TÉCNICO LISBOA

Implementing Network Level High-Availability and Load-Balancing on OpenStack, using SDN and NFV

Filipe Emanuel Lourenço Ramalho Fernandes

Thesis to obtain the Master of Science Degree in

Telecommunications and Informatics Engineering

Supervisor: Prof. Fernando Henrique Côrte-Real Mira da Silva

Examination Committee

Chairperson: Prof. Ricardo Jorge Fernandes Chaves
Supervisor: Prof. Fernando Henrique Côrte-Real Mira da Silva
Member of the Committee: Prof. João Nuno De Oliveira e Silva

November 2019

Acknowledgments

I would like to thank my family for their friendship, encouragement and caring over all these years, for always being there for me and without whom this journey would not be possible.

I would also like to give a very special thank you to my colleagues and friends, Tomás Jacob, Tiago Martins and José Pedro Gomes for their friendship, support and motivation during the development of this thesis, and also during my journey on IST.

I would also like to acknowledge my dissertation supervisor Prof. Fernando Mira da Silva for his insight, expertise and availability that has made this Thesis possible.

I want to thank all the DSI Taguspark team members, for giving great ideas to the work, support and for borrowing the necessary equipment used in this work implementation.

Last but not least, to all my friends, specially my girlfriend that helped me grow as a person and were always there for me during the good and bad times in my life. Thank you.

Abstract

Software Defined Networks (SDN) and Network Functions Virtualization (NFV) offers a new way to design, deploy and manage networking services. SDN relies on separating the control plane from the data plane, in data networks. NFV decouples network functions from proprietary hardware appliances so they can run in software. These network functions are *softwarized* and then consolidated on standard Commercial off-the-shelf (COTS) equipment. This work proposes a solution to implement high-availability and load-balancing as a Virtual Network Function (VNF) between OpenStack regions using SDN concepts and NFV Open Source MANO (OSM). High-Availability is a characteristic of a system that usually as an uptime period higher than normal. Load-Balancing refers to efficiently distribute workload between two or more resources, in this case, two OpenStack regions. As use-case, it is used Instituto Superior Técnico (IST) OpenStack infrastructure to deploy and evaluate the implemented system.

Keywords

Network Functions Virtualization; Software Defined Networks; OpenStack; Open Source MANO; High-Availability; Load-Balancing.

Resumo

As Redes Definidas por Software (RDS) e a Virtualização de Funções de Rede (VFR) oferecem uma nova forma de desenhar e gerir serviços de rede. As RDS consistem na separação do plano de controlo do plano de dados, em redes de dados. A VFR desassocia funções de rede de hardware proprietário para que estas possam ser executadas por software. As respectivas funções de rede são retiradas dos dispositivos físicos e proprietários, passando a ser executadas por software em dispositivos não proprietários e de baixo valor comercial. Este trabalho propõe uma solução para implementar alta disponibilidade e balanceamento de carga entre regiões de OpenStack, recorrendo ao uso de conceitos de RDS e VFR. Considera-se que um sistema é altamente disponível, se os serviços que este fornece continuarem disponíveis durante o máximo de tempo possível, apesar de possíveis falhas de energia, software ou hardware. O balanceamento de carga, consiste em distribuir carga, entre dois ou mais nós, neste caso, pacotes de dados entre duas regiões de OpenStack. Neste trabalho, como caso de teste, é usada a infraestrutura de OpenStack do Instituto Superior Técnico (IST) para implementar e avaliar a solução proposta.

Palavras Chave

Virtualização de Funções de Rede; Redes Definidas por Software; OpenStack; Open Source MANO; Alta-Disponibilidade; Balanceamento de carga.

Contents

List of Figures

List of Tables

Listings

Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
CAPEX	Capital Expenditures
CLI	Command Line Interface
COTS	Commercial off-the-shelf
CPU	Central Processing Unit
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DPI	Deep Packet Inspection
DSI-IST	Direção dos Serviços de Informática do Instituto Superior Técnico
EMS	Element Management Systems
ETSI	European Telecommunications Standards Institute
ForCES	Forwarding and Control Element Separation
GRE	Generic Routing Rncapsulation
GUI	Graphical User Interface
HA	High-Availability
HTTP	HyperText Transfer Protocol
ICT	Information and Communication Technologies
IDS	Intrusion Detection System

IETF Internet Engineering Task Force

IP Internet Protocol

IP Internet Protocol

ISG Industry Specification Group

IST Instituto Superior Técnico

IT Information Technology

LB Load-Balancing

MAC Media Access Control

MANO Management and Orchestration

NAT Network Address Translation

NBI NorthBound Interface

NFV Network Functions Virtualization

NFVI Network Functions Virtualization (NFV) Infrastructure

NFVO NFV Orchestrator

NIC Network Interface Card

NS Network Service

NSD Network Service Descriptor

NTP Network Time Protocol

NVA Network Virtual Appliances

ONAP Open Network Automation Platform

ONF Open Networking Foundation

ONOS Open Network Operating System

OPEX Operational Expenses

OSM Open Source MANO

OVS Open vSwitch

RAM Random Access Memory

REST Representational State Transfer

SBI SouthBound Interface

SDN Software Defined Networks

SLA Service Level Agreement

TCP Transmission Control Protocol

TPR Transparent Packet Redirection

UI User Interface

URL Uniform Resource Locator

VDU Virtual Deployment Unit

VIM Virtualized Infrastructure Manager

VLAN Virtual LAN

VM Virtual Machine

VNF Virtual Network Functions

VNFD Virtual Network Function Descriptor

VNFM Virtual Network Functions (VNF) Manager

vNIC virtual Network Interface Card (NIC)

VPN Virtual Private Network

VXLAN Virtual Extensible LAN

1

Introduction

Contents

Over the years, since the beginning of the Information Technology (IT) era, computer systems have drastically changed and improved. Storage and memory evolved and new concepts like clouds and virtualization arose. But over most of these years data network architecture and management have been somehow stagnant.

The first packet-switching network was established in 1969 and was named ARPANET. Since then, several new concepts and standards were proposed in computer/data networks, but most basic principles remained the same. For example, switches and routers, having a ”dedicated” proprietary black box to perform a specific action. This implies a huge lack of flexibility, because if an organisation deploys its network using hardware from a specific brand, they will most likely have to keep using the same brand products if they want to update and manage the overall network on a consistent homogeneous platform. This happens because all hardware runs closed and proprietary software while communication protocols are well standardized, management and operation software are usually brand specific. (And not programmable).

With Information and Communication Technologies (ICT) evolution, Networks are becoming more and more complex, new services are always being deployed and traditional networks are becoming inflexible and difficult to manage.

On the other hand, open-source communities had a huge impact on IT evolution and modernisation. The main proof of that are Linux based operating systems like Debian, Red Hat and Android.

Over the last few years, open-source started to spread to computer networks, first with Software Defined Networks (SDN) and *OpenFlow* [7], and now with Network Functions Virtualization (NFV) concept. SDN decouples data and control planes, centralising all the control functions on a single network controller. This does not mean the existence of only one node as network controller, but the existence of a centralised cluster with a view of the entire network. This controller is able to define network policies and configure the entire network using simple software implemented functions. For the first time in computer networks history, it is becoming possible to decouple network functions from proprietary hardware (the main goal of NFV), and run it on commodity standard machines (for instance a machine running some Linux distribution). This means that much proprietary hardware may be replaced by Commercial off-the-shelf (COTS) hardware running open-source implementation of traditional network functions.

Apparently, this evolution may seem bad news to network hardware vendors, because they will lose market share. However, if they adapt their business to focus on what they are truly good and specialize in it, computer networks might have a huge leap forward shortly.

This thesis explores the potential of these new network paradigms to build complex, yet flexible, network services.

1.1 Motivation

Instituto Superior Técnico (IST) has an OpenStack infrastructure composed by two regions, *Alameda* and *Taguspark*. OpenStack is a cloud operating system, composed of several services/projects which are explained in section 2.5. On this OpenStack infrastructure, the existing regions are independent. One of the main features of such multi-region architecture is to offer a simple solution for High-Availability (HA) and load-balancing. Whenever an application or service requires high availability, it may be deployed in parallel in two regions. If one of these becomes unavailable, the application or service remains active and operational on the surviving region.

One main challenge of such configuration is to deal with public Internet Protocol (IP) addresses since these are usually distinct in both regions. In other words, if a service that requires HA is published with an IP address of one region, and if this region for some reason becomes unavailable, the service will also become unavailable, since the access IP becomes unreachable. To overcome this limitation, there is the need to dynamically serve requests on both regions exposing the same public IP to external clients.

This work proposes and discusses a solution for this issue using SDN and NFV, using as orchestrator the Open Source MANO (OSM) project.

1.2 Research Goals

The main goal of this work is to implement HA and load-balancing between N OpenStack regions ($N \geq 2$) using NFV and SDN concepts.

To achieve the aforementioned goal, this work required some intermediate steps. Namely:

- Deploy a multi-region OpenStack infrastructure, similar to IST OpenStack infrastructure, for testing purposes;
- Deploy NFV Management and Orchestration (MANO) platform to manage and orchestrate the deployed OpenStack infrastructure;
- Design and deploy Network Service Descriptor (NSD) and Virtual Network Function Descriptor (VNFD) able to solve: HA, load-balancing and automatic fail-over;
- Finally, perform the necessary tests to evaluate the functionality and impact of the proposed solution.

1.3 Organization of the Document

This document is structured as follows: in chapter 2 we review the State of the Art. In chapter 3 it is presented the architecture of the solution. In chapter 4 it is presented the implementation of the architecture, where all the components of the architecture are described and their behaviour is explained. In chapter 5 we describe the tests performed, and respective results, to evaluate the proposed solution. And finally, in chapter 6, the main conclusions of this work are detailed.

2

State of the Art

Contents

Within the telecommunication industry, service provisioning has been based on network operators deploying physical proprietary hardware for each function of a given service. Also, this hardware must be interconnected statically in order to implement a fixed sequence of pre-defined operations. These requirements, along with the need for high quality, availability, stability and strict protocols led to long product cycles, heavy dependence on specialised hardware and extremely low service agility. However, applications today require more diverse and new services with higher data rates and availability.

Since hardware-based appliances rapidly reach end of life, requiring much of the procure-design-integrate-deploy cycle to be repeated with little or no revenue benefit, new approaches are being explored.

In the next sections, we address the main technologies and concepts that these approaches comprise.

2.1 Virtualization

Virtualization creates layer of abstraction on top of hardware, making it flexible. Most common virtualized resources are storage, memory and operating systems. Virtualization enables to abstract the computer hardware and software resources from the operating system.

In this work, the main focus is network virtualization.

In a traditional/legacy networking infrastructure (figure 2.1), a set of physical servers hosts the application set. The communication between them is made via switches, which interconnect servers using their Network Interface Card (NIC). The NIC along with with a networking software stack allows communication among endpoints.

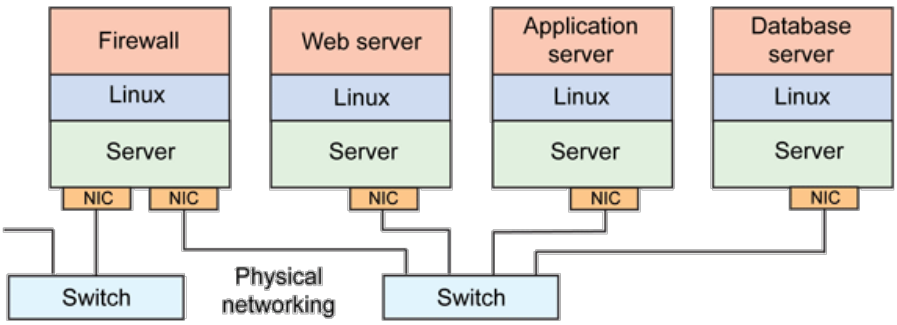

Figure 2.1: Traditional/legacy networking infrastructure [1]

Server consolidation aims to reduce the total number of servers, or server locations that an organisation requires, taking advantage of efficient usage of server resources.

The key innovation behind server consolidation is an abstraction of the physical hardware to allow multiple operating systems and applications to share the same hardware, as displayed in figure 2.2 [1].

This innovation is called Virtual Machine (VM) monitor, or hypervisor. Each VM "sees" the underlying hardware as a complete machine. These VMs can be connected to the physical infrastructure, and other VMs, using one or more virtual NIC (vNIC). Once again, each VM "sees" its vNIC as a physical NIC. This is done by attaching the server's physical NICs to the hypervisor's logical infrastructure.

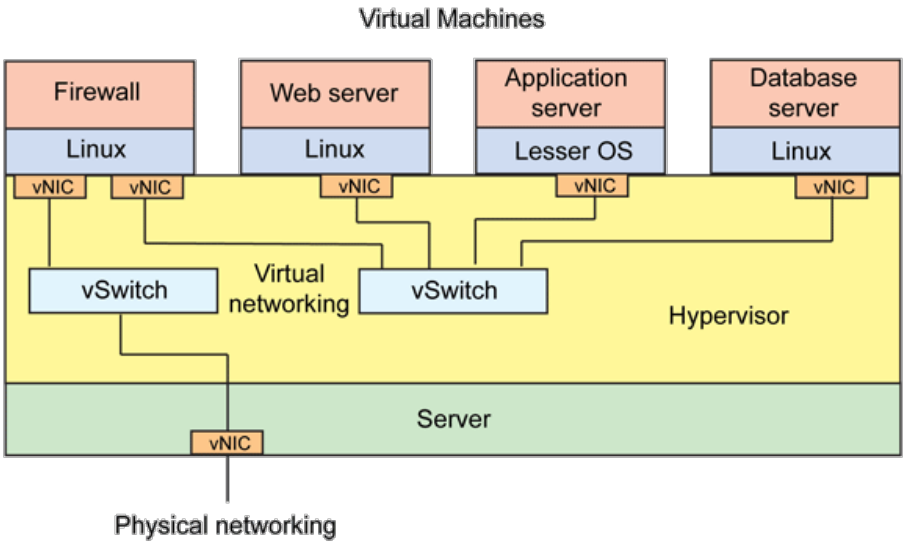

Figure 2.2: Virtualized networking infrastructure [1]

Another key development is Virtual Switching [1], where switching is implemented at software level and limits are related to memory bandwidth instead of network speeds. This allows efficient communication between local VMs and minimizes the overhead of network infrastructure.

Open vSwitch (OVS) is a multilayer virtual switch that supports the leading open source hypervisors like KVM and VirtualBox. It consists of a switch daemon and companion kernel module that manages the flow-based switching. OVS is licensed under the open source Apache 2 license. For this work the major advantage of OVS is its support of latest release of the OpenFlow protocol which will be described in more detail in section 2.2.

As mentioned before, virtualization started with server virtualization, but this was only the beginning. After server virtualization, new concepts appeared as virtual appliances and Network Virtual Appliances (NVA). A virtual appliance is a virtual machine image, pre-configured with the desired software stack to perform some specific function. Virtual appliances are cost-effective, since it is possible to dynamically alter their resources based on their needs.

NVAs are developed for network applications, and may include routers, firewalls, Intrusion Detection System (IDS) and Virtual Private Network (VPN).

Starting with NVA, new approaches to network data architecture and management are possible, such as Virtual Network Functions (VNF) and NFV. These approaches may further be leveraged using SDN. SDN and NFV are further discussed in sections 2.2 and 2.4.

2.2 Software Defined Networks

The concept of SDN appeared intending to separate the control and data planes in data networks.

The main concept of SDN relies on the separation of the data and control planes, centralising all control functions in a single network controller. In SDN, network devices implement only the data plane, where simple packet forwarding is processed. Forwarding rules and actions are programmed on a central network controller [8], which runs on a standard computing server [9].

Since in SDN control and forwarding functions are fully decoupled, network devices can be programmed with high flexibility and dynamically adapted to application requirements, which enables the creation and quick deployment of new types of applications and services.

By decoupling control and forwarding functions, SDN enables to automate provisioning and orchestration which reduces overall management time and the chance of human error, therefore reducing (Operational Expenses (OPEX)) costs. On the other hand, since in theory, it may rely on simplified network nodes that implement only the data plane, it may also limit the need to purchase specialized networking hardware, reducing (Capital Expenditures (CAPEX)) costs.

In 2004 a standard called Forwarding and Control Element Separation (ForCES) was published by Internet Engineering Task Force (IETF). This standard considered various ways to decouple the control and forwarding functions [10]. The first attempts of decoupling these functions failed because the internet community viewed the separation of those as too risky and vendors got concerned that creating a standard Application Programming Interface (API) between control and data planes would increase competition.

Along with the concept of SDN, *OpenFlow* [7, 11] was created by Stanford's computer science department. The first API for *OpenFlow* was created in 2008, later that year appeared an operating system for networks (*NOX*). Over the next two years both SDN and *OpenFlow* got supporters and in 2011 Open Networking Foundation (ONF) was founded to promote these technologies.

SDN-based networks are composed of three planes/layers, Application, Control and Data planes. Between Application and Control planes there is the Northbound Interface and between Application and Data planes, there is the Southbound Interface, as shown in figure 2.3.

The main component of an SDN network is the controller, which is the application that acts as a centralized control point in the SDN network, manages the flow and forwarding rules of the switches/routers, and works at the same time as a broker that manages and configures the network resources according to the requirements of higher-level applications. The controller communicates through two standard interfaces: the SouthBound Interface (SBI) and the NorthBound Interface (NBI).

Southbound API is the controller interface that communicates with network nodes. There are different protocols that implement the communication between the controller and the network nodes, but the most popular and widely used is the *OpenFlow* [7] protocol.

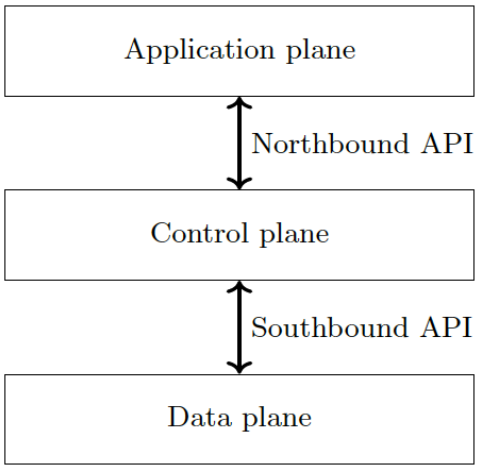

Figure 2.3: SDN node planes

Northbound API is the controller interface that communicates with higher-level applications. There is not a standard protocol defined for the NBI [12–14]. However, it is usually implemented through a REST API [15] or directly on a programming language. While there were some efforts to develop a standard for the NBI protocol, such standard was not yet established. In practice, each SDN controller has its own specification.

2.3 SDN Controllers

In this section we discuss the the main open-source SDN controllers. As explained before, the SDN controller has a global view of the entire network, and provides a central control to the network devices.

Most open-source SDN controllers are composed of multiple moduleswhich can be selectively enabled in order to perform different network tasks.

In this work, we need an open-source SDN controller with the following characteristics: support for OpenFlow 1.3, a module to install and remove flow rules, well documented and widely used by the community. Considering these requirements, we will review in more detail Floodlight and Open Network Operating System (ONOS) [16].

Floodlight

Floodlight is an open-source SDN controller, implemented in Java and Apache-licensed. It has a modular architecture and uses OpenFlow protocol as SBI and a Representational State Transfer (REST) API as NBI. Since it has a modular architecture, it allows developers to easily extend its functionality with new custom modules.

Floodlight is a good solution for this work since it has a module that allows to easily add and remove flow rules on network devices. The Static Entry Pusher API exposes a REST API that enables the

addition of detailed flows with a simple HyperText Transfer Protocol (HTTP) request.

ONOS

ONOS is also an open-source SDN controller and it has very similarities with Floodlight. It is also written in Java and Apache-licensed. It also has a modular architecture and it is very well documented.

Like Floodlight, ONOS uses OpenFlow protocol as SBI and a REST API as NBI.

Over the last years ONOS popularity highly increased and it is currently one of the open-source SDN controllers with most contributors on GitHub platform. This turned ONOS into one of the most complete SDN controllers and also very complex.

For the scope of this work, it is worth to note that ONOS also has a module that enables the addition and deletion of flow rules on network devices.

2.4 Network Functions Virtualization

As seen in the previous section, SDN established new principles regarding network management and network programmability. One of the outcomes of this principle is *network softwarization*, which is based on the possibility to virtualize network functions [17]. VNF are virtualized tasks that used to be performed on proprietary, dedicated hardware. With NFV, these tasks are moved from dedicated hardware devices to virtual machines or containers that may be run on COTS hardware. Examples of VNFs include firewalls, Domain Name System (DNS), Network Address Translation (NAT), VPN gateways, Deep Packet Inspection (DPI) and IDS [18].

In October 2012, at a conference on SDN and *OpenFlow*, a specification group ("Network Functions Virtualisation") that was part of European Telecommunications Standards Institute (ETSI) published a white paper [18] regarding NFV. Since that white paper, the specification group produced several materials including a standard terminology definition and use cases for NFV that act as references for the adoption of Network Virtualization by vendors and operators.

As stated before, one of NFV goals is to decouple network functions from proprietary hardware appliances. It means that applications/functions that typically had to be ran on specific proprietary hardware are *virtualized/softwarized* to run on COTS equipment. These applications are executed and consolidated on standard IT platforms like high-volume servers, switches, and storage which reduces CAPEX and support pay-as-you-grow models, eliminating over-provisioning. NFV also reduces OPEX, since the specialized hardware devices may be replaced with software running in virtual servers, saving space, power and cooling requirements and simplifying the overall management of network services.

NFV provides several other benefits, as it enables shorter development cycles, openness of platforms, scalability and flexibility and allows the use of a single platform for different applications, users and tenants [18–20].

While NFV offers increased management and operational flexibility, it also encompasses several challenges. One of such challenges is the migration of conventional network functions to NFV and the co-existence of NFV with legacy platforms. Furthermore, achieving high performance virtualized networks appliances which are portable between different hardware vendors and different hypervisors can be a complex task. As seen before, one of the NFV benefits is scalability, but this is only possible if all functions can be automated. Moreover, NFV domains must be secure, resilient, and VNFs should be decoupled from any underlying hardware and software [18, 19, 21, 22].

NFV is applicable to any data plane packet processing and control plane function in mobile and fixed networks.

With the aim to develop a unified framework for NFV, the ETSI Industry Specification Group (ISG) developed the ETSI NFV reference framework, presented in figure 2.4. The ETSI architecture comprises three main domains [2]:

1. The first domain are VNF and their corresponding VNF Managers. These represent the communications applications (previously deployed as custom appliance type hardware solutions) now fully virtualized, supporting elasticity, orchestration, and SDN enablement. Closely correlated with the VNFs are also the associated Element Management Systems (EMS) as well as northbound connectivity to OSS/BSS systems for provisioning, accounting and other functions.

2. The second one is NFV Infrastructure (NFVI) which is comprised of the underlying data center hardware (compute, networking and storage) with the associated virtualization layer, abstracting that hardware to a software controlled Cloud and SDN environment. The Virtualized Infrastructure Manager (VIM) is closely correlated to this [23, 24]; i.e., OpenStack control functions.

3. The last domain is NFV Management and Orchestration. This one can be divided into three functional blocks: NFV Orchestrator (NFVO), VNF Manager (VNFM) and VIM.

   The NFVO is responsible for the on-boarding of Network Service (NS), VNF packages and the lifecycle management of the NSs. These packages are described using NSD and VNFD, respectively.

   The VNFM coordinates the configuration and event reporting between NFVI and EMS. It also supervises the lifecycle management of VNF instances.

   The VIM manages and controls resources from the NFVI. OpenStack, described in section 2.5, is a VIM.

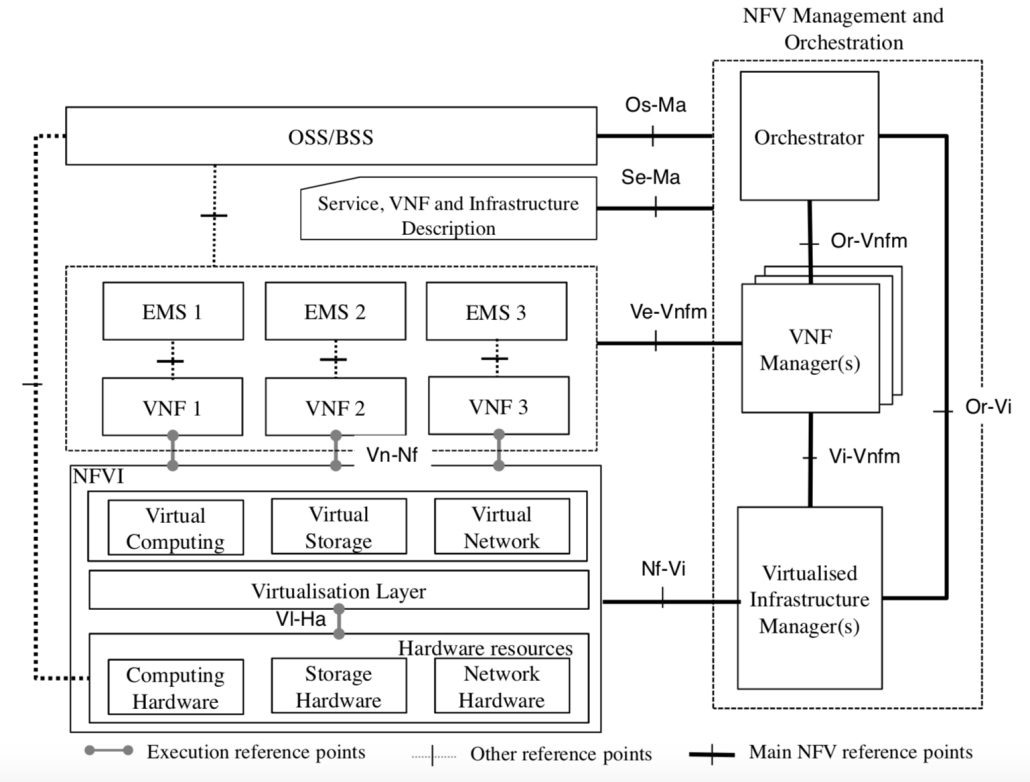

Figure 2.4: NFV Reference Architecture [2]

2.5 OpenStack

OpenStack is a cloud operating system, also known as a Virtualized Infrastructure Manager. It controls large pools of computing, storage, and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface [25].

OpenStack is composed of several projects [26], but the main ones are:

- Ironic (Bare Metal Provisioning Service) - the project that enables scalable, on-demand, self-service access to bare metal machines instead of virtual machines.
- Heat (Orchestration) - the project responsible to orchestrate the infrastructure resources for a cloud application. The orchestration is based on templates, in the form of text files, that can be treated like code.
- Horizon (Dashboard) - it provides a graphical, web-based User Interface (UI) to OpenStack services.
- Nova (Compute) - it enables scalable, on demand, self service access to compute resources and virtual machines.

- Neutron (Networking) - due to the nature of this work, this service is discussed on the following chapter.
- Swift (Object Storage) - the project responsible for object storage. It provides a highly available, distributed and eventually consistent object-store.
- Glance (Image Service) - the image service/project that enables discovering, registering, and retrieving virtual machine images.
- Keystone (Identity Service) - OpenStack service that provides API client authentication.
- Cinder (Block Storage) - it virtualizes the management of block storage devices. It provides, end-users, an API to request and consume block storage resources.

A map with a list of current OpenStack projects [4] is presented in appendix D.

As explained before, OpenStack is composed of several services/projects. Figure 2.5 shows the relationship between them.

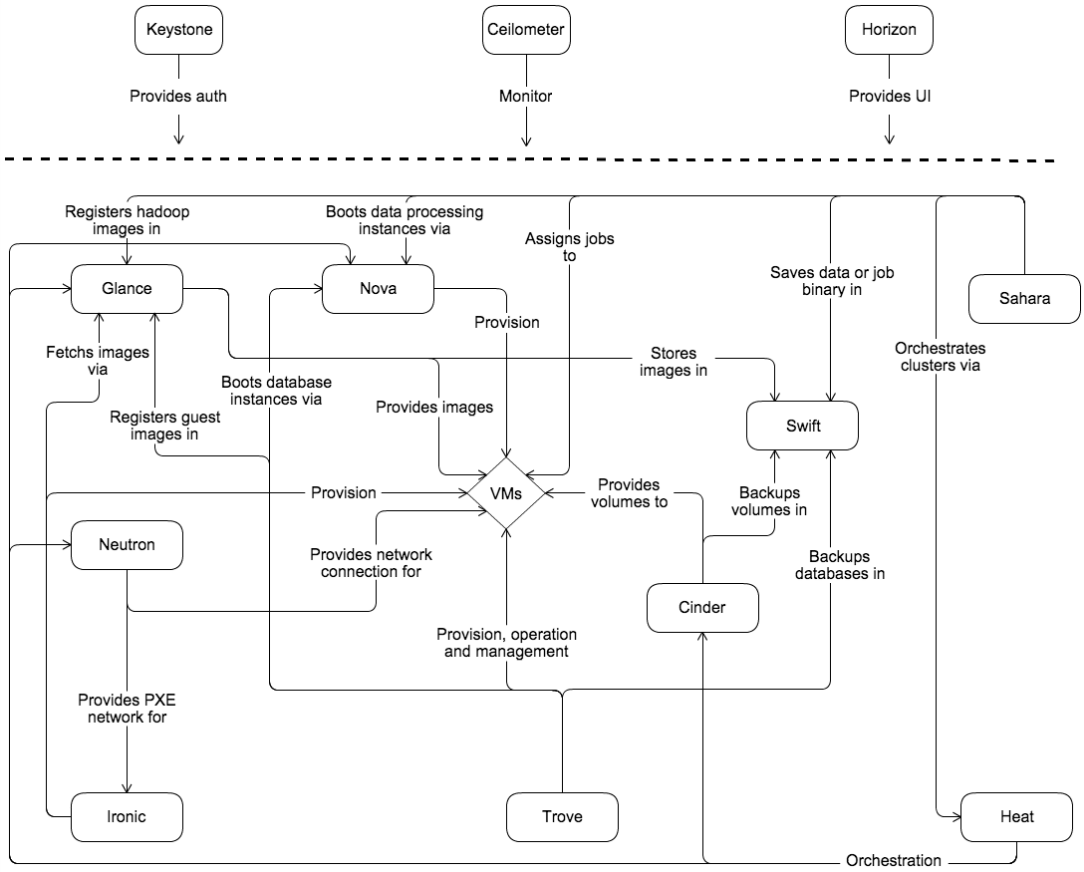

Figure 2.5: OpenStack conceptual architecture [3]

All OpenStack services authenticate through a common identity service. Individual services interact with each other through public APIs, except where privileged administrator commands are necessary.

Regarding OpenStack Services, they are composed of several processes. Each service has at least one API process that listens for API requests, preprocesses them and passes them to other parts of the service. Except for the Identity service (Keystone), the actual work is done by distinct processes.

OpenStack can be accessed via web-based UI (if Horizon is implemented), via Command Line Interface (CLI) or by issuing API requests.

In OpenStack each user can have one or more organisational space, this space is, also, called a project. Each user has a role that defines which actions he can perform. Users, projects and roles are managed independently from each other.

2.6 Neutron

Neutron is the OpenStack Networking service. It provides an API that allows users to set up and define network connectivity and addressing in the OpenStack system. Neutron is the service in charge of the creation and management of virtual networking infrastructure [27], including networks, switches, subnets, and routers for devices managed by Nova (OpenStack compute service).

Neutron consists of the neutron-server, a database, and any number of plug-in agents, which provide other services such as interfacing with native Linux networking mechanisms, external devices, or SDN controllers. It supports several network services namely L3 forwarding, NAT, load balancing and VPN are some examples. Neutron supports public and private IP addresses, and one-to-one NAT, which is used to implement floating IP addresses. One-to-one NAT is a mapping between private and public IP addresses (floating IPs). One of its advantages is that they can be instantly moved between virtual machines.

Networking Service allows the creation of networks, subnets and ports, which other OpenStack services can use. Neutron plugins enable support for different networking equipment and software. These plugins provide flexibility to OpenStack architecture and deployment.

Neutron supports each project having multiple private networks and enables projects to choose their IP addressing scheme.

Neutron enables two types of layer-3 connectivity, provider and self-service networks. Provider networks maps to existing networks managed by OpenStack administrators. Self-service networks are assigned to projects and enable users to manage their own networks without involving administrators, they are entirely virtual and require virtual routers to interact with provider and external networks. Both network types can provide Dynamic Host Configuration Protocol (DHCP) service to instances.

Regarding network isolation and overlay technologies, neutron supports Flat, Virtual LAN (VLAN),

Generic Routing Rncapsulation (GRE) and Virtual Extensible LAN (VXLAN).

Another key feature of OpenStack networking is security groups, which provide a container for virtual firewall rules. These rules control ingress and egress network traffic at the port level. The firewall driver translates security group rules to a configuration for packet filtering technology such as *iptables*.

2.7 High-Availability and Load-Balancing

HA is a system feature which aims to ensure a certain Service Level Agreement (SLA), usually uptime, for a period higher than normal [28]. Each SLA is traditionally measured by how close it is to 100%, a service that is available during 99,9% of time corresponds to a SLA of ”three nines”. On table 2.1 it is presented the maximum time of unavailability for each SLA.

SLA	Downtime/year	Downtime/month	Downtime/week	Downtime/day
”three nines”	8.77 hours	43.83 minutes	10.08 minutes	1.44 minutes
”four nines”	52.60 minutes	4.38 minutes	1.01 minutes	8.64 seconds
”five nines”	5.26 minutes	26.30 seconds	6.05 seconds	864.00 milliseconds
”six nines”	31.56 seconds	2.63 seconds	604.80 milliseconds	86.40 milliseconds
”seven nines”	3.16 seconds	262.98 milliseconds	60.48 milliseconds	8.64 milliseconds

Table 2.1: Time of unavailability per SLA [5]

In order to achieve HA three principles must be met:

- Elimination of single points of failure;
- Reliable crossover, ability to switch from node A to node B without losing data;
- Detection of failures as they occur.

Load-Balancing refers to efficiently distribute workload between two or more resources [29]. This workload usually is network traffic, and these resources can be servers, network links, Central Processing Unit (CPU)s and disk drives.

There are several load-balancing policies, most common are round-robin (there are some variants), least connections and IP hash. Round-robin policy is distributing requests sequentially among a server pool. Least connections policy always distribute requests to the server with the fewest current connections. And IP hash policy uses a hash of the client IP address to determine which server will serve the request.

A simple HA technique is hosting two web servers with one load-balancer splitting traffic between them, and another backup load-balancer on standby. If one web server becomes unavailable all traffic is redirected to the available one. The same occurs with load-balancers, if the primary becomes unavailable or malfunctioning, the replica assumes its command as the primary one.

2.8 NFV Managers and Orchestrators

Regarding NFV MANO there are several frameworks. The most popular and open-source are OpenStack Tacker, Open Network Automation Platform (ONAP) and OSM. On table 2.2 it is presented a comparison between them [6].

As explained in section 2.4, this domain, NFV MANO, is responsible to manage and orchestrate VNFs.

One of the advantages of ONAP and OSM is the support of Juju API as VNFM. It allows to define complex configurations, easily, on the VNFD.

Juju is a technology that simplifies the configuration and scalability of software. It enables day-0, day-1 and day-2 configurations. These configurations concern to the deployment of NSs and VNFs (day-0), the configuration right after deployment (day-1) and configurations that can be triggered manually when desired(day-2).

	OpenStack Tacker	**ONAP**	**OSM**
Status:	Working solution	Difficult to get started	Working solution
Activity:	Limited activity	Huge activity	Large activity
Completeness:	Technically complete but too integrated with VIM	Technically complete but too wide for this use-case	Huge leap forward in releases 4 and 5

Table 2.2: Comparison between different NFV MANO frameworks [6]

OpenStack Tacker

Tacker is an official OpenStack project building a Generic VNF Manager (VNFM) and an NFV Orchestrator (NFVO) to deploy and operate Network Services and Virtual Network Functions (VNFs) on an NFV infrastructure platform like OpenStack. It is based on ETSI MANO Architectural Framework and provides a functional stack to Orchestrate Network Services end-to-end using VNFs [30].

ONAP

ONAP provides a comprehensive platform for real-time, policy-driven orchestration and automation of physical and virtual network functions that will enable software, network, IT and cloud providers and developers to rapidly automate new services and support complete life-cycle management [31, 32].

OSM

OSM is an ETSI-hosted project to develop an open-source NFV Management and Orchestration (MANO) software stack aligned with ETSI NFV, described in section 2.4.

In this moment Tacker seems to be more integrated(at least officially) with OpenStack. On the other hand, OSM had a great development recently with both releases 4 and 5, and has a great number of significant supporters (including operators). Besides that, OSM has an excellent documentation and

seem to has a perfect integration with OpenStack [6]. This fact, combined with the evidence that ONAP is too overwhelming for this work's use-case, and difficult to get started, OSM was the selected NFV MANO for this work.

2.9 Summary

In this chapter we discussed the main concepts and technologies that this work addresses.

SDN and NFV are changing the architectures of traditional data networks. SDN separates data and control planes, and uses a centralised controller that has a global view of the network and is responsible to program its network devices. NFV decouples network functions from proprietary hardware appliances in order to run those functions as software on COTS equipment.

We discussed the main SDN controllers and NFV orchestrators and propose Floodlight and OSM as the more suitable to the solution of this work, which aims to enable access redundancy and load-balancing on OpenStack, whose main applications and components were also discussed in this chapter.

3

Architecture of the proposed solution

Contents

In this chapter, it is described the architecture of the proposed solution. As stated in chapter 1, the main goal of this work is to develop access redundancy and load-balancing, between two or more OpenStack regions, using SDN and NFV concepts. As use-case it is used the current IST OpenStack infrastructure.

On an OpenStack environment, when a VM needs to be accessed from the public network, it is usually assigned a floating IP address. If for some reason, one region becomes temporarily out of service, the services running on VMs from that region will become unavailable. The service only survives if there is a fast handover from the services in the failure region to the other one. There is the need to dynamically configure the network between those regions, to serve requests on the available region and respective VM. This work also proposes to leverage this dynamic network configuration and implement load-balancing between multi-region OpenStack environments.

The core idea of this work is to use NFV MANO to deploy a NS with one VNF capable of detecting when a service is unavailable, dynamically configuring the underlying network of OpenStack and filter/route traffic to the desired host.

In this chapter it is first presented IST infrastructure, then the overall architecture of the proposed solution.

The architecture can be divided in two views, physical and virtual. Physical architecture is composed of all physical nodes supporting the OpenStack infrastructure and proposed solution. Nodes deployed on each OpenStack region belong to the virtual architecture.

3.1 IST Infrastructure

IST OpenStack environment is composed of two regions, one at Alameda Campus (region A) and the other at Taguspark campus (region T). The architecture is presented in figure 3.1.

Both regions of the infrastructure have precisely the same core services and are symmetric. There are two services deployed on remote servers, Horizon and IST Custom Logic. OpenStack services that support IST infrastructure are: Designate, Keystone, Nova, Neutron, Cinder, Glance, Horizon. These OpenStack core services were described in chapters 2.4 and 2.5. IST Custom Logic service, represented in the overall architecture, is not relevant for this work, since it refers to specific IST business logic services and it is independent from OpenStack services.

These core OpenStack services are running over six VMs running on the same hypervisor. This hypervisor is replicated on hardware (*localcontroller1* and *localcontroller2*) with automatic fail-over between them.

The core OpenStack services previously mentioned have several dependencies. There is a module to coordinate operations and exchange messages between core services, a database module to store

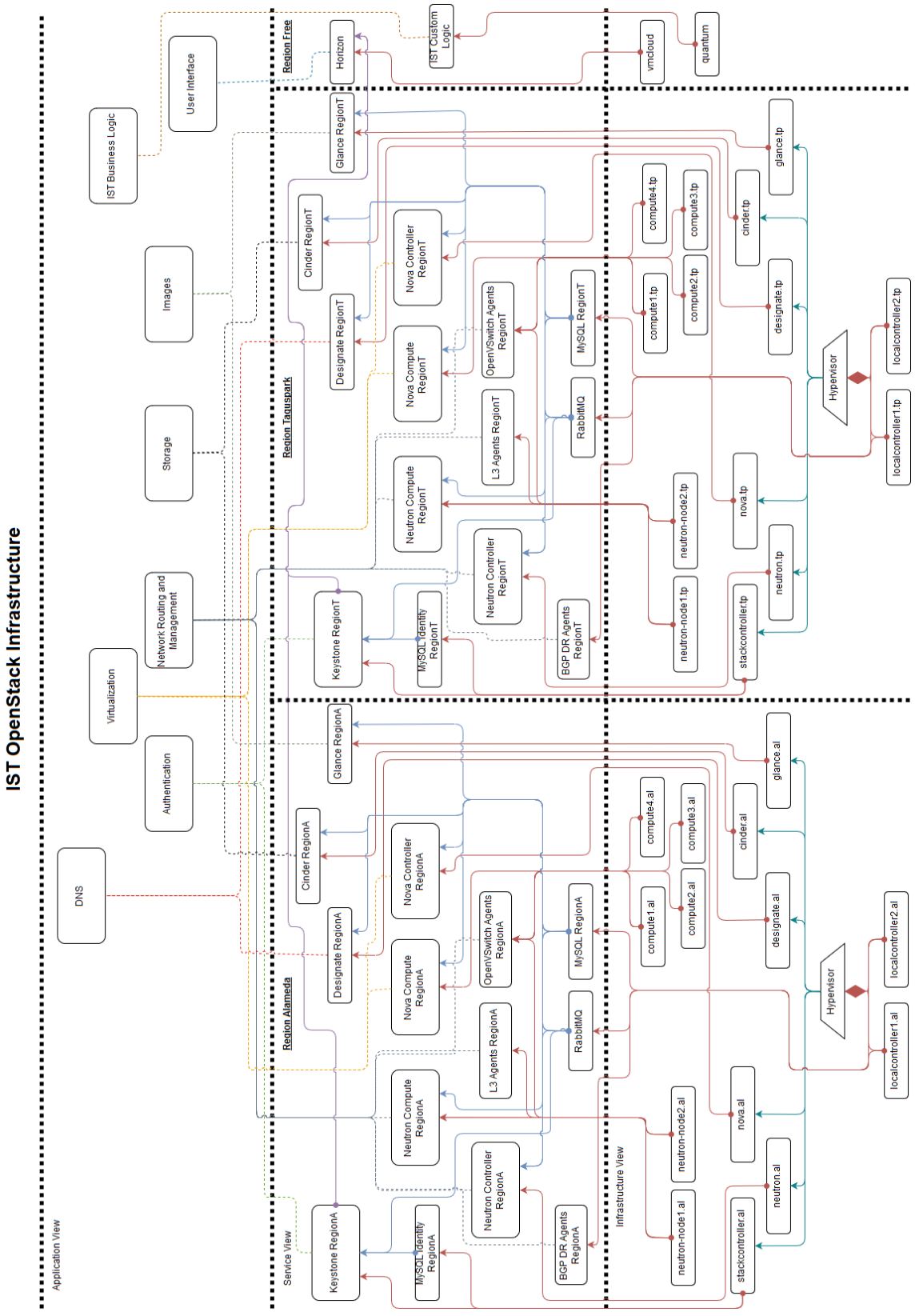

Figure 3.1: IST OpenStack Architecture

persistent data, and additional modules that support Neutron operation as described in chapter 2.5.

Core OpenStack services from region A and T are completely independent, with exception to the authentication service (keystone), that is deployed on a remote server and is used by both regions.

Between regions A and T, currently, there is only one physical link, which implies a single point of failure. Since we aim to implement high-availability between OpenStack regions, some sort of triangulation through a third site is mandatory to have redundant paths available. Although triangulation between regions is not implemented yet, it is foreseen in the near future. In this work, we assume that it is already available. However, note that this redundancy is transparent at the logical level, and therefore will not be mentioned further in this document.

Both OpenStack regions are connected to IST network and have connectivity to/from public network through a cluster of routers named *Gatekeeper IST*. A global view of OpenStack underlying network architecture is presented in figure 3.2. Also, all network devices on the underlying network shall be replicated, with automatic fail-over between them (due to the scope of this work). It means that each network node is actually a cluster of network devices with automatic fail-over. These clusters are represented as a group of three overlaying devices, as illustrated in figure 3.2.

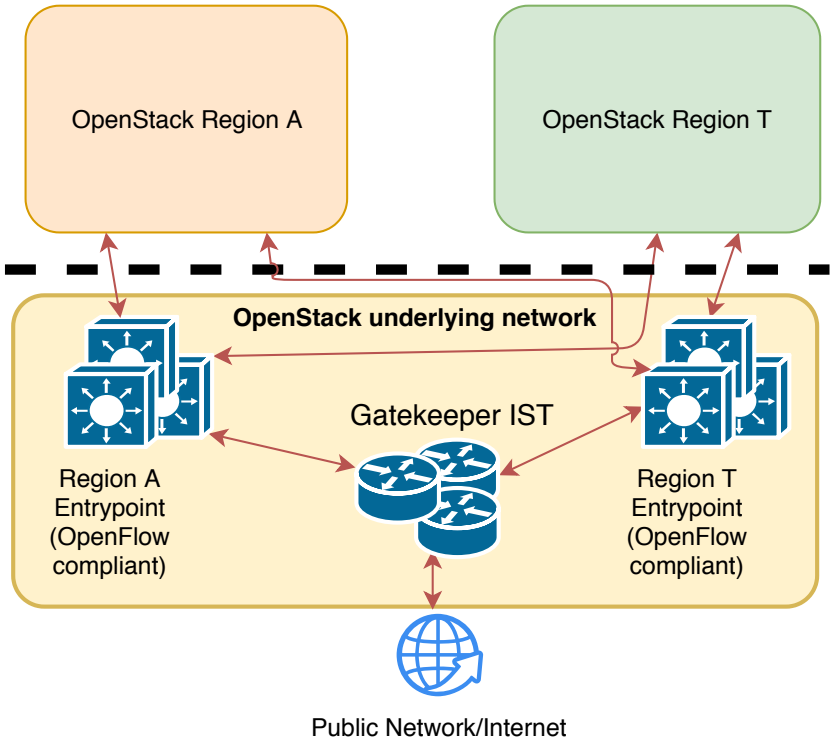

Figure 3.2: Global view of IST OpenStack underlying network

3.2 Architecture

The proposed solution is presented in figure 3.3 and is further divided into physical and virtual architectures.

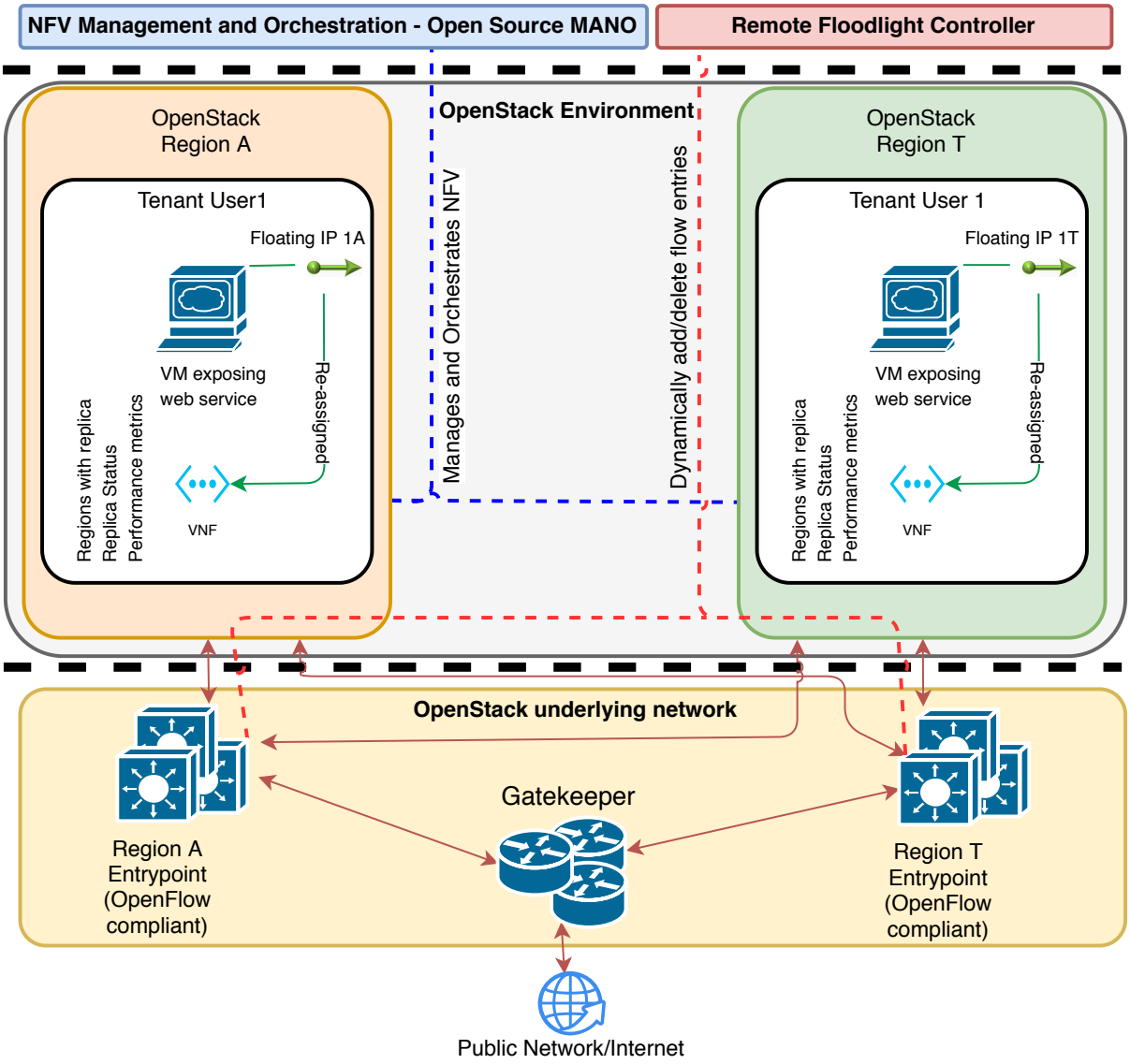

Figure 3.3: Architecture of the proposed solution

This solution is divided in three main blocks, separated with dashed lines and different colours.

The yellow and bottom block represents the underlying OpenStack network, composed of physical network devices. These network devices are responsible to forward incoming and outcoming packets to the desired OpenStack region. Each network device shall be replicated to avoid a single point of failure.

The top block, blue and red, is composed of NFV MANO and SDN controller. These components can be deployed on the same physical or virtual machine. Due to the nature of this work, this node

shall also be replicated. The proposed NFV MANO software is OSM, as explained on chapter 2.8, and the SDN controller is Floodlight. OSM is responsible to manage and orchestrate network services and VNFs deployed within OpenStack tenants. Floodlight is responsible to dynamically configure OpenStack underlying network.

The middle block is the OpenStack environment, which is composed by two regions. The OpenStack environment can be divided in two layers, physical and virtual. Physical layer is composed of the physical machines running core OpenStack services. The virtual layer is presented on orange and green blocks, and concerns to the network services deployed on OpenStack tenants of both regions.

Each one of these blocks have a specific function that is described below on physical and virtual architectures. Physical architecture is composed of NFV manager and orchestrator, SDN controller and OpenStack infrastructure and underlying network. Virtual architecture is composed of the network services and respective VNFs deployed at each OpenStack tenant.

3.2.1 Physical architecture

Each block that relies on a machine outside of OpenStack virtual environment belongs to the physical layer. On figure 3.4 it is presented a summary of the components that belong to the physical layer. The function of each block is described below.

Open Source MANO

OSM is the component responsible for the orchestration of the VNFs and management of NFV. It has a list of NSs and the respective VNFs that compose them. Since OSM is aligned with ETSI NFV reference architecture, it uses the defined interfaces to communicate with the VIM (OpenStack), which are Vi-VNfm and Or-Vi.

This work uses OSM to deploy, previously defined, NSs that are configured to ensure HA and Load-Balancing (LB). These NSs are defined using NSDs and VNFDs and uploaded to OSM through CLI or Graphical User Interface (GUI).

After NSs are deployed, OSM can use *Juju* charms to instruct the deployed VNFs to perform previously defined actions. These actions must be configured and described on the VNFD.

Although we propose to use NFV and OSM mainly to deploy some specific NSs that ensure HA and LB, our proposed solution/infrastructure can later benefit from some advantages like reduced CAPEX and OPEX, shorter development cycles, and the use of a single platform to deploy different services on different users and tenants.

OpenStack/VIM

Since this work focus on the specific use case of IST OpenStack infrastructure, we propose the use of OpenStack as the VIM. Yet, in theory, our proposed solution can also be applied to OpenVim, Amazon Web Services (AWS), and VMware vCD, since all are currently supported VIMs of OSM.

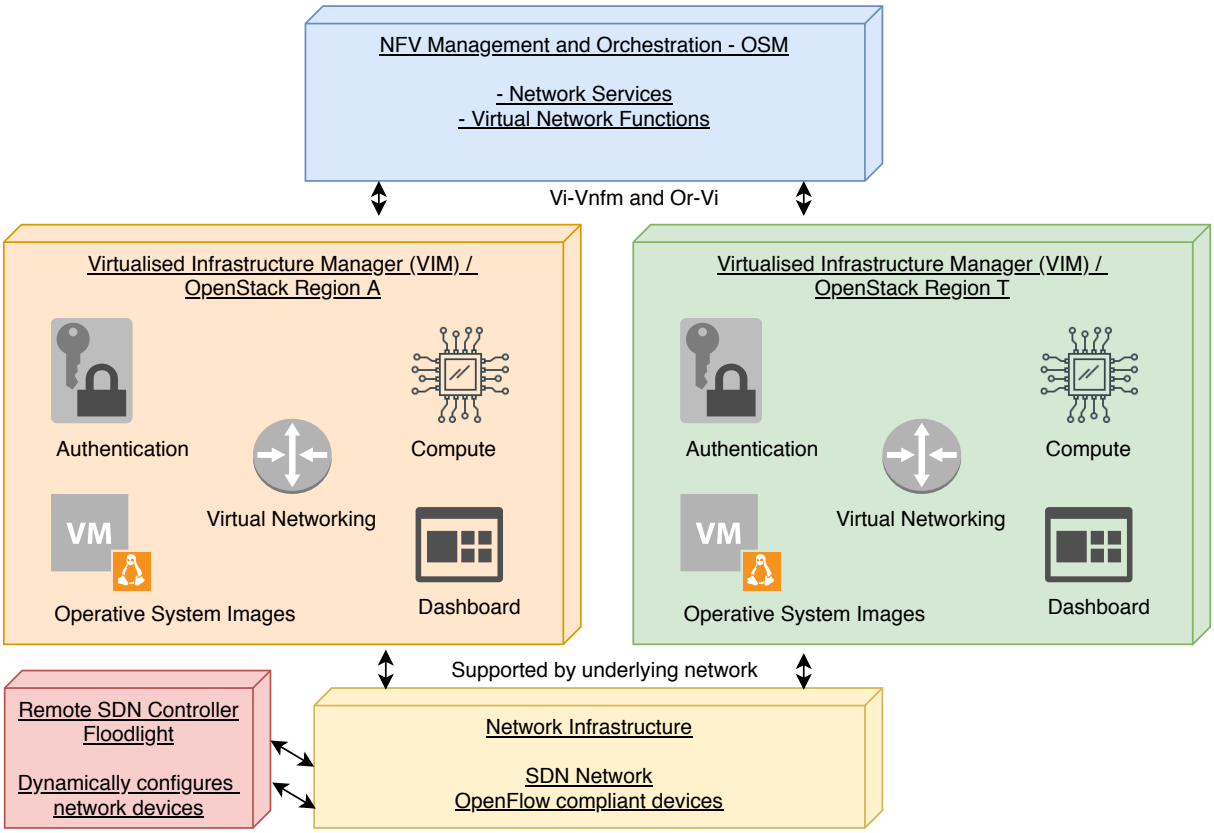

Figure 3.4: Physical architecture

OpenStack infrastructure is composed of two or more regions. If one becomes unavailable for some reason, incoming network traffic is forwarded to another one. Traffic can also be split between available regions.

Each region shall have services for authentication, computing resources, virtual networking, operative system images and a dashboard. These services are enabled with the core OpenStack services: Keystone, Nova, Neutron, Glance and Horizon, described in section 2.5.

An OpenStack environment can have multiple architectures and our solution is in theory compatible with all of them. Since we abstract from the inner components of our main blocks, if the ETSI NFV reference architecture is ensured, our proposed solution shall perform as expected.

Network architecture

Our network architecture is quite simple although it has two mandatory requirements. It shall exist triangulation between OpenStack regions and the public network. Network devices must support OpenFlow protocol, version 1.3 since it enables changes on headers of data packets.

As SDN controller, we propose the use of Floodlight, since it has a module to push static flow entries to network devices via HTTP requests using REST API. This allows a system administrator to dynamically configure the network from almost every computing device with the required access.

Floodlight controller should run on a remote server accessible from the OpenStack provider network. The controller is mainly used to install and remove table flow entries on the network devices providing public network connectivity to OpenStack regions.

These network devices have the following behaviour:

1. Receive and analyse incoming packet;
2. Check if it matches any entry on flow table;
3. Perform packet changes if it is needed;
4. Route the packet to the desired output port.

Floodlight can program network devices to perform different changes on a packet, as explained in section 2.2. In this work, we only need to change the packets source and destination of IP and Media Access Control (MAC) addresses.

3.2.2 Virtual architecture

Virtual architecture is the set of machines deployed within OpenStack tenants and their respective behaviour to ensure HA and LB. On figure 3.5 it is presented only the virtual architecture and the underlying network is abstracted as a cloud.

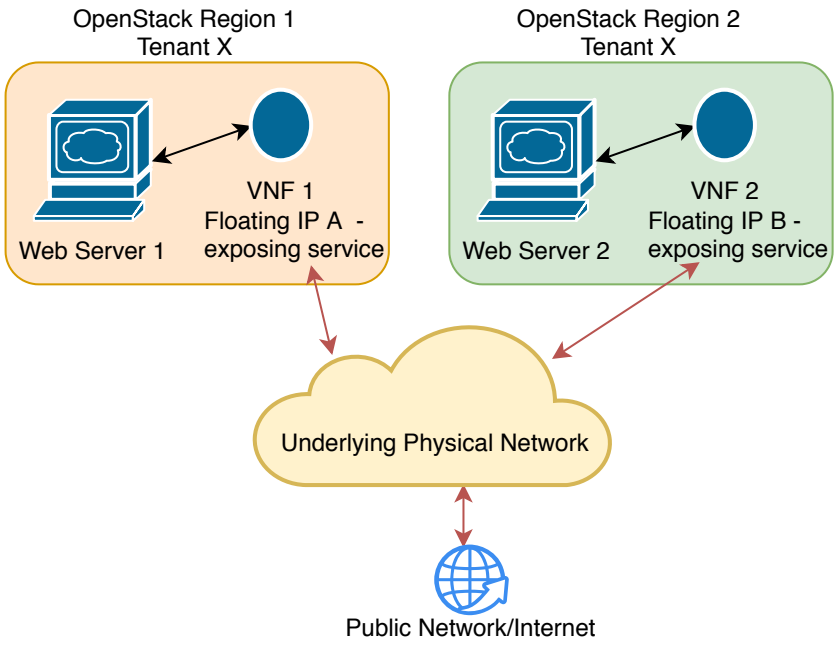

Figure 3.5: Virtual Architecture

Problem overview

As explained before, in order to expose a certain service on an OpenStack environment, a VM is deployed, and it is assigned a floating IP address to it, enabling access to and from the public network. If we require HA or LB, we must replicate this service on other OpenStack region, in order to provide redundancy if the primary region becomes unavailable or overloaded. In this scenario, HA and LB is only ensured if we have a fast handover of the incoming/outcoming traffic to/from the desired available region.

To implement this fast handover we propose to deploy an additional VM, at each region, along the original one. In case of failure or overloading, the floating IP address exposing the service is re-assigned to this new VM. By default, it forwards incoming traffic to the original VM.

This additional VM is actually a VNF with a predefined behaviour. And this pair of VMs can be replaced with a NS previously defined and deployed by OSM.

The architecture of this solution is presented on figure 3.5 and the network service is explained below.

Proposed network service

Assume that Web Server 1 and 2 are the VMs exposing some service with the need of HA and/or LB. Consider also that VNF 1 and 2 are the VMs responsible to ensure a certain SLA.

VNFs deployed at each region of a tenant send a heartbeat, every T_h second, to the other region. An action is taken if, at the other end, n heartbeats are missed. T_h and n are parameters that can be initialised using default values or defined according to the respective SLA.

The action taken when n heartbeats are missed consists of re-configuring the underlying network. If a certain service was being served at region 1, an *OpenFlow* message will be sent to change the flow of the respective service, in order to be served on region 2. VNFs instantiated on each region of a tenant are the ones responsible to send that control message.

The same action can be taken under other predefined metrics in order to implement LB. For instance, CPU utilisation, network links load or the number of active connections, scheduled maintenance, etc. All these metrics may be measured by each VNF, which may take appropriate actions in each case.

With this solution, the network automatically adapts to the user needs, keeping services available.

If at least one region is available, services worst time of unavailability (T_{un})is:

$$T_{un} = T_h * n + T_{nrp} + T_{nd}$$

where T_{nrp} is the network reprogramming time and T_{nd} is the network delay.

Due to the nature of this work, the VNFs and the underlying network devices are replicated with the behaviour depicted in figure 3.6.

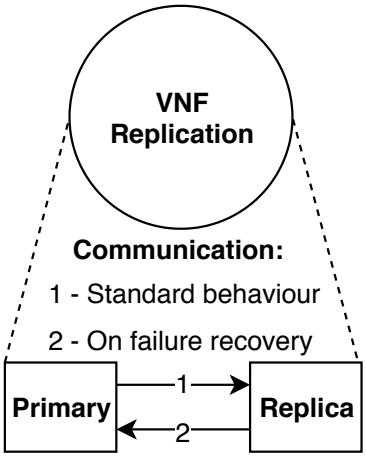

Figure 3.6: VNF replication

3.3 Summary

The main goal of this work is to leverage SDN and NFV concepts to implement HA and LB on IST OpenStack environment. Since this environment is in production, we detail it and give a template of a basic/core OpenStack environment to implement the proposed solution. The presented architecture has two layers, one physical and one virtual. The physical layer is the components that support the OpenStack infrastructure and the proposed solution. The virtual layer is the set of VMs to be deployed within OpenStack environment. All nodes shall be replicated and network nodes shall support OpenFlow protocol version 1.3+. VMs are responsible to verify the operational status of the deployed components and trigger the necessary addition of table flow entries to serve requests on the desired region.

4

Implementation

Contents

On this chapter, it is presented the implementation of the proposed solution.

As mentioned in the previous chapter, this work uses IST OpenStack environment as use-case. But since this environment is in production and there is no test environment, two identical regions of OpenStack were deployed, in association with Direção dos Serviços de Informática do Instituto Superior Técnico (DSI-IST), that borrowed all the necessary equipment and space to accommodate it.

As already described in chapter 3, the deployed scenario can be divided into two layers, physical and virtual. The physical layer is composed of OpenStack infrastructure, physical network devices and servers. The virtual layer is composed of the devices deployed within OpenStack cloud environment.

Physical and virtual layers are composed of several nodes. As stated in the previous chapter, since this work aims to enable access redundancy and load-balancing between OpenStack regions, on a production scenario, all nodes described on this chapter shall be replicated with automatic fail-over between them. On this implementation, due to the lack of resources and time constraints, these nodes do not have replication nor automatic fail-over. However, the implemented system has all the requirements to be used as a proof-of-concept and is further evaluated in chapter 5.

Both layers, physical and virtual, are described below in this chapter.

4.1 OpenStack Infrastructure

The deployed test infrastructure is presented in figure 4.1. It is similar to the one in production on IST, although it does not have Designate and Cinder OpenStack services. Note that these are not core services of basic OpenStack architecture and are not required in the scope of this work.

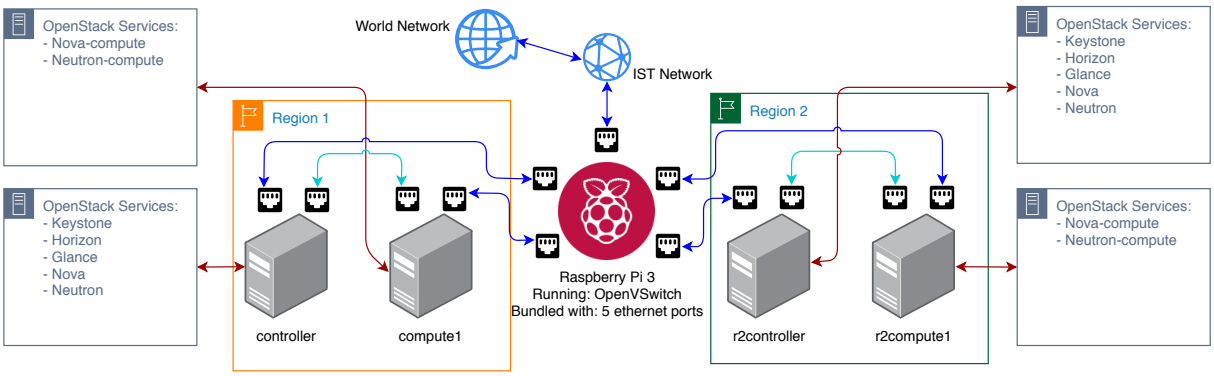

Figure 4.1: OpenStack Infrastructure

The deployed infrastructure is composed of two OpenStack regions (region 1 and 2) and the respective connection to the public network. Each region is composed by two nodes, *controller* and *compute1*, running core OpenStack services (described in figure and detailed below). All nodes have a connection to IST network and the public network through a switch. Since we had not available a physical switch

with support to OpenFlow 1.3, we used a Raspberry Pi 3 Model B+ running OpenVSwitch module. This node is described in more detail in chapter 4.2.1.

Each server node has two NICs, one connecting to the provider network (dark blue links) and the other connecting to the management network (light blue links). Each that supports the same OpenStack region must be connected to the same private network (management network), in this case, nodes are connected to each other.

Since region 1 and 2 are completely symmetric and have the same configurations except for network configurations, the procedures described below apply for both regions.

As stated before, each region is supported by two nodes. These nodes were set up with OpenStack release Rocky (the currently supported release at the time of the deployment), following its installation guide available at OpenStack official website. The installation process is not explained in detail, but the most important packages and procedures are described below.

An OpenStack environment relies on its core services and the respective dependencies. OpenStack services between nodes are synchronised using *Chrony*, an implementation of Network Time Protocol (NTP). *Controller* node sync with Official Ubuntu NTP server (`ntp.ubuntu.com`) and *compute1* node sync with *controller*.

Apart from time synchronisation, OpenStack services rely on *MySQL* to store information, *RabbitMQ* to coordinate operations among them, *Memchached* to cache tokens and *Etcd*, a distributed reliable key-value store is used to store configurations, keeping track of services availability and some other scenarios. All of these dependencies run on *controller* node.

Due to the lack of resources and need for other services, the deployed OpenStack environment is composed of the core services of a basic architecture and a dashboard.

System specifications of all nodes that compose OpenStack infrastructure are presented in appendix A.

4.2 Physical Layer

From the physical point of view, the scenario represented in figure 4.2 was deployed. It comprises OpenStack nodes from both regions, one node running *OpenVSwitch* and one node running OSM and Floodlight controller, further called OF-Node.

Nodes from both regions were deployed and connected to IST Taguspark campus facilities subnet *172.20.126.0/24*. Each region node communicates directly with the other through an Ethernet cable, on private network *10.0.0.0/24*.

OF-Node is a VM, running on IST OpenStack Region T (production environment), with the system specifications presented on appendix A. Although this node is a VM it belongs to physical architecture,

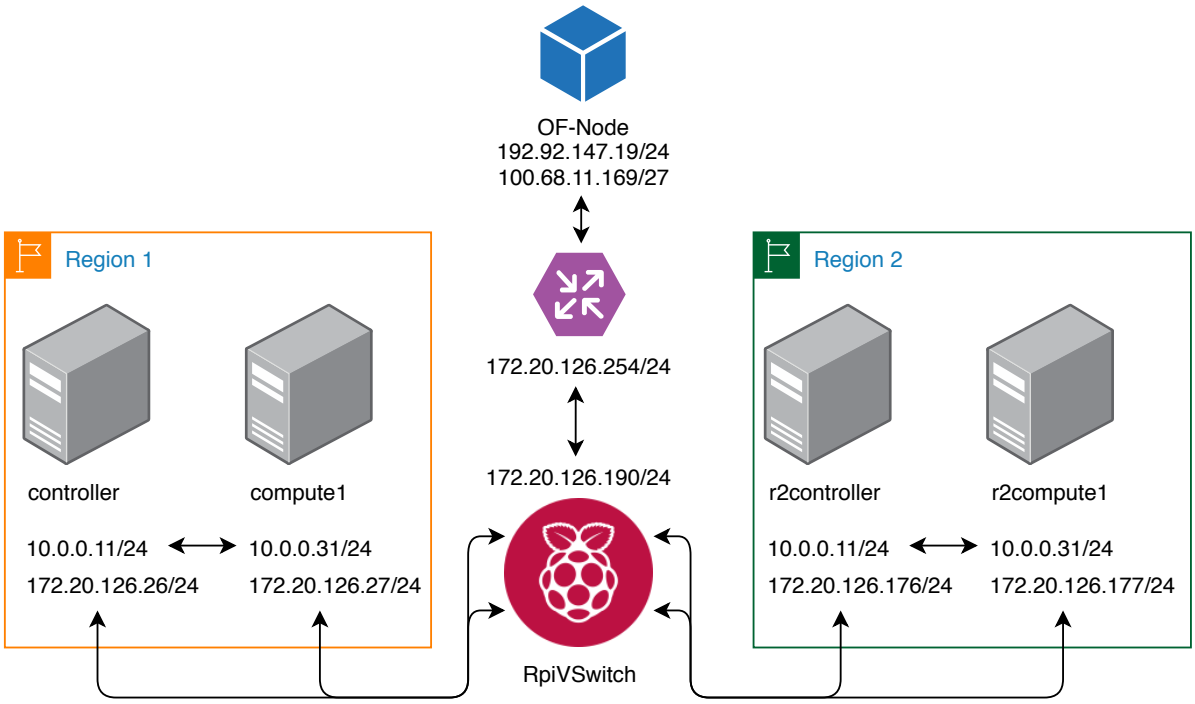

Figure 4.2: Physical Network Topology

only nodes running on the deployed OpenStack environment are considered virtual nodes.

A summary of IP addresses of physical nodes is presented in appendix B.

Controller and compute nodes were already described in chapter 4.2, RpiVSwitch and OF-Node are described below.

4.2.1 RpiVSwitch

One of the main components of this work is the node *RpiVSwitch*, a Raspberry Pi working as an OpenFlow switch. Since the design of the proposed solution, we realised the need for a network device compliant with OpenFlow protocol.

During the development of this thesis we tried to use *Mikrotik* routers, but the respective Operative System, *RouterOS*, is only compatible with version 1.0, which does not support the required features. Then, we tried to use two different *Alcatel-Lucent* switches recently withdrawn from production on IST facilities. One had its software outdated and the respective license to update also expired. The other was updated, compliant with OpenFlow 1.3+, but it only had interfaces for optical cables and there were no available adaptors to use.

After all those tries we had the ideia of using one Raspberry Pi 3 Model B+, bundled with 4 USB-to-Ethernet adaptors, running OVS.

RpiVSwitch is connected to a remote controller and performs Transparent Packet Redirection (TPR).

TPR, in this case, is the following: when a certain user sends packets to a VM on region 2, when packets arrive at *RpiVSwitch*, packets headers are modified (MAC and IP destination addresses are switched from region 2 VM to region 1 VM) and sent to region 1. The reverse also occurs, in the sense that response packets sent in the opposite direction from region 1 VM to original user VM are also modified (MAC and IP source addresses are switched from region 1 VM to region 2 VM) and sent to the original user.

An example of TPR implementation is presented on listing 4.1, using a *python3* script.

Listing 4.1: Python3 script that performs TPR

```
#!/usr/bin/python3

import urllib3, requests
import json
http = urllib3.PoolManager()

def addFlow():

    API_ENDPOINT = "http://192.92.147.19:8080/wm/staticflowpusher/json"

    flow1 = {
        "switch": "00:00:50:3e:aa:dc:be:04",
        "name": "flow-1",
        "priority": "32768",
        "in_port": "1",
        "active": "true",
        "eth_type": "0x0800",
        "ipv4_src": "10.2.0.82",
        "ipv4_dst": "172.20.126.202",
        "actions": "set_field=eth_dst->fa:16:3e:d8:26:42,
        set_field=ipv4_dst->172.20.126.110,output=7"}

    flow2 = {
        "switch": "00:00:50:3e:aa:dc:be:04",
        "name": "flow-2",
        "priority": "32768",
        "in_port": "7",
        "active": "true",
        "eth_type": "0x0800",
        "ipv4_src": "172.20.126.110",
        "ipv4_dst": "10.2.0.82",
        "actions": "set_field=eth_src->fa:16:3e:76:7e:2a,
        set_field=ipv4_src->172.20.126.202,output=1"}

    print(requests.post(url = API_ENDPOINT, data = json.dumps(flow1)))
    print(requests.post(url = API_ENDPOINT, data = json.dumps(flow2)))

addFlow()
```

This script sends a HTTP *POST* request to Floodlight controller. This request order the controller to add two flow rules to the identified switch. These flow rules perform the mentioned TPR packet header changes.

These flows can be deleted, when required, with a single request using *curl*:

```
curl -s http://192.92.147.19:8080/wm/staticentrypusher/clear/all/json
```

The *RpiVSwitch* behaviour is very simple. By default it only performs packet forwarding and when programmed by the controller, this node also modifies packet headers and re-routes them. An example of this behaviour is presented on message diagram on figure 4.3.

Since RpiVSwitch has very limited resources, and the goal was to develop only a proof-of-concept,

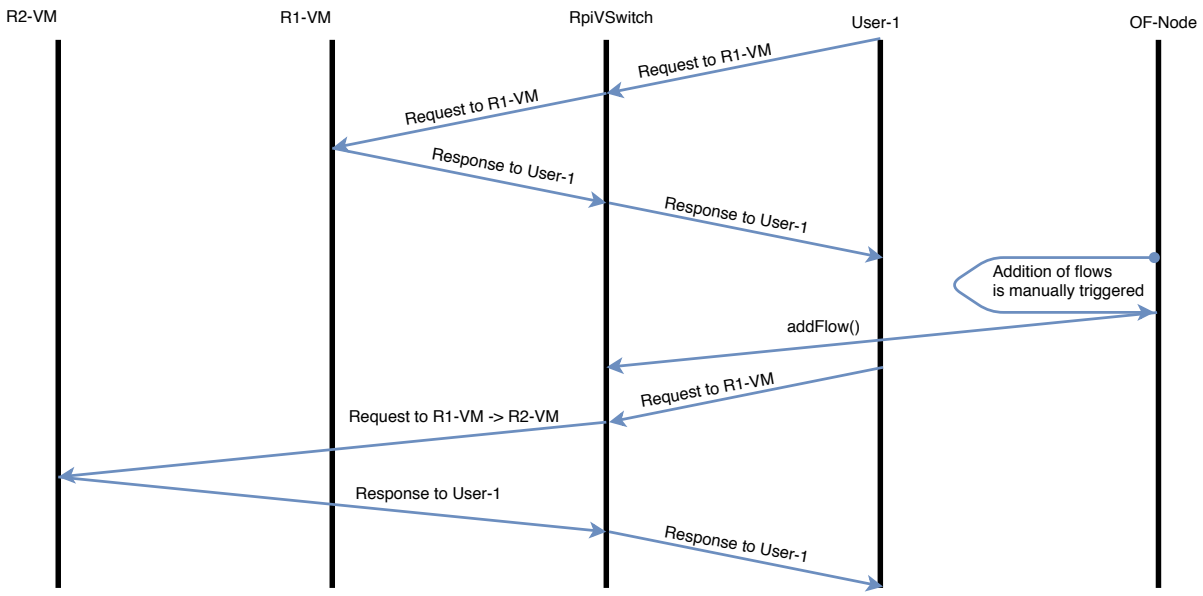

Figure 4.3: Message diagram of TPR example

we opted to use it only to forward packets. On a production scenario, using an OpenFlow compliant switch, this node could also be used to ensure LB. There would be a pool of servers, to serve requests with a given pattern, and the network node would load-balance these requests to servers on that pool. Since RpiVSwitch gets easily overloaded of packets, LB is ensured by other nodes, this behaviour is explained in chapter 4.3.

4.2.2 OF-Node

As mentioned before, nodes on the virtual layer were deployed with OSM running on OF-Node.

OF-Node was setup with OSM release 5, the most recent release at the time of installation, using a *dockerized* installation.

In this work we use OSM to deploy a network service, named thesis-ns, that will ensure our main goals, HA and LB.

After OSM installation on OF-Node, we configured it to access our VIM, the deployed OpenStack environment. First, on each OpenStack region, we created an account for OSM user with one project and admin role. Then we added the VIM to OSM. This procedure was done with the commands presented on appendix C.1.

This network service is described using the NSD presented on appendix C.2.

This descriptor mainly describes a NS with one VNF, member with index 1 of this VNF is connected to an external network named ”provider” on the VIM tenant.

As described before, our NSD itself is quite simple. The VNFD is a bit more complex, since it has

day-0, day-1 and day-2 configurations. Our VNF is composed of two VMs, DataVM and MgmtVM which is the Virtual Deployment Unit (VDU). Both are connected to each other through an internal, private network. MgmtVM is also connected to the provider network.

MgmtVM day-0 and day-1 configurations change the password of user *ubuntu*, enable *ssh* password authentication, enables IPV4 forwarding and creates a file at home directory of *ubuntu* user. This file is only created to test the functionality of the configuration. Regarding day-2 configurations, OSM sets a Juju charm that enables the network administrator to run python scripts on MgmtVM via OSM interface.

An excerpt of the VNFD code is presented on appendix C.3.

OF-Node is also running the OpenFlow Floodlight controller. It communicates with RpiVSwitch (listening on port 6653) via SBI using the OpenFlow protocol. It uses NBI to communicate with the application plane, which exposes a REST API on port 8080.

4.3 Virtual Layer

The virtual layer is composed of the VMs deployed within OpenStack tenants.

Regarding the virtual layer, the scenario presented in figure 4.4 was deployed.

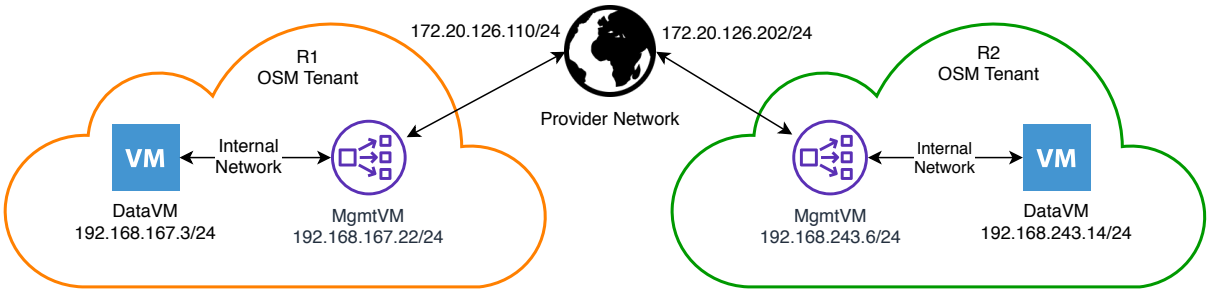

Figure 4.4: Virtual Network Topology

As stated before, this scenario was deployed using OSM running on the controller node (OF-node).

Each OpenStack region has two networks, provider and internal.

In both regions the provider network have the same subnet (*172.20.126.0/24*), but different allocation pools, which are *172.20.126.100-172.20.126.120* for region 1 and *172.20.126.200-172.20.126.220* for region 2.

Internal networks have: subnet *192.1268.167.0/24* on region1 and subnet *192.1268.243.0/24* on region 2.

Each region have two VMs. In region 1 *DataVM* is connected only to internal network with IP address *192.168.167.3*. *MGMTVM* is connected to both networks, with IP addresses *192.168.167.22* and *172.20.126.110*.

On region 2 the topology is identical. *DataVM* has IP address *192.168.243.14* and *MGMTVM* has IP addresses *192.168.243.6* and *172.20.126.202*.

A summary of these IP addresses are included on appendix appendix B.

Virtual nodes

On chapter 4.2 it were described the physical devices and environment required to support the virtual layer. Below it is explained the role of the virtual nodes that ensure HA and LB.

To explain the role of each virtual node, a summary of the scenario deployed and explained on the previous sections is presented in figure 4.5.

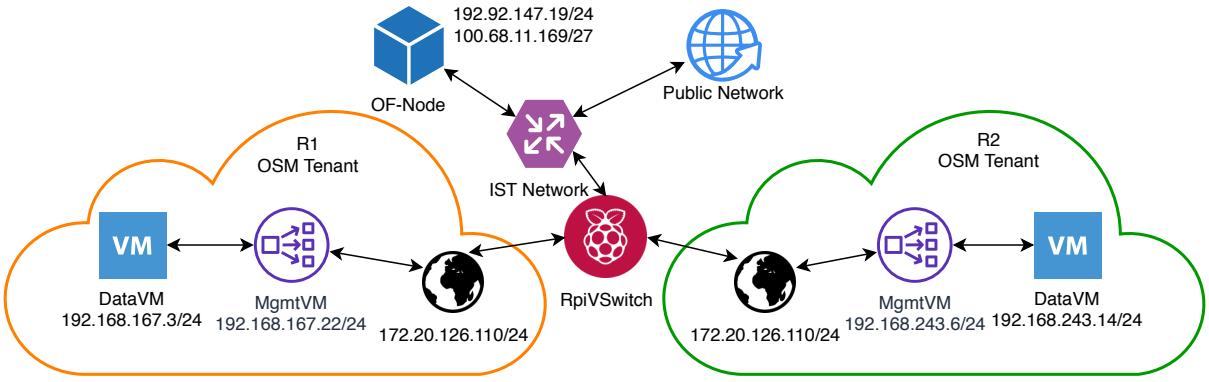

Figure 4.5: Summary of the deployed scenario

On this figure, it is represented OSM tenants of region 1 and 2, its connection to the IST and public networks through RpiVSwitch, and also the controller node (OF-Node) connected to IST network.

The system implemented to achieve the goals of this work lay on the transparent packet redirection presented in chapter 4.3. It allows us to re-route incoming and out-coming traffic to the desired VM exposing a certain service. On this implementation, we opted to expose an web-server using *python3 http.server* module.

Since DataVM is protected on a private network, inaccessible from outter networks, MgmtVM needs to forward traffic with destination to that node. This is performed using *IPTables prerouting* and *postrouting* rules presented on appendix C.4.

Each MgmtVM is responsible to verify if his peer DataVM is working as expected and exchange this information with MgmtVM from the other region. This verification is made with the script presented on appendix C.7.

High-availability

Regarding HA, MgmtVMs establish a Transmission Control Protocol (TCP) socket between them. Primary-MgmtVM runs *primary.py* and Replica-MgmtVM runs *replica.py*. These scripts works as a client-server architecture and they are presented on appendix C.5 and C.6. Both nodes listen and replies to heartbeat messages.

On *primary.py* it is called a function named *checkHTTPServer*. This function makes a request to the web-server and verifies its operation. This function is presented on appendix C.7 and described below.

In this case we don't have a DNS record for our website, otherwise, we would replace the IP address with the Uniform Resource Locator (URL). Our goal with this script is to verify the status of the web-server. If we get any status different from 200, which is the code for request succeed, we assume that the web-server has some malfunction and will redirect to the server on the other region.

Our verification does not care if it is the domain translation that failed, if the web-server crashed or if it was some other issue, we replicate the request/(s) that a normal user would perform and if the result is not the expected, we redirect traffic to the other region.

On the replica region, its MgmtVM runs *replica.py* script, which ensures that the service exposed on the web-server has a limited time of unavailability.

Assuming that region 1 is the primary region, we elect Mgmt and Data nodes of region 1 as Primary-MgmtVM and Primary-DataVM. Nodes of region 2 are elected as Replica-MgmtVM and Replica-DataVM.

We expose our web-server with the IP address of Primary-MgmtVM. This node will forward packets with destination port 80 to Replica-DataVM.

Replica-MgmtVM sends a heartbeat, ("Alive" message), every T_h second. If Primary-MgmtVM receives the heartbeat, it sends a request to Primary-DataVM to check the status of the HTTP server. If Primary-DataVM returns *200 status code*, Primary-MgmtVM replies to Replica-MgmtVM heartbeat with "Alive" message. In this scenario, everything is working as expected.

If Primary-MgmtVM does not receive "Alive" message or Primary-DataVM does not reply with *200 status code*, the number of heartbeats misses H_m is increased at Replica-MgmtVM. Being n the max number of missed heartbeats, if $H_m >= n$, Replica-MgmtVM checks the status of Replica-DataVM. If it returns *200 status code*, flows to implement TPR are installed. If it returns a different status code there is no available region and node to serve the requests of the exposed service.

T_h and n are parameters that can be initialised using default values or defined according to the respective SLA. Message diagrams of the described behaviour are presented on figures 4.6 and 4.7.

Load-Balancing

Regarding LB, we implemented three metrics that trigger TPR.

The first metric is packets per second. As explained before, MgmtVM is exposed on the public network and forwards HTTP traffic do DataVM. This metric counts the number of packets passing on the internal network interface and if it reaches a predefined number, TPR is triggered.

The second metric implemented is similar. Since triggering TPR on a predefined number of packets per second will probably reset an established connection, the second metric, counts the number of active HTTP flows on DataVM instead. This way established connections are not interrupted.

The third implemented metric concerns with the load of DataVM instead of network load. When the

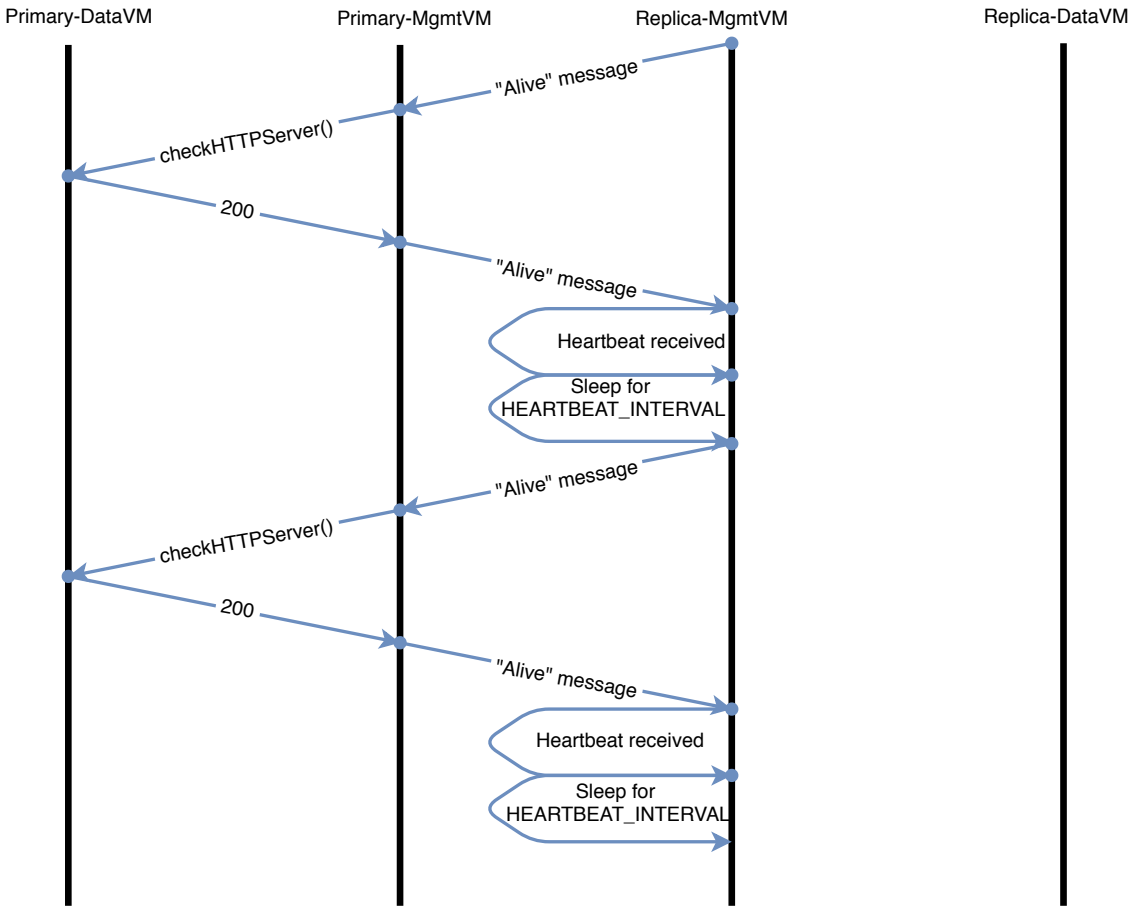

Figure 4.6: Message diagram of primary-replica communication on success scenario

CPU utilisation of DataVM reaches a certain percentage, MgmtVM triggers TPR.

4.4 Summary

On this chapter, it was described the implementation details of the developed solution. Two OpenStack regions were deployed, on two physical servers per region. A Raspberry Pi 3 Model B+ were used as an OpenFlow compliant switch to provide public network access to OpenStack nodes. A VM were setup with Floodlight as SDN controller and OSM as NFV orchestrator. Virtual nodes deployed within OpenStack environment compose a NS, previously described, that ensures HA and LB at the system. This NS implement a client-server architecture between nodes of both regions. If any core node of a region becomes unavailable, the node from the other region triggers the addition of flows that re-route incoming and outgoing traffic.

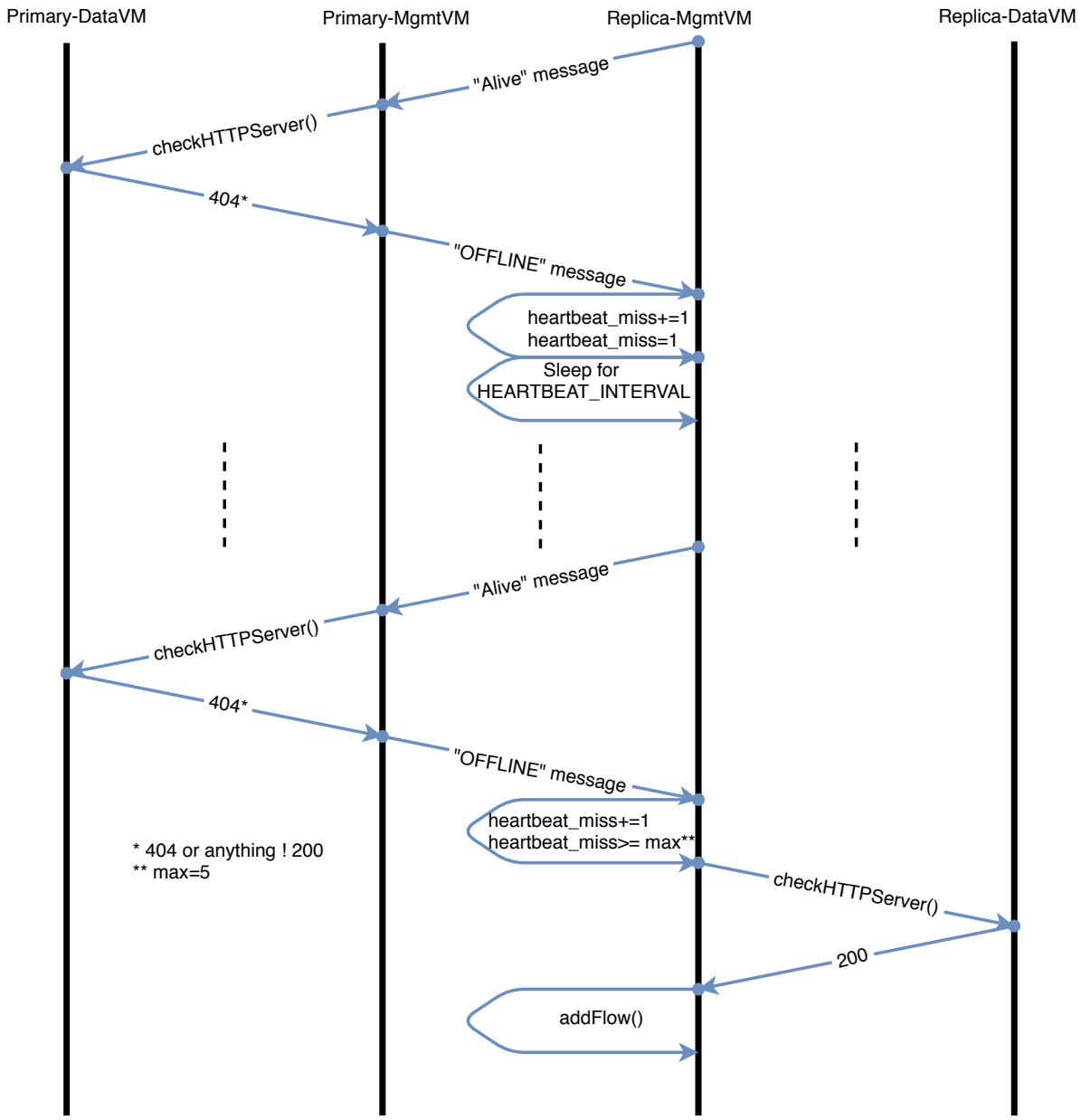

Figure 4.7: Message diagram of primary-replica communication on fail scenario

5

Evaluation

Contents

On this chapter, it is presented the tests performed to test and evaluate the implemented solution. We present functional and performance tests. Functional tests aim to test whether the implemented system has the expected behaviour. Performance tests aim to test the system under extra workload to check if it works efficiently and if the system handles the extra workload. Given that functional tests are quite simple, both types of tests are presented simultaneously. As stated in the previous chapter, the system was deployed on DSI-IST Taguspark *campus* facilities.

Some tests were performed during the development of the solution, namely, connectivity and bandwidth tests, which are presented below.

After these preliminary tests, the scenario presented in figure 5.1 was consolidated to further test the implemented system.

The test scenario is the one described in the implementation chapter with the addition of an extra PC to make requests from the IST network.

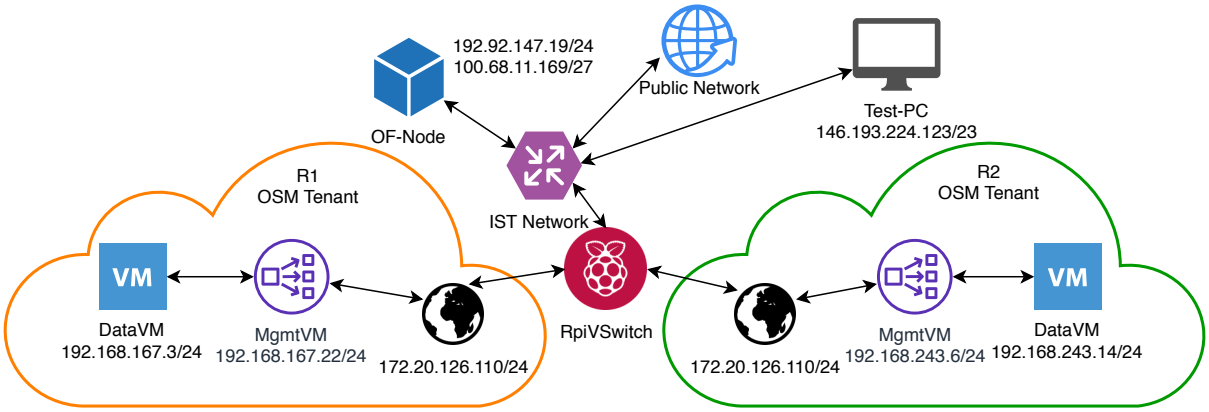

Figure 5.1: Test scenario implemented to evaluate proposed solution

5.1 Connectivity and speed test

To test connectivity between nodes a simple script that pings every node and prints if the host is up or not, were run on all nodes except DataVM. This script is presented in appendix C.8.

On every node, there is an entry on $/etc/hosts$ file pointing each name to the respective IP address.

Since there is connectivity between all nodes, the output was always the same:

```
node controller is up
node compute1 is up
node r2controller is up
node r2compute1 is up
node OF-node is up
node rpivswitch is up
node r1-mgmtvm is up
node r2-mgmtvm is up
```

Regarding bandwidth test, on each node, it was measured download and upload speeds. The results are presented in table 5.1.

	Download (Mbit/s)	Upload (Mbit/s)
controller	92.52	87.49
compute1	85.44	88.47
r2controller	92.59	87.86
r2compute1	92.54	70.54
OF-node	512.93	130.81
RpiVSwitch	162.41	171.89
r1-mgmtvm	86.76	51.57
r2-mgmtvm	93.29	64.13
Test-PC	602.02	590.97

Table 5.1: Bandwidth test results

Analysing these results we conclude that every node has reasonable bandwidth available. However, all nodes have internet connectivity through RpiVSwitch (with exception of Test-PC) and therefore have limited bandwidth, since they are connected through Ethernet adaptors with a maximum bandwidth of 100 Mbits/s and this can be further restricted due to Raspberry Pi 3 limitations.

5.2 Latency and bandwidth from Test-PC

After debugging connectivity between hosts and test the bandwidth to servers on the public network, latency and bandwidth tests between Test-PC and some nodes were performed using *iperf* [33].

Results are show in table 5.2.

	Min (ms)	Average (ms)	Max (ms)	Stdev (ms)
R1-MgmtVM	1.599	3.484	6.711	1.248
R2-MgmtVM	1.488	2.925	5.923	1.273
RpiVSwitch	0.662	1.152	6.753	0.896

Table 5.2: Latency test results from Test-PC

Results of the bandwidth test are shown in table 5.3.

	Time interval (s)	Transfer size (Mbytes)	Bandwidth (Mbits/s)
R1-MgmtVM	10	108	90.2
R2-MgmtVM	10	103	86.6
RpiVSwitch	10	123	103

Table 5.3: Bandwidth test results from Test-PC

From the results presented we can infer the impact that our network node (RpiVSwitch) has on the proposed solution. We conclude that RpiVSwitch add an extra latency off +/- 3 ms and the bandwidth

has a reduction of +/- 10%. This occurs beacuse Ethernet adaptors providing connectivity between RpiVSwitch and OpenStack nodes have limited bandwidth, as explained before.

5.3 Web-server bench-marking

To test OpenStack underlying network and the web-server, the system was bench-marked using *Apache bench* [34] and a python script that performs consecutive requests to the web-server during a certain period. Results of these tests are presented in tables 5.4 and 5.5.

Dest.	N	C	Elapsed time (s)	Avg # of requests/s	Transfer rate	Avg Conn. time (ms)
R1	1000	10	5.745	174.05	64.42	56
R1	1000	20	4.863	205.64	76.11	87
R1	1000	30	5.809	172.13	63.71	120
R1	1000	40	4.8	208.34	77.11	174
R1	1000	50	4.348	230.01	85.13	207
R1	999	60	ERROR #54	ERROR #54	ERROR #54	ERROR #54
R2	1000	10	7.152	139.83	51.75	56
R2	1000	20	4.767	209.77	77.64	92
R2	1000	30	4.357	229.52	84.95	121
R2	1000	40	4.371	228.8	84.68	152
R2	995	50	ERROR #54	ERROR #54	ERROR #54	ERROR #54

Table 5.4: Apache benchmark test results from Test-PC

As presented on table 5.4, when we perform 50/60 concurrent requests the connection is reset due to the lack of the *acknowledge* message from TCP [35]. This is caused by the incapacity of the *RpiVSwitch* CPU of performing packet switching with the desired frequency. A diagnosis of *RpiVSwitch*, using *htop* [36], during 60 concurrent requests is presented in figure 5.2.

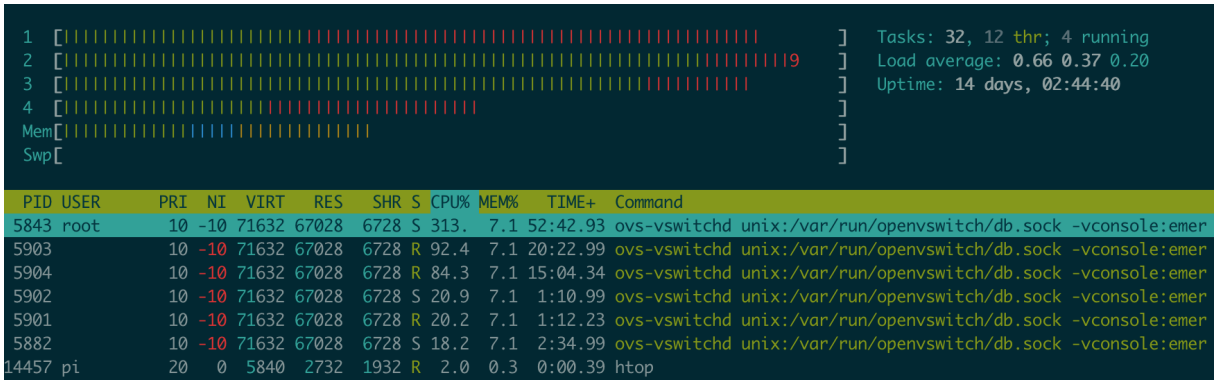

Figure 5.2: Screenshot of *htop* output on RpiVSwitch

Regarding the results presented in table 5.5, we conclude that the network and web-server are capable of serving an average of 36 requests per second.

Destination	Elapsed time (s)	# requests	Requests/s
R1-DataVM	20	687	34.35
R1-DataVM	40	1465	36.625
R1-DataVM	60	2202	36.7
R2-DataVM	20	688	34.4
R2-DataVM	40	1562	39.05
R2-DataVM	60	2282	38.03

Table 5.5: Performance test script results from Test-PC

5.4 Transparent Packet Redirection

As explained in chapter 4, this work depends on a core feature which is TPR, it allows to modify a packet that would be served on one machine, to be served on another one. It re-routes and modifies the necessary headers of a given packet or set of packets that match some predefined rules.

In this chapter, on figures 5.3 and 5.4, it is presented the tests performed during development in order to test the addition and deletion of flows, responsible to ensure TPR. It is expected that, before addition and after deletion of flows, requests to region 2 are served by region two. And after addition and before deletion, the same request is served by region 1.

Figure 5.3: Screenshot of two consecutive requests to R2-MgmtVM, one before and one after addition of flows

As it is shown, before the addition of flows, a request to the IP address of region 2 has a response from region 2. After the addition of flows, the same request has a response from region 1. Although it is not shown in the figures above mentioned, after the deletion of flows, the same request has a response from region 2. Given that, we conclude that the addition and deletion of flows that ensure TPR are

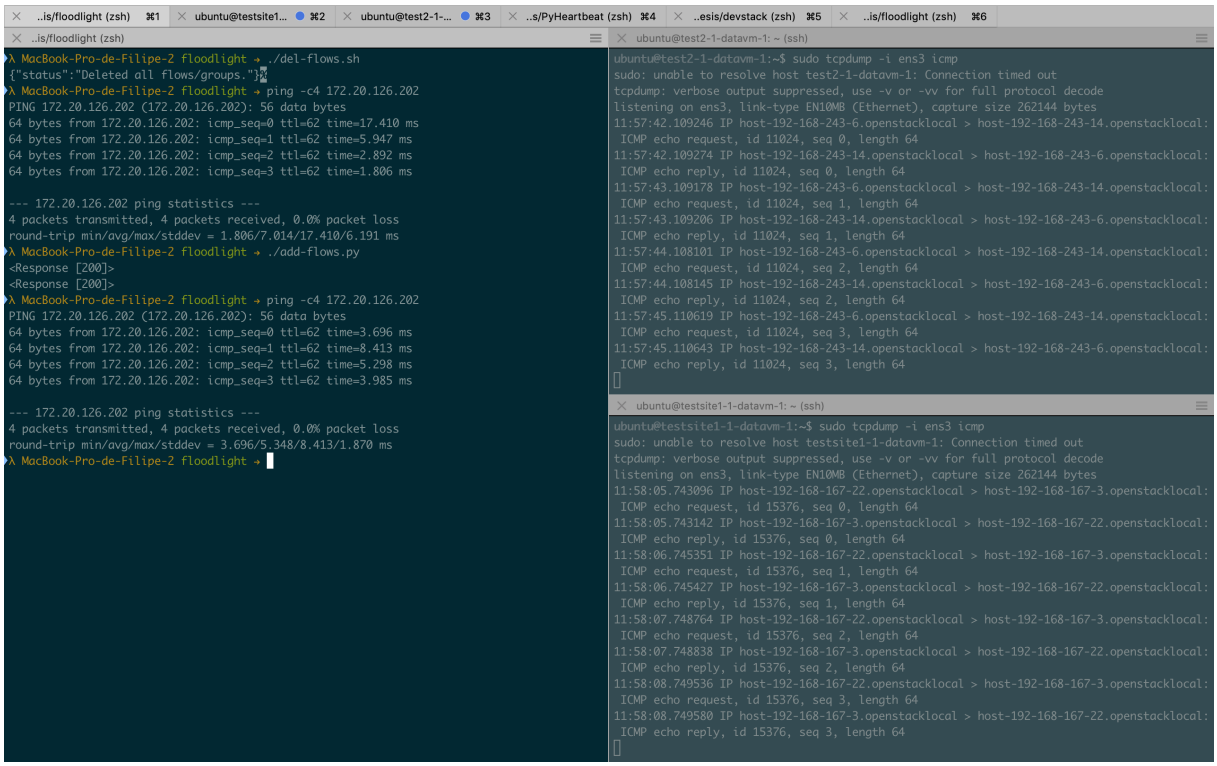

Figure 5.4: Screenshot of two pings to R2-MgmtVM, one before and one after addition of flows

working as expected.

5.5 High-availability

In this section, it is presented the tests performed to evaluate HA and time of unavailability when one region becomes unavailable for some reason.

Region 2 is in this scenario the primary region, R2-MgmtVM is running *primary.py* and R1-MgmtVM is running *replica.py*.

To measure the time of unavailability of our solution, we produced *perftest.py*, a script that performs consecutive requests to region 2 and counts the number of requests, the number of successful and failed requests and the time of unavailability. On table 5.6, it is presented the results outputted by *perftest.py*, using different values of T_h and n. Note that during the execution of *perftest.py*, R2-DataVM is shut down in order to force the system to reconfigure the network and start serving requests on the other, available, region.

Analysing results of the previous table, we can conclude that success rate increases and the time of unavailability drastically decreases, as we decrease parameters T_h and n.

While performing consecutive requests over 60 seconds, with parameters $T\_h = 0.5$ and $n = 5$, the

Elapsed time (s)	# requests	Success rate (%)	Fail rate (%)	T_un (s)	T_h (s)	n
60	8398	91.24	8.76	24.72	2	10
60	8631	96.85	3.15	9.88	2	5
60	9631	98.66	1.34	5.09	1	5
60	9614	99.5	0.5	3.25	0.5	5

Table 5.6: Results of tests evaluating HA with different parameters T_h and n

time of unavailability is only 3.25 seconds and the success rate is 99.5%. This means that, with our solution, the primary region could fail almost two times a week and we would guarantee a ”five nines” SLA. In this case we calculate that the $T\_nrp + T\_nd = 0.753s$.

To ensure a ”five nines” SLA a system can have a maximum downtime of 6.05 seconds per week. If our solution fails two times a week, we calculate it to be unavailable: $2 * 3.25seconds = 6.5seconds > 6.05seconds$.

With a specialised network device instead of RpiVSwitch, our solution will most likely guarantee ”five nines” SLA with the mentioned parameters.

5.6 Load-Balancing

In this section, it is presented the tests and respective results regarding the evaluation of LB.

We produced *lb-test.py* to evaluate the system, an extension of *perftest.py*. This script, instead of the time of unavailability, measures and count which region served each request.

The system will be tested with three LB policies, packets per second, number of active flows and CPU usage.

Results using packets per second policy are presented in table 5.7. In this scenario, each MgmtVM trigger TPR as soon as they forward one packet to DataVM.

Elapsed time (s)	# requests	Success rate (%)	Fail rate (%)	T_un (s)	R1 (%)	R2 (%)
20	755	99.47	0.53	0.1	49.7	50.3
60	2271	99.51	0.49	0.29	49.8	50.2

Table 5.7: Results of LB using packets per second policy

Analysing the experimental results, we conclude that the system has a behaviour very similar to round-robin. It serves almost 50% of requests on each region and has a low rate of requests not served. The percentage of requests are not exactly 50%, probably due to the network reprogramming time and minor random fluctuations. There are requests not served since TPR is triggered when a single packet passes the interface connected to the inner network. It means that at least some packets of a single request are split and routed to different regions.

Regarding active number of flows policy, which means that LB is triggered when a certain number

of active flows is reached, the results are presented in table 5.8. Each MgmtVM redirect traffic on 1000 active flows, test script performs 2000 requests, given that, it is expected that each region serves 1000 requests. This policy is tested with different times of update, (T_update), this time refers to the periodicity which MgmtVM asks DataVM for its active number of flows.

Elapsed time (s)	# requests	Success rate (%)	T_un (s)	R1 (%)	R2 (%)	T_update (s)
78.5	2000	100	0	45.8	54.2	5
80	2000	100	0	46.5	53.5	0.5

Table 5.8: Results of LB using number of active flows policy

In this case, the same request distribution issue occurs, there are more requests served in the primary region. Once again, this probably occurs due to the network reprogramming time. Using this policy there is 100% success rate.

Regarding CPU utilisation policy, since RpiVSwitch gets overloaded of packets before DataVMs significantly increases CPU usage, all packets are always served by the same region. Results are presented in figure 5.9.

Elapsed time (s)	# requests	Success rate (%)	Fail rate (%)	T_un (s)	R1 (%)	R2 (%)
20	780	100	0.00	0.00	0.00	100
60	2224	100	0.00	0.00	0.00	100

Table 5.9: Results of LB using CPU usage policy

5.7 Summary

In this chapter we presented functional and performance tests used to evaluate the implemented system.

Regarding functional tests, the implemented system performs as expected and it enables access redundancy and load-balancing between two OpenStack regions.

Regarding performance tests, it is clear that the implemented system is easily overloaded on node RpiVSwitch. Consecutively, some of the performance tests are not capable of evaluating the overall system, since the system always fails, firstly, on the same node. Yet, the presented tests can be used as a reference, to evaluate similar infrastructures and systems with dedicated hardware instead of node RpiVSwitch.

6

Conclusion

Contents

This work had a clear goal, how to leverage SDN and NFV concepts to implement HA and LB on IST OpenStack infrastructure. As explained before, IST has two regions of OpenStack, symmetric and independent. Notwithstanding it is not possible, or at least there is no simple way of load-balance traffic between a service replicated in both regions. The same happens concerning HA, given that a service running on one region is exposed via its IP address or DNS record pointing to that address, it is not simple to keep the availability of the service if the region where it is deployed become unavailable for some reason.

6.1 Discussion

The premise of this work was implementing HA and LB on OpenStack, with the above-mentioned concepts, and if possible, design and implement a solution compatible with different OpenStack architectures, with two or more regions, and also compatible with different clouds, for instance AWS. As a proof-of-concept, during the development of this work, an OpenStack infrastructure with two regions, for test purposes, was deployed on IST Taguspark *campus* facilities. On this OpenStack environment, we deployed and tested the solution proposed in this work, and despite some limitations regarding limited resources and time constraints, the system implemented works as expected.

Although the implemented system had proven to achieve its goal, in the previous chapter there is pretty obvious that the main limitation of this work is the network node, RpiVSwitch. As described before this node is a Raspberry Pi 3 Model B+ and has limited resources, mainly the CPU in what concerns to the needs of packet processing using OVS. A simple improvement that could be done with reasonable costs is the replacement of this node with the recently released Raspberry Pi 4 Model B.

6.2 Future work

Although it will not most likely be enough for a system in production, benchmarks to Raspberry Pi Models 3 B+ and 4 B show that the last one can be three to four times faster. This improvement could enable further testing of the presented proof-of-concept.

Due to logistics issues, both regions of the deployed OpenStack infrastructure were setup on Taguspark campus, ideally, the proof-of-concept shall be done with one region deployed at each campus and if possible test the system with an additional third region.

The architecture of the proposed solution assumes that all nodes are replicated, yet mainly due to limited resources, time constraints and the fact that replication is not the focus of this work, the presented proof-of-concept does not have any node replicated. In future work, before implementing this solution on a cloud in production, a more complete deployment shall be tested.

Bibliography

[1] M. T. Jones, “Virtual networking in linux - nics, switches, networks, and appliances,” Oct 2010.

[2] “Etsi, “network function virtualization: Architectural framework,”,” 2013, https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf, visited 2018-12-28.

[3] “Conceptual architecture,” https://docs.openstack.org/install-guide/get-started-conceptual-architecture.html#get-started-conceptual-architecture, visited 2018-12-28.

[4] “Openstack map,” https://www.openstack.org/assets/software/projectmap/openstack-map.pdf, visited 2018-12-28.

[5] “High-availability,” https://en.wikipedia.org/wiki/High_availability, visited 2019-06-13.

[6] G. Lavado, “(a true story on) achieving end-to-end nfv with openstack and open source mano,” May 2018.

[7] O. N. Foundation, “Openflow switch specification - version 1.5.1,” march 2015.

[8] X. Foukas, M. K. Marina, and K. Kontovasilis, “Software defined networking concepts,” *Software Defined Mobile Networks (SDMN): Beyond LTE Network Architecture*, p. 21, 2015.

[9] H. Kim and N. Feamster, “Improving network management with software defined networking,” *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, February 2013.

[10] R. Gopel L, “Separation of control and forwarding plane inside a network element,” in *5th IEEE International Conference on High Speed Networks and Multimedia Communication (Cat. No.02EX612)*, July 2002, pp. 161–166.

[11] R. Narisetty, L. Dane, A. Malishevskiy, D. Gurkan, S. Bailey, S. Narayan, and S. Mysore, “Openflow configuration protocol: Implementation for the of management plane,” in *2013 Second GENI Research and Educational Experiment Workshop*, March 2013, pp. 66–67.

[12] J. Dix, “Clarifying the role of software-defined networking northbound apis,” *Network*, vol. 4, p. 11, 2013.

[13] W. Zhou, L. Li, M. Luo, and W. Chou, "Rest api design patterns for sdn northbound api," in *2014 28th international conference on advanced information networking and applications workshops*. IEEE, 2014, pp. 358–365.

[14] S. R. David Lenrow, "Nothbound interface working," *Open Network Foundation*, 2013.

[15] A. Kondwilkar, P. Shah, S. Reddy, and D. Mankad, "Can an sdn-based network management system use northbound rest apis to communicate network changes to the application layer?" *Capstone Research Project*, pp. 1–10, 2015.

[16] L. Zhu, M. M. Karim, K. Sharif, F. Li, X. Du, and M. Guizani, "Sdn controllers: Benchmarking & performance evaluation," *arXiv preprint arXiv:1902.04491*, 2019.

[17] O. S. Brief, "Openflow-enabled sdn and network functions virtualization," 2014.

[18] N. W. Paper, "Network functions virtualisation: An introduction, benefits, enablers, challenges call for action. issue 1," Oct. 2012, https://portal.etsi.org/NFV/NFV_White_Paper.pdf, visited 2018-12-28.

[19] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal, "Nfv: state of the art, challenges, and implementation in next generation mobile networks (vepc)," *IEEE Network*, vol. 28, no. 6, pp. 18–26, Nov 2014.

[20] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys Tutorials*, vol. 18, 09 2015.

[21] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, Feb 2015.

[22] D. Hausheer, O. Hohlfeld, D. R. López, B. M. Maggs, and C. Raiciu, "Network Function Virtualization in Software Defined Infrastructures (Dagstuhl Seminar 17032)," *Dagstuhl Reports*, vol. 7, no. 1, pp. 74–102, 2017. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7246

[23] A. Kavanagh, "Openstack as the api framework for nfv: the benefits, and the extensions needed," *Ericsson Review*, vol. 2, p. 102, 2015.

[24] F. Callegati, W. Cerroni, C. Contoli, and G. Santandrea, "Performance of network virtualization in cloud computing infrastructures: The openstack case," in *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, Oct 2014, pp. 132–137.

[25] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Openstack: toward an open-source solution for cloud computing," *International Journal of Computer Applications*, vol. 55, no. 3, pp. 38–42, 2012.

[26] T. Rosado and J. Bernardino, “An overview of openstack architecture,” in *Proceedings of the 18th International Database Engineering & Applications Symposium*. ACM, 2014, pp. 366–367.

[27] J. Denton, *Learning OpenStack Networking (Neutron)*. Packt Publishing Ltd, 2014.

[28] J. Gray and D. P. Siewiorek, “High-availability computer systems,” *Computer*, vol. 24, no. 9, pp. 39–48, Sep. 1991.

[29] V. Cardellini, M. Colajanni, and P. S. Yu, “Dynamic load balancing on web-server systems,” *IEEE Internet Computing*, vol. 3, no. 3, pp. 28–39, May 1999.

[30] “Tacker,” https://wiki.openstack.org/wiki/Tacker, visited 2018-12-30.

[31] “Onap,” https://www.onap.org/, visited 2018-12-30.

[32] “Onap architecture overview,” https://www.onap.org/wp-content/uploads/sites/20/2018/11/ONAP_CaseSolution_Architecture_112918FNL.pdf, visited 2018-12-28.

[33] S. Ubik and A. Král, “End-to-end bandwidth estimation tools,” *Technical report, CESNET Technical Report 25/2003*, 2003.

[34] D. Rahmel, “Testing a site with apachebench, jmeter, and selenium,” in *Advanced Joomla!* Springer, 2013, pp. 211–247.

[35] B. A. Forouzan and S. C. Fegan, *TCP/IP protocol suite*. McGraw-Hill, 2006, vol. 2.

[36] H. Muhammad, “htop-an interactive process viewer for linux,” *Zugegriffen, unter http://hisham.hm/htop*, 2015.

System specifications

On this appendix it is presented a list of the system specifications of physical nodes.

A.1 *controller* and *compute1*

These nodes compose the deployed OpenStack Region 1 described on chapter 4.1. Both are physical servers with the specifications listed below.

- Operative System: Ubuntu 18.04.2 LTS
- CPU: Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
- Graphics card: Intel Corporation HD Graphics 530
- Random Access Memory (RAM): 8 G
- Disk size: 200 G
- Number of NICs: 2

A.2 *r2controller* and *r2compute1*

Similar to the nodes described above, these nodes compose OpenStack region 2. Both are physical servers with the specifications listed below.

- Operative System: Ubuntu 18.04.2 LTS
- CPU: Intel(R) Core(TM)2 Quad CPU Q9500 @ 2.83GHz
- Graphics card: Intel Corporation 4 Series Chipset Integrated Graphics Controller
- RAM: 4 G
- Disk size: 300 G
- Number of NICs: 2

A.3 *RpiVSwitch*

This node is a Raspberry pi 3 B+, his role is described on chapter 4.2.1. Respective system specifications are listed below.

- Operative System: Raspbian GNU/Linux 10 (buster)
- CPU: Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz
- Graphics card: Intel Corporation 4 Series Chipset Integrated Graphics Controller
- RAM: 1 G
- Disk size: Micro SD card with 16G
- Number of NICs: 5; 1 embed on board and 4 TP-Link USB 2.0 Ethernet UE200
- Power Source: 5.1V 2.5A Charger

A.4 *OF-Node*

This node is actually a VM deployed on IST OpenStack infrastructure, full list of system specifications e described below. His role is explained on chapter 4.2.2.

- Operative System: Ubuntu 16.04.6 LTS
- CPU: Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz

- Graphics card: Red Hat, Inc. QXL paravirtual graphic card
- RAM: 8G
- Disk size: 50 G
- Number of vNICs: 1

B

IP addresses of deployed nodes.

On this appendix it is presented a summary of all IP addresses, from both physical and virtual nodes.

B.1 Physical nodes

IP addresses of physical nodes:

- *controller*: 172.20.126.26/24 and 10.0.0.11/24
- *compute1*: 172.20.126.27/24 and 10.0.0.31/24
- *r2controller*: 172.20.126.176/24 and 10.0.0.11/24
- *r2compute1*: 172.20.126.177/24 and 10.0.0.31/24
- *RpiVSwitch*: 172.20.126.190/24
- *OF-Node*: Private IP: 100.68.11.169/27 and Floating IP: 192.92.147.19/24

B.2 Virtual nodes

IP addresses of virtual nodes:

- *R1-DataVM*: 192.168.167.3/24
- *R1-MgmtVM*: 192.168.167.22/24 and 172.20.126.110/24
- *R2-DataVM*: 192.168.243.14/24
- *R2-MgmtVM*: 192.168.243.6/24 and 172.20.126.202/24

C

Code of Project

On this appendix it is presented the most relevant excerpts of the code produced during the development of this work. Also, it is included some commands performed to enable desired functions.

Listing C.1: List of commands to add OpenStack environment on OSM

```
#Create OSM User, tenant and assign admin role
$ openstack user create --domain default \
  --password-prompt osm

$ openstack project create --description "OSM project" osm-project \
  --domain default

$ openstack role add --project osm-project --user osm admin

 #Setup VIM on OSM host
 $ osm vim-create --name openstack-site --user osm --password ***** \
    --auth_url http://controller:5000/v3 --tenant osm-project \
    --account_type openstack

$ osm vim-create --name openstack-site2 --user osm --password ***** \
    --auth_url http://r2controller:5000/v3 --tenant osm-project \
    --account_type openstack
```

Listing C.2: NSD of thesis-ns

```
nsd:nsd-catalog:
    nsd:
    -   constituent-vnfd:
        -   member-vnf-index: '1'
            vnfd-id-ref: thesis-vnf
        description: NS with 1 VNF thesis-vnf connected by internal VL
        id: thesis-ns
        logo: osm.png
        name: thesis-ns
        short-name: thesis-ns
        version: '1.0'
        vld:
        -   id: mgmtnet
            mgmt-network: 'true'
            name: mgmtnet
            short-name: mgmtnet
            type: ELAN
            vim-network-name: provider
            vnfd-connection-point-ref:
            -   member-vnf-index-ref: '1'
                vnfd-connection-point-ref: vnf-mgmt
                vnfd-id-ref: thesis-vnf
```

Listing C.3: Excerpt of VNFD of thesis-vnf

```

        vnf-configuration:
            config-primitive:
            -   name: touch
                parameter:
                -   data-type: STRING
                    default-value: /home/ubuntu/touched
                    name: filename
            -   name: python
                parameter:
                -   data-type: STRING
                    default-value: /home/ubuntu/file.py
                    name: filename
            initial-config-primitive:
            -   name: config
                parameter:
                -   name: ssh-hostname
                    value: <rw_mgmt_ip>
                -   name: ssh-username
                    value: ubuntu
                -   name: ssh-password
                    value: osm4u
                seq: '1'
            -   name: touch
                parameter:
                -   name: filename
                    value: /home/ubuntu/first-touch
                seq: '2'
            juju:
                charm: simple
```

Listing C.4: IPTables rules to forward traffic from outter networks to DataVM

```
# MgmtVM of Region 1
sudo iptables -t nat -A PREROUTING -p tcp --dport 80 \
-j DNAT --to-destination 192.168.167.3:80
sudo iptables -t nat -A POSTROUTING -j MASQUERADE

# MgmtVM of Region 2
sudo iptables -t nat -A PREROUTING -p tcp --dport 80 \
-j DNAT --to-destination 192.168.243.14:80
sudo iptables -t nat -A POSTROUTING -j MASQUERADE
```

Listing C.5: Excerpt of primary.py script

```
def start_server():
    host = '0.0.0.0'
    port = 5000
    backlog = 5
    size = 1024
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((host,port))
    s.listen(backlog)
    logger.info('Server listening on port %s', port)

    while 1:
        try:
            client, address = s.accept()
            logger.info('New client: %s', address)
            data = client.recv(size)
            if data and checkHTTPServer() == "ONLINE":
                client.send(data)
            else:
                client.send(str.encode('OFFLINE'))
            client.close()
            #logger.info('Client %s disconnected', address)
        except KeyboardInterrupt:
            s.close()
            print('Server shutting down!')
            logger.warning('Server shutting down!')
            sys.exit(1)
```

Listing C.6: Excerpt of replica.py script

```
def connect():
    heartbeat_miss = 0

    while 1:
        try:
            # Connect to server
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect((HOST, PORT))
            logger.info('Connected to %s on port %d', HOST, PORT)

            # Send Data
            #if checkHTTPServer() == "ONLINE":
            s.send(str.encode('Alive'))
            data = s.recv(size)
            s.close()
            if data.decode() == 'Alive':
                heartbeat_miss = 0
                print('Received from ', HOST, ': ', data.decode(),
                " \nMiss = ", str(heartbeat_miss))
                logger.info('Heartbeat received from %s \nMiss = %d'
                ,HOST, heartbeat_miss)
            else:
                heartbeat_miss+=1
                if heartbeat_miss >= HEARTBEAT_MAX_MISS:
                    print('REMOTE HTTPServer OFFLINE - REDIRECTING -')
                    logger.warning('REMOTE HTTP Server OFFLINE
                    - REDIRECTING -')
                    if checkHTTPServer() == "ONLINE":
                        print('Redirecting traffic')
                        addFlow()
                    else:
                        print('All regions unavailable')
                        logger.warning('All regions unavailable')
                    break
                else:
                    print('REMOTE HTTP Server missed: {}
                    times.'.format(heartbeat_miss))
                    logger.warning('REMOTE HTTP Server missed: %d times.',
                    heartbeat_miss)
                    pass

            # Sleep for HEARTBEAT_INTERVAL seconds
            time.sleep(HEARTBEAT_INTERVAL)

        except ConnectionRefusedError:
            heartbeat_miss += 1
            if heartbeat_miss >= HEARTBEAT_MAX_MISS:
                print('There has been some problem with the connection...')
                logger.warning('Connected to %s on port %d', HOST, PORT)
                if checkHTTPServer() == "ONLINE":
                    print('Redirecting traffic')
                    addFlow()
                else:
                    print('All regions unavailable')
                    logger.warning('All regions unavailable')
                break
            else:
                print('Server missed: {} times.'.format(heartbeat_miss))
                logger.warning('Server missed: %d times.', heartbeat_miss)
                time.sleep(HEARTBEAT_INTERVAL)
                pass
        except KeyboardInterrupt:
            s.close()
            print('Disconnecting!')
            logger.warning('Disconnecting!')
            sys.exit(1)
```

Listing C.7: Python3 script to verify HTTP Server status

```
#!/usr/bin/python3

import urllib3, requests
import json

def checkHTTPServer():
    http = urllib3.PoolManager()
    try:
        resp = http.request('GET', 'http://192.168.167.3:80')
        if resp.status==200:
            print("ONLINE")
            return ("ONLINE")
        else:
            print ("ERROR")
            return ("ERROR")
    except:
        print("OFFLINE")
        return ("OFFLINE")

checkHTTPServer()
```

Listing C.8: Content of pingall.sh and thesis-servers.txt

```

# File: pingall.sh

#!/bin/bash
cat thesis-servers.txt |  while read output
do
    ping -c 1 "$output" > /dev/null
    if [ $? -eq 0 ]; then
    echo "node $output is up"
    else
    echo "node $output is down"
    fi
done
# ------------------------

# File: thesis-servers.txt
controller
compute1
r2controller
r2compute1
osm
rpivswitch
r1-mgmtvm
r2-mgmtvm
```

D

OpenStack Map

On this appendix it is presented a map with all OpenStack services.

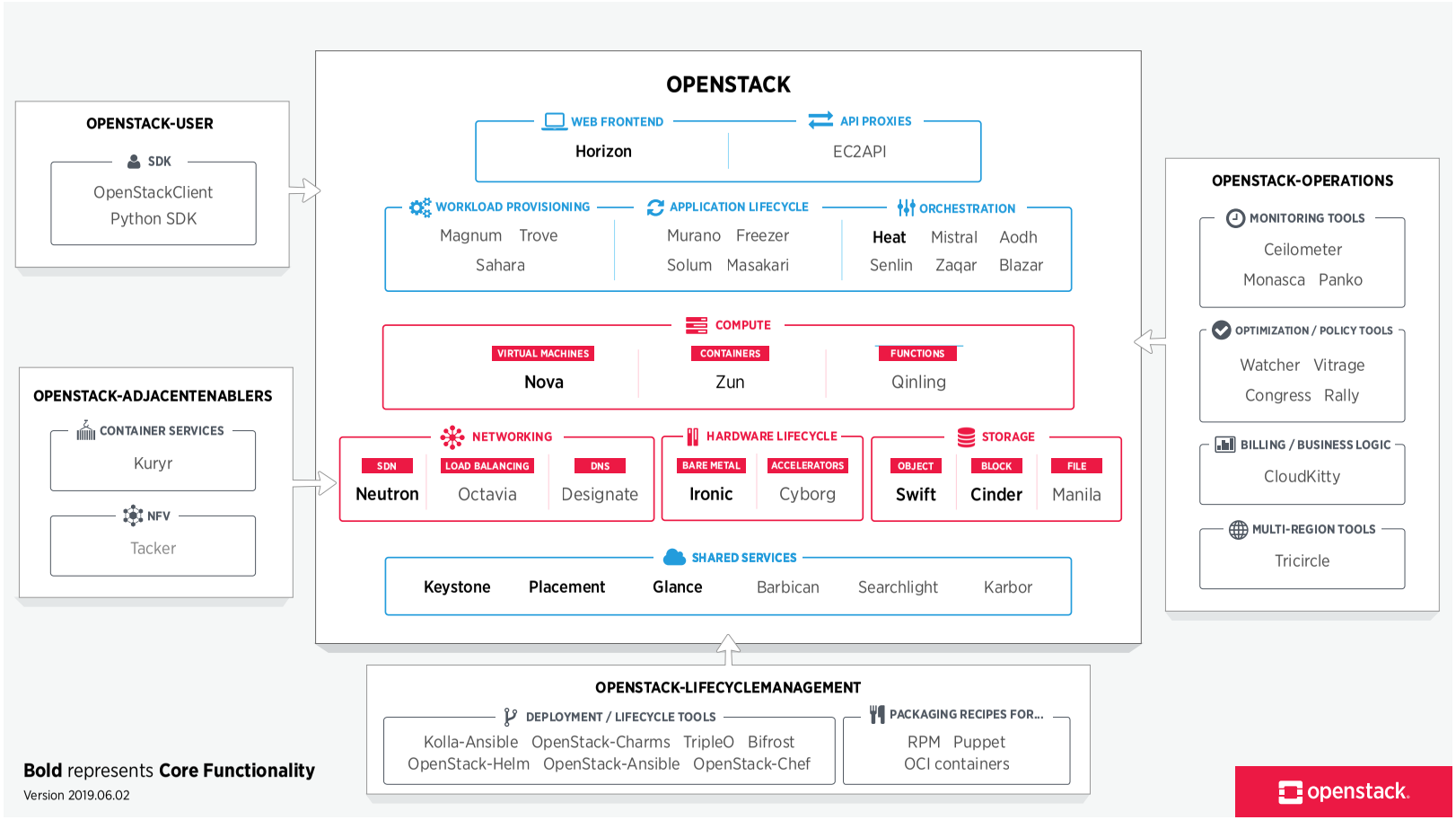

Figure D.1: OpenStack map of current services [4]