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

Filipe Emanuel Lourenço Ramalho Fernandes
Instituto Superior Técnico, Universidade de Lisboa

***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.**

***Index Terms*—NFV, SDN, OpenStack, Open Source MANO, High-Availability, Load-Balancing**

I. Introduction

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.

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

A. 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.

II. State of the art

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.

A. 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 [1], which runs on a standard computing server [2].

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 [3]. 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* [4] 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.

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* protocol.

Northbound API is the controller interface that communicates with higher-level applications. There is not a standard protocol defined for the NBI [5]–[7]. However, it is usually implemented through a REST API [8] 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.

B. 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 [9]. Virtual Network Functions (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), Virtual Private Network (VPN) gateways, Deep Packet Inspection (DPI) and Intrusion Detection System (IDS) [10].

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 [10] 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 [10]–[12].

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 [10], [11], [13], [14].

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. The ETSI architecture comprises three main domains [15]:

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 [16], [17]; 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 Network Service Descriptor (NSD) and Virtual Network Function Descriptor (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.

III. Architecture of the proposed solution

In this chapter, it is described the architecture of the proposed solution. As stated in chapter I, 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 Virtual Machine (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 Management and Orchestration (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.

A. IST Infrastructure

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

Both regions of the infrastructure have precisely the same core services and are symmetric. OpenStack services that support IST infrastructure are: Designate, Keystone, Nova, Neutron, Cinder, Glance, Horizon.

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 persistent data, and additional networking modules that support Neutron operation.

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 1. 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 1.

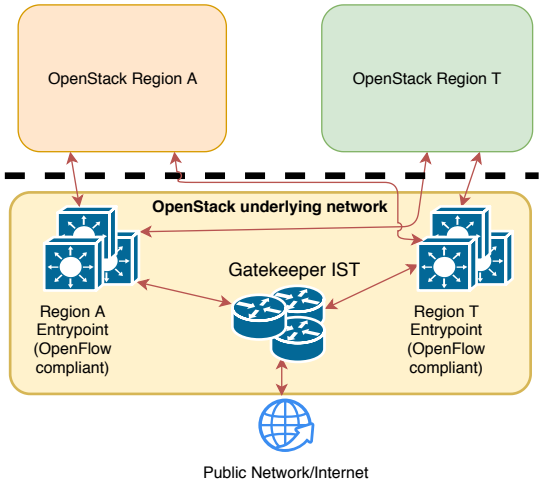

Fig. 1: Global view of IST OpenStack underlying network

B. Architecture

The proposed solution is presented in figure 2 and is further divided into physical and virtual architectures.

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 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.

C. Physical architecture

Each block that relies on a machine outside of OpenStack virtual environment belongs 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 Command Line Interface (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.

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

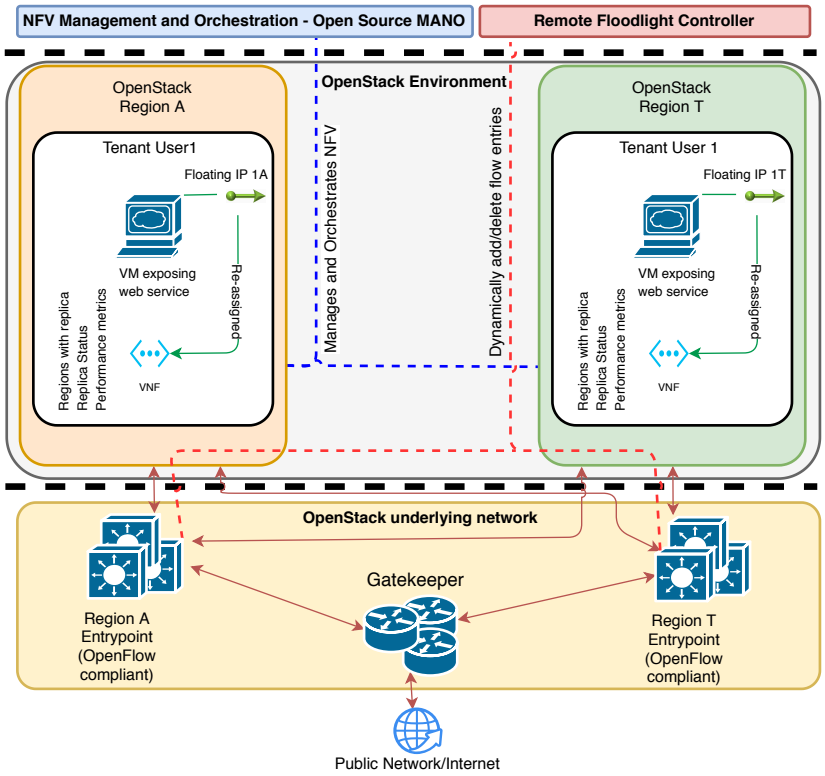

Fig. 2: Architecture of the proposed solution

OpenStack services: Keystone, Nova, Neutron, Glance and Horizon.

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 HyperText Transfer Protocol (HTTP) requests using Representational State Transfer (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. In this work, we only need to change the packets source and destination of IP and Media Access Control (MAC) addresses.

D. 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 it is presented only the virtual architecture and the underlying network is abstracted as a cloud.

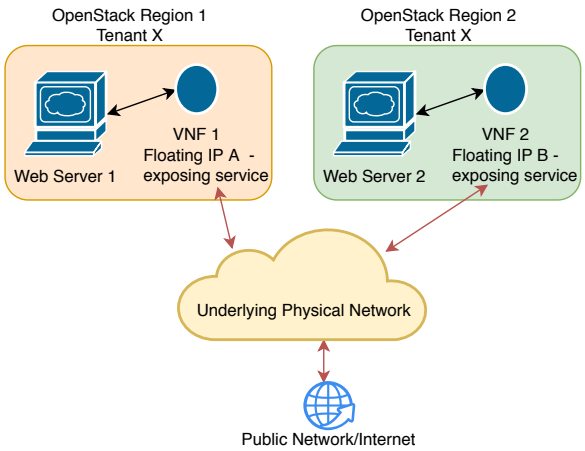

Fig. 3: Virtual Architecture

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 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 Service Level Agreement (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, Central Processing Unit (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 automatic fail-over between them.

IV. Implementation

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 III, 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 V.

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

A. OpenStack Infrastructure

The deployed test infrastructure is presented in figure 4. 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.

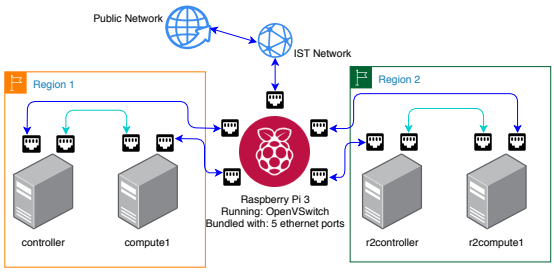

Fig. 4: OpenStack Infrastructure

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 section IV-C.

Each server node has two Network Interface Card (NIC)s, 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.

B. Physical Layer

From the physical point of view, the scenario represented in figure 5 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.

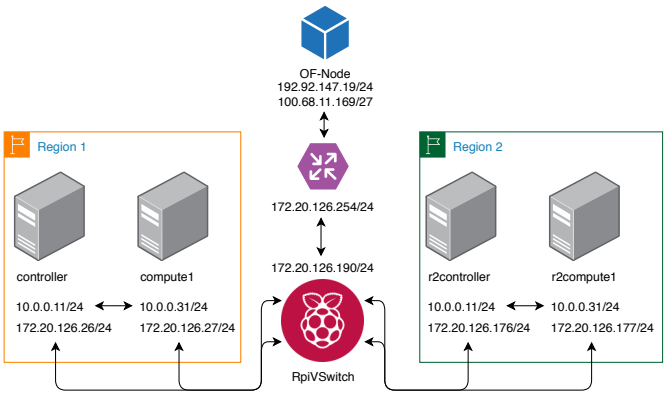

Fig. 5: Physical Network Topology

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). Although this node is a VM it belongs to physical architecture, only nodes running on the deployed OpenStack environment are considered virtual nodes.

Controller and compute nodes were already described before, RpiVSwitch and OF-Node are described below.

C. 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 Open vSwitch (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.

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.

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 reroutes them.

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

D. 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 network service is described using a NSD which 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.

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.

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.

E. Virtual Layer

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

In section IV-B 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 6.

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 which 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.

Each MgmtVM is responsible to verify if his peer DataVM is working as expected and exchange this information with MgmtVM from the other region.

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. Both nodes listen and replies to heartbeat messages.

On *primary.py* it is called a function named *check-HTTPServer*. This function makes a request to the web-server and verifies its operation. This function 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.

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.

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

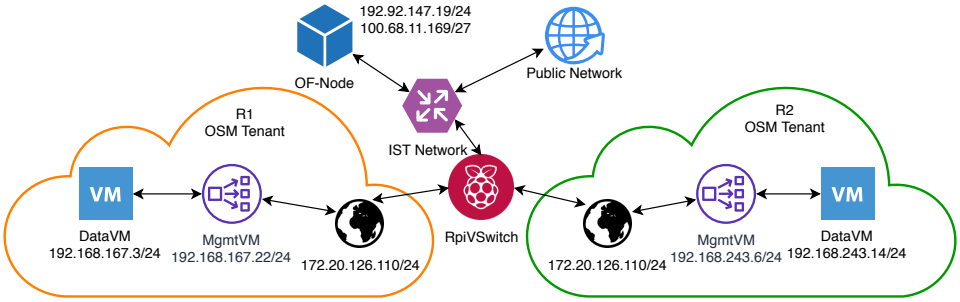

Fig. 6: Summary of the deployed scenario

V. Evaluation

In this chapter, it is presented some of the tests performed to test and evaluate the implemented solution.

As stated in the previous chapter, the system was deployed on DSI-IST Taguspark *campus* facilities.

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.

A. 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, during 60 seconds, and counts the number of requests, the number of successful and failed requests and the time of unavailability. On table I, 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.

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

TABLE I: Results of tests evaluating HA with different parameters T_h and n

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 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.

B. 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 II. In this scenario, each MgmtVM trigger TPR as soon as they forward one packet to DataVM.

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 III. 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.

In this case, the same request distribution issue occurs, there are more requests served in the primary region. Once again,

Time(s)	#requests	Success(%)	Fail(%)	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 II: Results of LB using packets per second policy

Time(s)	T_un (s)	R1 (%)	R2(%)	T_update(s)
78.5	0	45.8	54.2	5
80	0	46.5	53.5	0.5

TABLE III: Results of LB using number of active flows policy

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 with a success rate of 100%.

VI. Conclusion

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.

A. 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, were 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.

B. Future work

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.

References

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

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

[3] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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

[15] "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.

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

[17] 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.