise *Network Address Translation (NAT)* to share one public IP address for multiple virtual machines or connect floating IP address to specific virtual machine. NAT allows tenants to have internal IP address blocks for virtual machines and one (or more) public address blocks for internet connection. All this preserves expensive public IP addresses.

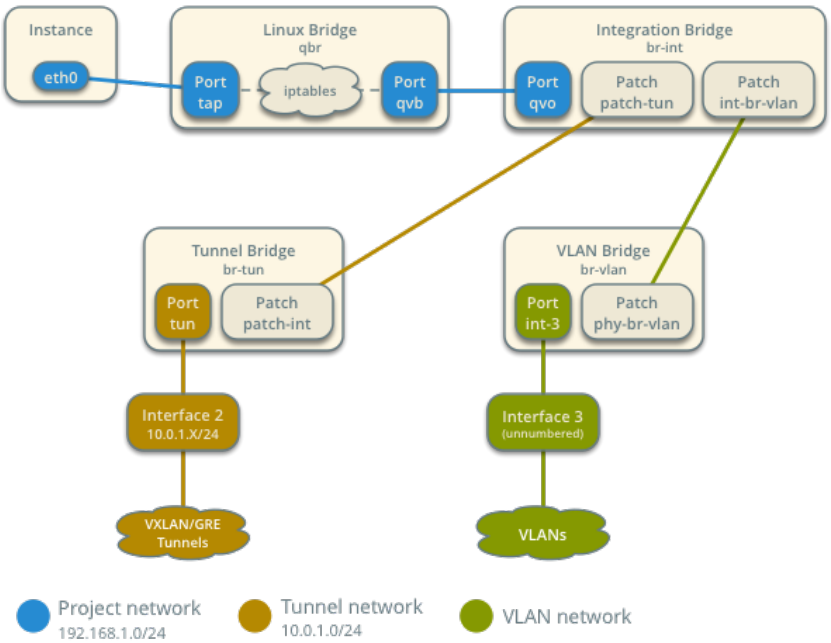

Figure 2.3: Overview of network components running inside compute nodes in Open vSwitch configuration.[29]

Figure 2.3 details the networking components running inside the compute nodes. The Open vSwitch architecture uses both Open vSwitch switches and Linux bridges to provide network connectivity to tenant networks and to separate networks belonging to different tenants. Each compute node has one bridge for each physical network interface, br-tun for gre tunneled networks or br-vlan for vlan tunneled networks. Integration bridge br-int is used for connecting virtual machines to tunneled networks. In addition there is one Linux bridge for each virtual machine running on the compute node. This additional bridge is needed for OpenStack security groups which are implemented using iptables functionality in Linux kernel. Ultimately this is

a workaround for the fact that iptables rules can not be directly attached to Open vSwitch ports [1]. In total every IP packet leaving virtual machine must travel through 9 virtual network devices before reaching the physical network interface of the compute node (counting virtual ports and bridges as devices).

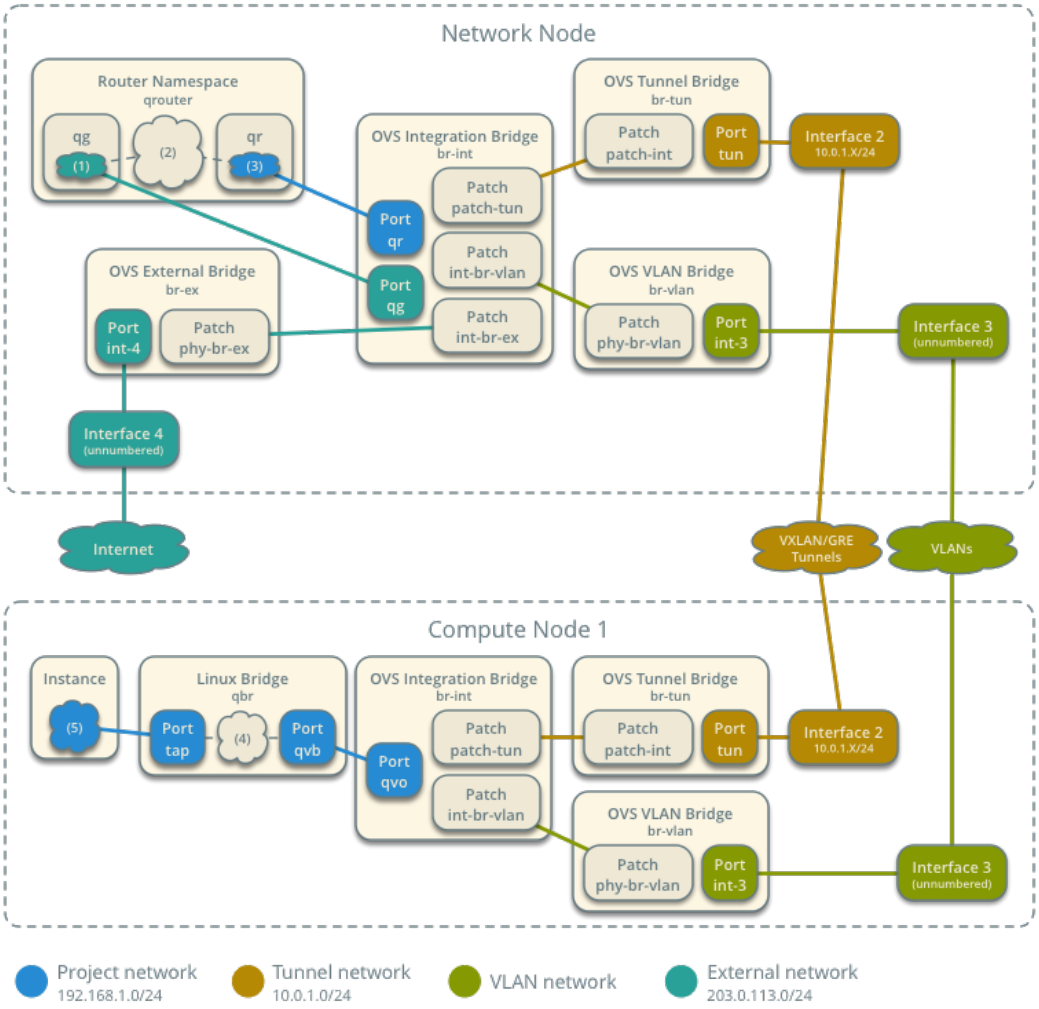

Figure 2.4: Network traffic flow from virtual machine in OpenStack to internet. [29]

Figure 2.4 gives overview of the components involved in delivering IP packets from virtual machine to public internet. In this case packets travel from virtual machine to compute node, which delivers them to network node. Network node then sends the packets to internet. Basically all components in

Neutron networking detailed before are involved when communicating with hosts outside of the OpenStack cloud. The network node acts as a router which is connected to internet and which then directs incoming traffic to correct virtual machine.

In total each IP packet will travel through 27 virtual or physical network devices before reaching the public internet. All this causes significant overhead and adds delay to communication. In some cases the processing can also limit the available throughput. The network performance of OpenStack is further analyzed in section 5.1.

2.3.3 OpenStack Glance Image Service

Glance is the primary image service in OpenStack installation. It provides virtual machine images for Nova. Essentially each image contains installed operating system. When launching virtual machine Nova connects the new instance to a user-specified image from Glance and provides configuration for the operating system.

Glance can use multiple different backends for storing the images. In the simplest case images are stored as files on the file system, either locally or using network filesystems. Other popular choices for storing images are OpenStack Swift object storage, Amazon *Simple Storage Service (S3)* and Ceph[11] distributed file system.

The high-level architecture of Glance is depicted in Figure 2.5. The most important component is the rest API provided for clients. Using the API clients such as Nova can retrieve images from Glance or store new images. Glance handles authentication using OpenStack keystone and then provides the image data for client. Client does not need to care about how the images are stored inside Glance, so changing storage backend only influences Glance itself, not clients. Also the database which Glance uses internally to store information about images is invisible to clients.

2.4 Software Defined Networking and Network Functions Virtualisation

Software Defined Networking (SDN) [26] is an emerging network architecture which decouples network control and actual user data forwarding. The network control is centralized and the underlying infrastructure is abstracted

[11] `http://ceph.com/`

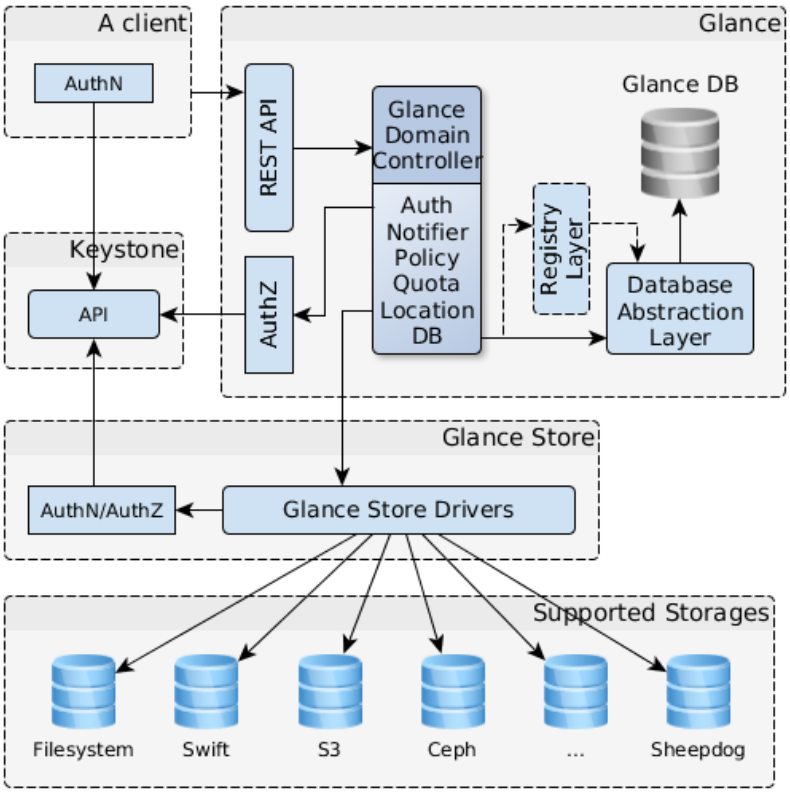

Figure 2.5: OpenStack Glance architecture [28].

to achieve easier network management. On a conceptual level software defined networking is divided to three layers, application layer, control layer and infrastructure layer.

On the application layer reside normal applications which utilize the network. Traditionally these applications have no control over the network they are operating on. However, SDN offers northbound API for application layer for modifying the logical network topology. Applications could for example request specific *Quality of Service (QoS)* class from SDN controller, which then would configure the network to fulfill the needs of all applications.

The OpenStack Neutron networking can be interpreted as software defined networking platform. The Neutron service offers API for other OpenStack services to request virtual networks. When tenants request network connections, Neutron networking delegates the responsibility of creating the networks on underlying hardware to appropriate plugin. The plugin then configures the networks so that tenants get the requested virtual network

connections. The Neutron networking also isolates tenant's network traffic from other tenants even though all of them share a single physical network.

Network Functions Virtualisation (NFV) [14] relies on network functions implemented as software components running on standard IT infrastructure.

European Telecommunications Standards Institute (ETSI) has identified multiple use cases for NFV. Use case #5: Virtualisation of Mobile Core Network and IMS as presented in [16] describes how NFV can be utilized to build mobile core networks.

2.5 Long Term Evolution (LTE)

Long Term Evolution (LTE) is the newest mobile network standard by the *Third Generation Partnership Project (3GPP)*. It is designed to improve mobile data rates and spectrum utilization while simplifying the network architecture. All communications in LTE happen using packet-switched IP protocol instead of multiple specialized and circuit-switched protocols used in previous wireless networks. As the core network is completely IP based, it is possible to build LTE-compliant networks using *commercial of-the-self (COTS)* hardware and existing operating systems and network applications. [35, Chapter 4]

The architecture and different components of LTE network are specified and standardized by ETSI 3rd Generation Partnership Project (3GPP) [2]. 3GPP is a global initiative consisting of seven telecommunications standard development organizations. Due to the global nature, the same network technologies are deployed across the world. Furthermore, the openness of the specification allows anyone to manufacture standard-compliant network entities.

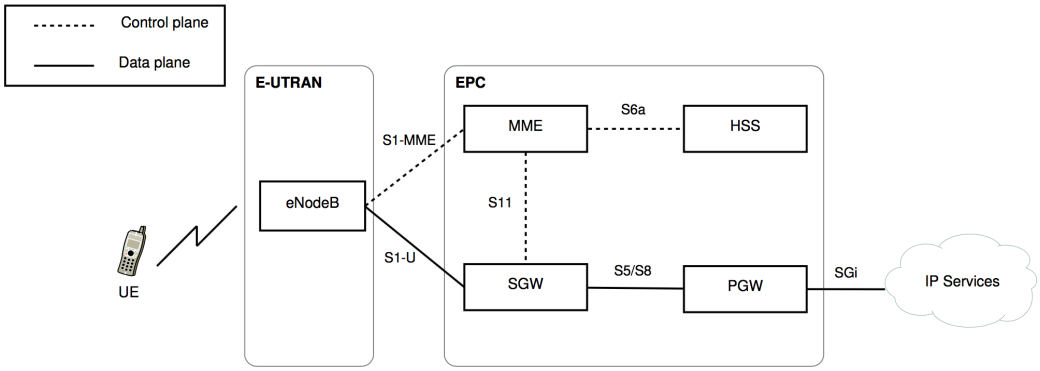

Figure 2.6: LTE network elements and interfaces overview [32].

On a high level the LTE network can be divided to two parts, the *Evolved Universal Terrestrial Radio Access Network (E-UTRAN)* and *Evolved Packet Core (EPC)*. The E-UTRAN is the wireless part of the LTE network and it consist of *User Equipments (UE)*, such as mobile phones, and *Evolved Node B (eNB)* which are the base stations in LTE network. The EPC contains all other network elements required to deliver data from eNodeB to public internet. Among these elements are *Mobility Management Entity (MME)*, *Serving Gateway (S-GW)*, *Packet Data Network Gateway (PDN-GW)*, and *Home Subscriber Server (HSS)*. Figure 2.6 presents the overview of different components in LTE network and the interfaces between the components.

The eNB is the most complex element of the LTE network as it has to implement both the radio communications interfaces and EPC communication interfaces. For the radio communication eNB needs antennas and radio module that modulates digital signals sent over radio interface. These are physical devices than can not be implemented in software, thus the whole eNB must be a physical device. In fact it is the only LTE network element that must be a standalone physical device. In addition to transmitting data between UE and EPC the eNB is responsible for scheduling air interface and managing quality of service, partly handles the mobility of UEs and manages interference between base stations. The eNB handles both signaling and user data. Usually there are hundreds of eNodeBs in a single network.

Mobile management entity is the main signaling entity in LTE network. It handles only signaling, never receiving or sending of any user data. When UE joins the LTE network, MME is responsible for authenticating the user using information from HSS. After authentication MME establishes bearers, which are IP tunnels that connect UE to internet through S-GW and PDN-GWs. MME also participates in the mobility management of UEs. If the UE moves between tracking areas or eNodeBs are not connected to each other then MME is needed for signaling. Also interworking with older phone networks and SMS and voice support are handled by MME. There are multiple MMEs in single network and each of them are responsible for part of the eNodeBs in the network.

The Home Subscriber Server is a database that keeps records of the subscribers of the operator. For each subscriber the HSS maps subscriber's *International Mobile Subscriber Identity (IMSI)* to other information of the subscriber. IMSI is used fro tracking the subscriber in home and foreign networks and it uniquely indentifies the subscriber. The IMSI is stored on the *Subscriber Identity Module (SIM)*-card of the user. Among the data that HSS stores is the MME in which responsibility area the subscriber currently is, the subscriber phone number and *Access Point Names (APN)* the subscriber is allowed to use. Each operator has usually a single HSS.

Serving Gateway is responsible for managing GTP tunnels between eNB and PDN-GW. To allow mobility while keeping existing TCP connections open all the user traffic in LTE network is transmitted in *GPRS Tunnelling Protocol (GTP)* tunnels. As long as the end of the tunnel that is connected to internet does not change, the UE can keep using the same IP address. The other end of the tunnel is transferred from eNodeB to another anytime the UE moves in the network. If the UE moves away from the area that is served by previous MME, the S-GW responsible for the tunnel is usually changed to a S-GW related to new MME. There are usually one S-GW for each MME in the network.

PDN-GW is the gateway to internet from LTE network. It terminates the GTP tunnel from eNodeB and transmits the user data to internet using standard IP protocol. PDN-GW is also responsible for assigning IP addresses for UEs and possibly doing Network Address Translation (NAT) if that technology is used by operator. In mobility scenarios the PDN-GW is the single entity that won't change during active session. This allows the UE to keep the same IP address and keeps network connections alive while user roams in the network. The amount of PDN-GWs in network depends on the amount of traffic service provider wants to be able to support.

2.5.1 Distributed MME

In this thesis we use LTE core network as an example application domain for our multi-cloud load balancing setup. Specifically we deploy distributed MME on the developed platform. This chapter briefly introduces the distributed MME designed and implemented by Gopika Premsankar [32].

Traditionally the LTE core network elements are static and it's not easy to add or remove these elements in a running network. However, using cloud techniques for scaling the network based on demand requires that we can add or remove virtual entities automatically based on demand. Thus, distributed architecture for MME is proposed. In this architecture the single logical MME is divided in three logical parts. These parts are MME front-end, MME worker node(s) and state database. The architecture of distributed MME is presented in Figure 2.7.

First part is front-end that handles all the communications with other LTE network components and MME worker nodes. It is designed to be just a thin proxy that relays traffic from other LTE network entities to worker nodes. Furthermore, it balances the traffic among worker nodes.

Second part is MME worker nodes that handle the computation required for different procedures the MME is involved with. These are the most computationally intensive part of the MME and the ideal subject for scaling.

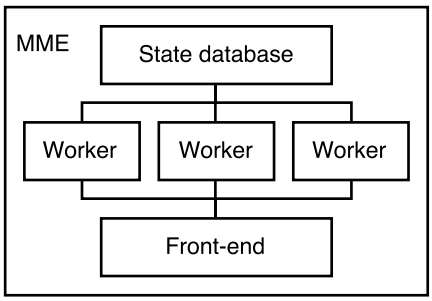

Figure 2.7: The architecture of distributed MME.

When there is only small amount of subscribers online, only few worker nodes are required. When the network load increases, new worker nodes can be added. As the worker nodes are not directly connected to other entities in LTE network, rest of the network does not need to know about the adding or removing MME workers. The worker nodes are stateless, thus removing a worker node is easy as no data needs to be saved.

The third part is a state database which keeps track of the internal status of the whole MME. This status includes information of connected clients, ongoing LTE procedures, and which worker is currently handling which client. All the information workers need to complete requests is stored in the state database. In the proof of concept implementation the author uses Redis[12] cluster as the state database. In this case the database itself is distributed among multiple nodes but worker nodes see it as a single database.

In addition to supporting EPC protocols support for the load balancer implemented in this Master's thesis was implemented in the MME front-end. Thus the MME front-end is able to request adding or deleting worker nodes from our load balancer. As the front-end makes the decision when to add or remove workers, the system has multiple metrics available for doing the scaling decisions. In this implementation the number of attach requests per second is used as a metric which triggers scaling.

2.6 Web Technologies

In current web services often provide Application Programming Interfaces (API) for other developers to interact with the service. Regarding this thesis

[12] `http://redis.io/`

the OpenStack API and our own load balancing API are in a central position. This section explains the relevant technologies related to these APIs.

Both of these APIs are built around *Representational State Transfer (REST)* [17] architecture using *Hypertext Transfer Protocol (HTTP)* [7]. In REST architecture servers should hold no client state. Instead every request coming from client should contain all the information required for completing the request. Conforming to REST architectural constraints makes the interfaces generalized and applications easily scalable. As all requests contain all the needed information subsequent requests from single client can be handled by different servers. Thus scaling can be done horizontally, adding multiple workers and distributing requests among workers.

Both of these APIs are built using Python programming language. It is not economical to implement web server with Python, so both of these APIs rely on *Web Server Gateway Interface (WSGI)* [15]. WSGI describes an easy to implement method for communication between web servers and web applications. Clients send HTTP requests to web server, which parses the request data and calls Python application object which calculates the result for given request and return the data back to web server. The web server then delivers the data to the client.

The WSGI architecture separates socket handling from the actual application logic. There are multiple high-performance web servers available, but extending those to generate dynamic content is out of their scope. Instead WSGI allows generating the content using external application. Furthermore, this approach allows implementing the web server and application using different programming languages. Common choice is to implement the web server with low-level, high performance language such as C and then use higher-level languages such as Python or Ruby for application logic.

The WSGI application can return any static or dynamic data to the web server. Generally it can be text as *Hypertext Markup Language (HTML)*, *Cascading Style Sheet (CSS)* or JavaScript documents, images, audio, video or any other document format. In the case of web APIs most modern applications use *Extensible Markup Language (XML)* [10] or *JavaScript Object Notation (JSON)* [9] as a data interchange format. Both the OpenStack API and our load balancer use JSON by default.

The JSON format has become incredibly popular in web development lately. Most people consider JSON as easier to read and write, for both humans and computers. JSON specification does not specify schema unlike XML, so developers save time as they don't need to write schema files for APIs. JSON is also the native data interchange format for JavaScript, so it is logical choice for web applications.

2.7 Related Work

Virtualisation of mobile core networks and inter-cloud federation are hot topics among the industry at the moment. Thus, a lot of research has been conducted on these topics. This section presents the most important related research results.

2.7.1 Cloud federation and load balancing

The interoperability of multiple cloud services is often called cloud federation. Multiple architectures have been proposed for connecting clouds from different vendors. Kurze et al. [24] investigates both commercial cloud providers and open-source cloud systems and possibilities for interoperability. Yang et al. [42] proposes cloud federation architecture especially for real-time applications such as games. Their architecture has layer for enforcing business SLA and lower layer for actual cloud control.

Celesti et al. [13] proposes usage of *Cross-Cloud Federation Manager (CCFM)* for intelligently managing resources across multiple clouds. In their solution users interact with so-called home cloud, which provides at least part of the computation capabilities needed bu users. When the home cloud runs out of capacity, it intelligently discovers other clouds that are capable of providing needed services and automatically schedules parts of computation to other clouds. As the functionality is built into the home cloud, users are not aware that part of their virtual machines are running on different clouds. The CCFM component handles discovering other clouds, selecting most prominent foreign cloud for needed resources and authenticating the user to foreign cloud. However, the research does not assess the performance of their solution or the network connections between these clouds.

In the paper by Villegas et al. [40] a layered architecture for cloud federation is presented. The idea is that federation can happen on SaaS, PaaS or IaaS layer. The different layers are responsible for different

Automatic scaling is an essential component of multi-cloud load balancing setup. Beernaert et al. [6] presents an elastic load balancer for OpenStack. Their implementation monitors CPU load across VM instances connected to given service. If the load is too high, the system starts new instance and then notifies the service that new instance is available. When CPU load decreases, the system will stop instances accordingly. To avoid oscillation the load balancer requires that two consecutive load checks yield the same action. While the solution is effective at preventing oscillation, it adds delay to load balancing decisions.

2.7.2 Virtualization performance

The performance of virtual machines in cloud is widely studied subject. Wang and Ng [41] studies the impact of virtualization for network performance in Amazon EC2 cloud. They show that especially with the smaller instances in cloud the network performance varies significantly and quite randomly due to shared physical resources. Both the throughput and latency between virtual machines varies during their measurements. Therefore they conclude that using cloud for network measurements and research should be done carefully. According to Ristov et al. [34] the cloud decreases the performance of applications to around 73% compared to similar hardware without virtualisation. Ou et al. [31] studies the variations in cloud performance. They note that the performance of instances with the same type can vary depending on the physical hardware. Callegati et al. [12] analyze the network performance of virtual machines in OpenStack Havana cloud. Their measurements indicate that OpenStack negatively affects the network performance of applications.

2.7.3 Virtual Evolved Packet Core

The virtual telecommunication infrastructure has been studied from multiple different viewpoints. Especially the MME has received wide attention. Bosch et al. [8] present virtual telco as means for consolidating hardware. As an example use case for their architecture, they present distributed MME. An et al. [4] presents distributed MME architecture where all MME nodes are identical, but are responsible for serving closest clients. When clients move the state is transfered between MME nodes. Thus the architecture is not hierarchical.

Raivio et al. [33] presents hybrid cloud setup for *Short Message Service Center (SMSC)*. Their solution uses proactive model for predicting service load and launching new virtual machine instances before they are actually needed. The developed prototype system uses OpenNebula as private cloud and Amazon EC2 as public cloud. The paper also presents thorough cost estimation of the hybrid cloud setup and proposes how to optimize the capacity of private cloud to gain maximal cost savings.

Chapter 3

System Design

This chapter details the design of our load balancer software for multiple OpenStack installations. First we introduce the high-level architecture of the system and the interactions between client applications, load balancer and OpenStack clouds. Second, we present the internal architecture of our load balancer.

3.1 High-level Architecture

On a high-level our load balancer acts basically as a proxy between client application and multiple OpenStack installations. Instead of using OpenStack API directly the client application sends RESTfull HTTP requests to load balancer which then figures out to which OpenStack instance is most appropriate to complete the action. The requested action is then completed using the OpenStack API. The application itself is responsible for forwarding the actual user data to the new virtual machines. One possible implementation is to use special front-end application to handle user data as presented in the Figure 3.1. Another possibility would be to direct traffic directly from clients to virtual machines without any central node. This kind of traffic routing could be done for example by leveraging *Domain Name System (DNS)* load balancing mechanism [11].

All the actions in the system are always initiated by the client. The client API was designed to be minimal and as easy to use as possible to make integration to client applications easy. The main operations available to client are creation and deletion of instances. In addition the possibility to list instances is provided. At this point the load balancer does not implement any kind of encryption or authentication in communication with the client application. Thus it's not suitable for use in public networks.

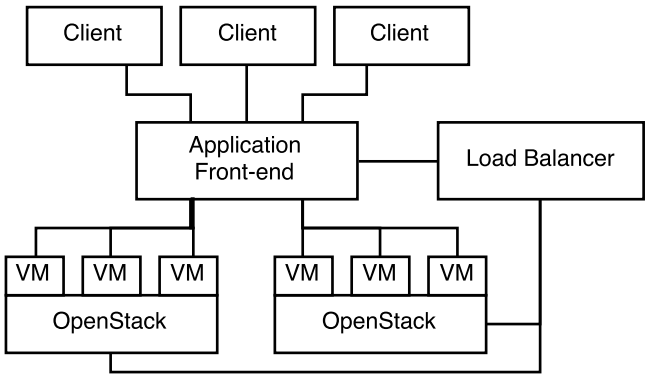

Figure 3.1: The high-level architecture of our load balancing system. The load balancer only communicates with the OpenStack platform and the application needing load balancing service, not with clients or virtual machines.

When client wants to create new instance, it sends `HTTP POST` request to url `/vm` with json body indicating the type of VM to create. The possible types are defined in load balancer configuration and specify all the attributes of new VM instances. This makes integrating the load balancer to applications easy, as the application does not need to know the details of virtual machines in the OpenStack clouds, only names of available types. However, administrator must carefully configure the load balancer and provide the details of virtual machines to create. Administrator needs to specify flavor of the virtual machine, disk image to use and connected networks for each possible virtual machine type.

It is important to note that the actual application data newer goes through the implemented load balancer. The load balancer only manages virtual machines and does not handle the load balancing on application level. This was conscious decision to make the load balancer compatible with wide variety of applications. In some papers this same technique is referred as cloud scheduling [25] or cloud federation[24].

Keeping the virtual machine scheduling separate from application data allows using the load balancer for any application independent of used network protocols and other application properties. Application implementers must only implement a small amount of code to collaborate with the load balancer. Especially keeping in mind the LTE use case implementing the required protocols, such as SCTP on the load balancer would have been significant amount of work and out of the scope of this thesis.

3.2 Internal Architecture

The load balancer is implemented as a Python WSGI application running inside Gunicorn[1] WSGI server. Everything runs inside a single Python process, so complicated inter-process communication is not needed. The Gunicorn server listens for incoming TCP connections, parses the HTTP requests and then delegates the actual request handling to a WSGI application written in Python. The Python application extensively uses Flask[2] web framework and Flask-restful[3] plugin to handle the incoming requests and for formatting responses.

The load balancer application has four main component as presented in Figure 3.2. Each of these components is a separate python module that provides classes and functions for other components in the system. The OpenStack manager and VM scheduler components are built from scratch by the author of this thesis. The Flask-restful API logic is implemented by the author using components provided by the Flask-Restful framework. OpenStack API is a collection of classes related to different OpenStack services and it's provided by OpenStack open-source project. The OpenStack API components have not been modified for this project. Unlike typical web applications, the load balancer does not need any database. All relevant information is queried from OpenStack on-demand. Furthermore, the load balancer is stateless, so it's possible to connect multiple similar load balancers to same clouds. This could be used for example to have separate load balancer with different configuration for each application.

In this implementation the scheduler component is severely limited. The scheduler counts the virtual machines in the related project on all configured OpenStack clouds and uses the counts as the only metric for load balancing situation. The aim is to have the same amount of virtual machines on each cloud. When launching new instances, the cloud with least amount of virtual machines is chosen as the target for new VM. If all clouds have the same amount of VMs, the first configured cloud is selected. Similarly when deleting virtual machines the load balancer selects one machine from the cloud with largest amount of machines for deletion. User application can override this behavior by selecting specific virtual machine for deletion.

When the Gunicorn server is started, it loads the application initialization code. First the initialization code reads configuration file specified by the user. Then it tries to contact the keystone services of each OpenStack

[1] `http://gunicorn.org/`
[2] `http://flask.pocoo.org/`
[3] `https://flask-restful.readthedocs.org`

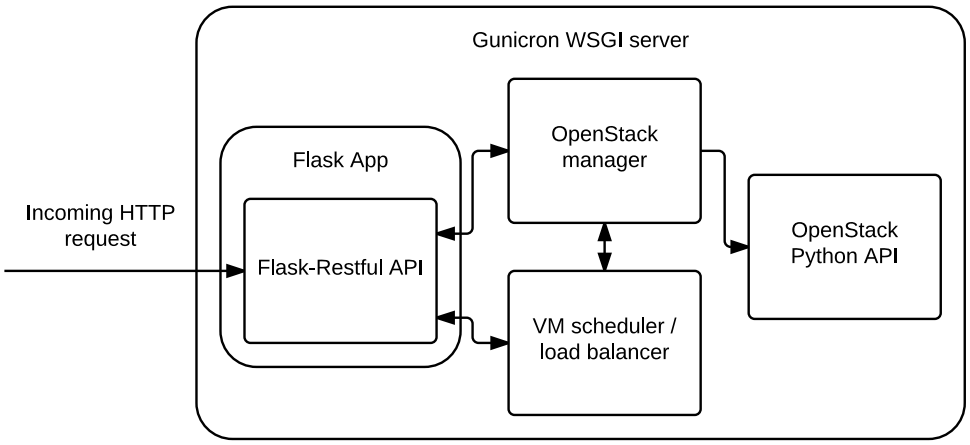

Figure 3.2: The internal architecture of the load balancer and interactions between components.

instance specified in the configuration file. After connecting the OpenStack instances Flask app object is created and flask-restful api object attached to it. Then all available api url endpoint resources are attached to the api and server starts listening for connections.

When server receives new HTTP connection from client, it parses the HTTP headers and calls the Flask app object with the information from request. The Flask app determines which API endpoint was called and then calls the related view function. The view function calls OpenStack manager and load balancer components to accomplish the results required by client. Finally the OpenStack manager uses OpenStack Python API calls to complete the actions. After the selected OpenStack cloud completes the action, the results are returned to OpenStack manager which then returns information to the view function and thus to the client.

In real environments Gunicorn should not be exposed directly to public network, but instead it is recommended to use reverse proxy such as Nginx[4] or HA Proxy[5] in front of Gunicorn. The reverse proxy could for example handle SSL encryption, caching and buffering and load balancing among multiple Gunicorn instances. All these practices improve security and scalability, but in this thesis those are not used to keep the setup as simple as possible.

In total the load balancer consists of 16 Python source files containing about 1100 lines, including comments. This figure does not include the con-

[4] `http://nginx.org/`
[5] `http://www.haproxy.org/`

tents of used libraries. The code is mostly really simple and implementing similar functionality should not be too difficult. The load balancer implementation uses synchronous functions for everything to avoid weird behavior with threading. This makes it easy to understand what happens in the code. However, this means that the load balancer can handle only one HTTP request at a time and that returning the results to client happens only after OpenStack has completed the request. Thus the client application must be prepared to wait for the answer over two seconds in most cases.

The whole implementation phase took about a month in total. However, only couple of days were spent actually coding the load balancer. Most of the time spent implementing the load balancer was consumed by understanding the OpenStack API and the related interactions between OpenStack components. Also designing the system and selecting tools and libraries to use took substantial amount of time.

Chapter 4

Testbed Architecture

This chapter describes the architechture of our testbed. The testbed is an extension to the testbed described by Antti Tolonen in his master's thesis [39]. It is a joint operation between Department of Computer Science and Engineering (CSE) of School of Science and Department of Communications and Networking (COMNET) of School of Electrical Engineering. CSE department is responsible for main computation units while COMNET department is responsible for the LTE side of the project. The OpenStack architecture used in the testbed is based on the reference architecture described in OpenStack installation guide [30].

4.1 Physical Architecture

The physical architecture of the testbed is detailed in Figure 4.1. The system consists of two OpenStack installations, both including four identical blades connected to single 10 Gb ethernet switch. The blade switch is connected to university network with 10 Gb link. The university network provides 10 Gb connection between CSE and COMNET departments. There are no firewalls between the departments and VLAN trunking is supported end-to-end. University network also provides internet connection and connection to shared network storage which is used for storing virtual machine images. Echo server is used as a ssh gateway for connecting to the testbed from internet and for hosting the load balancing application. eNB is a commercial LTE base station which creates the E-UTRAN radio network. However, the eNb is not connected to antenna module, so the network range is limited.

The main part of the testbed is HP BladeSystem c7000 Enclosure G2 chassis which includes eight blade servers, redundant power supplies and a HP FlexFabric 10Gb interconnect. The hardware configuration of each blade

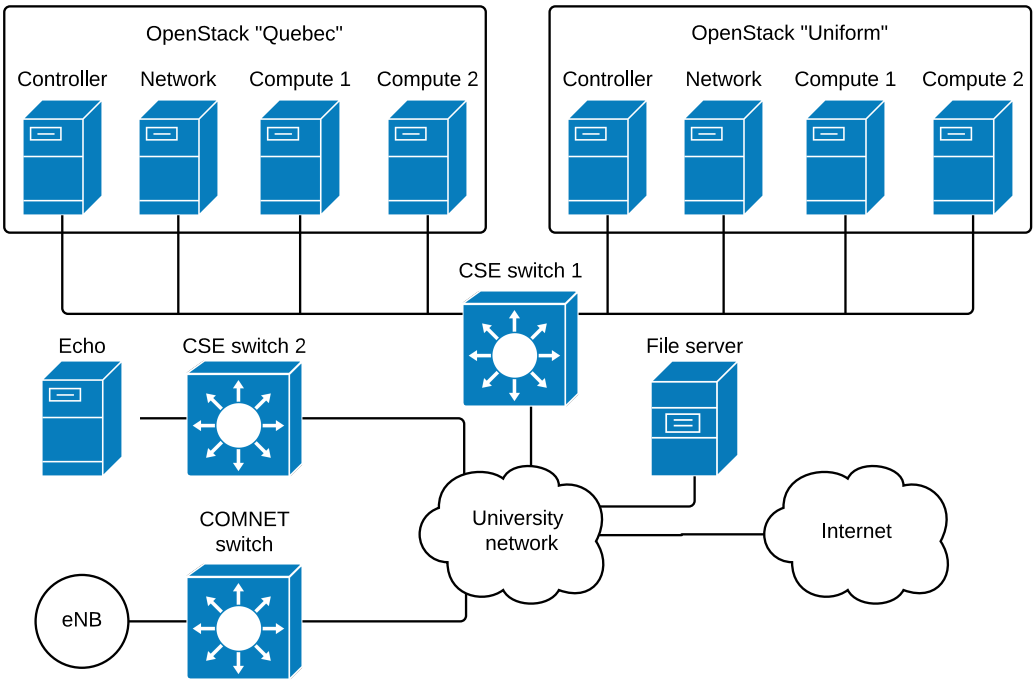

Figure 4.1: The physical architecture of the testbed.

is detailed in Table 4.1. The FlexFabric networking module is divided to two logical ethernet ports, each providing 5 Gb link capacity. Unfortunately in subsection 5.1.2 we notice that this virtual network interface is the limiting factor for vm-to-vm throughput in our testbed. First of the ports is used for OpenStack internal networks and internet connection, the second is used for VLANs connecting the virtual machines and eNB. For more details refer to Section 4.3.

Model	HP ProLiant BL460c Gen8
CPU	2 * Intel Xeon E5-2665 (2.4-3.1 GHz, 64-bit, 8 cores, Hyper-Threading)
RAM	128 GB DDR3-1600
Storage	2 * 150 GB 10K rpm hard disk drives in raid 0
Networking	HP FlexFabric 10Gb 2-port 554FLB Adapter

Table 4.1: OpenStack blade hardware configuration

Echo is a HP ProLiant BL280c G6 blade server with Intel Xeon E5640 CPU (2.67GHz, 64-bit, 4 Cores, Hyper-Threading), 8 Gb of RAM and 1 Gb ethernet card. It has only 1 Gb ethernet link to university network but as it

does not handle any user traffic it is deemed adequate.

The university network is managed by university IT department and acts as a black box to us. It offers 10 Gb end-to-end connectivity between CSE and COMNET departments. The links are shared with other traffic so in theory there might be traffic spikes that affect measurements. However, in practice we have not noticed any delays or bandwidth drops due to other network traffic.

The file server is also managed by university IT department and it is shared with other projects. It offers us 2 Tb of storage over *Network File System (NFS)* version 4 protocol. Again, in theory there might be load spikes due to other projects using same resources, but we have not observed any spikes or drops on the file server performance.

The OpenStack installation guides recommend that there are four physically separated networks: One for control traffic, one for virtual networks (tunnels), one for storage and one for external network (internet) [30]. However, our testbed has only one physical network available, which is shared to service all four purposes simultaneously. In the security and reliability point of view this is not an optimal setup, but for the testbed it is good enough trade-of between usability and cost.

4.2 Software Architecture

All of the physical hosts on the system are running Ubuntu 14.04 operating system with latest patches from the OS vendor. Both OpenStacks are running OpenStack Icehouse and the OpenStack services are distributed to the physical hosts according to the following list. The architecture is based on the reference architecture described in OpenStack installation guide [30].

- Controller node
  - Keystone
  - Horizon
  - Nova API
  - Nova cert
  - Nova consoleauth
  - Nova conductor
  - Nova scheduler
  - Nova novncproxy

  - Neutron server
  - Glance API
  - RabbitMQ
  - MySQL
- Network node
  - Neutron DHCP agent
  - Neutron metadata agent
  - Neutron Open vSwitch agent
  - Neutron L3 agent
- Compute nodes
  - Nova compute
  - Neutron Open vSwitch agent

As the name implies, controller node runs the control components of all OpenStack services. In addition the controller also runs RabbitMQ message broker and MySQL server which are needed by all OpenStack services. The network node runs only Neutron services. Basically it acts as a router, firewall and DHCP server for all the virtual networks in OpenStack. The compute nodes are responsible for hosting the actual virtual machines.

The controller and compute nodes are connected to single NFS mount and all Glance and Nova services are configured to store their data on this file system. This way virtual machine images are accessible from any node in the OpenStack installation and copying those between machines is not needed in any situation. The physical disks in machines are only used for storing the main operating system, caching and log files, not for any user data.

Additionally, all physical hosts are running ssh servers to allow easy remote management and ntp software to keep the clocks synchronized.

The echo server is used for running our custom OpenStack load balancing software. The software could also run on virtual machine on either of the OpenStack installations, but we decided to run it on a separate host to isolate it from the OpenStack installations.

4.3 Network Architecture

Both of the OpenStack installations are configured to use both VLAN tunneled and GRE-tunneled virtual networks. By default all virtual netoworks are configured to use GRE-tunneled networks, as these do not require any special support from network hardware. Administrator can manually configure virtual networks to use VLAN-tagged networks when needed. The VLAN-tagged networks in both OpenStack installations are configured to use the same VLAN tags. Essentially this means that virtual machines in both OpenStack clouds are connected to same LANs and can communicate together without any routing or firewalling in between. As the network setup is quite complex, we have included the commands for configuring SIGMONA project networks in the OpenStack as Appendix A.

There are 5 different virtual networks as shown in the Table 4.2 and in the following list. The only difference in networks between different OpenStack installations is that the `ue_net` is not configured to the Quebec OpenStack instance and that the IP address allocation ranges differ to prevent collisions. Our MME implementation does not support multiple SPGWs, so it is provisioned manually to Uniform, thus the network for user equipments is not needed on the other OpenStack instance. The functions of the virtual networks are as follows:

- The `ext-net` is GRE-tunneled external network with public IP addresses. It is used for allocating IP addresses to virtual routers and floating IPs to virtual machines when needed.
- The `openstack` network is VLAN-tagged virtual network that is used for administrator access to virtual machines. It also provides VMs with internet access through virtual router.
- The `ltemgnt` network is VLAN-tagged virtual network that is used for LTE control plane traffic.
- The `lteuser` network is VLAN-tagged virtual network that is used for LTE user plane traffic.
- The `ue_net` network is GRE-tunneled network that is used for allocating IP addresses for LTE user devices. It also provides the internet connection for user devices through virtual router.

The virtual networks also share the same IP subnet on both installations. Only the DHCP allocation ranges are different to prevent IP address collisions. The IP address allocation pools for the networks are described in Table 4.2.

Network	Subnet	Uniform IP pool	Quebec IP pool
ext-net	130.233.42.0/24	130.233.42.33-62	130.233.42.128-254
openstack	10.1.0.0/24	0.1.0.2-100	10.1.0.102-200
ltemgnt	10.2.0.0/24	10.2.0.3-50	10.2.0.52-100
lteuser	10.3.0.0/24	10.3.0.3-50	10.3.0.52-100
ue_net	10.10.0.0/24 and 10.10.10.0/24	10.10.0.2-254 and 10.10.10.2-254	Not configured

Table 4.2: IP allocation in virtual networks

Only required traffic is allowed to pass to virtual machines. To limit the allowed traffic we use OpenStack security groups, which essentially control iptables[1] firewalls in compute nodes. For each virtual machine we associate two security groups: the default security group and a security group determined by the usage of virtual machine. In total we have configured 5 different security groups with names that describe the usage. The default security group allows SSH connections from anywhere and *Internet Control Message Protocol (ICMP)* messages and it's associated to every machine to allow controlling the VM. For example a node that runs MME front-end software will be associated with security groups `default` and `mme_frontend`. All the security groups and the firewall rules belonging to given security group are listed in Table 4.3.

Security group	Protocol	Remote IP prefix	Destination Ports
default	icmp		
default	tcp	10.1.0.0/24	22
mme_frontend	sctp	10.1.0.0/24	any
mme_frontend	udp	10.2.0.0/24	2123
mme_worker	sctp	10.1.0.0/24	any
mme_worker	udp	10.2.0.0/24	2123
redis	tcp	10.1.0.0/24	7000 - 7002
redis	tcp	10.1.0.0/24	17000 - 17002
spgw	udp	10.2.0.0/24	2123
spgw	udp	10.3.0.0/24	2152

Table 4.3: Security group rules for incoming traffic.

[1] `http://netfilter.org/`

Chapter 5

Evaluation

This chapter presents the different measurements conducted to analyze system performance, observed measurement results and analysis based on the results. For all the measurements using virtual machines we used Ubuntu 14.04 instance with two CPU cores, 2Gb of RAM and 10Gb hard-disk space.

5.1 OpenStack Testbed Network Performance

The network performance of virtual machines significantly affects the performance of deployed virtual network functions. When using automatic load balancing, the system should have information of the traffic characteristics and requirements set by the application. Load balancer should then schedule the new virtual machine to proper location to offer acceptable performance for application. For example if application requires high throughput to some other application, in optimal case these would be scheduled in same physical host.

In this section we analyze the network performance of virtual machines running on OpenStack using synthetic and repeatable test procedures in 5 different test cases. Furthermore, we try to identify the reasons why we get these results. For each test case we measure machine-to-machine latency and TCP throughput.

In the system there are 5 different cases for virtual machine location with respect to each other. The test cases are ordered based on the number of physical and virtual networking components involved in transferring the packets. We expect that the throughput will be highest in the first test case on lowest on the last test case. Similarly, we expect that latencies will be higher with later tests. The test cases are as follows:

1. Physical machine to physical machine without virtualisation. This test

case is included to set the baseline and to measure how our environment performs without the overhead added by virtualisation. In this test case packets travel only through the Linux kernel in both hosts and the physical network in between.

2. Virtual machines running on single OpenStack instance on same physical virtualisation host. In this test case the packets travel from virtual machine through linux bridge to the second virtual machine. Specifically this is only part of the path depicted in Figure 2.3.

3. Virtual machines running on single OpenStack instance but different virtualisation hosts, VLAN network. In this case network packets must travel twice through the path depicted with blue and green colors in Figure 2.3

4. Virtual machines running on different OpenStack instances, connected through shared VLAN network. This implies that the virtual machines are also on different virtualisation hosts. The network path packets must travel is similar to previous test case.

5. Virtual machines running on different OpenStack installations as in the fourth test case. OpenStack internal communication using VLAN tunnels and external communication using Floating IP. In this test case packets must travel through project network, vlan network and external network as depicted in Figure 2.4 in both OpenStack clouds.

5.1.1 Network Latency

The network latency between virtual machines was measured using standard Linux ping application. The application sends 10 ICMP echo request messages and measures time between sent messages and received echo reply messages. The table 5.1 Presents the minimum, maximum and average *Round Trip Times (RTT)* for all the different test cases.

In all five test cases the complexity of the system increases compared to the previous test case. As can be seen from the table 5.1 this increasing complexity leads to increased network latency. Furthermore, the variations in round trip time increase with the added complexity.

The first test case shows that the average round trip time between the physical host machines is about 0.2ms which is very good result and sets the baseline of our underlying network hardware performance. In practice improving this value is not possible without significant hardware investments and specialized network stack.

	RTT (ms)		
Case	min	avg	max
1.	0.126	0.208	0.291
2.	0.410	0.446	0.504
3.	0.693	0.754	0.819
4.	0.624	0.743	0.813
5.	1.051	1.239	1.401

Table 5.1: Network round trip time between virtual machines in different test cases.

The second test case indicates that adding virtualisation layer at least doubles the latencies, even when the packets don't need to travel through physical network. In this case only few virtual network components must be travelled through so added latency is small.

In the third and fourth test case the latencies are nearly identical. Even though the virtualisation hosts are part of different OpenStack clouds in the later case, the network path used is identical because of the shared VLAN network between OpenStacks. Each packet travels the path depicted in Figure 2.3 twice, once in both compute nodes.

In the fifth test case each packet must travel from compute node to network node and then through second network node to second compute hosts. In practice every packet has to travel twice through the path depicted in Figure 2.4.

It is evident that the network stack OpenStack uses adds significant amount of latency to each network packet. However, even in the most complicated scenario the maximum round trip time of 1.4ms is small compared to the latencies in public internet.

5.1.2 Network Throughput

For measuring the network throughput we used iperf[1] in TCP mode and test duration of 30s. The test for each test case was run five times to get reliable measurements. Table 5.2 shows the minimun, average and maximum measured throughput in each of the five test cases.

From the first test case we see that our hardware almost accomplishes the theoretical throughput of 5Gbps provided by the virtual network interfaces in blade servers. The most likely reason for small difference between measured and theoretical performance is the overhead caused by adding eth-

[1] `https://iperf.fr/`

	Throughput (Gbits/sec)		
Case	min	avg	max
1.	4.97	4.97	4.97
2.	12.1	14.06	15.6
3.	4.86	4.88	4.90
4.	4.87	4.88	4.89
5.	2.44	2.52	2.61

Table 5.2: Network throughput between virtual machines in different test cases.

ernet headers to each packet. By using ethernet jumbo frames we might have achieved slightly better throughput.

In the second test case when the virtual machines are running on same physical hardware the achieved throughput is significantly higher than in any other test case. This clearly exposes the fact that the physical network is the limiting factor for throughput. In this test case the limiting factor is the throughput of Linux network stack. We also see that the throughput varies between 12.1 and 15.6 MBps, which is quite significant. Any extra work that the hardware or kernel must handle during the transmission can affect the throughput significantly.

Again the third and fourth test case achieve nearly identical performance for the same reason as with the latency measurements. However, regarding throughput the virtualisation does not cause significant decrease.

In fifth test case the throughput drops to half compared to test cases 1, 3 and 4. There are multiple possible explanations for this, but most likely the reason is that the network node in both OpenStack clouds must simultaneously receive and send the same data through same physical interface. Other possible reason is the Network Address translation conducted by both network nodes. The NATing is expensive operation as the IP headers in every packet must be modified and a lookup table updated or queried to get the correct destination addresses. Furthermore, involving four physical hosts instead of two physical hosts adds significant amount of complexity to the route packets must travel.

5.2 MME Load Balancing

To demonstrate that the prototype multi-cloud load balancer works with distributed MME we used eNB emulator to constantly send one attach request per second to the MME. We then set the internal load balancing so that

MME would trigger load balancing after few requests. We captured all IP packets on the MME front-end host and draw Figure 5.1 which shows the timeline of communications. The components are deployed in a fashion which corresponds with the components in Figure 3.1. In this case the eNB emulator acts as a client, MME front-end is the single application in the center and the MMW worker nodes are running on the virtual machines on top of the OpenStack installations. The state database does not map directly to any of the elements in the figure even though it's running on virtual machines in the same OpenStack installation.

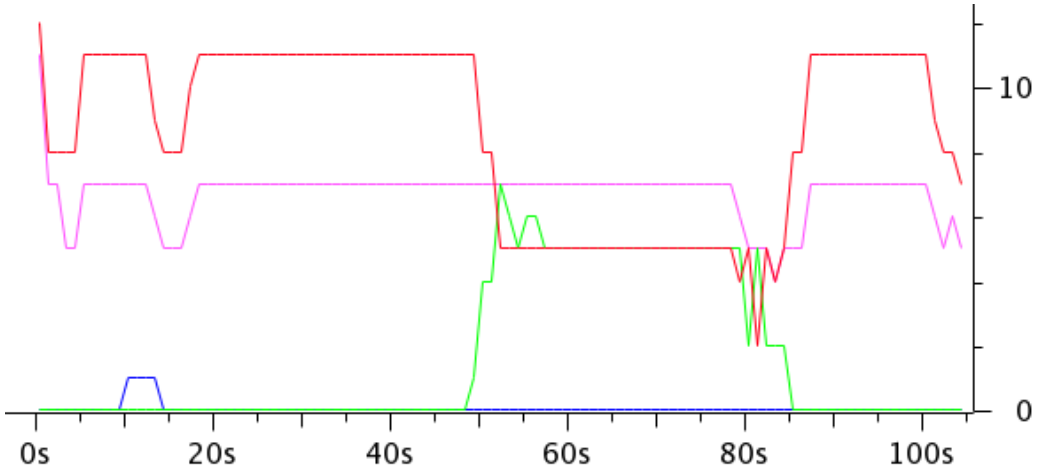

Figure 5.1: IP packets/s between MME front-end and other nodes during automatic load balancing test. RED: worker #1, GREEN: worker #2, BLUE: Load balancer, PINK: eNB emulator

From the figure we can see that in the beginning MME front-end sends all the traffic to worker node #1. Approximately after 10 seconds the front-end decides to make a load balancing request and contacts the load balancer. At 49 seconds the worker node #2 comes online and takes half of the load from node #1. We can see that while the node #2 registers to front-end, couple of more packets are exchanged than during normal operation. At approximately 80 seconds the front-end decides that second node is not needed and stops sending traffic to it. However, it does not send command to shut down the instance yet.

Interestingly there are some drops in the traffic between front-end and eNB emulator. We expected that the traffic should be constant, but clearly this is not the case. The fluctuations in the traffic are analyzed more thoroughly in section 5.3.

The experiment demonstrates that the load balancer is able to launch new instance when required and that it can then connect to the system automatically. In this case the startup of new instance took approximately 39 seconds. The delay depends on the size of the VM and the load of virtualisation platform. If the load balancing decision is done based on actual load, this delay might be too long and the service might experience slowness during this period. That's why methods for predicting the load are needed and decisions must be done using predicted values.

5.3 MME Performance on Multiple Clouds

To validate the performance of our multi-cloud setup for the network functions virtualisation use case, we deployed the distributed MME described in Subsection 2.5.1 on the testbed. The test cases are designed to measure the performance of MME in all the different cases that could happen when using the load balancer. The load balancer is not enabled during these tests, as it does not allow manual control of the virtual machine location. On the contrary, worker nodes are started manually the achieve desired location for them.

Based on the results of the testbed performance measurements, we decided to measure MME performance in three different cases. Each of the test cases correspond roughly to the test cases 2-4 described in Section 5.1. The three test cases coorespond with the test cases 2-4 from Section 6.1. Test case 1 is not relevant here, as it does not use cloud technologies or virtualization. Similarly test case 5 was lef out as floating IPs are not meant for application internal communication, thus they should not be used in our use case.

In all three test cases three Redis nodes are running on first OpenStack and distributed to both available compute nodes. The test cases are as follows:

1. MME front-end and worker running in single OpenStack in same compute node.

2. MME front-end and worker running in single OpenStack but different compute node.

3. MME front-end running on first OpenStack and worker running on second OpenStack.

We used eNodeB emulator developer by Vicent Ferrer Guasch [19] to send one LTE attach request per second for 100 seconds to the distributed MME.

We captured all the IP packets on the MME front-end host with tcpdump[2] and then calculated the delay between receiving attach request packet and sending the last packet indicating completion of the attach procedure. The same test was conducted separately for each of the three test cases and the results are presented in Figure 5.2. In each of the test cases we had 1-2 outliers in the range of 2000ms (2s) which is two orders of magnitude higher than the typical results. These data points were completely omitted in the following analysis.

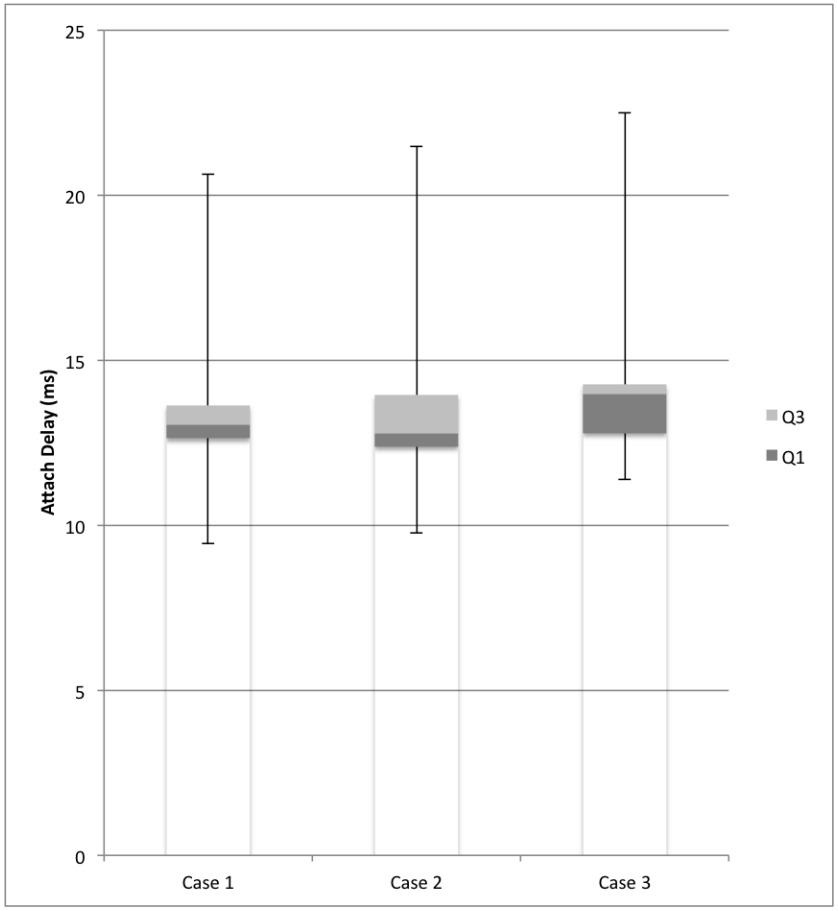

Figure 5.2: The attach delay of clients in each of the test cases.

The attach delay has significant variation in each of the three test cases, the difference between fastest and slowest attach procedure being about 10ms in each of the test cases. In the first and second test case median attach delay

[2] `http://www.tcpdump.org/`

is roughly 13ms and in the third test case 14ms. In 50% of attaches the attach happens within 3ms of the median attach delay, which can be considered as a good result. Furthermore, even the maximum measured attach delay of 23ms is 27ms below the limit of 50ms allowed by LTE specification [3].

Compared to the resutls achieved in [19] and [39], the attach proceudere using distributed MME is slower. We achieved average attach delay of 13ms, while the other two authors achieved MME attach delay of 5,3ms. The incresed delay is probably mostly caused by the incresed amount of signaling traffic between the front-end and worker nodes. If the other parts of vEPC are fast enough, our solution can still achieve the required performance. However, the measurements in [19, 39] indicate that the most time-consuming part of the full attach procedure is caused be the radio network. Furthermore, we only tested the system with only one simultaneous ongoing attach procedure as we did not have full eNB emulator available for testing. The real load with multiple ongoing procedures might affect the results significantly.

Chapter 6

Discussion

This chapter presents discussion of our testbed, the developed load balancer and the aplicability of these technologies to LTE core network use case. Furthermore, we anlyze future work that would be needed to widely implement this kind of system.

6.1 Testbed Performance

The networking in the deployed testbed performed surprisingly well despite of reasonably cheap hardware. With the 2.5Gbps VM-to-internet bandwidth it would be able to relay traffic for approximately 20-25 mobile LTE devices all transferring data at the 100Mbps peak rate required by the LTE-specification [3]. Furthermore, improving the bandwidth should be reasonably cheap by adding multiple network adapters to each host in OpenStack. Also the additional latency caused by OpenStack networking is low enough to support wide variety of network functions virtualisation use cases. Only applications requiring extremely low latencies of less than 5ms might be affected by the extra latency.

The network performance of our testbed is a little unrealistic, as all the hardware is deployed really close to each other. In real-world use there would usually be a longer distance between different components and some other load on the network as well. Especially the cloud-to-cloud bandwidth would definitely be slower, as well as the latency would be higher.

Furthermore, we were able to use VLANs to connect the two cloud installations, which is not a realistic requirement. VLANs are usually available only in networks that are controlled by single operator and implemented using ethernet technology. With hybrid clouds other networking technologies are needed to provide similar connectivity. One possibility would be using

site-to-site VPN connection. For example IPSec [22] tunneling would allow secure and high-performance tunneling of arbitrary IP-based network traffic between multiple clouds.

The only underperforming part in the testbed was the network storage system. Launching new virtual machines took quite a long time, which was caused by the low bandwidth of the storage system. Moreover, all disk operations took significant amount of time, which might have added delays especially in the distributed MME test case. The redis cluster used as a state database needs to write to disk periodically, and due to slow disk performance the writes might delay other operations. Some of these problems might have been avoided if we used the local storage in compute nodes for virtual machine disks. However, we only had 150Gb of storage available in each host, which prevented us from storing virtual machine disk images to local storage. Furthermore, using local storage for virtual machines would decrease the performance of live migrations as data would be copied from host to another. Similarly the resiliency of the system would suffer as a host failure would destroy all VM instances residing on that host.

6.2 Multi-cloud Load Balancing

Our implementation for the multi-cloud load balancer is a simple prototype which provided only the necessary operations for testing the system. Even with the simple implementation we were able to test the automatic scaling functionality built into the distributed MME. The load balancer was able to launch and terminate instances on request from the most appropriate cloud. However, more sophisticated algorithm for load balancing would be needed.

In real use this kind of application-independent load balancing and scheduling is not feasible solutions. Every application has specific performance metrics that are optimal for determining the optimal location for new virtual machines. Furthermore, determining which virtual machine should be shut down when load decreases could be better optimized based on internal performance metrics. In the current solution the load balancer does not handle any application traffic. Thus separate load balancing system is required for application data. In the MME use case the front-end performed the load balancing function and distributed traffic among connected worker nodes.

There are at least two different possible scenarios for more optimal load balancing with multiple clouds. First is to provide the load balancer as highly controllable microservice. In this case the load balancer would still be independent component that connects to target clouds and provides API for the client application. However, the application should provide meaningful

performance metrics and decision rules for the load balancer, so that the balancer can do good enough load balancing decisions autonomically. The load balancer should have some kind of plugin architecture which allows it to be extended for the needs of specific use cases. Even some kind of support for modules for relaying application traffic would be useful. For example common protocols such as HTTP would benefit from this kind of setup.

The second option would be to provide the load balancer as a library which would be integrated into the application itself. The library would provide the functions for querying cloud status, launching new VM to specific cloud and terminating specific cloud. Then the application should implement the logic for determining the optimal location for new VM, deciding when to launch or destroy a VM and which VM do destroy. After the decision the application would call the functions provided by the library to do the required tasks. The tight integration would allow the application to control the clouds better. Furthermore, the traffic relaying and load balancing decisions would be made by a single system. Only the core logic would be implemented by the library, rest would be freely customized by application developer. However, this solution requires more work from application developer.

6.3 Distributed MME on Multiple Clouds

The current implementation of distributed MME is mainly constrained by the context retrieval latency [32]. As the whole connection state is stored to the database and workers are stateless, every operation requires querying the database. Effectively this requires that the latency between workers and database must be small. Furthermore, the database load can increase query times, so optimizing database performance is important as well.

In the current setup the delay caused by the connection between front-end and the worker nodes was not as critical as the database connection. Currently the front-end only relays SCTP connections from eNB to the workers, thus multiple round-trip times is needed for setting up the connection. If the network delay increases, the connection setup phase might affect the delay of LTE procedures significantly. Thus, using same connection between front-end and workers for multiple sessions and minimizing the amount of transferred data might improve performance in network-constrained setups.

In our testbed the connection between clouds performed extremely well, so running the workers and database on different clouds performed nearly as well as with everything running on single cloud. However, as mentioned before the connection between clouds would be significantly slower in real situations.

6.4 Future Work

All the solutions studied in this thesis are just proof of concepts and significant amount of work would be required to make these techniques work in real networks. This section presents our ideas of most compelling future work.

Regarding the testbed we have identified two important items for future research. First of all the clouds should be installed in different buildings with slower network connecting them to imitate real-world performance more closely. Furthermore, employing more widely supported networking protocols to connect the clouds should be done. This change would as well improve the test results to predict real-world performance more precisely.

The second interesting research direction for the testbed would be to use Docker[1] containers instead of virtual machines for running the network applications. Docker containers are a lightweight alternative for virtual machines. Instead of virtualizing the whole operating system, Docker containers share the same operating system kernel but the libraries and applications are different for each container. As the abstraction layer with Docker is much thinner than with virtualisation, the containers should start considerably faster than with virtual machines. Furthermore, distributing Docker images is simpler than distributing other types of applications. However, Docker containers are not as independent as virtual machines and do not provide as much security.

Replacing virtual machines with Docker containers requires significant amount of work. First of all, it is not certain that the LTE components and their networking is supported inside Docker. Second, the support for Docker in OpenStack might not be mature enough for our use case. Third, the rapidly evolving platform might introduce unexpected challenges. But if everything works, Docker might be the future of application virtualisation.

For the load balancer there are few issues that need more thorough research. First research goal would be addressing the design concerns described in Section 6.2 and determine the most optimal way of implementing the whole load balancing system. The current design where the load balancer is just a tiny wrapper around OpenStack APIs does not provide good enough experience for application developers.

Second, the conditions that trigger load balancing should be considered. Virtual machine resource utilization such as CPU capacity, memory capacity, network performance and disk performance are all figures that should be monitored and taken into consideration in load balancing. Furthermore, application specific metrics might be more useful than generic metrics. In addition to reacting to changes in application load, building models of the

[1] `http://docker.com/`

traffic patterns and scaling the service in advance might result in improved user experience.

Third, the placement algorithm for new virtual machines should be studied. As evident from Section 5.1, the placement has huge impact on network performance between virtual machines. Furthermore, the network performance for end-users around the world is significantly affected by the physical location of virtual machines. Also the computational performance of virtual machines might be affected by the placement as the variation in performance might be significant depending on the load of host machine and the used hardware [31].

Chapter 7

Conclusions

In this thesis we have build a testbed consisting of two independent OpenStack clouds for testing network applications. We have implemented a simple prototype load balancing API that is capable of communicating with multiple OpenStack clouds and scheduling resources across these clouds. We have tested the load balancer on the deployed platform and analyzed the network performance of the platform. Finally, we tested the load balancer and our testbed as a NFV platform by running a distributed MME on the testbed.

Our research indicates that OpenStack can be used as a NFV platform as the networking overhead of the platform is not too large in most cases. Furthermore, OpenStack network performance should scale decently if more powerful hardware is used. The network performance in setups involving multiple clouds is probably constrained by the network connection between clouds instead of the network performance of OpenStack clouds.

Our load balancer implementation shows that it is possible to use OpenStack for building hybrid clouds and automatically balance the load between these clouds. With modest amount of code we were able to implement system that communicates with multiple OpenStack clouds and is easy to use for application developers. However, the OpenStack Python API used for this load balancer was difficult to use and barely documented, so real integrations should use the OpenStack HTTP REST API instead. Moreover, integrating the load balancer to application should be considered more thoroughly as discussed before.

The tests done with distributed MME indicate that it indeed would be possible to deploy similar MME on multiple clouds and meet the LTE specification requirements. Both the throughput and latency requirements of LTE core network can be met on OpenStack clouds at least in situations when the cloud is not overloaded. However, significant amount of work is required to optimize the performance of the distributed MME for real-world use.

Bibliography

[1] Open vSwitch quantum plugin documentation. http://openvswitch.org/openstack/documentation/. [Online; Accessed 21.10.2015].

[2] ETSI 3rd Generation Partnership Project (3GPP). ETSI TS 123 002 V12.7.0: LTE; Network architecture. Techincal specification, European Telecommunications Standards Institute ETSI, July 2015.

[3] 3rd Generation Partnership Project (3GPP). 3GPP TR 36.913 V12.0.0: Requirements for further advancements for Evolved Universal Terrestrial Radio Access (E-UTRA) (LTE-Advanced). Techincal specification, 3rd Generation Partnership Project (3GPP), September 2014.

[4] Xueli An, Fabio Pianese, Indra Widjaja, and Utku Günay Acer. Dmme: A distributed lte mobility management entity. *Bell Lab. Tech. J.*, 17(2):97–120, September 2012. ISSN 1089-7089. doi: 10.1002/bltj.21547. URL http://dx.doi.org/10.1002/bltj.21547.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010. ISSN 0001-0782. doi: 10.1145/1721654.1721672. URL http://doi.acm.org/10.1145/1721654.1721672.

[6] Leander Beernaert, Miguel Matos, Ricardo Vilaça, and Rui Oliveira. Automatic elasticity in openstack. In *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, SDMCMM '12, pages 2:1–2:6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1615-6. doi: 10.1145/2405186.2405188. URL http://doi.acm.org/10.1145/2405186.2405188.

[7] Tim Berners-Lee, Roy Fielding, and H Frystyk. RFC 1945: Hypertext Transfer Protocol— HTTP/1.0. RFC 1945, RFC Editor, May 1996. URL http://www.rfc-editor.org/rfc/rfc1945.txt.

[8] P. Bosch, A. Duminuco, F. Pianese, and T.L. Wood. Telco clouds and virtual telco: Consolidation, convergence, and beyond. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pages 982–988, May 2011. doi: 10.1109/INM.2011.5990511.

[9] T. Bray. RFC 7159: The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, RFC Editor, March 2014. URL `http://www.rfc-editor.org/rfc/rfc7159.txt`.

[10] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0. Recommendation REC-xml-19980210, The World Wide Web Consortium (W3C), February 1998. URL `http://www.w3.org/TR/1998/REC-xml-19980210`.

[11] T. Brisco. RFC 1794: DNS support for load balancing. RFC 1794, RFC Editor, April 1995. URL `https://tools.ietf.org/html/rfc1794`.

[12] F. Callegati, W. Cerroni, C. Contoli, and G. Santandrea. Performance of network virtualization in cloud computing infrastructures: The openstack case. In *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pages 132–137, October 2014. doi: 10.1109/CloudNet.2014.6968981.

[13] A. Celesti, F. Tusa, M. Villari, and A. Puliafito. How to enhance cloud architectures to enable cross-federation. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 337–345, July 2010. doi: 10.1109/CLOUD.2010.46.

[14] Margaret Chiosi, Don Clarke, Peter Willis, Andy Reid, James Feger, Michael Bugenhagen, Waqar Khan, Michael Fargano, Chunfeng Cui, Hui Deng, Javier Benitez, Uwe Michel, Herbert Damker, Kenichi Ogaki, Tetsuro Matsuzaki, Masaki Fukui, Katsuhiro Shimano, Dominique Delisle, Quentin Loudier, Christos Kolias, Ivano Guardini, Elena Demaria, Roberto Minerva, Antonio Manzalini, Diego López, Francisco Javier Ramón Salguero, Frank Ruhl, and Prodip Sen. Network functions virtualisation. White paper, European Telecommunications Standards Institute, ETSI, October 2012. URL `https://portal.etsi.org/NFV/NFV_White_Paper.pdf`.

[15] Philip J. Eby. Python Web Server Gateway Interface, WSGI v1.0.1. Python Enhancement Proposal, PEP 3333. `https://www.python.org/dev/peps/pep-3333/`, September 2010. [Online; Accessed 21.05.2015].

[16] ETSI Industry Specification Group (ISG) Network Functions Virtualisation (NFV). ETSI GS NFV 001 v1.1.1: Netowrk Functions Virtualisation (NFV); Use Cases. Group spesification, European Telecommunications Standards Institute, ETSI, October 2013.

[17] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[18] Daniel F Garcia, G Rodrigo, Joaquın Entrialgo, Javier Garcia, and Manuel Garcia. Experimental evaluation of horizontal and vertical scalability of cluster-based application servers for transactional workloads. In *8th International Conference on Applied Informatics and Communications (AIC'08)*, pages 29–34, 2008.

[19] Vicent Ferrer Guasch. Lte network virtualization. Master's thesis, Aalto University School of Electrical Engineering, October 2013.

[20] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal. Nfv: state of the art, challenges, and implementation in next generation mobile networks (vepc). *Network, IEEE*, 28(6):18–26, Nov 2014. ISSN 0890-8044. doi: 10.1109/MNET.2014.6963800.

[21] Qingye Jiang. Open source IaaS community analysis, OpenStack vs OpenNebula vs Eucalyptus vs CloudStack, April 2015. URL `http://www.qyjohn.net/?p=3801`. [Online, Accessed 12.09.2015].

[22] S. Kent and K. Seo. RFC 4301: Security architecture for the internet protocol. RFC 4301, RFC Editor, September 2005. URL `https://tools.ietf.org/html/rfc4301`.

[23] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[24] Tobias Kurze, Markus Klems, David Bermbach, Alexander Lenk, Stefan Tai, and Marcel Kunze. Cloud federation. In *Proceedings of the 2nd International Conference on Cloud Computing, GRIDs, and Virtualization (CLOUD COMPUTING 2011)*, 2011.

[25] Wubin Li, J. Tordsson, and E. Elmroth. Modeling for dynamic cloud scheduling via migration of virtual machines. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 163–171, 2011. doi: 10.1109/CloudCom.2011.31.

[26] Open Networking Foundation. Software-defined networking: The new norm for networks. White paper, Open Networking Foundation, 2012.

[27] OpenStack Foundation. OpenStack admin guide. Nova system architecture. `http://docs.openstack.org/admin-guide-cloud/compute_arch.html`, . [Online, Accessed 07.09.2015].

[28] OpenStack Foundation. OpenStack Glance basic architecture. `http://docs.openstack.org/developer/glance/architecture.html`, . [Online, Accessed 03.10.2015].

[29] OpenStack Foundation. OpenStack networking guide. scenario: Legacy with Open vSwitch. `http://docs.openstack.org/networking-guide/scenario_legacy_ovs.html`, . [Online, Accessed 07.09.2015].

[30] OpenStack Foundation. OpenStack installation guide for Ubuntu 14.04. `http://docs.openstack.org/kilo/install-guide/install/apt/content/`, . [Online; Accessed: 17.05.2015].

[31] Zhonghong Ou, Hao Zhuang, Jukka K Nurminen, Antti Ylä-Jääski, and Pan Hui. Exploiting hardware heterogeneity within the same instance type of Amazon EC2. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[32] Gopika Premsankar. Design and implementation of a distributed mobility management entity (MME) on OpenStack. Master's thesis, Aalto University School of Science, August 2015.

[33] Yrjo Raivio, Oleksiy Mazhelis, Koushik Annapureddy, Ramasivakarthik Mallavarapu, and Pasi Tyrväinen. Hybrid cloud architecture for short message services. In *The 2nd international conference on cloud computing and services science CLOSER 2012*, pages 489–500, 2012.

[34] Sasko Ristov, Goran Velkoski, Marjan Gusev, and Kiril Kjiroski. Compute and memory intensive web service performance in the cloud. In Smile Markovski and Marjan Gusev, editors, *ICT Innovations 2012*, volume 207 of *Advances in Intelligent Systems and Computing*, pages 215–224. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-37168-4. doi: 10.1007/978-3-642-37169-1_21. URL `http://dx.doi.org/10.1007/978-3-642-37169-1_21`.

[35] Martin Sauter. *From GSM to LTE: An Introduction to Mobile Networks and Mobile Broadband*. John Wiley & Sons, Ltd, 2011.

[36] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, 2012.

[37] B. Sotomayor, Ruben S. Montero, I.M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *Internet Computing, IEEE*, 13(5):14–22, September 2009. ISSN 1089-7801. doi: 10.1109/MIC.2009.119.

[38] Microsoft TechNet. Virtual private networking: An overview. `https://technet.microsoft.com/en-us/library/bb742566.aspx`. [Online; Accessed 24.10.2015].

[39] Antti Tolonen. Dynamic virtualized network functions on an OpenStack cloud. Master's thesis, Aalto University School of Science, September 2014.

[40] David Villegas, Norman Bobroff, Ivan Rodero, Javier Delgado, Yanbin Liu, Aditya Devarakonda, Liana Fong, S. Masoud Sadjadi, and Manish Parashar. Cloud federation in a layered service model. *Journal of Computer and System Sciences*, 78(5):1330 – 1344, 2012. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/j.jcss.2011.12.017. URL `http://www.sciencedirect.com/science/article/pii/S0022000011001620`. {JCSS} Special Issue: Cloud Computing 2011.

[41] Guohui Wang and T.S.E. Ng. The impact of virtualization on network performance of Amazon EC2 data center. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, March 2010. doi: 10.1109/INFCOM.2010.5461931.

[42] Xiaoyu Yang, Bassem Nasser, Mike Surridge, and Stuart Middleton. A business-oriented cloud federation model for real-time applications. *Future Generation Computer Systems*, 28(8):1158 – 1167, 2012. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2012.02.005. URL `http://www.sciencedirect.com/science/article/pii/S0167739X12000386`. Including Special sections SS: Trusting Software Behavior and SS: Economics of Computing Services.

Appendix A

Commands for configuring SIGMONA project in OpenStack

Create tenant and user

This step must be done as admin!

```
keystone tenant-create --name sigmona \
    --description "SIGMONA EPC virtualization project"
keystone user-create --name sigmona-bot --pass 2ylg3e4qJb4aJQ
keystone user-role-add --user sigmona-bot --tenant sigmona --role _member_
```

Create ssh keypair

```
ssh-keygen -C "SIGMONA" -f ~/.ssh/sigmona
nova keypair-add --pub-key ~/.ssh/sigmona sigmona
```

Configure networks

This step must be done as admin!

The networks must be configured as admin, because neutron only allows administrators to use provider extensions required for setting specific VLAN.

```
neutron net-create openstack \
    --tenant-id $(keystone tenant-get sigmona | awk '/id/ { print $4}') \
    --provider:network_type vlan \
    --provider:physical_network physnet2 \
    --provider:segmentation_id 3284
```

```
neutron net-create ltemgnt \
    --tenant-id $(keystone tenant-get sigmona | awk '/id/ { print $4}') \
    --provider:network_type vlan \
    --provider:physical_network physnet2 \
    --provider:segmentation_id 3285

neutron net-create lteuser \
    --tenant-id $(keystone tenant-get sigmona | awk '/id/ { print $4}') \
    --provider:network_type vlan \
    --provider:physical_network physnet2 \
    --provider:segmentation_id 3286
```

Configure subnets and router

```
neutron subnet-create --name openstack_subnet \
    --gateway 10.1.0.1 \
    --allocation-pool start=10.1.0.2,end=10.1.0.100 \
    openstack 10.1.0.0/24

neutron subnet-create --name ltemgnt_subnet \
    --no-gateway \
    --allocation-pool start=10.2.0.2,end=10.2.0.100 \
    ltemgnt 10.2.0.0/24

neutron subnet-create --name lteuser_subnet \
    --no-gateway \
    --allocation-pool start=10.3.0.2,end=10.3.0.100 \
    lteuser 10.3.0.0/24

neutron router-create sigmona_router
neutron router-gateway-set sigmona_router ext-net
neutron router-interface-add sigmona_router openstack_subnet
```

Configure security groups

When configuring security groups we must use ip prefixes instead of security groups as the target for traffic because the security groups have different IDs in both OpenStack instances.

```
# Allow all incoming icmp traffic by default
neutron security-group-rule-create --protocol icmp --direction ingress \
    default
# allow incoming ssh from openstack net
neutron security-group-rule-create --protocol tcp --port-range-min 22 \
    --port-range-max 22 --remote-ip-prefix 10.1.0.0/24 default

neutron security-group-create --description "SSH gateway" \
    ssh_gw
neutron security-group-rule-create --protocol tcp --port-range-min 22 \
    --port-range-max 22 ssh_gw

neutron security-group-create --description "MME_FE" \
    mme_fe
neutron security-group-rule-create --protocol 132 \
    --remote-ip-prefix 10.1.0.0/24 mme_fe
neutron security-group-rule-create --protocol udp --port-range-min 2123 \
    --port-range-max 2123 --remote-ip-prefix 10.2.0.0/24 mme_fe
neutron security-group-rule-create --protocol tcp --port-range-min 22 \
    --port-range-max 22 --remote-ip-prefix 10.1.0.0/24 mme_fe

neutron security-group-create \
    --description "MME worker" mme_worker
neutron security-group-rule-create --protocol 132 \
    --remote-ip-prefix 10.1.0.0/24 mme_worker
neutron security-group-rule-create --protocol udp --port-range-min 2123 \
    --port-range-max 2123 --remote-ip-prefix 10.2.0.0/24 mme_worker
neutron security-group-rule-create --protocol tcp --port-range-min 22 \
    --port-range-max 22 --remote-ip-prefix 10.1.0.0/24 mme_worker

neutron security-group-create \
    --description "Redis servers" redis
neutron security-group-rule-create --protocol tcp --port-range-min 17000 \
    --port-range-max 17002 --remote-ip-prefix 10.1.0.0/24 redis
neutron security-group-rule-create --protocol tcp --port-range-min 7000 \
    --port-range-max 7002 --remote-ip-prefix 10.1.0.0/24 redis
neutron security-group-rule-create --protocol tcp --port-range-min 22 \
    --port-range-max 22 --remote-ip-prefix 10.1.0.0/24 redis

neutron security-group-create \
    --description "S-PGW" spgw
```

```
neutron security-group-rule-create --protocol udp --port-range-min 2123 \
    --port-range-max 2123 --remote-ip-prefix 10.2.0.0/24 spgw
neutron security-group-rule-create --protocol udp --port-range-min 2152 \
    --port-range-max 2152 --remote-ip-prefix 10.3.0.0/24 spgw
neutron security-group-rule-create --protocol tcp --port-range-min 22 \
    --port-range-max 22 --remote-ip-prefix 10.1.0.0/24 spgw
```