POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master's Degree Thesis

Offering Cloud-Native Network Services to Residential Users

Supervisors

Prof. Fulvio RISSO

Prof. Guido MARCHETTO

Ing. Roberto MORRO

Candidate

Francesco PAVAN

December 2020

“Perseverance is the hard work you do after you get tired of doing the hard work you already did”
Newt Gingrich

Abstract

The recent spread of cloud technologies and new virtualization paradigms has created new possibilities in the way telecom operators offer network services to residential clients. The transition to a cloud-native-based context marked the evolution of a well-known technology in the world of computer networks: Network Function Virtualization (NFV). In a standard NFV scenario, Virtual Netowork Functions (VNFs) are deployed as virtual machines on physical servers; in a cloud-native NFV scenario, however, network functions, which are now called Cloud Native Network Functions (CNFs), run inside containers and are possibly managed through a container orchestrator system, such as Kubernetes. This evolution has forced telecom operators to also evolve their residential broadband access technologies, pushing them towards the adoption of a cloud-native model.

This thesis analyzes SDN-Enabled Broadband Access (SEBA), one of the most promising technologies regarding the cloud-native residential broadband access management, and its integration with the Network Service Mesh project, which enables the creation of multiple CNFs meshes in Kubernetes. The fusion of these two technologies allows telecom operators to offer its customers network services fully deployed in Kubernetes with CNFs.

In the first part of the work a brief analysis of CNFs in Kubernetes is provided, also addressing the problem of creating service chains with Network Service Mesh. However, NSM alone cannot be directly used to drive end-users' traffic inside a service mesh, since they does not run pods inside their devices. It is therefore necessary to find a way to bring cloud-native technologies closer to residential users.

One of the most prominent solutions to the problem is the SEBA framework, which leverages SDN, NFV, Kubernetes and other technologies to provide a cloud-native, open and programmable architecture to manage the central office and the access network of an internet service provider. A deep analysis of this framework is carried out, both from the point of view of the control plane and data plane, also explaining the differences between the broadband access management system currently used by operators and the one managed with SEBA.

Finally, this thesis presents also the technical details to integrate the two systems together, showing two different deployment scenarios and analyzing the performance impact of the fusion of the two systems.

Table of Contents

List of Tables

List of Figures

Acronyms

AAA

Authentication, Authorization and Accounting

API

Application Programming Interface

BSS

Business Support System

BNG

Broadband Network Gateway

CIDR

Classless Inter-Domain Routing

CLI

Command Line Interface

CNF

Cloud Native Network Functions

CNI

Container Network Interface

CO

Central Office

CORD

Central Office Re-architected as a Data Center

CPE

Customer Premise Equipment

CRD

Custom Resource Definition

DHCP

Dynamic Host Configuration Protocol

DNS

Domain Name System

FTTH

Fiber To The Home

GUI

Graphic User Interface

IDS

Intrusion Detection System

IPS

Intrusion Prevention System

IP

Internet Protocol

IPoE

IP over Ethernet

ISP

Internet Service Provider

K8s	Kubernetes
LAN	Local Area Network
M-CORD	Mobile CORD
MAC	Medium Access Control
NAT	Network Address Translation
NEM	Network Access Mediator
NFV	Network Function Virtualization
NIC	Network Interface Controller / Card
NSM	Network Service Mesh
OAM	Operations, Administration and Maintenance
ODN	Optical Distribution Network
OF	OpenFlow

OLT	Optical Line Termination
ONF	Open Networking Foundation
ONOS	Open Network Operating System
ONT	Optical Network Terminal
ONU	Optical Network Unit
OSS	Operational Support System
POD	Point Of Delivery
PON	Passive Optical Network
PONSIM	PON Simulator
PPP	Point-to-Point Protocol
PPPoE	PPP over Ethernet
QoS	Quality of Service

R-CORD	Residential CORD
RADIUS	Remote Authentication Dial-In User Service
RG	Residential Gateway
RPC	Remote Procedure Call
SDN	Software Defined Networking
SEBA	SDN-Enabled Broadband Access
SiaB	SEBA-in-a-Box
Veth	Virtual Ethernet
VLAN	Virtual LAN
VM	Virtual Machine
VNF	Virtual Network Function
VOLTHA	Virtual OLT Hardware Abstraction
VPP	Virtual Packet Processor

Chapter 1

Introduction

In recent years, cloud technologies, which previously were the prerogative only of large providers and companies that had the financial resources to create an on-premises environment, are spreading in all sectors of information technology, including the networking environment.

It all started with virtualization, which had as its objective the optimization of the number of servers used to deploy applications. When virtualization became common it began to spread to network functions as well, starting a process called Network Function Virtualization (NFV). Network functions, which are now called Virtual Network Functions (VNFs), became virtualized services running on open computing platforms instead of proprietary, dedicated hardware technology.

VNFs have introduced a number of advantages in terms of network programmability and services management, but they have also introduced some drawbacks. These drawbacks mainly concern the performance required by the devices where these functions are deployed, much greater than traditional network devices, in which software and hardware are highly optimized for the purpose.

A possible solution to this problem is represented by containers, which are based on a lighter form of virtualization that allows you to create multiple network functions even within the same virtual machine. Containers are less power consuming than virtual machines, deploy faster and large set of containers can be easily managed with container orchestrator systems, such as Kubernetes. Thus, thanks to these new virtualization techniques, Cloud Native Network Functions (CNFs), which are network functions that run inside containers, and cloud-native technologies are born.

All these new innovations have created new possibilities in the way telecom operators offer network services to residential clients and the evolution from VNFs, widely used by operators, to CNFs, very efficient but difficult to integrate into a non-cloud-native environment, has forced them to also evolve their residential broadband access technologies, pushing towards the adoption of a cloud-native model.

This thesis analyzes the SDN-Enabled Broadband Access (SEBA) framework, one of the most promising technologies regarding the cloud-native residential broadband access management. SEBA leverages SDN, NFV, Kubernetes and other mature technologies to provide a cloud-native, open and programmable architecture to manage the central office and the access network of an internet service provider.

Thanks to this cloud-native context extended to all the elements of the edge network of an internet service provider it is now possible to fully exploit the potential offered by Cloud Native Network Functions, since they can be integrated with the existing cloud-native architecture offered by SEBA.

The most valid way to exploit the potential of CNFs is Network Service Mesh, which enables the creation of multiple CNFs meshes in Kubernetes. In the last part of this work the integration between SEBA and NSM is analyzed in two different scenarios, because the fusion of these two technologies allows telecom operators to offer its customers network services fully deployed in Kubernetes with CNFs, improving at the same time the management of these services thanks to the advantages offered by the cloud environment.

Finally, the results obtained in the integration phase are summarized, also addressing the possible next steps to improve the integration of the two systems, and discussion about the performance of the framework is made.

Chapter 2

Background

In this thesis many technologies concerning the world of computer networks and residential access have been used. In this chapter they are explained in the shortest but at the same time effective way possible, to give a basic understanding of the working context.

2.1 SDN: Software Defined Networking

Software-defined networking is an approach to networking that uses software-based controllers or APIs to communicate with the underlying hardware infrastructure, to direct traffic on a network. This model differs from the one of traditional networks, which use dedicated hardware devices, such as routers or switches, to control network traffic. SDN can create and control an entire virtual network, or control a traditional hardware component, via software [1].

A typical SDN architecture, is made up of three parts, which may be located in different places:

- Applications: they communicate resource requests or information about the network as a whole;
- Controllers: they implement the control plane functionalities, use the information from applications to decide how to route a data packet;
- Networking devices (physical or virtual): they implement data plane functionalities, receive information from the controller and actually move the data through the network.

The key difference between SDN and traditional networking is the infrastructure: SDN is software-based, while traditional networking is hardware-based. Because the control plane is implemented in software decoupled from the hardware, SDN is much more flexible than traditional networking. It allows administrators to control the network, change configuration settings, provision resources, and increase network capacity, all from a centralized user interface.

Software Defined Networking offers the following advantages:

- Increased control with greater speed and flexibility: instead of manually programming multiple vendor-specific hardware devices, developers can control the flow of traffic over a network simply by programming an open standard software-based controller. Networking administrators also have more flexibility in choosing networking equipment, since they can choose a single protocol to communicate with any number of hardware devices through a central controller;

- Customizable network infrastructure: with a software-defined network, administrators can configure network services and allocate virtual resources to change the network infrastructure in real time through one centralized location. This allows network administrators to optimize the flow of data through the network and prioritize applications that require more availability;

- Robust security: a software-defined network delivers visibility into the entire network, providing a more holistic view of security threats. With the proliferation of smart devices that connect to the internet, SDN offers clear advantages over traditional networking. Operators can create separate zones for devices that require different levels of security, or immediately quarantine compromised devices so that they cannot infect the rest of the network. However, because software-defined networks use a centralized controller, securing the controller is crucial to maintaining a secure network.

SDN represents a substantial step forward from traditional networking, enabling a new way of controlling the routing of data packets through a centralized server.

2.2 Cloud Native Network Functions (CNFs)

Before starting to talk about Cloud Native Network Functions it is necessary to take a step back and give a brief definition of what are network functions (NFs) and virtual network functions (VNFs) [2].

Network functions (NF) are physical devices that process packets supporting a network and/or application services. Routers, switches and firewalls are network function examples. More generally a network function is a physical component which has two network interfaces, one interface is used to send traffic to the NF while the other interface is used to send it out of the function. What distinguishes one function from another is the functionality it implements, which is determined by the software that runs inside the hardware component: the application processes every packet that arrives at the function and, once it finished, it sends out the packet through the output interface. In traditional network functions hardware and software are closely coupled, making interoperability between devices from different manufacturers difficult and introducing some problems from the management point of view.

To solve these problems virtual network functions (VNF) were introduced. A virtualized network function (VNF) is an NF designed to run in a virtual machine, normally deployed on a common hardware infrastructure. Being applications deployed within a virtual machine, they inherit all the advantages introduced by virtualization, such as isolation, automated configuration and operations and multi-tenancy.

Moreover, they introduced some improvements with respect to normal NF, and some of them are listed below:

- Lower costs: VNFs reduce costs in purchasing network equipment via migration to software on standard servers;
- Service chaining: a uniform customer application or service is assembled from a set of interconnected VNFs, which can be deployed inside the same virtual machine;
- Scalability: scaling a network architecture made of virtual machines is faster and easier, and it does not require purchasing additional hardware.

This newer technology, which is nowadays the current de facto standard for standing up network functions in cloud environments, has not only led to advantages but also introduced new problems to be solved, mainly concerning the performance of the new network functions. In fact, due to the virtualization overhead introduced by the guest OS/kernel, host OS/kernel and hypervisor, VNFs require much more powerful servers than the standard hardware network functions. Moreover, long boot times under normal maintenance, failure restart, and burst scenarios can impact availability.

A solution to the performance problem is given by containers, which take advantage of a lighter virtualization paradigm and do not have all the overhead introduced by VMs. Containers introduced a new approach for developing and running applications in a cloud environment, called cloud-native approach, which is based on microservices and container orchestration systems, such as Kubernetes, in which CNFs run inside a pod (section 2.3).

Thanks to this new approach Cloud Native Network Functions are born. A CNF is a network function designed and implemented to run inside a container instead of a virtual machine. They inherit all the principles of the VNFs and of the cloud native model, introducing a number of advantages [2]:

- Smaller footprint compared to physical or virtual devices: this reduces resource consumption, with the potential savings allocated to applications and/or infrastructure expansion;

- Rapid development, innovation and immutability: CNFs should run in user space where it is easier, faster and less risky to develop and deploy new features and innovations. There is no need to interact with the stable and mature Linux kernel used by other components in the environment;

- Agnostic to host environments: bare-metal or VMs are the most common. Since VMs do not add value to CNFs, bare-metal could become the preferred host of choice.

CNFs and the cloud-native paradigm are spreading more and more also in the world of telecom operators and not only in the one of developers, leading them to search for cloud-native solutions also for the management of residential access networks, as will be seen in the next chapters.

2.3 Kubernetes (K8s)

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem [3].

When Kubernetes is deployed, it created a cluster. A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node. The worker nodes host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

Figure 2.1 shows a diagram of a Kubernetes cluster with all the components tied together.

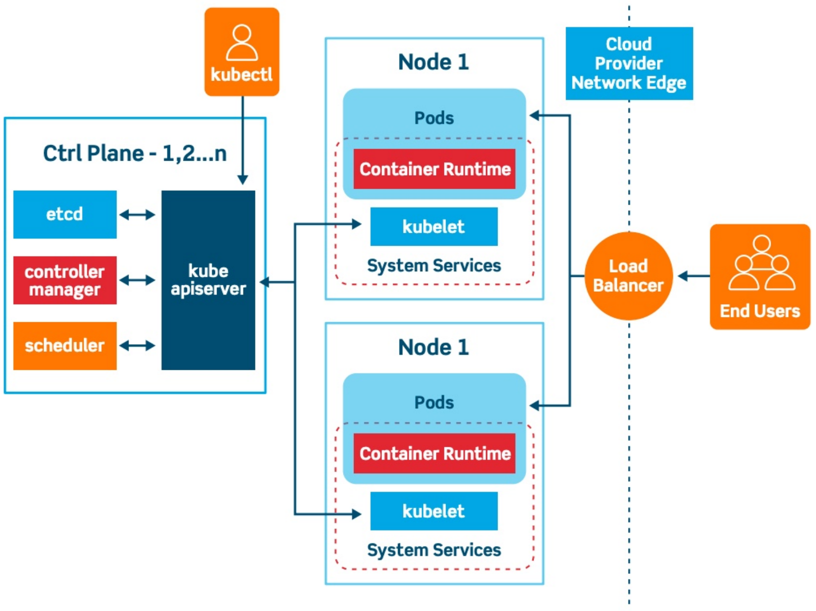

Figure 2.1: Kubernetes Main Components [4]

A Pod, which is the smallest deployable units of computing that you can create and manage in Kubernetes, is a group of one or more containers, with shared storage/network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled and run in a shared context.

A Pod models an application-specific *logical host*: it contains one or more application containers which are relatively tightly coupled.

2.3.1 Control Plane Components

The control plane's components make global decisions about the cluster, as well as detecting and responding to cluster events. Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine.

Components of the control plane are briefly described below:

- kube-apiserver: the API server exposes the Kubernetes API and represents the front end for the Kubernetes control plane. kube-apiserver is designed to scale horizontally - that is, it scales by deploying more instances. It is possible to run several instances of kube-apiserver and balance traffic between those instances;

- etcd: it is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data;

- kube-scheduler: it is the control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on;

- kube-controller-manager: it is the control plane component that runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process;

- cloud-controller-manager: it is the control plane component that embeds cloud-specific control logic to interact with the underlying cloud providers.

2.3.2 Node Components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

Node components are described below:

- kubelet: it is an agent that runs on each node in the cluster. It makes sure that containers are running in a Pod. kubelet doesn't manage containers which were not created by Kubernetes;
- kube-proxy: kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept. kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster;
- Container runtime: container runtime is the software that is responsible for running containers.

To interact with the cluster REST APIs are used. The API server exposes an HTTP API that lets end-users, different parts of your cluster, and external components communicate with one another, allowing the user to query and manipulate the state of the objects. Depending on the kind of request, the kube-apiserver can interact with other control plane components on master node, for example forwarding the request to them so that they can elaborate it, or interact directly with worker nodes. The interaction with worker nodes is made possible by the kubelet, which receives and reply to messages from kube-apiserver. There is never direct interaction from kube-apiserver to pods deployed in worker nodes, communication is always mediated by the kubelet.

2.4 PON: Passive Optical Network

A passive optical network (PON) [5], represented in Figure 2.2, is a fiber-optic network utilizing a point-to-multipoint topology and optical splitters to deliver data from a single transmission point to multiple user endpoints. Passive, in this context, refers to the unpowered condition of the fiber and splitting/combining components. Passive optical networks are often referred to as the *last mile* between an Internet service provider (ISP) and its customers.

In contrast to an active optical network, electrical power is only required at the send and receive points, making a PON inherently efficient from an operation cost standpoint. Passive optical networks are used to simultaneously transmit signals in both the upstream and downstream directions to and from the user endpoints.

A PON consists of an optical line termination (OLT) and a number of optical network units (ONUs). Normally, the OLT is placed at the provider's central office and the ONUs are put near end-users. Up to 32 ONUs can be connected to an OLT.

A PON system makes it possible to share expensive components for FTTH. A passive splitter that takes one input and splits it to broadcast to many users, which help cut the cost of the links substantially by sharing, for example, one expensive laser with up to 32 homes. PON splitters are bi-directional, that is signals can be sent downstream from the central office, broadcast to all users, and signals from the users can be sent upstream and combined into one fiber to communicate with the central office.

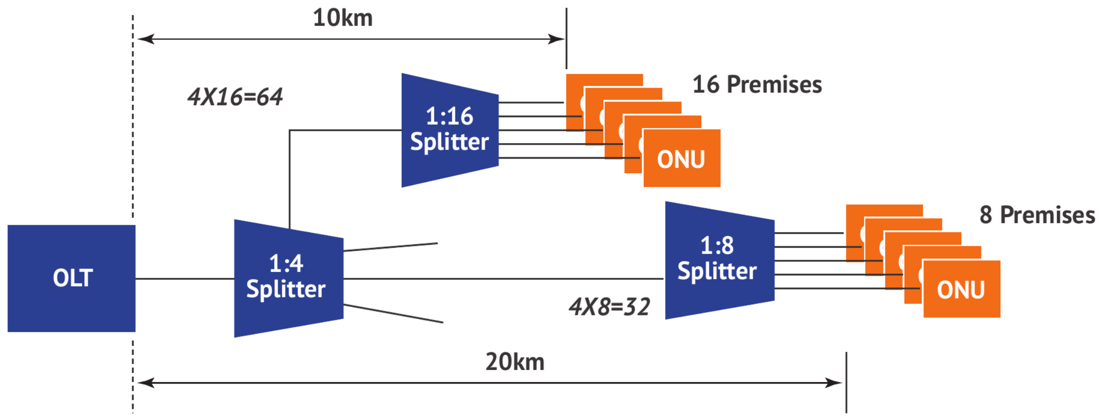

Figure 2.2: PON Main Components [6]

In addition to high energy efficiency, PON also has other advantages, such as a simple and upgradable infrastructure and high ease of maintenance. The main drawbacks are instead:

- Distance: this is the main drawback when PON is compared to active optical networks. The range for PON is limited to between 20 to 40 km, while an active optical network may reach up to 100 km;
- Test access: the PON cannot be isolated so test tools must allow in-service troubleshooting without disrupting service to other end-users on the same PON;
- Little redundancy: there is little redundancy in PONs and in case of an accidental fiber cut or a faulty OLT the service disruption can be extensive.

Overall, the inherent benefits of passive optical networks substantially outweigh these limitations.

2.4.1 OLT: Optical Line Termination

The OLT (Figure 2.3) is the starting point for the passive optical network, which is connected to a core switch (aggregation switch) through Ethernet cables [7]. The primary function of the OLT is to convert frames coming from the aggregation switch to optical signals and transmit towards the ONUs and to coordinate the ONUs multiplexing for the shared upstream transmissions, converting optical signals into Ethernet frames and sending them to the aggregation switch.

In general, OLT equipment contains rack, CSM (Control and Switch Module), redundancy protection, -48V DC power supply modules or one 110/220V AC power supply module and fans. In these parts, PON card and power supply support hot-swap while another module is built inside. The OLT has two float directions: upstream (getting distributing different types of data and voice traffic from users) and downstream (getting data, voice, and video traffic from the metro network or from a long-haul network and send it to all ONU modules on the Optical Distribution Network). The maximum distance supported for transmitting across the ODN is 20 km.

Figure 2.3: OLT Device

2.4.2 ONU: Optical Netowork Unit

The ONU converts optical signals transmitted via fibers by the OLT to electrical signals and vice-versa [7]. These electrical signals are then sent to individual subscribers. In general, there is a distance or other access network between ONU and end-user's premises. Furthermore, ONU can send, aggregate, and groom different types of data coming from the customers and send it upstream to the OLT. Grooming is the process that optimizes and reorganizes the data stream so it would be delivered more efficiently. ONU supports bandwidth allocation that allows making smooth delivery of data float to the OLT, which usually arrives in bursts from the customers. ONU could be connected by various methods and cable types, like twisted-pair copper wire, coaxial cable, optical fiber, or through Wi-Fi.

ONU may also be referred to as the optical network terminal (ONT), ONT is an ITU-T term, whereas ONU is an IEEE term.

2.4.3 ODN: Optical Distribution Network

The ODN provides the optical transmission medium for the physical connection of the ONUs to the OLTs, with 20 km or further reach. Within the ODN, fiber optic cables, fiber optic connectors, passive optical splitters, and auxiliary components collaborate with each other to provide internet access to end-user [7].

Chapter 3

Cloud Native Network Function Chains in Kubernetes

In this chapter the problem of creating CNF chains in Kubernetes is addressed and the most valid solution to this problem is briefly described, which is Network Service Mesh.

CNFs meshes are important, especially for telecom operators, because as in the NFV paradigm more VNFs are chained together in a specific order to offer one or more services, in a cloud-native environment the services are offered through the concatenation of several CNFs. Currently, however, the concatenation of VNF is easy to carry out and well established while the ordered concatenation of CNF in Kubernetes remains an open problem, since it is not natively supported by the framework.

The content of this chapter is freely drawn from Raffaele Trani's Master's Degree thesis, which analyzes the whole NSM project with a greater level of detail [8].

3.1 Network Service Mesh

Kubernetes default networking system is based on a Container Network Interface (CNI), which consists in a set of APIs that are introduced in K8s cluster and are used to provide network connectivity to the pods of the cluster. This system cannot solve the problem of creating service meshes, mainly due to the following two limitations:

- Lack of multiple network interfaces on pods: CNFs requires two network interfaces to properly work, one for traffic to get inside the CNF and one to get outside of it. Kubernetes should configure these interfaces in pods that implement CNFs, but it does not provide this behavior, limiting the configuration to a single interface (Multus CNI[1] solves this problem with a special configuration);
- Unsupported service chain implementation: no CNI allows the creation of pod chains, therefore it is not possible to natively create CNF chains. This is due to the fact that the CNI provides connectivity between pods and deploys necessary elements to allow routing not only inside the node but also between the nodes, but it does not establish how links between pods need to be created to implement a chain of CNFs.

Even if Multus CNI can provide more than one interface per pod the service chain implementation is still missing, so it is necessary to use the Network Service Mesh framework to solve the problem.

Network Service Mesh (NSM) is a novel approach to solve complicated L2/L3 use cases in Kubernetes that are tricky to address withing the existing Kubernetes Network Model. Inspired by Istio, Network Service Mesh maps the concept of a service mesh to L2/L3 payloads [9].

3.1.1 Main Concepts

Network Service Mesh offers the possibility to create service chains of pods in Kubernetes. This is made possible by two abstractions: the Network Service and the Cross-connection, which are represented in Figure 3.1.

Network Service

The network service is a Kubernetes Custom Resource Definition (CRD), an ad-hoc resource (standard resource are pods, deployments, services, etc.) which implements a user-defined function. Once a CRD is installed it can be created in the same way

[1] `https://github.com/intel/multus-cni`

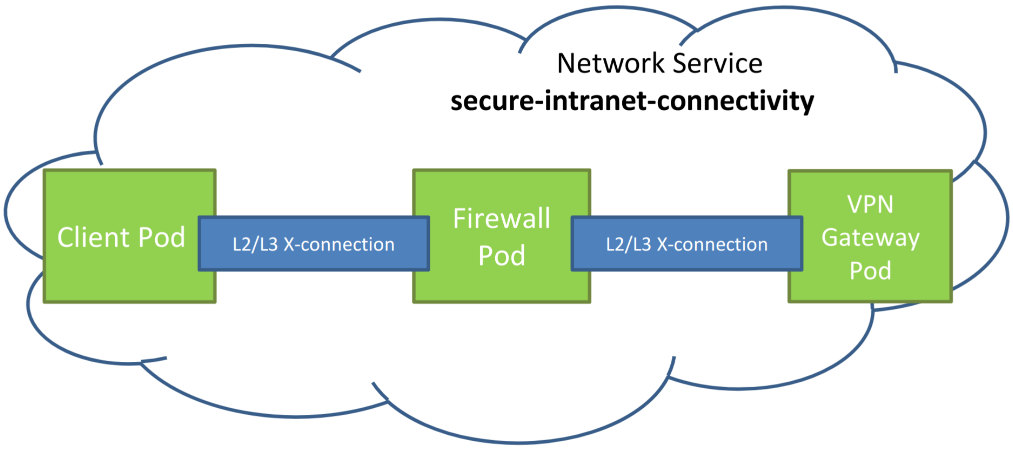

Figure 3.1: NSM Main Concepts representation [8]

as any other resource. A Network Service represents the logical implementation of a chain of CNFs implemented as pods in the cluster. If, like in Figure 3.1, a chain of CNFs is composed by a firewall and a VPN gateway, the Network Service would represent the chaining of these virtual network functions. The network service also specifies the order of CNFs, so traffic that traverses the chain will follow the order specified during the network service definition.

Cross-connection

A cross-connection represents a virtual wire which is created between two pods in the chain. Referring Figure 3.1, each link between two pods represents a cross-connection. Cross-connections are created by the control plane elements of NSM and are composed of two interfaces, configured and injected in pods belonging to the CNF chain by these same components. Network Service Mesh supports both L2 and L3 cross-connections, thus allowing Ethernet frames or IP packets to travel along the chain from pod to pod.

3.1.2 Main Elements

To properly create cross-connections and allow communications between pods, NSM provides both control plane and data plane elements. Figure 3.2 gives a graphical representation of these elements.

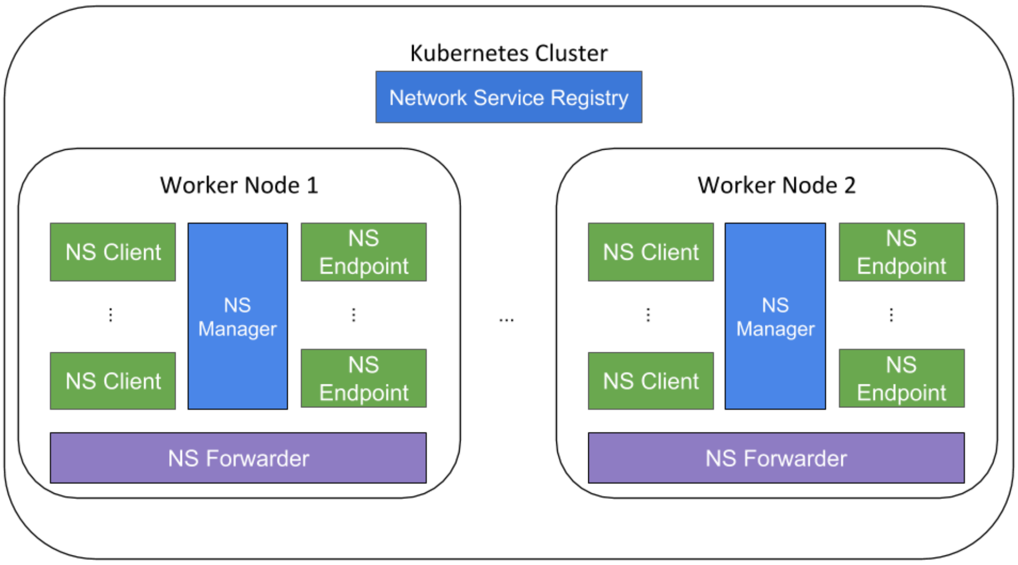

Figure 3.2: NSM Main Elements representation

Network Service Client

The Network Service Client is deployed as a pod in Kubernetes environment and its main aim is to request a cross-connection to a specific Network Service thanks to a container made on purpose (there may also be other containers inside the pod). This request is sent to the manager using gRPC and includes all the necessary parameters needed to allow the manager to understand which Network Service is required. Once the request is sent, the container awaits to receive a reply from the Manager. The process described above starts as soon as the pod is deployed in K8s, so that the cross-connection is immediately established and the main container of the NSC can communicate to the pods implementing the CNFs involved in the requested Network Service as soon as possible.

Network Service Endpoint

This element is in charge of implementing a CNF specified in a Network Service. Referring to Figure 3.1, it represents the firewall pod as well as the VPN gateway pod belonging to the Network Service named *secure-intranet-connectivity*. As for Network Service Client, also this pod can be composed by more containers, for example one implementing the NSM control plane functionalities and the other implementing the required cloud native network function. The control plane container receives request of cross-connection forwarded by NSM Manager from NSC. Once it receives the request, in may behave in two different ways: if it is not the last hop of the service chain it belongs to it will create a new request in order to create a cross-connection to the next hop of the chain, otherwise, if it is the last hop of, it will immediately elaborate the request and create an appropriate reply which then is sent to NSM Manager.

It is worth to note that control plane containers and CNF containers can work concurrently in the pod and, for this reason, it may be necessary for them to coordinate. There are two different ways to make them coordinate: the first one involves a specific configuration of the CNF container, so that it can work properly with the cross-connection built by NSM. The second way involves merging the two containers into a single container, which implements both NSM control plane functionalities and the CNF logic. In the latter case, the part of the container implementing the virtual network function does not need further configuration, as it receives direct instructions from the control plane part of the container.

Network Service Manager

Network Service Manager main aim is to receive and transmit all messages involved in control plane communications to build cross-connections. These messages are implemented by NSM using gRPC technology. First of all, it is implemented as a DaemonSet in Kubernetes environment, which means that a pod implementing this resource is deployed in each node of the cluster. In this way, each node has its own Network Service Manager, which guarantee that each pod in a node that is part of NSM will receive control plane messages. NSM Managers can interact with each other to exchange both direct messages and messages that need to be sent to other NSM elements in the node.

It is composed by three different containers:

- nsmd: this container is the heart of the NSM control plane implementation of Network Service Manager. It is in charge of elaborating all kind of requests involved in the cross-connections construction: it elaborates and possibly modify the requests coming from the Network Service Client and the Network Service Endpoint to allow their proper forwarding, it communicates with the NS Forwarder to actually create cross-connections and it monitors cross-connection-related messages, to get updates about cross-connections between pods registered with the NSM Manager.

- nsmd-k8s: this container manages the local registers and interfaces with the remote ones. It also communicates with Network Service Endpoints that are deployed on its same node, registering them to the NS Manager at their startup. It also communicates internally with nsmd container to interact with data stored locally and with NS Managers in other K8s nodes for data stored remotely;

- nsmdp: this last container is in charge of checking that all the elements involved in NS Manager functions are working properly. To do so, it interacts with them, periodically interrogating their state and, in case something is wrong, it provides necessary functionalities to correctly react to the event.

Network Service Forwarder

The main aim of this pod is to implement NSM data plane functionalities, in particular it is in charge of the creation and configuration of the additional interfaces and of the building of the cross-connection between involved pods.

It receives cross-connection requests from the NS Manager and, if all the parameters are correct and supported, it builds the cross-connection, by injecting the necessary interfaces into pods. Once the cross-connection is built, the forwarder advertises the NSM Manager about it, so that the latter can replies to the Network Service Client, which will then be able to use the cross-connection to communicate with the other pod. Network service Mesh provides two different implementations of this forwarder:

- VPP Forwarder: this implementation is the default one provided by NSM. In this case the forwarder behaves differently depending on the technology of the cross-connection indicated in the request that the forwarder receives from the NSM Manager. This forwarder can in fact support both *standard* technology and VPP technology[2] and it configures different kind of interfaces basing on the technology that is implemented in the pods involved in the chain;

- Kernel Forwarder: it is the forwarder used in this thesis. Differently from the previous implementation, in this case the forwarder supports only the construction of cross-connection which consists of a *veth* pair between the

[2]`https://wiki.fd.io/view/VPP/What_is_VPP%3F`

two pods involved in the communication. To set up the cross-connection, the forwarder uses the NetLink Linux library[3], which not only injects *veth* interfaces in pods, but it also configures appropriate entries in the routing tables to allow pods to reach each other.

NSM Admission Webhook

In a Kubernetes cluster, normally, when the api-server receives a request of pod creation, it will immediately elaborate the request, sending it to the proper elements inside the master node. However, the api-server can be configured to work in a slightly different way: when the api-server receives a request of pod's creation, instead of elaborating immediately the request, it will previously send the request to a chain of controllers, called webhooks, which analyze the request and can perform some action on the request itself, modifying or even rejecting it.

In this optic, NSM provides its own admission webhook: this elements intercepts pod creation requests directed to the api-server and, depending on its configuration and annotations present in the YAML file, it can modify the request, injecting specific code in the file associated to the request. This element is necessary to allow a client pod to be able to communicate with the other pods involved in NSM, because the Admission Webhook injects the container which begins the process of connecting the new pod to the NS Manager.

3.1.3 NSM Operations

Now that all the main elements have been presented, it is possible to briefly describe the operations of Network Service Mesh in a simple use case, taking into consideration the deployment of NSM on a single worker node and a pod chain made of a single CNF. For a more detailed analysis always refer to Raffaele Trani's Master's Degree Thesis [8].

Once the network service has been defined and the pods that implement the NS Client and the CNFs (NS Endpoints) of the chain have been deployed through a set of YAML files that make up a Helm[4] *chart*, the system is ready to start.

First of all, NS Endpoints register themselves with the NS Manager installed on

[3]`https://man7.org/linux/man-pages/man7/netlink.7.html`

[4]`https://helm.sh`

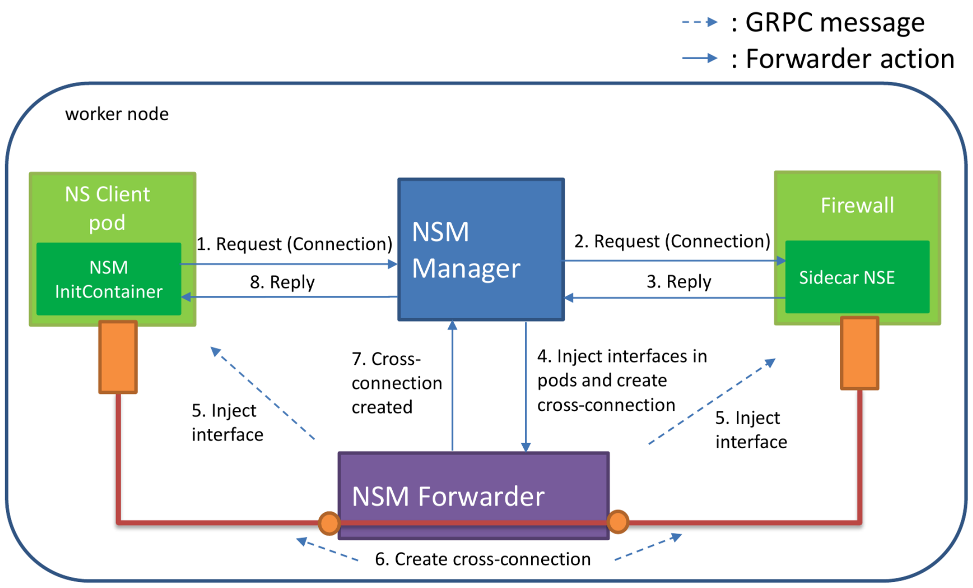

Figure 3.3: Cross-connection to one NS Endpoint [8]

the node with a gRPC message, containing information about the pod's role in NSM. If the registration is successful, local NS Manager will know that in its node there is a pod implementing a CNF function of that specific Network Service and it will send it requests directed to that pod if necessary.

Then, the cross-connection building step begins. The NS Client sends a gRPC message to the NS Manager containing the cross-connection request to a Network Service. When the NS Manager receives the request it tries to find the Network Service specified in the request and, if present, it identifies which CNF is required by looking in the local registry. Then, it sends the request to the selected NS Endpoint, that analyzes it to understand if the parameters associated to the request (i.e. specific required technology of cross-connection) can be satisfied. If so, the NS Endpoint replies to the NS Manager which in turn contacts the NS Forwarder sending the specific instructions and parameters about the cross-connection that needs to be built.

Then, the NS Forwarder injects the required interfaces in pods involved in the chain and, depending on the selected technology, it may also create interfaces on itself and properly configure them to receive traffic from newly injected interfaces and forward it between them. Once the interfaces are injected and properly configured, forwarder creates the cross-connection and advertises the NS Manager that the process is completed.

Finally, once the NS Manager receives confirmation about the creation of the cross-connection, it advertises the NS Client that the cross-connection is created and it can be used to communicate with NSEs involved in the Network Service.

Figure 3.3 gives a graphical representation of the steps explained above to create a cross-connection.

3.1.4 Firewall Use Case

In this section the NSM use case used for the development of this thesis is briefly illustrated, which is represented in Figure 3.4.

In this scenario three pods are deployed[5]: NS Client and NS Server, which are made of a simple Ubuntu container with no additional configurations and the Firewall, which is an Ubuntu-based container with the addition of one or more iptables[6] rules to implement firewall functionalities.

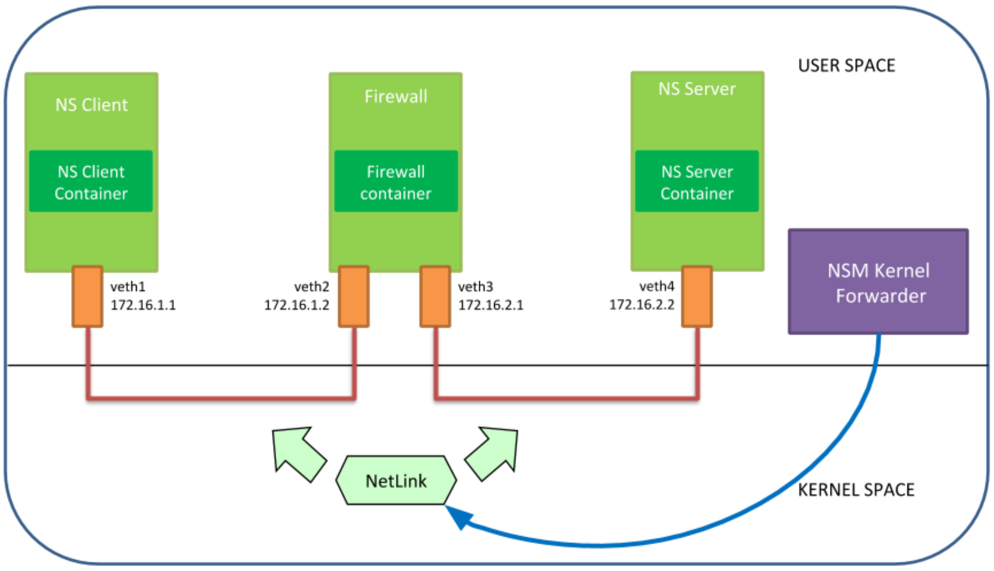

Figure 3.4: Firewall (iptables) Use Case

[5]Helm charts to reproduce this configuration are available here: `https://github.com/PPaviii/my-nsm`

[6]Linux Man Page of iptables: `https://linux.die.net/man/8/iptables`

As can be seen from Figure 3.4, NSM kernel forwarder was used: this element configures *veth* pairs between pods involved in communications using the NetLink Linux library, which provides all the necessary functions to properly configure and inject the interfaces. Thus, the forwarder is not directly involved in the communication, it just provides the cross-connection between pods involved in the communications, without configuring further interfaces.

IP addresses assigned to each *veth* pair are different and they are isolated from each other. For this reason, it was necessary to manually configure routing tables entries in the NS Client and the NS Server, to allow each one of them to reach the other one. On the other hand, no manual configuration of routing table was needed inside the firewall pod, because automatic routing tables were injected during its creation, so that it could reach both *veth* pairs. As said before, the firewall application consists of one or more iptables rules injected in the kernel.

3.2 NSM Advantages and Limitations

Network Service Mesh project is nowadays the most promising solution to allow integration of Cloud Native Network Functions in Kubernetes. After this brief analysis some positive aspects can be pointed out:

- Chaining of pods: once the chain is established and pods implementing CNFs are deployed, the cross-connections really allows the communication between pods and the order of CNFs in the chain is respected;
- Self-healing system: NSM comes with a full working monitoring system, which does not only concerns with cross-connections' state, but also with pods involved in NSM environment. Specifically, if a cross-connection for some reasons fails, the NSM Manager restarts the creation process to substitute the failed one. On the other hand, if a pod implementing a CNF stops working, the NSM Manager tries to create a cross-connection with the replica pods that implement the same CNF or, if no replicas are present, with the new same pod as soon as it is available.

Besides positive aspects, there are also some important limitations:

- Support for dynamic change of service chain: in a CNF service chain is sometimes necessary to dynamically change the chain but NSM does not supports this aspect;

- Load balancing: NSM does not provide load balancing between NSEs that are replicas of the same CNF. This means that all the cross-connections created by the NS Manager and the NS Forwarder involve only on replica of the CNF, not balancing requests to other possible pods which implements the same CNF.

One last important limitation, in particular regarding the use of NSM in the context of telecom operators, is that it cannot be directly used to drive end-users' traffic inside a service mesh, since they do not run pods inside their devices. A possible solution to this problem is to deploy Network Service Mesh in the central office of an operator, in order to filter the traffic that arrives from users without forcing them to use Kubernetes pods.

Unfortunately, in the context of the central offices of the majority of operators, NSM would be the only cloud-native technology, making the flexibility and scalability benefits brought by the cloud environment and the advantages brought by Network Service Mesh very limited. It is therefore necessary to find a way to bring cloud-native technologies closer to residential users, in order to make better use of the NSM framework.

Chapter 4

SEBA: SDN-Enabled Broadband Access

The advent of the cloud-native paradigm and the birth of new technologies, such as Network Service Mesh, which use the advantages of the cloud environment has forced telecom operators to also evolve their residential broadband access technologies, pushing them towards the adoption of a cloud-native model to manage their resources.

In this chapter the SDN-Enabled Broadband Access (SEBA) framework is analyzed in detail, which is one of the most promising technologies regarding the cloud-native residential broadband access management. However, before analyzing the framework, a brief description of how residential access network is currently managed and of SEBA's ancestor (CORD) is presented.

4.1 Standard Residential Access Management

Central offices (COs) are the central part of the telecommunication infrastructure of a telecom operator. They are the part of the edge network where subscriber homes and business lines are connected to the backbone network. From here the traffic is then aggregated, sent to the network of the ISP and finally forwarded towards Internet destinations.

Figure 4.1 represents the typical residential access network of a telecom operator, which is composed by a PON network (section 2.4) with three ONT, one optical splitter and three OLT devices, two aggregation switches and two Broadband

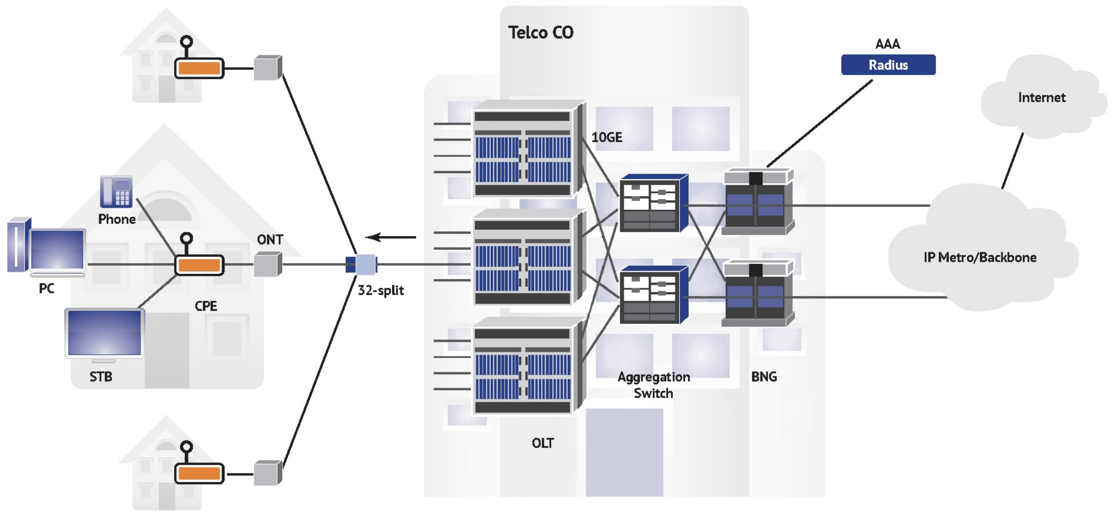

Figure 4.1: Residential Access Network [10]

Network Gateways (BNG). These last two devices play an important role in the forwarding of traffic from end-users to the Internet, so their operations are briefly explained below.

Aggregation Switch

The aggregation switch is the fundamental component of the aggregation network present in the central office. The main purpose of this component is to aggregate the large amounts of traffic coming from the OLTs into a smaller number of data streams and to send them to the BNG. Normally, more switches than necessary are deployed within the central office and are managed with the VRRP protocol, to prevent any packet loss or malfunction. The switch must provide the following capabilities [11]:

- A means to prioritize traffic in order to handle congestion at points of over-subscription;
- Multicast deployments support;
- Quality of Service (QoS) support;
- High availability (e.g. by multi-homing) support;
- Multiple VLANs aware bridging support, according to 802.1ad principles;

- Maintain user isolation.

These features are not only required by the switches but must be supported throughout the entire aggregation network they make up.

Broadband Network Gateway (BNG)

Broadband Network Gateway (BNG) connects residential subscribers to the broadband network of an Internet Service Provider. It is an active part of this broadband network, also participating in routing protocols. CPEs, which are also called Residential Gateways (RGs), and the BNG establish a direct connection, through which the subscriber can access the broadband services provided by the Internet Service Provider [12]. Beyond the BNG, traffic can be correlated to a subscriber using the IP address, but the complete view is lost.

BNG establishes and manages subscriber sessions. When a session is active, BNG aggregates traffic from various subscriber sessions from the aggregation network, and routes it to the network of the service provider, managing the following subscriber management functions:

- Connecting with the RG: BNG connects to the RG through a multiplexer. The three user devices pictured in Figure 4.1 represents the represent the three most common types of traffic: voice, video, and data. The individual subscriber devices connect to the RG;

- Establishing Subscriber Sessions: each subscriber device (the application which is executed on the RG) connects to the network by a logical session. Based on the protocol used, subscriber sessions are classified into two types, PPPoE and IPoE;

- Interacting with the RADIUS Server: BNG exploits an external Remote Authentication Dial-In User Service (RADIUS) server to provide Authentication, Authorization, and Accounting (AAA) functionalities to end-users. During the AAA process, BNG uses the RADIUS protocol to authenticate a subscriber before establishing a subscriber session, to authorize the subscriber to access specific broadband services or resources and to track the usage of this services for accounting or billing purposes. This server contains a database with all the subscribers' information of an Internet service provider, and provides to the BNG subscriber data updates. BNG, on the other hand, to keep data updated, sends to the RADIUS server session usage (accounting) information;

- Interacting with the DHCP Server: BNG relies on an external Dynamic Host Configuration Protocol (DHCP) server for address allocation and to correctly configure clients' devices. The DHCP server contains an IP address pool, from which it allocates addresses to the residential gateways.

The collaboration of all these devices allows a telecom operator to offer Internet access and network services to residential users but as can be seen from this brief explanation this model is very static and difficult to integrate with a cloud-native environment, due to the lack of programmability of the infrastructure. It is therefore necessary to find a new model that uses more recent technologies to achieve this integration, without however upsetting the operator's network architecture.

4.2 CORD: Central Office Re-architected as a Data Center

The first attempt to design and develop a platform that allowed central office management in a more similar way to the cloud-native paradigm was made by the Open Networking Foundation (ONF) and produced the CORD project, which stands for Central Office Re-architected as a Data Center. The main aim of the project was to transform the central office into an agile service delivery platform enabling the operator to deliver the best end-user experience along with innovative next-generation services.

The CORD framework presents a reference implementation which is open and complete, integrating everything needed to create a fully operational edge data center with built-in service capabilities. The implementation is all based on commodity hardware and white box switches and follows the latest cloud-native design principles [13].

The implementation specifies four architectural requirements [14]:

- Commodity servers and white box switches: the framework must be based on commodity hardware and leverage merchant silicon as mush as possible. There must not be dependencies on proprietary or specialized hardware to reach high performance. This requirement is dictated by goal of supporting cloud infrastructure economics and, by implication, the software running on that commodity hardware must deliver the same performance and reliability as today's purpose-built hardware;

- Support of a wide range of services: the framework must not be limited to access services and must not constrain the implementation process of services. In particular, it must support services belonging to the following four fields: access services and conventional cloud services, data plane and control plane services, operator-provided and third-party services and bundled legacy services. Finally, CORD must manage services without knowing their implementation details or how these services implements isolation, high, availability, scale and performance;
- High configurability: CORD is by definition a configurable framework, not a ready to use solution. An operator must have the possibility to specify the services it wants to deploy and how to manage them. Thanks to this configurability feature CORD can be used to manage different markets and all the three access technologies: residential, enterprise and mobile;
- Support of multiple domains of trust: in CORD it is not sufficient to only distinguish between CORD users and CORD operators, but it is necessary to include a wider range of intermediate roles, for example including global operators, site-specific operators, etc.;
- Support for partial and intermediate failures and incremental upgrades: CORD is a system built by integrating multiple, independently developed and deployed software components so it must support incremental upgrades and partial/intermediate failures;

CORD runs on commodity servers and white-box switches, coupled with disaggregated packaging of media access technologies. These hardware elements are then organized into a rackable unit which exploits a leaf-spine switching fabric, called POD, that is suitable for the deployment in a telecom operator central office. White box switches implements the data plane functionalities of the elements analyzed in section 4.1 (OLT and aggregation switch), while the control plane software runs on the commodity servers. The disaggregation of the control plane and data plane functionalities of the various devices is a key point to understand the new architecture, so it is analyzed in more detail in the next sections, since the same principle is also exploited by SEBA (section 4.3.1). Figure 4.2 represents a possible configuration for a reference implementation of CORD.

As far as software components, CORD exploits four different open source projects [15]:

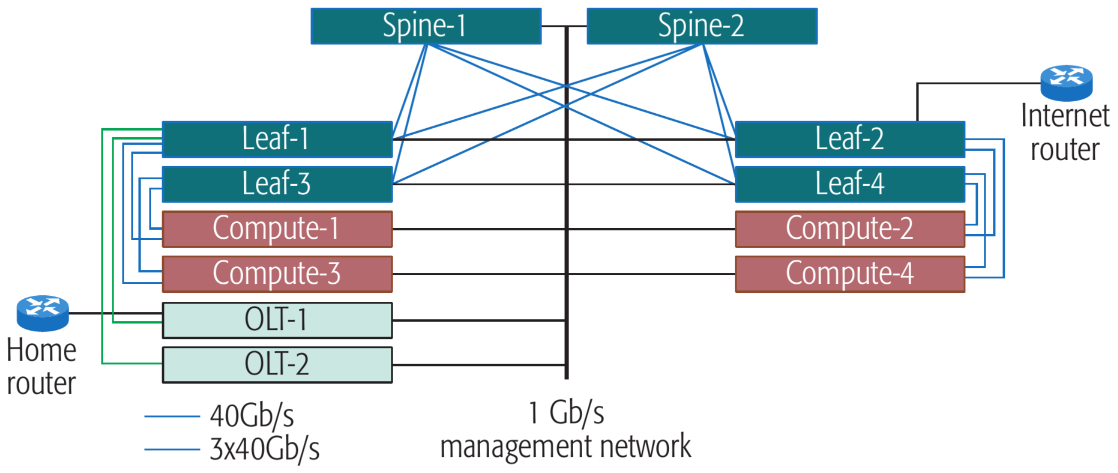

Figure 4.2: CORD Possible Reference Hardware Architecture [15]

- OpenStack: it is the framework which provides the Infrastructure as a service (IaaS) capability, and is responsible for creating, provisioning and managing virtual machines and virtual networks;
- Docker: services are deployed as Docker container and interconnected with Docker features. It also plays a role in deploying CORD itself (e.g., some management elements are instantiated in Docker containers);
- ONOS: it is the network operating system which is in charge of managing both software switches and the physical switching fabric. All the subscriber services but also the switching fabric itself are managed through applications hosted in ONOS;
- XOS: it is a framework for assembling and composing services, based on the Everything as a Service principle. It unifies OpenStack, which provides the infrastructure services, ONOS, which provides the control plane services, and all the data plane or cloud services running in VMs or containers.

To support the widest possible number of services, the reference implementation supports services running in VMs, in containers running directly on bare metal, and in containers deployed inside VMs. ONOS plays two roles in CORD: it interconnects VMs implementing virtual networks and manages flows across the switching fabric and provides a platform for hosting control programs that implement CORD services.

Finally, each unique CORD configuration is defined by a *Profile*. It consists of a set of services (e.g., access services, VNFs, other cloud services, etc.), including both abstract services on-boarded into XOS and SDN control apps running on ONOS. The two main profiles are M-CORD and R-CORD, which are CORD-based solutions to manage, respectively, mobile wireless access networks and ultra-broadband residential services. The latter is briefly described in the next section, as it was starting point for the development of SEBA.

4.2.1 R-CORD: Residential CORD

To deliver ultra-broadband residential services using the CORD platform and principles R-CORD was born. R-CORD is an open source solution which aims to transform the edge of an internet service provider's network into an agile service delivery platform, to allow operators to exploit innovative next-generation technologies to deliver the best end-user experience [16].

Compared to the standard version of CORD, R-CORD also exploits the disaggregation/virtualization of the residential user's RG. This device runs a collection of essential functions (e.g., DHCP, NAT) and optional services (e.g., Firewall, Parental Control, VoIP) on behalf of residential subscribers. By extending the capabilities of the RG in the cloud, new added-value services as well as customer care capabilities can be provided more easily.

The virtualized RG runs a bundle of functions selected by the subscriber, but it does so on commodity hardware located at the Central Office rather than on the customer's premises. There is still a device at home (which is still called RG), but it can be reduced to a bare-metal switch, with most of the functionality that ran on the original RG moved into the central office and running in a virtual compute instance (e.g., a VM or container) on commodity servers. In other words, the *customer LAN* includes a remote VM or container which resides in the central office, effectively providing every subscriber with direct ingress into the telecom operator's cloud.

4.3 SEBA Main Components

SEBA is a lightweight platform based on a variant of R-CORD, born from an initiative of AT&T and the ONF. The main difference between the two systems is that SEBA does not exploits the virtualization of the residential gateway, which

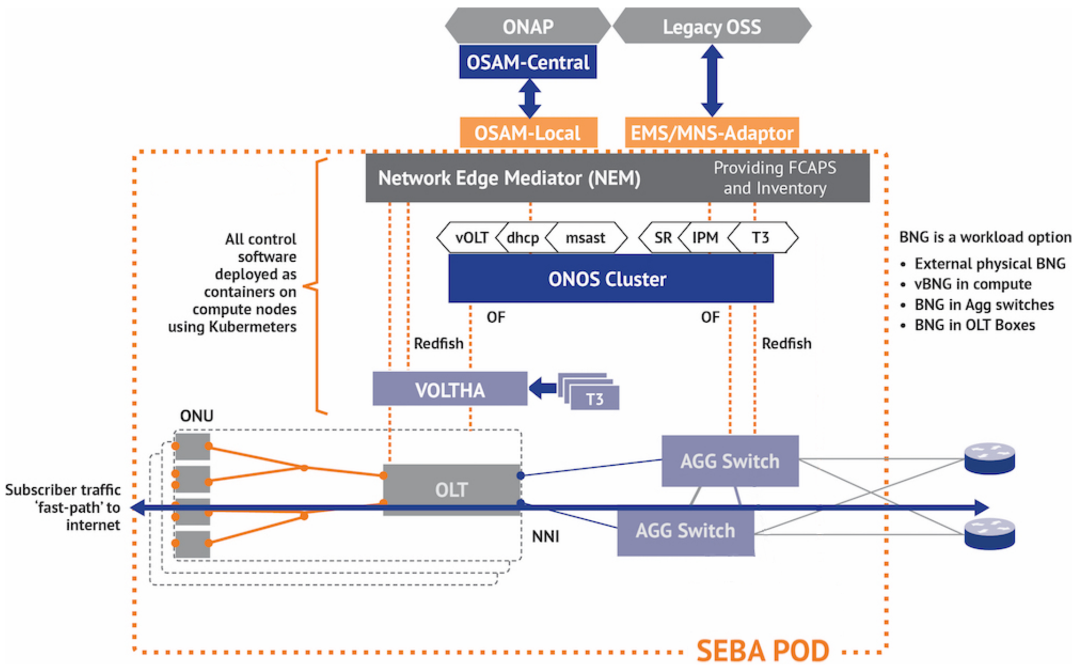

Figure 4.3: SEBA Hardware and Software Architecture [17]

caused a considerable overhead. In SEBA dataplane traffic for a subscriber just goes through the hardware and out to the Internet, creating a *fast-path* to the Internet for subscribers' traffic. This fast path stays in hardware and only goes out to compute nodes when necessary, for example, when a subscriber needs to exploit one or more third-party services deployed in compute nodes. By comparison, in R-CORD, the traffic that comes from a residential subscriber does not always stay in hardware, in fact it has to go to compute nodes trough a virtual switch to visit the virtualized residential gateway. Once finished, traffic goes back to the hardware and out to the Internet [17].

SEBA inherits all the principles of CORD (it runs on commodity servers and white-box switches, it is a configurable platform, etc.) but uses the latest cloud technologies, such as Kuberentes. In fact, SEBA is designed as a set of container elements which run in a Kubernetes environment. The system is modularized per typical microservice system architectures, and there is a hierarchy of modularity used to allow flexible compositions at different scales [18].

Figure 4.3 represents SEBA's hardware and software architecture, including the main elements of a PON network (ONU and OLT), the aggregation switches of the central office and all the software components which implement the control plane of the system and run on compute nodes. In the picture are missing the residential gateway, since it is no more virtualized, and the BNG, because SEBA can operate with both an external physical BNG and a virtual BNG, with the control plane software implemented in the compute nodes and the data plane functionalities in the aggregation switches, which now support aggregation, switching and routing of data plane and control plane traffic within the SEBA Point Of Delivery (POD).

4.3.1 Hardware Components Disaggregation

The first step to transform the today's central office following the principles of SEBA is to disaggregate and virtualize the devices, that is, turn each purpose-built hardware device into its software counterpart running on commodity hardware [15]. The Ethernet aggregation switch is not virtualized because the switching fabric, under the control of ONOS, effectively replaces it.

OLT Disagreggation/Virtualization

OLTs represents a large financial investment, consisting of a large amount of closed and proprietary hardware equipment, used to terminate access for tens of thousands of subscribers. The virtualization of an OLT devices is especially challenging because, unlike network functions that are actually implemented by software applications which run on commodity servers, OLTs are implemented primarily in hardware. Network traffic is currently distributed to tens of thousands of customer sites per CO, making them a significant operational burden.

OLT terminates the optical link coming from the splitters in the central office, with each physical termination point aggregating a set of subscriber connections. The first step is to create an I/O device with the PON OLT medium access control (MAC) and all the merchant silicon chips, which is under the control of an OpenFlow application deployed in ONOS. Figure 4.4 shows this new I/O blade and the difference with a standard device is immediately evident, especially in terms of size.

These boxes are then brought under the same SDN-based control paradigm as the white-box based switching fabric. The control plane application, which is called virtual OLT (vOLT) and runs on top of ONOS, implements all the functionality normally contained in a legacy OLT chassis: it manages per-subscriber authentication, establishes and manages VLANs connecting subscriber's devices to the central office and finally manages the other control plane functions of the OLT.

Figure 4.4: Disaggregated OLT

BNG Disaggregation/Virtualization

As previously said, the virtualization of the BNG is not strictly necessary in SEBA, but it is reported here for completeness.

BNG can be decomposed into the following control plane (CP) and user plane (UP) functionalities:

- Control plane: Authentication, Authorization, and Accounting (AAA) and session control;
- User plane: per subscriber packets and flow processing.

The routing Control Plane (CP) as well as User Plane (UP) are not dependent on the per-subscriber functions and can thus become separated. By doing so leads to a look at a BNG as being split into a subscriber-facing functions which is called Service Edge (SE) and a router. The SE component can be now implemented as a VNF or CNF in compute nodes while the router part can be embedded in the aggregation switches.

Further steps of disaggregation are also possible. For example, control and user plane can be split to allow the User Plane to stay on specialized hardware. This further division enables service providers to scale both layers independently of each other and to centralize the control plane.

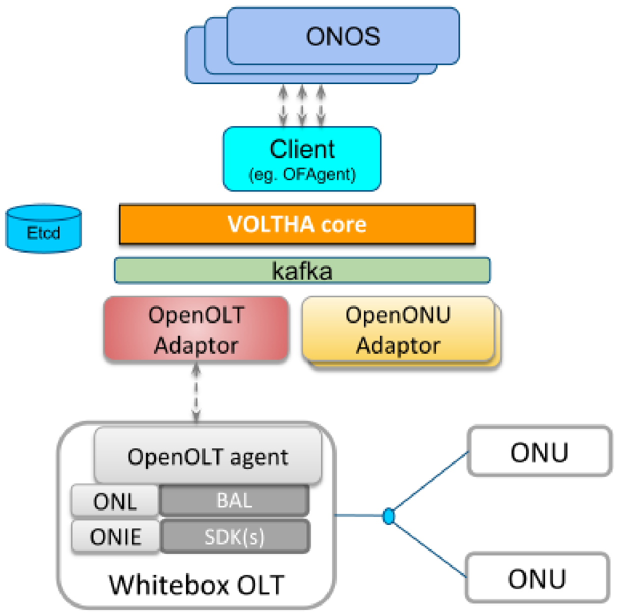

Figure 4.5: VOLTHA Software Components [19]

4.3.2 VOLTHA: Virtual OLT Hardware Abstraction

Control and management in the access network space is complex. Each access technology brings its own set of hardware devices and protocols, above which vendors deploy their own interpretation or extension of the same network standards.

VOLTHA is an open source project of the Open Networking Foundation (ONF) to create a hardware abstraction for broadband access equipment, to unify the control and management of network devices. It supports the principle of multi-vendor, disaggregated, *any broadband access as a service* for the Central Office of a telecom operator[19].

VOLTHA currently provides a common, vendor agnostic, PON control and management system, for a set of white-box and vendor-specific PON hardware devices. It has four key concepts:

- Network as a Switch: VOLTHA abstracts all the connected PON access network devices as a SDN programmable L2/L3 switch, isolating the PON management system and the PON devices. In this way the SDN controller does not have to know the PON-level details to work;

- Evolution to virtualization: VOLTHA supports a lot of access network technologies and devices and can work with them seamlessly;
- Unified Operations, Administration and Maintenance (OAM) abstraction: service lifecycle, troubleshooting, device lifecycle (including upgrade and discovery), alarms, security, system monitoring, etc. are handled in an unified, vendor and technology agnostic manner;
- Cloud/DevOps bridge to modernization: all the previous key concepts are implemented and deployed using a microservices architecture which runs on top of Docker and/or Kubernetes.

In Figure 4.5 the software structure of VOLTHA is visible, which also includes ONOS, OLT and ONU adapters and a white-box OLT device.

VOLTHA communicates with other systems via two interfaces: the northbound interface, through which it hides all PON-level details from the SDN controller and abstracts each PON as a pseudo-Ethernet switch easily programmed by the controller itself with an ONOS app, and the southbound interface, through which it communicates with PON hardware devices using vendor-specific protocols through OLT and ONU adapters. These adapters manage and send commands to the devices through agents installed in them (in Figure 4.5 only the one for the OLT is visible) with the gRPC protocol.

VOLHA is in charge of of establishing the data plane connections through the hardware by interpreting service requests from the SDN controller and transforming them into requests to be fulfilled by the appropriate adapter. It has also the responsibility of forwarding control plane requests to the SDN controller, for example requests concerning authentication protocols like 802.1x and PPoE or multicast services such as IGMP.

Finally, telecom operators should be able to view logs from all the VOLTHA components as well as from white box OLT and ONU devices in a single stream. This aspect is managed through an EFK (Elasticsearch[1], Kibana[2] and Fluentd[3]) setup for VOLTHA which enables the operator to push logs from all components in a single place.

[1]`https://www.elastic.co/elasticsearch`

[2]`https://www.elastic.co/kibana`

[3]`https://www.fluentd.org`

4.3.3 ONOS: Open Network Operating System

The possibility offered by SDN to take control functions out of a dedicated box to centralize them and create applications for such purposes and to dynamically program data paths through the network plays a key role when steering subscribers' traffic. In SEBA, when looking at a residential gateway attachment process, there are two major stages, which can be implemented differently by the various telecom operators [18]:

- Device attachment and recognition: when device is powered on and attached to the access node in the service provider domain (usually the ONU) layer 1 comes up and VOLTHA needs to enable the L2/L3 connection. To do so, it can create an event that is processed by the SDN control framework. Then, the port is authenticated and a network is created to enable step 2, where the device attaches to the BNG;

- Subscriber session establishment: the new device is connected to the BNG by inserting one or more forwarding rules (flow rules) in aggregation switches. The newly created flow is then registered in a central database inside or attached to the SDN controller if the BNG is virtualized or its registration is delegated to the BNG if it is external.

In SEBA, these two features are provided by ONOS, the Open Network Operating System, which is a distributed SDN controller developed in Java. ONOS can run as a distributed system across multiple servers, allowing it to use the CPU and memory resources of multiple servers while providing fault tolerance and potentially supporting live/rolling upgrades of hardware and software without interrupting network traffic.

It has four main characteristics [20]:

- Code modularity: it is possible to introduce new functionalities as self-contained units built independently;

- Configurability: it is possible to load and unload various features, whether it be at startup or at runtime;

- Separation of concern: there are clear boundaries between subsystems to facilitate modularity and northbound and southbound APIs are provided to interact with each subsystem;

- Protocol agnosticism: ONOS and its applications are not bound to specific protocol libraries or implementations. If ONOS needs to support a new protocol, it is possible to build a new module as a plugin that may be loaded into the system.

ONOS System Components

ONOS is architected with tiers of functionality, represented in Figure 4.6. The three main tiers are the App component, in which applications consume and manipulate information aggregated by the managers, the Core or Manager component, which receives information from Providers and serves it to applications and other services, and the Providers component, which interface with the network via protocol-specific libraries and with the core via a specific interface. A service/subsystem is defined as a unit of functionality that is comprised of multiple components that create a vertical slice through the tiers as a software stack.

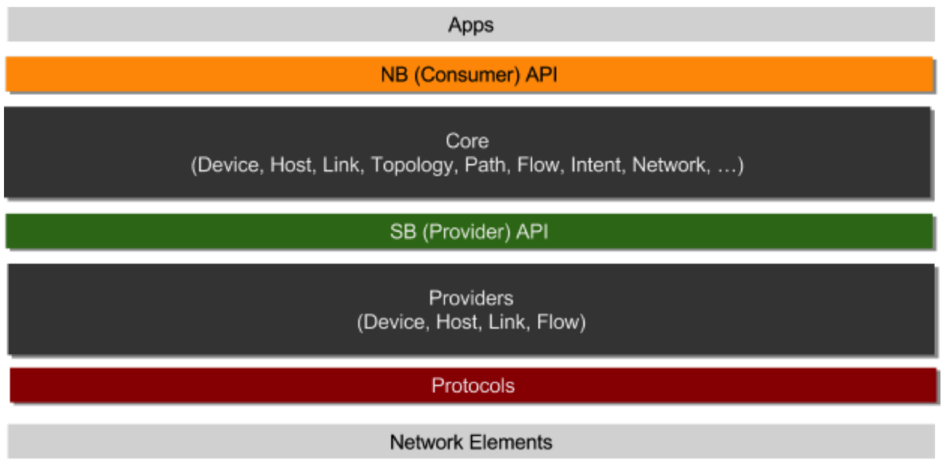
Apps
NB (Consumer) API
Core
(Device, Host, Link, Topology, Path, Flow, Intent, Network, ...)
SB (Provider) API
Providers
(Device, Host, Link, Flow)
Protocols
Network Elements

Figure 4.6: ONOS System Components [21]

There are a lot of services (device service, host service, link service, topology service, etc.) and each of them manages a specific functionality (inventory of devices, inventory of hosts, inventory of links, snapshots of the network graph view, etc.). Each of a service's components resides in one of the three main tiers and the relationships between those tiers are illustrated in Figure 4.7, where the top and bottom dashed lines represent the boundaries created by the northbound and southbound interfaces.

In this thesis three services have been mainly used: device subsystem, application

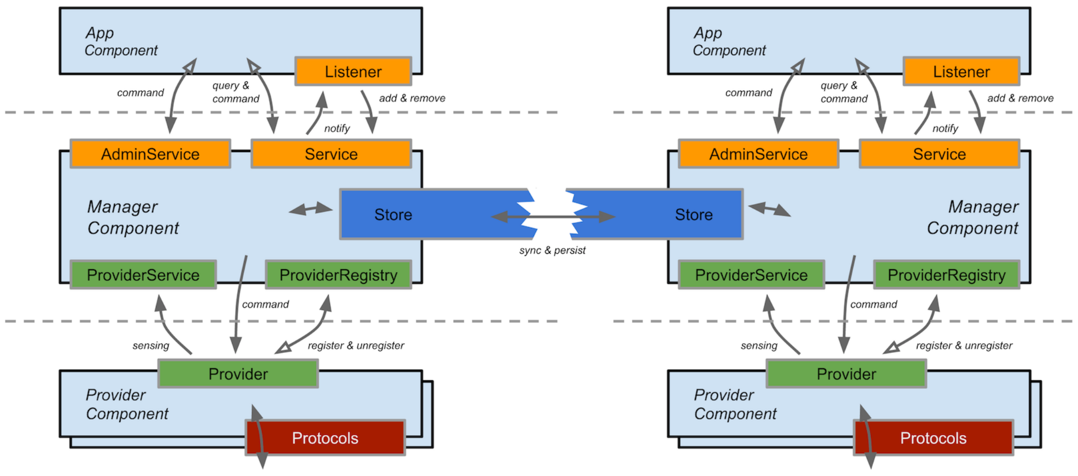

Figure 4.7: ONOS Component Interactions [21]

subsystem and flow rule subsystem, which are briefly described below.

Application Subsystem

The main aim of the application subsystem is to facilitate applications deployment and management across all the ONOS instances in a cluster. The subsystem exploits ONOS eventually consistent data structures and inter-node communication procedures to make available the full inventory of applications in all the ONOS instances [22].

All built in sample and test applications provided by ONOS are delivered exploiting those two features and are pre-installed in the standard ONOS distribution, but are not activated by default. This includes any providers, such as OpenFlow providers. In this way, there is no need to rebuild, or even reconfigure, ONOS itself when optional software components have to be installed or withdrawn from it.

Device Subsystem

The main responsibility of the device subsystem is to discover and track the devices that comprise the network and to enable operators and applications to control such devices. Most of ONOS's core subsystems rely on the Device and Port model objects, which are an abstraction of a device with its ports, created and managed by this subsystem, and on its provider for interacting with the network [23].

The provider component of this subsystem is made up of multiple providers, each with support of their own network protocol libraries or means to interface with the network. In this thesis the *OpenFlowDeviceProvider* is used, which allows ONOS to interact with OpenFlow networks, those used in SEBA.

Flow Rule Subsystem

The main aim of the flow rule subsystem is to manage flow rules which are present in the system and to install them into specific devices in the network [24].

This subsystem implements the semantics of a distributed authoritative flow table where the master copy of the flow rules is contained in the controller and then it is pushed down to the devices. According to this mechanism the subsystem never tries to discover information from the network itself and never tries to deal with flows that are already present on devices. If a flow is detected by ONOS on a device that should not have it according to its authoritative flow table, ONOS will remove that flow.

The providers of this subsystem are responsible for collecting statistics on flows installed in the network and to report this data to the flow rule subsystem itself. Then, the subsystem uses these reports to ensure that its representation of the network state is still present on the devices on the network.

4.3.4 NEM: Network Access Mediator

Transforming today's CO into CORD is a two-step process. The first step is to disaggregate and virtualize the hardware devices, so what has been illustrated in the previous sections, while the second step is to provide a framework into which the resulting disaggregated elements can be plugged, producing a coherent end-to-end system. This framework defines the unifying abstractions that forge this collection of hardware and software elements into a scalable and agile system [15].

The Network Edge Mediator (NEM) serves as the mediation layer between the edge/access system and the service provider backend and global automation frameworks. NEM will provide the interfaces and components required by the service provider for managing the access network components and broadband service subscribers the SEBA POD is designed to offer and support. A variety of operator OSS/BSS and global orchestration frameworks can be integrated northbound for specific deployment needs. NEM plays two important roles in SEBA [25]:

- NEM forms a complete and unified solution. Each telecom operator wants a different subset of the available disaggregated components, which can be specified in NEM at configuration time with a resource called *Profile*. In the case of SEBA, for example, some operators want the BNG to be internal to the solution and some want it to be external. Moreover, when the BNG is an internal, some operators want it to be implemented in containers and others in the switching fabric. NEM puts together all the components specified in the declarative profile into a self-contained resource, by generating the necessary *glue code*. It offers a coherent Northbound Interface to the operator's OSS/BSS, and it manages any state that needs to be shared among the components. This avoids hardwiring dependencies into each component;
- NEM manages all the environment dependencies defining a runtime *Workflow*, which has to be created by the telecom operator. In SEBA the workflow specifies how the integrated solution interacts with the surrounding operational environment, as individual subscribers are managed throughout their lifecycle (e.g., new RG online/existing RG offline). The workflow specifies all aspects of what happens as RGs and ONUs are detected and approved, authentication packets are sent by the subscriber and are authenticated, DHCP requests are received and IP addresses assigned, and, finally, packets start flowing. Different workflows can be defined for different operators, yet reuse all the same components without modification.

At a high level, NEM consists of three subsystems: an Authoritative State Manager (implemented by XOS), which manages the state needed to configure and control the collection of backend components, a Monitored State Manager (implemented by Prometheus, Grafana, Elk Stack and Kibana), which collects and manages the logs, metrics, and events produced by the backend components and an Event Bus (implemented by Kafka), which shares events among all the backend components and NEM subsystems.

Among the three components the most important is certainly XOS, used in the context of SEBA to control the virtualized OLTs via VOLTHA and the aggregation switches via ONOS, which is described below.

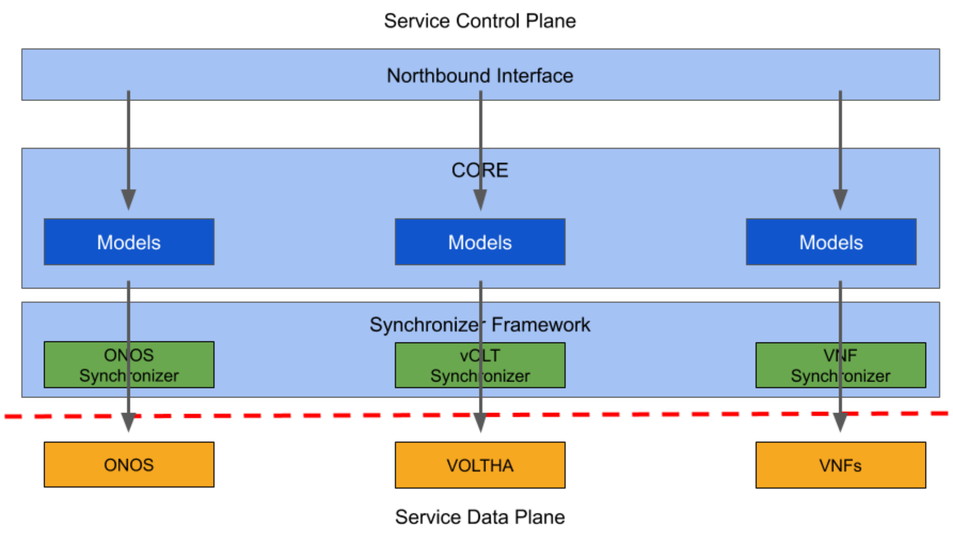

Figure 4.8: XOS Internal Structure

XOS

XOS is a logically centralized, model-based service for operating micro-service-based cloud applications. XOS complements cloud orchestration tools like Kubernetes, which manages the set of containers that implement each micro-service, focusing instead on managing the state necessary to configure and control the collection of micro-services as a whole, and doing so at the granularity of individual subscribers. In this sense, XOS can be thought of as augmenting the micro-service-based architecture with an Extensible Service Control Plane [25].

XOS plays a central role in operationalizing the disaggregated backend components in SEBA. It takes a set of model definitions (schema) as input, and auto-generates the glue code needed to integrate those components. XOS internal structure is shown in Figure 4.8.

The models define both the *Profile* of disaggregated components the operator wishes to deploy and the *Workflow* required to integrate those components into a surrounding operational environment. The auto-generated code includes the Northbound Interface used by the operator to manage the deployment and the *Synchronization Framework* needed to keep the backend components synchronized with the authoritative state defined in the data models.

The XOS Data Model is initialized with a set of core models. The core defines common abstractions, such as services and service dependencies. Each component that is to be integrated into a particular Profile then loads one or more component-specific models into XOS (these typically extend the core models), along with a Synchronizer plugin (an imperative program) that map the abstract declarative state about how the system is supposed to behave (as defined by the XOS data model) into the concrete operational state of the backend components that implement the system. XOS also supports purely logical services that have no corresponding backend component. These logical services create new/composite functionality by programming the data model. Models used by XOS are written with the XOS Modeling Framework, which defines a language for specifying data models and a tool chain for generating code based on the set of models.

Finally, XOS also plays an important role in SEBA's security, providing hooks for specifying security policies, which are generic logic expressions that can operate on any model or on the environment. These policies determine what a user, represented by a user model or by an API context, can do on a given resource (read, write, update, etc.). The policies are enforced at the API boundary. When an API call is made, the appropriate policy is executed to determine whether or not access should be granted, and an audit trail is left behind. The policy enforcers are auto-generated by the generative toolchain as part of the model generation process.

Chapter 5

SiaB: SEBA-in-a-Box

SEBA is a big framework and it requires a lot of hardware components to be deployed and tested, making a first approach impossible for a user interested in the technology, who would have to buy and configure a lot of network equipment. Some problems arise even if it is an internet operator or a new possible SEBA partner which is interested in the framework, since developing and testing new features or new system integrations in such a complex system takes a long time and it is impossible without having a complete system deployed.

To solve these problems the developers of SEBA created a fully virtualized and easy to setup version of the framework, which can be installed on a single Ubuntu server together with Docker and Kubernetes. Thanks to this lightweight version of SEBA, called SEBA-in-a-Box (SiaB), in this thesis it was possible to analyze the SEBA system and look for a method to integrate it with NSM.

In this chapter SiaB is analyzed, highlighting the differences with the standard version of SEBA and showing the *operator workflow* it uses to manage residential access.

5.1 Differences with SEBA

The main difference between SEBA and SiaB is hardware virtualization, in fact control plane software is the same in both systems. The default configuration of SiaB incorporates an emulated RG/ONU/OLT provided by PONSIM and an emulated AGG switch provided by Mininet. Mininet is also configured with a host that stands in as the BNG and runs a DHCP server. The RG is able to authenticate itself via 802.1x, run dhclient to get an IP address from the DHCP server in Mininet, and finally ping the BNG. This demonstrates end-to-end connectivity between the RG and BNG via the ONU, OLT, and aggregation switch [26].

With SiaB two other configurations are possible: the first allows to use a physical aggregation switch instead of an emulated Mininet topology, reducing hardware virtualization to PON components only, the second one allows to use a disaggregated BNG, instead of an external one, with the BNG data plane implemented in the aggregation switch and the control plane functionalities implemented in ONOS (section 4.3.1). Thanks to these possible additional configurations, many more use cases can be tested, also depending on the hardware available and the type of BNG to be tested.

5.1.1 PON Virtualization

PON virtualization is based on two different Helm charts: PONSIM and PONNET. The first chart to be installed is PONNET, which creates two or more Linux bridges, *pon0* and *nni0*, *nni1*, etc. depending on the selected number of simulated OLTs, that allow a L2 dataplane to be created between the PONSIM RG and components upstream of the PONSIM OLT. To create these bridges, the chart modifies the underlying Kubernetes setup by installing the bridge CNI and adding configuration files to create the two bridges. During the installation the number of desired ONUs and OLTs must be specified (between 1 and 4) to configure the right number of interfaces in the bridges.

Once the PONNET chart has been installed PONSIM can be installed too. It deploys the hardware component of a PON, so one or more RGs, ONUs and OLTs, as a set of containers in as many Kubernetes pods, in the context of VOLTHA. All deployed pods are independent of each other and are scalable. OLT pods communicates with ONU pods and with VOLTHA through the gRPC protocol, while RGs and ONUs communicates through the *pon0* bridge.

Finally, PONSIM also offers a CLI tool to use the simulator outside of a Kuberentes cluster, giving the possibility to independently create, manage and test one or more RGs, ONUs and OLTs.

5.1.2 Mininet

As mentioned previously, the aggregation switch and the BNG are simulated through Mininet. Mininet is a network emulator which creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and Software-Defined Networking [27].

One of the main advantages of Mininet is that its networks run real code including standard Unix/Linux network applications as well as the real Linux kernel and network stack (including any kernel extensions which may have been available, as long as they are compatible with network namespaces).

Because of this, code developed and tested on Mininet for an OpenFlow controller, modified switch, or host, can move to a real system with minimal changes, for real-world testing, performance evaluation, and deployment. Importantly this means that a design that works in Mininet can usually move directly to hardware switches for line-rate packet forwarding.

In SiaB, Mininet is deployed through a Helm chart, which depends on the PONNET one. This chart creates a virtual aggregation switch with Open vSwitch[1], that connects to the OLT(s) created by PONSIM through the *nniX* bridges, and a host that stands in as the BNG and runs a DHCP server. This host has a static IP address determined by the number of ONUs and OLTs present in the deployment (e.g. 172.18.0.10 with 1 OLT and 1 ONU) and is configured with some double-tagged interfaces to untag incoming traffic and tag outgoing traffic [28]. The usage of VLAN tags is better explained in the following section while the usage of double tagged interfaces is explained in the next chapter.

5.2 AT&T SEBA Workflow

To have a fully functional and testable version of SEBA it is not enough to have the hardware available but it is also necessary to develop the software that regulates the functions of the control plane and that manage the interaction between the residential users and the devices managed by SEBA, starting from users' authentication.

[1]`https://www.openvswitch.org/`

The specification of this interaction is called *Workflow* and depends on the telephone operator who decides to use SEBA. Actually there are four available workflows described in the official documentation of SEBA[2] but SiaB uses only one, developed by the AT&T operator [29]. It authenticates end-users with the 802.1x protocol and exploits an external BNG.

The workflow is made of several steps, which are illustrated below:

1. A new ONU is turned on and discovered by the OLT. If its serial number is not allowed or unknown, it is disabled by default. Anyhow, an event stating that a new ONU has been discovered is generated and sent to the NEM, which determines whether the ONU is valid by consulting local pre-provisioned data;

2. If the ONU is valid the port is enabled and an authentication 802.1x trap flow is programmed in the OLT;

3. RG sends an 802.1x an EAPOL message which is trapped by the OLT and sent to ONOS. Here the ONOS AAA app adds options and sends to RADIUS server, which authenticates the request with its internal information;

4. If RG authentication fails, allow it to keep trying. If RG authentication succeeds, ONOS AAA app notifies via an event on the Kafka bus that authentication has succeeded;

5. NEM can listen for this event and program the network with its synchronizer modules (the process is graphically shown in Figure 5.1):

    - It creates a new subscriber service instance with two VLAN tags: the Customer tag (C-tag) which is added by the ONU and it is used for the customer's purposes and the Service tag (S-tag) which is added by the OLT and is used for forwarding purposes;
    - With the OLT synchronizer (*vOLT Synchronizer*), it programs a DHCP trap rule in the OLT to intercept DHCP packets coming from the RG;
    - With the ONOS synchronizer (*Fabric-xconnect Synchronizer*), it programs the aggregation switch to forward L2 Q-in-Q VLANs packets to the external BNG. BNG is always configured to strip VLAN tags.

[2]`https://wiki.opencord.org/display/CORD/SEBA+Workflows`

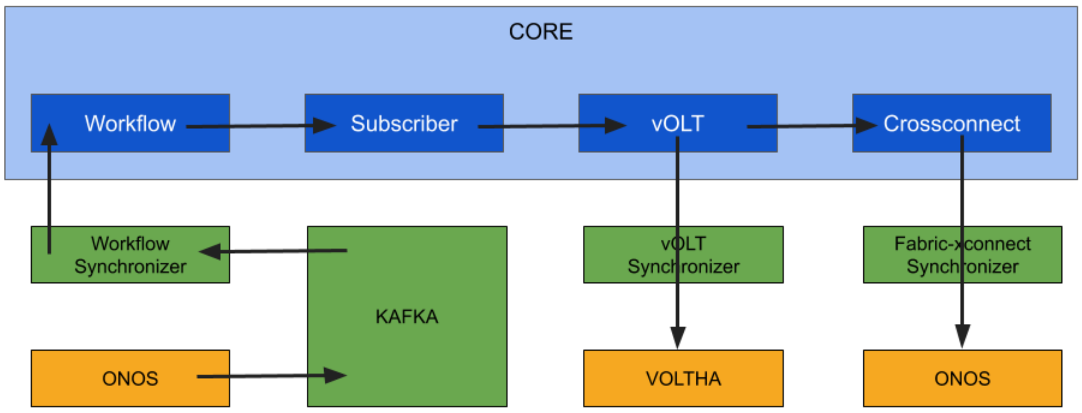

Figure 5.1: NEM steps for new user subscription

6. RG sends a DHCP request packet which is intercepted by the OLT and sent to ONOS. Here the ONOS DHCP L2 relay app adds some information to the packet and then sends it back to the data plane, where it is forwarded to the DHCP server in the BNG by the aggregation switch;

7. The DHCP response is sent directly to the RG from the BNG without the need of extra processing and an event is generated into the Kafka bus regarding the DHCP state machine for each subscriber (including IP + MAC information);

8. If RG is disconnected from the ONU, authentication is forced again. Upon reconnection to the ONU, RG must re-authenticate before DHCP/other-traffic can flow on the provisioned VLANs.

In Figure 5.2 is shown a high-level complete vision of the AT&T workflow, in which are indicated the functions of each component and the path of a packet in the network with a blue line.

The steps just illustrated are only the most important part of the AT&T workflow, which actually include other specifications, for example the hardware requirements, the maximum number of ONUs per OLT or the maximum number of OLTs connected to a single aggregation switch and many more. Finally, the workflow also indicates how to properly disable or remove one or more components without corrupting the system and how to react if a component fails.

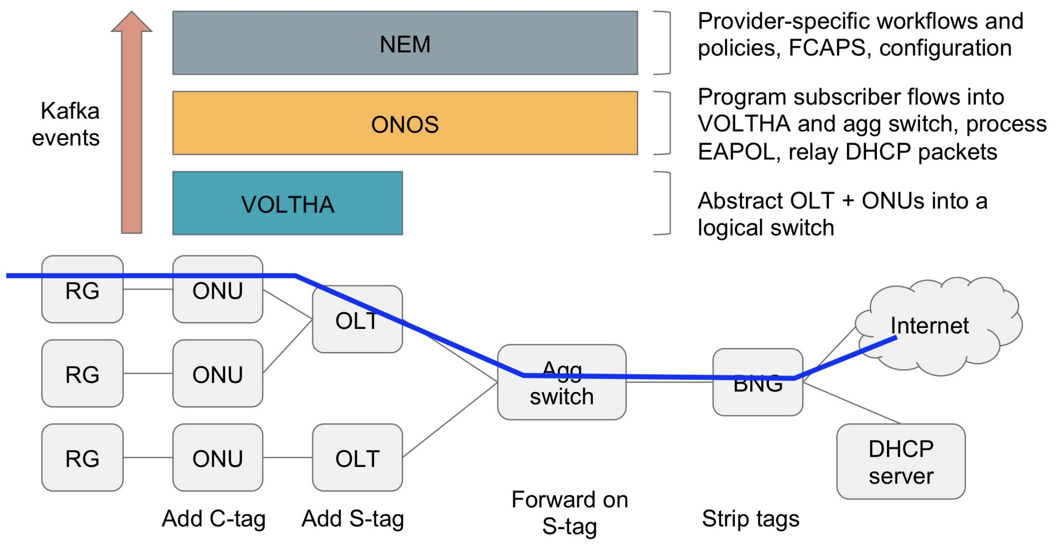
NEM
ONOS
VOLTHA
Kafka events
Provider-specific workflows and policies, FCAPS, configuration
Program subscriber flows into VOLTHA and agg switch, process EAPOL, relay DHCP packets
Abstract OLT + ONUs into a logical switch
RG
ONU
OLT
Agg switch
BNG
Internet
DHCP server
Add C-tag
Add S-tag
Forward on S-tag
Strip tags

Figure 5.2: SEBA AT&T Workflow Overview

5.3 SiaB Use Case

In Figure 5.3 it is visible the full dataplane of SEBA-in-a-Box with one ONU and one OLT, with all the elements explained in the previous sections. This is the scenario used for the study of the framework and for the integration part between SEBA and NSM.

In this section other details regarding the functioning of this scenario will be added and it will be shown how to obtain an IP address for the residential gateway deployed with PONSIM and how to perform a ping test between the RG and the BNG deployed in Mininet.

First of all, the gray squares and rectangles in Figure 5.3 represent pods in a Kubernetes cluster, deployed in different namespaces. In the picture are also visible the two Linux bridges (*pon0.0* and *nni0*) connecting the RG with the ONU and the OLT with the aggregation switch emulated in Mininet. Finally, inside the Mininet pod it is visible the BNG with the static IP address 172.18.0.10, connected to the aggregation switch with a virtual link emulated by Mininet.

Near the connection between ONU and OLT, made with gRPC, and between OLT and aggregation switch, made with the Linux bridge, the VLAN tags used in this scenario are also visible. The customer tag (inner tag), added by the ONU, is 111, while the service tag (outer tag), added by the OLT, is 222.

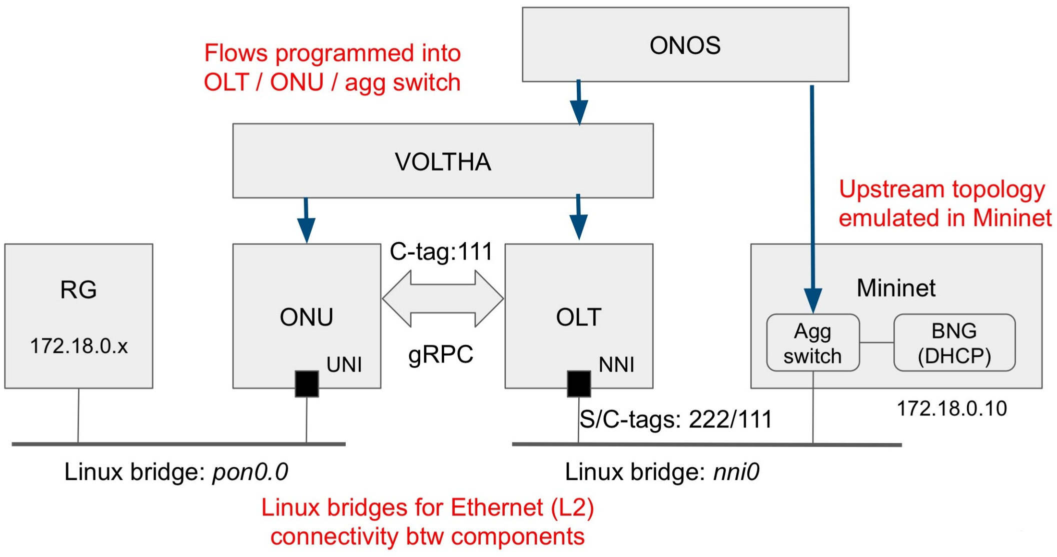

Figure 5.3: SEBA-in-a-Box Dataplane Components

As far as the control plane of SiaB, in the picture are visible only the VOLTHA pod and the ONOS pod, in their usual representation. The NEM pod is not represented because it is composed of several pods, in particular XOS alone is composed of six pods, which implement different functionalities such as the GUI, the core component and the database.

Once all the pods are available in Kubernetes and the system is fully deployed (a complete guide to deploy the system is available in the next chapter), it is possible to start the procedure to request an IP address for the RG, which will be in the 172.18.0.0/24 pool of addresses [26]:

1. Before starting the procedure it is necessary to remove from the *dhclient* profile from *apparmor* if present on the host:

```
$ sudo apparmor_parser -R /etc/apparmor.d/sbin.dhclient ||
    true
```

2. Enter the RG pod in the VOLTHA namespace and open a shell:

```
$ RG_POD=$( kubectl -n voltha get pod | grep rg0-0 | awk
    '{print $1}')
$ kubectl -n voltha exec -ti $RG_POD bash
```

3. Inside the pod, run the following command:

   ```
   $ wpa_supplicant -i eth0 -Dwired -c
       /etc/wpa_supplicant/wpa_supplicant.conf
   ```

   whose output must match the following one:

   ```
   Successfully initialized wpa_supplicant
   eth0: Associated with 01:80:c2:00:00:03
   WMM AC: Missing IEs
   eth0: CTRL-EVENT-EAP-STARTED EAP authentication started
   eth0: CTRL-EVENT-EAP-PROPOSED-METHOD vendor=0 method=4
   eth0: CTRL-EVENT-EAP-METHOD EAP vendor 0 method 4 (MD5)
       selected
   eth0: CTRL-EVENT-EAP-SUCCESS EAP authentication completed
       successfully
   ```

4. Remove the IP address assigned to the main network interface of the pod:

   ```
   $ ifconfig eth0 0.0.0.0
   ```

5. Request a new IP address to the DHCP server in the BNG:

   ```
   $ dhclient
   ```

   The output is the following:

   ```
   mv: cannot move '/etc/resolv.conf.dhclient-new.46' to
       '/etc/resolv.conf': Device or resource busy
   ```

   This error can be ignored, it is caused by the fact that */etc/resolv.conf* is mounted into the RG container by Kubernetes and dhclient wants to overwrite it.

At the end of this procedure a new IP address will be assigned to the *eth0* interface and it will therefore be possible to ping the BNG at its address 172.18.0.10.

Finally, it should be noted that it is not currently possible to contact hosts on the Internet via SiaB, all traffic ends at the BNG.

Chapter 6

Integrating SEBA (SiaB) and Network Service Mesh

This chapter describes the process of integrating SEBA-in-a-Box and Network Service Mesh.

The study was divided into two scenarios:

- First scenario: SiaB and NSM are developed in the same server, creating a full cloud-native environment where every component is managed with the same instance of Kubernetes. This first scenario simulates the real use case in which both SEBA and NSM are deployed within the central office of a telecom operator and can be applied when the CNF chains do not require a dedicated infrastructure;
- Second scenario: SiaB and NSM are deployed in two different servers, with two different instances of Kubernetes. This second scenario is applicable when the CNF chains require a lot of computational capabilities and therefore cannot be deployed on the same servers that manage the SEBA control plane. Thanks to the dedicated infrastructure, it is possible to create larger CNF chains or duplicating one or more existing chains, also introducing a load balancing functionality.

Before describing the integration process for each scenario, it is explained how to correctly deploy the two systems, highlighting some of the possible problems and the solutions to solve them.

6.1 First Scenario: SiaB and NSM in the same server

In this scenario SiaB and NSM are developed in the same server, creating a full cloud-native environment. As mentioned above, this scenario can be applied when the CNFs of the telecom operator do not require much computing power and therefore a dedicated infrastructure is not required.

Resource consumption is significant since the servers hosting the SEBA control plane applications do not require high performance and are therefore not suitable for processing large amounts of traffic, as it may be necessary for some network functions, such as firewalls or IDS/IPS.

On the other hand, always using high-performance servers in order to support any type of CNF workload can involve a large expense of money and could prove useless if there is not always the need to maintain high performance.

6.1.1 SiaB Deployment

SiaB developers have tried to make its installation as simple as possible, creating a special *Makefile* that takes about 10 minutes to install the whole system on a physical server or VM. The recommended operating system is Ubuntu 16.04.

To get the *Makefile* and start the installation just run the following commands:

```
$ git clone https://gerrit.opencord.org/automation-tools
$ cd automation-tools/seba-in-a-box
$ rm Makefile
$ git clone https://github.com/PPaviii/my-SiaB.git .
$ make [NUM_OLTS=n] [NUM_ONUS_PER_OLT=m]
```

The number of OLTs and the number of ONUs for OLT must be between 1 and 4, and by default they are both set to 1.

During the installation procedure the original *Makefile* is removed because to enable IPv4 forwarding in Kubernetes it is necessary a modified version of the Calico CNI configuration file, which is provided in the repository *https://github.com/PPaviii/my-SiaB.git* together with the modified *Makefile* that allows its correct use.

In some cases the commands listed so far may be sufficient to install the complete system, but it is possible that the following problems may arise:

- Errors due to the expiration of a timeout: in many places in the makefile timeouts have been implemented to wait for Kubernetes pods to become operational. On some occasions, depending on the performance of the server or VM, the timeout may not be sufficient, and the installation procedure fails. To solve the problem just restart the installation with the same initial command, which will restart from where it left off;

- Errors due to *CrashLoopBackOff* status of the CoreDNS pods: CoreDNS is a flexible, extensible DNS server that can serve as the Kubernetes cluster DNS [30]. Sometimes these pods can stop working properly due to a short loop of DNS requests, entering in the CrashLoopBackOff status. To solve the problem is sufficient to disable the loop detection functionality in the pod configuration file and to restart the pods, with the following commands:

  ```
  $ kubectl -n kube-system edit configmap coredns
  ```

  comment or delete line 19 of the file, which contains the word *loop* and then restart the pods:

  ```
  $ kubectl -n kube-system delete pod -l k8s-app=kube-dns
  ```

  Finally, resume the installation procedure with the same initial command;

- Missing ONOS applications: this problem occurs if the Kubernetes pods do not have direct internet access, for example due to a proxy. In this situation it is not possible to download the ONOS applications of the control plane and therefore they must be added manually to the system. First of all, download the ONOS app on the server with the following command:

  ```
  $ git clone https://github.com/PPaviii/onos-app-SiaB.git
  ```

  Then, it is necessary to activate the ONOS GUI (username: karaf, password: karaf) from the CLI and then manually upload the files:

  ```
  $ ssh onos@127.0.0.1 -p 30115 (password: rocks / karaf)
  onos@root > app activate gui
  ```

  Finally, upload and activate all the ONOS applications previously downloaded from the URL: `localhost:30120/onos/ui/index.html#/app` and resume the installation procedure with the same initial command.

After solving some or all these errors the installation should finish correctly, and it will be possible to carry out the steps listed in section 5.3 to authenticate the RG and ping the BNG. From the ONOS GUI it is also possible to see the full network topology, which will include the RG, the emulated OLT and ONU, displayed as a single device thanks to VOLTHA, the aggregation switch and the BNG.

6.1.2 NSM Deployment

The deployment of Network Service Mesh is divided into three parts: a first part where all the components to make NSM work are installed and a second part where the firewall use case with its three pods is installed (section 3.1.4) and a third part where some routes are added to the routing tables of the various pods so that the whole system works correctly. The steps to install the framework are as follows:

- Download the source code of NSM and enter the corresponding folder:

```
$ git clone https://github.com/networkservicemesh/
   networkservicemesh.git
$ cd networkservicemesh/
```

- Download the source code of the firewall use case, called example, and put it into the *deployments* folder:

```
$ git clone https://github.com/PPaviii/my-nsm.git
    deployments/helm/example
```

- Install Network Service Mesh, selecting the default Kubernetes namespace and the kernel forwarder:

```
$ NSM_NAMESPACE=default FORWARDING_PLANE=kernel
    INSECURE=true SPIRE_ENABLED=false make helm-install-nsm
```

- Install the firewall use case pods, with the same options as before:

```
$ NSM_NAMESPACE=default FORWARDING_PLANE=kernel INSECURE=true
    SPIRE_ENABLED=false make helm-install-example
```

To insert one or more route towards a specific subnet into the routing table enter the selected pod and use the command:

```
$ ip route add <subnet_CIDR> via <IP_address_next_hop>
```

The routes that must be entered are:

- NS client routes: in the client must be entered the route to reach the IP address of the NS server (172.16.2.2);
- Firewall routes: in the client must be entered the route to reach the IP address of the residential gateway;
- NS server routes: in the server must be entered the routes to reach the IP address of the NS client (172.16.1.1) and the one of the residential gateways.

Network Service Mesh installation should be successful in most cases, except in the presence of a proxy. In the latter case it is necessary to add two Linux environment variables in the */etc/environment* file:

```
NO_PROXY=nsm-admission-webhook-svc.nsm-system.svc
no_proxy=nsm-admission-webhook-svc.nsm-system.svc
```

where *nsm-system* is the name of the namespace where NSM is installed (*default* in the case of this thesis).

Once the installation is finished, the framework will work correctly and the NS client pod will be able to send and receive traffic from the NS server pod through the firewall.

6.1.3 Integration Procedures

Up to this point all the SiaB and NSM pods are deployed in the same instance of Kubernetes, but the systems are not connected and cannot yet exchange traffic. The first step is to connect the data planes of the two systems and then modify the control plane with XOS so that the traffic flows from SiaB to NSM and then comes back.

Figure 6.1 represents the final deployment after integrating the two systems. NSM is connected to SiaB between the aggregation switch and the BNG, with the NS client connecting to the aggregation switch via a veth pair. Figure 6.1 depicts a more general scenario, where a generic CNF is represented instead of NS client.

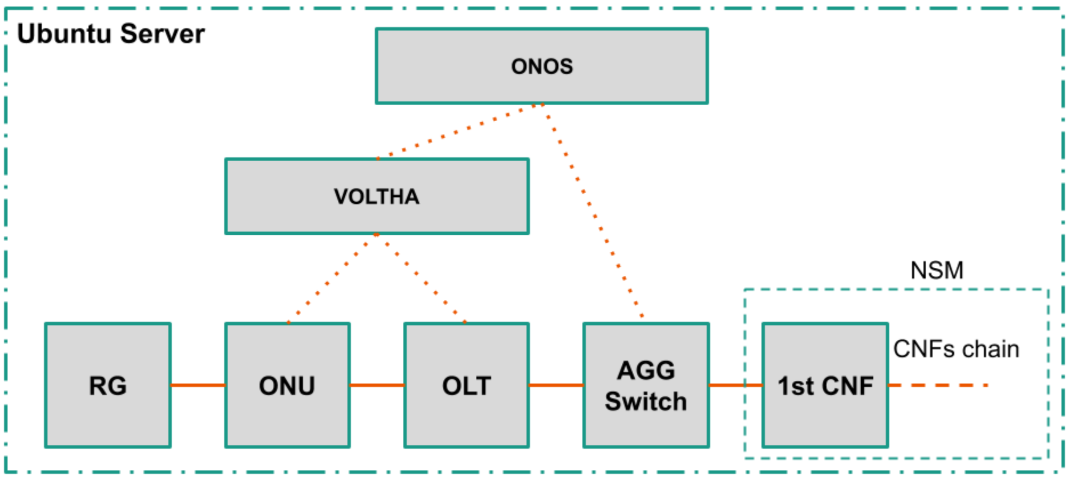

Figure 6.1: Graphical Representation of the First Integration Scenario

Connect Mininet pod with NS Client pod

A custom made *veth* pair was used to connect the two pods together, with one head in the network namespace of the Mininet pod and the other one in the network namespace of the NS client. In Kubernetes it is not easy to find the namespace of a pod, so it is necessary to find it with two Docker commands:

- Mininet network namespace:

```
$ DOCKER_ID_MN=$( docker ps | grep <mininet_pod_name> |
    head -n 1 | awk '{print $1}' )
$ POD_NETNS_MN=$( docker inspect --format '{{ .State.Pid
    }}' $DOCKER_ID_MN )
```

- NS Client network namespace:

```
$ DOCKER_ID_NSC=$( docker ps | grep <ns_client_pod_name> |
    head -n 1 | awk '{print $1}' )
$ POD_NETNS_NSC=$( docker inspect --format '{{ .State.Pid
    }}' $DOCKER_ID_NSC )
```

Once both network namespaces are found the command to create the veth pair is the following:

```
$ ip link add <mininet_interface_name> netns $POD_NETNS_MN
    type veth peer name <ns_client_interface_name> netns
    $POD_NETNS_NSC
```

where *mininet_interface_name* and *ns_client_interface_name* are the names assigned to the new veth interfaces in their respective pods. Once the two interfaces have been created enter both pod and activate them with the command:

```
$ ip link set dev <interface_name> up
```

Once all these operations are finished, the two pods will be correctly connected and will be able to exchange traffic. The two systems, however, are not yet connected because the new *veth* interface in the mininet pod is not directly connected to the aggregation switch, and therefore the RG cannot yet communicate with NSM.

Connect veth interface in Mininet to the Aggregation switch

The last step to connect the two systems is to connect the new Mininet pod interface to the simulated aggregation switch in it. Since the switch is emulated through OpenvSwitch its CLI will be used, which is called *ovs-vsctl*.

First of all, the name of the switch is needed, which can be found with the command:

```
$ ovs-vsctl show
```

Then, the command to add the port to the switch is the following:

```
$ ovs-vsctl add-port <switch_name> <veth_interface_name>
```

Finally, it is possible to check if the port was added successfully by looking at the list of all the ports of the switch, where the new port must be present:

```
$ ovs-vsctl list-ports <switch_name>
```

At this point the two systems are correctly connected from the point of view of the data plane, but there are still some problems to be solved:

- The BNG is still the default gateway for the RG so all traffic directed to CNFs deployed in NSM, which have an IP address in a different subnet compared to that of the RG, is still sent to the BNG instead of to the NS Client;
- NSM does not currently support VLAN tagged traffic forwarding and then all tagged packets that arrive at the NS Client are discarded by default;

- Currently the aggregation switch is configured to send all the traffic it receives from the RG to the BNG, without the possibility of steering it to the CNF chain in NSM;
- Calico source NATting prevents the RG pod from communicating with the IP address assigned to it by DHCP, replacing it with the one of the host server on which both systems are deployed. Due to this behavior the routing cannot work, and the use case would no longer be applicable to a real scenario, where the RG is not a pod and therefore does not suffer this effect.

Additional integration steps are required to resolve these issues, as detailed below.

Create double tagged VLAN interfaces on NS Client

NSM does not support and cannot forward VLAN tagged traffic from SiaB, which in turn cannot function without VLAN tagged traffic. It is therefore necessary to find a method to untag all packets arriving at NS client from the *veth* interface connected to the aggregation switch and to tag all those that leave the same *veth* interface towards SiaB.

The method is to create two 802.1Q VLAN interfaces on top of the *veth* interface of the NS client pod, which untag incoming traffic and tag outgoing traffic. These interfaces are logical interfaces that can be created on a hardware interface. These software-defined interfaces allow the segregation of traffic into separate logical channels on a single hardware interface, with which they share the MAC address and all the other physical characteristics. In this case two interfaces are needed since there are two VLAN tags, so the first interface inserts/removes the outermost (222) tag while the second interface the innermost one (111).

The procedure to create and activate the two interfaces is as follows:

- Create the first 802.1Q interface on top of the *veth* interface, to manage the outermost tag:

  ```
  $ ip link add link <veth_interface_name> name
      <veth_interface_name>.222 type vlan id 222
  ```

- Create the second 802.1Q interface on top of the one just created, to manage the innermost tag:

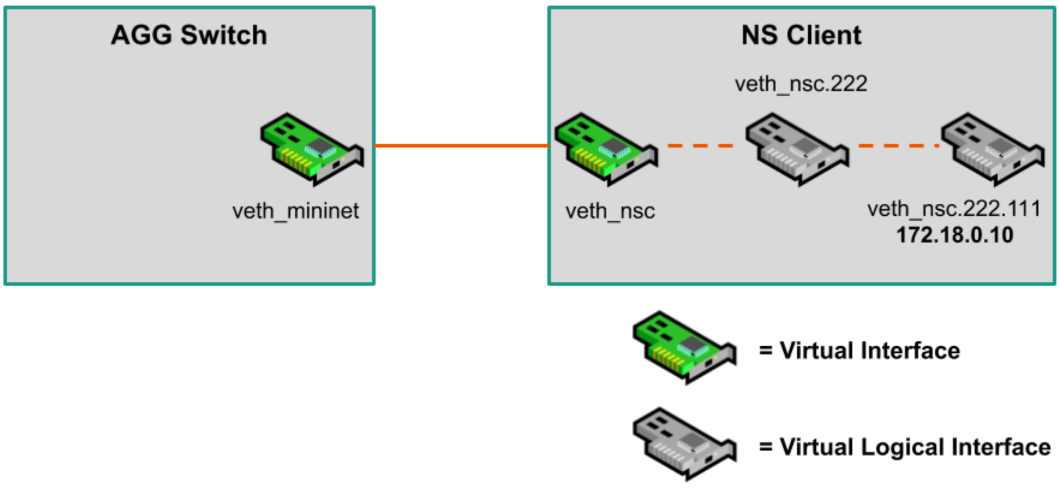

Figure 6.2: 802.1Q VLAN Tagged Interfaces Configuration

```
$ ip link add link <veth_interface_name>.222 name
    <veth_interface_name>.222.111 type vlan id 111
```

- Activate both interfaces:

```
$ ip link set dev <veth_interface_name>.222 up
$ ip link set dev <veth_interface_name>.222.111 up
```

Finally, thanks to these two new interfaces it is possible to easily solve the defalut gateway problem, assigning the same address as the BNG to the double tagged interface of NS client. In this way the traffic directed to the default gateway that arrives at the NS Client is not discarded but it is directed towards the CNFs chain without having to introduce changes to the configuration of the RG, thus making the solution more manageable. The Linux command to carry out this procedure is:

```
$ ip addr add 172.18.0.10/24 dev <veth_interface_name>.222.111
```

Figure 6.2 provides a graphical representation of the connection between the aggregation switch and the NS Client, with the virtual *veth_nsc* interface and the two logical virtual 802.1Q interfaces created on top of it.

Modify XOS configuration to steer RG's traffic

The next step to make the two systems communicate is to modify the forwarding rules of the aggregation switch to steer the traffic that originally was directed to the BNG towards the CNFs chain in NSM. Before illustrating the procedure to modify the existing forwarding rules, the mechanism of operation of XOS and the abstractions on which it is based are briefly presented.

The first abstraction used by XOS is the VLAN Cross Connect: VLAN cross connect creates a L2 bridge between two given ports on the same device. Once configured, every packet arriving at one of the ports with specific VLAN tag will be sent to another port directly. Current implementation only matches on the outermost VLAN tag. If the packet is double tagged, the S-tag will be matched and both the S-tag and C-tag will be persisted [31]. A cross connection is created on an OpenFlow device with the *segmentrouting* ONOS application, which is present by default on every ONOS instance.

This abstraction is used by the Fabric Crossconnect service in XOS [32], which creates an L2 bridge between two given ports on the same device.

The service is made up of three parts:

- *FabricCrossconnectService* global service-related parameters, such as the name of the service. There is currently no additional state here beyond the default XOS Service model;
- *FabricCrossconnectServiceInstance* represents one half of a VLAN crossconnect. Fields include the following:
  - *s-tag*: the VLAN id that will be connected;
  - *switch_datapath_id*: switch id where the VLAN crossconnect will be enacted;
  - *source_port*: the source port number on the switch.
- *BNGPortMapping* represents the other half of a VLAN crossconnect. Fields include the following:
  - *s-tag*: the VLAN id that will be connected. In addition to specifying a single id, the keyword ANY may be used, or also a range of ids;
  - *switch_port*: the output port number on the switch.

Property	**Value**
Name	<custom_name>
Owner id	fabric-crossconnect
S tag	222
Source port	2
Switch datapath id	of:0000000000000001

Table 6.1: Parameters to create the Fabric Crossconnect Service Instance

FabricCrossconnectServiceInstance and *BNGPortMapping* work together to create a VLAN crossconnect tuple, linked by a common *s-tag*.

The Fabric Crossconnect service uses the synchronization framework (Figure 5.1) to modify ONOS forwarding rules: When a *FabricCrossconnectServiceInstance* or a *BNGPortMapping* is created, updated, or deleted, the synchronizer will make a REST API call to ONOS. Appropriate cross connections are removed from ONOS and parallelly new BNG data and rules are pushed to it.

To create the cross connection in XOS there two alternatives: the GUI or the APIs exposed by XOS. In both cases all the parameters listed above are required (s-tag, switch_datapath_id, source_port and switch_port). The GUI is reachable at `http://localhost:30001` and can be accessed with *admin@opencord.org* as username and *letmein* as password.

From the home page of the system navigate to **Fabric crossconnect** -> **Fabric Crossconnect Service Instances**, delete the existing entry in the table and click the top right **Add** button. In the new page create a new instance with the values contained in Table 6.1.

Before creating the BNG Port Mapping it is necessary to discover the id of the *veth* interface connected to the aggregation switch. To find this id, simply navigate `http://localhost:30120/onos/ui/index.html#/device` (ONOS GUI) and click on the aggregation switch, called *leaf_1*, after which a window will open indicating the port id of all the switch ports.

Once the id is found (which is usually 3), navigate to **Fabric crossconnect** -> **BNGPortMappings**, delete the existing entry in the table and click the top right **Add** button. Here create a new port mapping with the values contained in Table 6.2.

Once the creation of the BNG port mapping is finished, it is sufficient to wait a few moments for XOS to synchronize the ONOS forwarding rules, after which the

Property	Value
S tag	222
Switch port	<port_id>

Table 6.2: Parameters to create the BNG Port Mapping

two systems will finally be correctly connected also from the control plan point of view. All that remains is to disable the calico source NATting to finish the systems integration.

Disable Calico source NATting

Calico CNI implements a source NATting policy by default in case a pod is trying to contact an IP address outside the Kubernetes cluster. In the case described in these pages, source NATting is enabled although the pods are all in the same cluster and this is due to the fact that the NSM pods are in a different subnet than the one of the RG.

To disable this functionality it is necessary to first install *calicoctl*, the CLI of Calico CNI:

```
$ cd /usr/local/bin
$ curl -O -L
   https://github.com/projectcalico/calicoctl/releases/
  download/v3.16.2/calicoctl
$ chmod +x calicoctl
```

Then, create a new YAML file with this content:

```
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: no-nat-rg
spec:
  cidr: 172.16.2.0/30
  disabled: true
  nodeSelector: all()
```

Where 172.16.2.0/30 is the subnet of the NS Server (the last CNF of NSM).

Finally, to disable the source NATting run the command:

```
$ DATASTORE_TYPE=kubernetes KUBECONFIG=~/.kube/config
    calicoctl apply -f <file_name>.yaml
```

After this last command all the problems have been resolved, the two systems will be perfectly integrated and they will be able to exchange traffic, more precisely the RG will be able to ping the last CNF of the NSM chain (NS Server).

6.2 Second Scenario: SiaB and NSM in two different servers

In this scenario SiaB and NSM are developed in two different servers, with two different instances of Kubernete. As mentioned at the beginning of this chapter, this scenario can be applied when the CNFs of the telecom operator do require much computing power and therefore a dedicated infrastructure is required.

A dedicated infrastructure not only brings advantages in terms of server performance management and number of CNFs deployed but also allows greater flexibility in the management of CNFs since the entire infrastructure can be changed without the risk of damaging the one that controls the central office pods, which are critical services and cannot stop working.

Furthermore, there is greater flexibility regarding the deployment of the CNFs, in fact a dedicated infrastructure can be positioned anywhere in the telecom operator's network and not necessarily in the central office.

6.2.1 Servers Configuration

Figure 6.3 represents the configuration of the two servers used in this scenario. SEBA-in-a-Box is deployed on the first server and is connected via a physical interface to the second server, where Network Service Mesh is deployed. Also on this server, there is a physical interface connected to the cluster. The connections between cluster and physical interfaces (represented in orange in the figure) are made through a *veth* pair between the host and a pod within the cluster.

The physical connection between the two servers is instead a normal Ethernet connection between the two NICs, both with an IP address assigned. A more complete characterization of the two servers will be carried out in section 7.1.

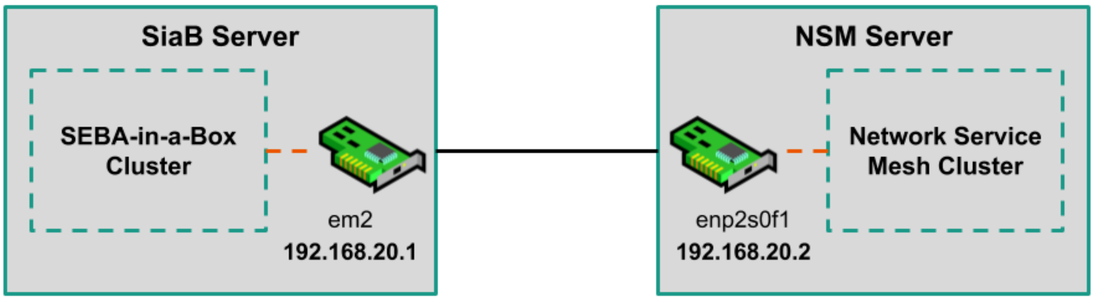

Figure 6.3: Graphical Representation of the two Servers

6.2.2 SiaB and NSM Deployment

The deployment of SiaB is exactly identical to the one illustrated in the previous sections. There is only a small difference in NSM deployment: in the NSM server, to avoid black holes or other routing problems, it is better to configure a different network CIDR for Kubernetes' pods than the one used in the SiaB cluster and the one used by the CNFs of Network Service Mesh.

Compared to the previous scenario, many of the integration procedures are the same, the main differences are in the connection of the two systems to physical servers that host them and in some additional entries in the routing tables.

6.2.3 Integration Procedures

Up to this point all the pods of SiaB are deployed in one server and all NSM pods are deployed in another one, with two different Kubernetes clusters. The two servers are connected with a Gigabit interface, as shown above.

Figure 6.4 represents the final deployment after integrating the two systems. NSM is connected to SiaB between the aggregation switch and the BNG, this time with a physical connection between the two servers. In this scenario *veth* interfaces are still used, but they are used to connect the virtual interface of a pod to the server that hosts it. Also in this case, Figure 6.4 depicts a more general scenario, where a generic CNF is represented instead of NS client.

Connect Mininet pod with the first server

This procedure to connect the Mininet pod to the host is very similar to the one described in section 6.1.3, the only difference is that only one network namespace

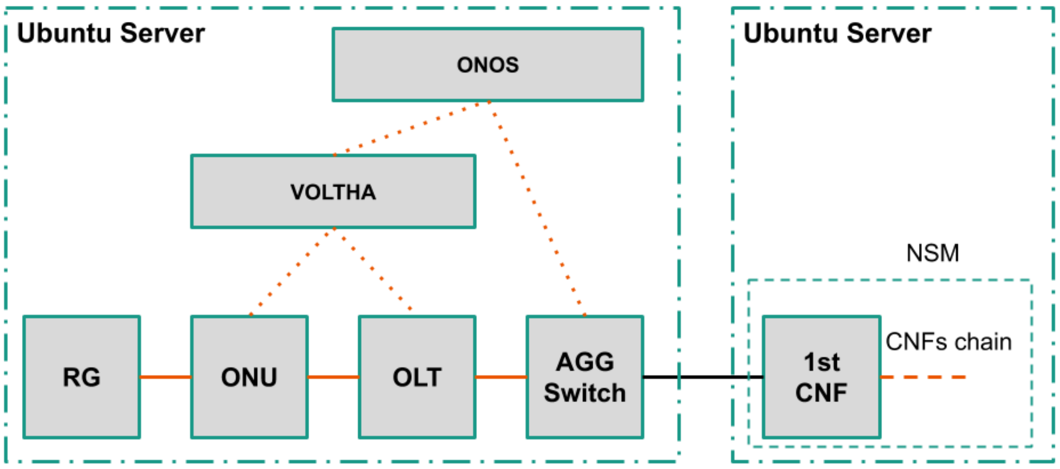

Figure 6.4: Graphical Representation of the Second Integration Scenario

is required. Here are the commands:

```
$ DOCKER_ID_MN=$( docker ps | grep <mininet_pod_name> | head
    -n 1 | awk '{print $1}' )
$ POD_NETNS_MN=$( docker inspect --format '{{ .State.Pid }}'
    $DOCKER_ID_MN )
$ ip link add <mininet_interface_name> netns $POD_NETNS_MN
    type veth peer <host_interface_name>
```

where *mininet_interface_name* and *host_interface_name* are the names assigned to the new veth interfaces in the Mininet pod and in the host. Once the two interfaces have been created activate them with the command:

```
$ ip link set dev <interface_name> up
```

In this way pod and host are connected. The next step is to connect the new interface of the Mininet pod to the aggregation switch, with the same set of commands explained in section 6.1.3.

Create double tagged VLAN interfaces on the first server

The principle of the operation of the interfaces and their creation is identical to the one explained above, the only difference is that in this case the commands are given from the host command line and not from the NS client. It is also necessary to set the IP address of the BNG on the double tagged interface, to correctly configure the default gateway of the RG. The commands are shown below for simplicity:

```
$ ip link add link <host_interface_name> name
   <host_interface_name>.222 type vlan id 222
$ ip link add link <host_interface_name>.222 name
   <host_interface_name>.222.111 type vlan id 111
$ ip link set dev <host_interface_name>.222 up
$ ip link set dev <host_interface_name>.222.111 up
$ ip addr add 172.18.0.10/24 dev <host_interface_name>.222.111
```

Once the two interfaces have been created, the following integration procedures explained in section 6.1.3 must be repeated, which do not require any modification: *Modify XOS configuration to steer RG's traffic* and *Disable Calico source NATting.*

Connect the second server with the NS Client pod

This is the part where there are the greatest differences compared to the first scenario, in fact although the commands to create the connection between host and pod are identical to those already illustrated, in this case it is also necessary to assign an IP address to each of the *veth* interfaces, otherwise the routing does not work properly.

```
$ DOCKER_ID_NSC=$( docker ps | grep <ns_client_pod_name> |
   head -n 1 | awk '{print $1}' )
$ POD_NETNS_NSC=$( docker inspect --format '{{ .State.Pid }}'
   $DOCKER_ID_NSC )
$ ip link add <ns_client_interface_name> netns $POD_NETNS_NSC
   type veth peer <host_interface_name>
```

Then, add the two IP addresses, one directly from the command line of the server and the second one from the NS Client pod command line, taking care not to create IP address conflicts:

```
ip addr add <IP_address> dev <interface_name>
```

At this point the two systems are correctly connected from both from the point of view of the data plane and the one of the control plane, it is only necessary to arrange the routing tables of the two servers and the various pods involved.

Routing tables configuration

The routing tables are not very different from the first scenario, but a greater number of entries must be added, illustrated below, divided according to the host/pod:

- First server: one route to reach the RG subnet via the *veth* interface towards the Mininet pod and one route to reach the NS Server via the physical interface towards the second server;
- Second server: one route to reach the RG subnet via the physical interface towards the first server and one route to reach the NS Server via the *veth* interface towards the NS Client pod;
- NS Client: one route towards the RG subnet via the *veth* interface towards the second server and one route to reach the NS Server through the interface connected to the firewall;
- NS Server and Firewall: their routes are identical to those of the first scenario.

The command to add a route is the usual one:

```
$ ip route add <subnet_CIDR> via <IP_address_next_hop>
```

At this point the two systems are perfectly integrated and the ping between the RG and the NS Server flows correctly in the network.

6.3 Issues and Gap Analysis

Most of the problems encountered integrating the two systems were caused by routing and by Calico's source NATting. In fact with source NATting enabled the source IP address of the RG packets is changed to the server IP and this prevented the correct forwarding of the traffic. In particular, when a packet with changed source IP arrived at the NS client it was discarded by the pod itself and was not forwarded in the NSM chain.

In any case, even if the packet had been forwarded along the chain, once it arrived at its destination it would never have had the opportunity to go back, since the NS server thinks it has to send the packet to the server and no longer to the RG.

The first solution to this problem was to manually change all the IP addresses of the NSM chain, so that they belonged to the same subnet as the RG. This solution can obviously be implemented only in the case of a very small CNFs chain, such as the one used in the thesis, and not in a real case. Moreover, in a real scenario the IP addresses assigned to users are not in the same subnet as the ones used by the telecom operator for its services. For these two reasons another solution was sought, bringing to light the phenomenon of source NATting and its subsequent disabling.

A final observation regarding this problem is that in a real case the source NATting would not be present since the RG is not a pod but a physical device in the user's home and therefore there would be no need to introduce further configurations in the cluster.

As far as the gap analysis, currently the system created by the integration of SEBA and NSM works, at least in its fully virtualized version. Although the integration works, it still requires a lot of manual interventions which, although reduced in a real case (for example there would be no need to create *veth* interfaces for the connection between pods or to disable the source NATting), cannot be performed in each central office of a telecom operator.

In the event that this integration occurs with a non-virtualized version of SEBA and in a real production scenario, the new system should certainly be accompanied by a controller or other mechanism that allows all the operations described in the chapter to be carried out, but also the dynamic deployment of CNF chains, in a more automated way, for example by leveraging ONOS and XOS APIs or custom scripts.

Chapter 7

Experimental Validation

This chapter analyzes the performance of SiaB and the system formed by SiaB and NSM in both scenarios, explaining the reasons why it was difficult and at times impossible to carry out real performance tests.

In any case, even if it had been possible to carry out real performance tests, since SiaB is a completely virtualized environment compared to a real SEBA installation, the tests carried out would represent an upper limit in terms of system performance, since generally virtual networking is faster than the real one. In the particular case of this thesis instead, due to some SiaB developers' implementation choices, the system performance is low even if the environment is virtual, and this had an impact on the tests and their feasibility.

Finally, the results obtained in the integration phase and those that have yet to be achieved to create a complete and automated system were summarized, also explaining the reasons why it was not possible to achieve them during the thesis.

7.1 Test Environment

Before explaining the tests and results obtained, a more detailed characterization of the environment is provided. The two servers used to carry out the integration procedures and the tests are located in one of TIM's network laboratories, company that collaborated in the development of this thesis. Here are hardware and software information about the environment:

- First server hardware characteristics (the one used in the first scenario and as the SiaB server in the second one):

  - Processor: Intel Xeon 2.4GHz, 8 cores;
  - Memory: 64 GB;
  - Disk: 1 TB HDD.

- Second server hardware characteristics (the one used in the second scenario as the NSM server):
  - Processor: Intel Xeon 2.6GHz, 12 cores;
  - Memory: 384 GB;
  - Disk: 4 * 900 GB HDD.
- Application level characteristics for both servers:
  - Ubuntu version: 16.04;
  - Kubernetes version: 1.12.7;
  - Kubernetes deployment: single node with Kubeadm[1];
  - Kubernetes CNI: Calico v3.3;
  - Docker version: 17.06;
  - Helm version: 2.14.2;
- Characteristics of the connection between the two servers:
  - Protocol: Ethernet;
  - Speed: 1 Gb;
  - Cable type: UTP RJ45.

7.2 Tests

This section deals with two different types of tests: throughput tests and system latency tests. The original idea was to run these tests to provide an upper limit to system performance of SEBA and those of the system made of SEBA and NSM, as virtual networking is usually faster than standard hardware. Carrying out the tests, however, it was seen that the system performance is very low and that in

[1]`https://kubernetes.io/docs/reference/setup-tools/kubeadm/`

general SiaB does not support large amounts of traffic, due to the usage of gRPC to simulate network traffic, making it impossible to test real traffic scenarios. In addition to the presentation of the results obtained, the reason for which gRPC is the cause of the slowness of the system will also be addressed in more detail, also showing the comparison with a non-gRPC connection.

7.2.1 Throughput Tests

Throughput test were carried out with Iperf3[2], with both TCP and UDP protocols.

Unfortunately the tests carried out with both protocols did not bring any results in both scenarios, since starting the tests causes immediate system failure. The cause of this break, which was also reported in the official SEBA Jira page[3] and currently has no solution, is due to the use of the gRPC protocol in the connection between the ONU and the OLT in PONSIM.

gRPC (Figure 7.1) is a modern open source high performance RPC framework. It can efficiently connect services in and across data centers. In gRPC, a client application can directly call a method on a server application on a different machine as if it were a local object [33].

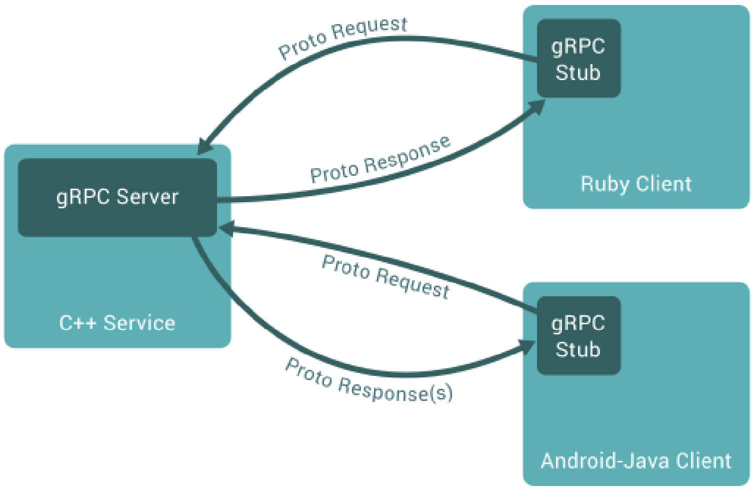

Figure 7.1: gRPC Overview [33]

[2]https://iperf.fr/

[3]https://jira.opencord.org/browse/SEBA-711?page=com.atlassian.jira.plugin.system.issuetabpanels%3Aall-tabpanel

As can be seen from this brief description, gRPC was not born to simulate an L2 network connection, in fact in the case of SiaB the ONU does not send a packet to the OLT in the usual way but instead calls a method within it, passing it as parameters the data related to the user's packet. Due to this implementation choice there is a large overhead in this phase of transmission of a packet, which becomes unsustainable in the case of Iperf3 tests, where the system is flooded with packets. Failing to dispose of the excessive number of requests, the system interrupts the connection between ONU and OLT, entering a state of error from which it is no longer possible to exit. The system crashes even with the smallest bandwidth value available for Iperf tests, which is 100 Kb/s.

Finally, to further investigate the system overhead caused by gRPC, latency tests have been performed on the system, also in this case in both scenarios.

7.2.2 Latency Tests

Latency tests were performed with the standard ping command and their purpose was to further investigate the overhead generated by the gRPC protocol. In both scenarios the following latency tests were performed:

- Ping between the RG and the BNG: this test was carried out to analyze the latency of the simulated environment as it is without NSM, to have a reference point for comparisons with the results of the other tests. This test that has been performed only in the first scenario, as SiaB is deployed in both cases only on the first server;

- Ping between the RG and the NS Client: this test was carried out to analyze the impact on the SiaB system of the change of a single element of the chain, in fact now the RG connects to the NS Client which is a separate pod and no longer to the BNG, which was instead emulated in mininet in the same pod of the aggregation switch. However, the number of devices that make up the chain does not change;

- Ping between the RG and the NS Server: this test aims to investigate the latency of the system obtained from the integration between SiaB and NSM, to investigate if the addition of the pods that make up the CNF chain has an impact on the ping results;

- Ping between the NS Client and the NS Server: this last test was carried out only in the second scenario, to test NSM alone, to have a reference regarding the latency of a virtualized system without gRPC, in order to evaluate the overhead introduced by the gRPC protocol. The test is particularly significant since also in this case it is not a direct ping between two pods, but the packet must cross a chain of pods, as in the case of SiaB.

In the next paragraphs the results of both scenarios will be presented, followed by a discussion on the data obtained. All tests consist of 50 pings with standard settings, of which only the first three and the last three are shown for each case. At the end of each section also a summary of the results is provided, where the minimum, average and maximum round trip times are indicated.

First Scenario: ping RG - BNG

In this first scenario it is already possible see how the latency of SiaB alone is very high for a fully virtualized system.

Sequence Number	Round Trip Time
1	25.0
2	18.9
3	18.4
...	...
48	33.5
49	20.8
50	34.7

Min RTT	Avg RTT	Max RTT
16.179	25.179	34.738

Table 7.1: First Scenario: Ping between RG and BNG

First Scenario: ping RG - NS Client

In this second case of the first scenario it is possible to see that adding a new pod to the chain increases the round trip time but not significantly, which confirms the hypothesis that gRPC is the system bottleneck.

Sequence Number	Round Trip Time
1	52.4
2	23.9
3	30.8
...	...
48	35.8
49	23.8
50	22.4

Min RTT	Avg RTT	Max RTT
18.608	26.430	52.498

Table 7.2: First Scenario: Ping between RG and NS Client

First Scenario: ping RG - NS Server

In this test of the two integrated systems it can be observed once again how the addition of two other pods to the chain does not entail a substantial difference in the values obtained, on the contrary there is a slight improvement. Also in this case the results show that the high latency is due to the SiaB system and not to the integration of the two systems.

Sequence Number	Round Trip Time
1	27.0
2	28.4
3	19.9
...	...
48	24.8
49	38.6
50	25.5

Min RTT	Avg RTT	Max RTT
18.564	26.299	38.636

Table 7.3: First Scenario: Ping between RG and BNG

Second Scenario: ping RG - NS Client

In this test with the two integrated systems on two different servers it can be seen how the separate deployment has almost no effect on the results obtained, which remain in line with the same case in the previous scenario.

Sequence Number	Round Trip Time
1	27.9
2	17.8
3	27.1
...	...
48	29.9
49	21.6
50	18.5

Min RTT	Avg RTT	Max RTT
16.895	27.136	39.941

Table 7.4: Second Scenario: Ping between RG and NS Client

Second Scenario: ping RG - NS Server

Also in this last test of the two systems it can be observed that the results do not differ much from those obtained in the previous scenario.

Sequence Number	Round Trip Time
1	28.9
2	18.4
3	27.4
...	...
48	30.4
49	23.3
50	22.0

Min RTT	Avg RTT	Max RTT
17.154	27.543	40.259

Table 7.5: Second Scenario: Ping between RG and NS Server

Second Scenario: ping NS Client - NS Server

This test is used as a reference to evaluate the latency of a fully virtualized system that does not use gRPC to connect pods together. As it is possible to see in the table, the latency of the system is very low compared to that of SiaB.

Sequence Number	Round Trip Time
1	0.122
2	0.061
3	0.065
...	...
48	0.074
49	0.082
50	0.072

Min RTT	Avg RTT	Max RTT
0.039	0.068	0.122

Table 7.6: Second Scenario: Ping between NS Client and NS Server

7.2.3 Final Discussion

The results of all latency tests in both scenarios confirmed the initial hypothesis that the cause of the system's inability to handle large amounts of traffic was the gRPC protocol. As can be seen from Table 7.1, the SiaB system alone has a very high latency, on average equal to 25 ms.

In all the other tests carried out, the average latency value never differs much from 25 ms (the maximum value is 27.543 ms in Table 7.5, obtained in the ping test between RG and NS Server in the second scenario), it can therefore be concluded that the integration of the two systems did not lead to significant decreases in performance. Comparing the values obtained in these tests with the values obtained in the ping between NS Client and NS Server it can be seen how NSM, which does not use gRPC but only *veth* interfaces, has instead a very low latency compared to SiaB, well below the millisecond.

It can therefore be concluded that gRPC is the cause of the high system overhead. Thanks to the latency data it is also possible to try to calculate the maximum throughput supported by SiaB: gRPC is a protocol that is based on requests made by a client to a server, so its throughput can be measured in requests per second (req/sec). A ping from the RG to the BNG takes about 25 ms to cycle through the pod chain twice, so it can be assumed that it takes about 12.5 ms to cycle through the chain once. As previously mentioned (Table 7.6) a ping between two or more pods connected by *veth* interfaces has a latency of less than one millisecond, so it is possible to assume that a gRPC request from the ONU to the OLT takes about 12 ms to be satisfied. The maximum throughput supported by gRPC is therefore:

$$\frac{1}{12 \times 10^{-3}} = 83.3 \sim 83 \; req/sec$$

gRPC supports about 83 pings per second, therefore knowing that the size of an ICMP packet that makes up a ping is 84 bytes (64 + 20) in total and that each request corresponds to a package, the following throughput is obtained:

$$83 \times (84 \times 8) = 55776 \; b/s \sim 56 \; Kb/s$$

Given this very low result, the breakdown of the system is therefore understandable even in the case of the minimum throughput of Iperf3 (100 Kb/s), which is almost double the one supported by SiaB.

Finally, it is fair to observe that these results refer only to SiaB in this particular context and do not concern the performance of SEBA, which should instead be tested more rigorously in a real scenario. Furthermore, it can be concluded that SiaB was not designed to simulate real traffic loads or to perform performance tests but was developed to be able to bring developers and operators closer to the SEBA project, allowing them to test new systems on SEBA and possible integrations of the framework.

7.3 Integration Summary

In this last section of the chapter there is a table (Table 7.7) that includes a summary of all the requirements for a correct integration of the two systems and their availability. For each element is not only reported its availability but it is also briefly explained how it was possible to implement that requirement. Finally, as can be seen in the table, only the last three requirements are not available, because they went beyond the objective of this thesis or due to some intrinsic limitations of SEBA-in-a-Box which cannot be overcome.

Requirement	**Availability**
SiaB deployment on a single server	Yes
SiaB double tagged traffic management in NSM	Yes, thanks to two specially created interfaces that untag incoming traffic and tag outgoing traffic
SiaB and NSM connection on a single server	Yes, the two systems are connected through a *veth* pair
SiaB and NSM connection on two different servers	Yes, the two systems are connected through two *veth* pairs and a physical Ethernet connection between the servers
Calico source NATting disabled	Yes, otherwise the IP–src of a packet would be modified, preventing its correct forwarding in the chain
Siab and NSM can correctly exchange traffic in both single server and two servers' cases	Yes
Traffic steering for one subscriber towards a single NFV chain	Yes, thanks to ONOS flow rule injection in the AGG switch made by XOS
Traffic steering for more than one subscriber towards a single NFV chain	Not available (not tested)
Traffic steering for more than one subscriber towards dedicated NFV chains	Not available, because it is more concerned with the NSM environment than the SEBA one
Traffic steering automation: upon subscription, associate the subscriber to a NFV chain, configure data-path and activate the chain	Not available, because SiaB currently does not support the dynamic subscription of new RGs, they can be configured only at deployment time

Table 7.7: Integration Procedures Summary

Chapter 8

Conclusions and Future Works

The SEBA framework is one of the most promising solution to manage residential broadband access in a cloud-native way. Exploiting SDN, NFV, Kubernetes and other technologies it provides a cloud-native, open and programmable architecture to manage the central office and the access network of an internet service provider. Furthermore, the introduction of a cloud environment closer to residential users allows telecom operators to introduce other frameworks into their infrastructure, such as Network Service Mesh.

After the analysis carried out in this thesis, following positive aspects of SEBA can be pointed out:

- SEBA allows telephone operators to program their own residential access network, while reducing the costs to build the network itself, introducing a pay-as-you-grow model to eliminate wasteful overprovisioning of network equipment;

- SEBA introduces greater agility into the network thanks to the separation of control plane and data plane, reducing the cost of operations and the equipment requirements;

- SEBA allows telecom operators to manage all central office devices through a single interface and to define their own workflow without the framework needing changes.

Another very positive aspect of SEBA is that its community, and more generally the Open Network Foundation community, is very active in developing the software that makes up the framework and also really open on its social channels, where developers try to help people asking questions as much as possible.

As far as the integration between SEBA and Network Service Mesh, the advantages are not limited to the possibility of offering network services deployed on NSM to residential users but some of the intrinsic limitations of NSM are also solved. As highlighted in section 3.2, two of the main characteristics of NSM are the impossibility to create a dynamic chain of Cloud-Native Network Functions and the missing support to load balancing between the various CNFs. Thanks to the characteristics of SEBA and in particular to the possibility of steering users' traffic in the aggregation switch through the injection of precise rules, both these problems find a solution, albeit partial:

- Impossibility to create a dynamic chain of CNFs: this problem can be solved by creating a new CNFs chain in NSM with the desired changes. To direct the traffic to the new chain without service interruptions, it is enough to change the forwarding rule in the aggregation switch;
- Missing load balancing support: also in this case the problem can be solved by creating a new CNFs chain identical to the one it has to be balanced but in this scenario all traffic should not be moved from one chain to another but rather balanced according to the parameters of the packets (for example the IP address or other possibilities offered by ONOS). Although this is not really a traditional load balancer and is very limited in functionality, it still allows to distribute traffic over two or more chains.

Finally, as far as the possible future developments of this thesis, the main one consists in increasing the automation of the system, for example by making automatic the deployment of network interfaces and the creation of VLAN interfaces to connect SEBA to NSM. Further developments may otherwise concern the tests carried out on the system and the integration scenarios: it would be interesting to try to integrate the two frameworks in a real case with SEBA and not with SiaB and try to steer user traffic with more residential gateways and/or more chains of CNFs. In a real case scenario it would finally be interesting to carry out the performance tests that did not provide a result in the case studied in this thesis.

Bibliography

[1] *Software Defined Networking.* `https://www.vmware.com/topics/glossary/content/software-defined-networking.html` (cit. on p. 3).

[2] *Cloud Native Network Functions.* `https://ligato.io/cnf/cnf-def` (cit. on pp. 5, 6).

[3] *Kubernetes Docs.* `https://kubernetes.io/docs/home` (cit. on p. 7).

[4] *Kubernetes Architecture.* `https://platform9.com/blog/kubernetes-enterprise-chapter-2-kubernetes-architecture-concepts` (cit. on p. 7).

[5] *Passive Optical Network.* `https://www.viavisolutions.com/en-us/passive-optical-network-pon` (cit. on p. 10).

[6] *Passive Optical Neworks and Optical Distribution Network.* Tech. rep. Northforge Innovations Inc., Feb. 2018 (cit. on p. 10).

[7] *OLT, ONU and ODN.* `https://community.fs.com/blog/abc-of-pon-understanding-olt-onu-ont-and-odn.html` (cit. on pp. 11, 12).

[8] R. G. Trani. «Integrating VNF Service Chains in Kubernetes Cluster». MA thesis. Politecnico di Torino, Mar. 2020 (cit. on pp. 13, 15, 19, 20).

[9] *Network Service Mesh.* `https://networkservicemesh.io/docs/concepts/what-is-nsm` (cit. on p. 14).

[10] *Northforge Solutions for PON using CORD and VOLTHA.* `https://gonorthforge.com/northforge-solutions-for-pon-using-cord-and-voltha` (cit. on p. 25).

[11] *Migration to Ethernet-Based Broadband Aggregation.* Tech. rep. The Broadband Forum, July 2011 (cit. on p. 25).

[12] *Broadband Network Gateway Overview.* https://www.cisco.com/c/en/us/td/docs/routers/asr9000/software/asr9k_r4-2/bng/configuration/guide/b_bng_cg42asr9k/b_bng_cg42asr9k_chapter_01.pdf (cit. on p. 26).

[13] *Central Office Re-Architected as a Data Center.* https://opennetworking.org/cord/ (cit. on p. 27).

[14] *CORD Architectural Requirements.* https://wiki.opencord.org/display/CORD/CORD+Requirements (cit. on p. 27).

[15] Larry Peterson, Ali Al-Shabibi, Tom Anshutz, Scott Baker, Andy Bavier, Saurav Das, Jonathan Hart, Guru Palukar, and William Snow. *Central Office Re-Architected as a Data Center.* Tech. rep. IEEE, Oct. 2016 (cit. on pp. 28, 29, 32, 39).

[16] *Residential CORD.* https://opennetworking.org/r-cord (cit. on p. 30).

[17] *SEBA (SDN Enabled Broadband Access) – The Next Generation of Broadband Access.* https://gonorthforge.com/seba-sdn-enabled-broadband-access-the-next-generation-of-broadband-access/ (cit. on p. 31).

[18] *SDN Enabled Broadband Access (SEBA) - Reference Design.* Tech. rep. ONF, Mar. 2019 (cit. on pp. 31, 36).

[19] *VOLTHA Documentation.* https://docs.voltha.org/master/index.html (cit. on p. 34).

[20] *ONOS Overview.* https://wiki.onosproject.org/display/ONOS/ONOS+%3A+An+Overview (cit. on p. 36).

[21] *ONOS System Components.* https://wiki.onosproject.org/display/ONOS/System+Components (cit. on pp. 37, 38).

[22] *ONOS Application Subsystem.* https://wiki.onosproject.org/display/ONOS/Application+Subsystem (cit. on p. 38).

[23] *ONOS Device Subsystem.* https://wiki.onosproject.org/display/ONOS/Device+Subsystem (cit. on p. 38).

[24] *ONOS Flow Rule Subsystem.* https://wiki.onosproject.org/display/ONOS/Flow+Rule+Subsystem (cit. on p. 39).

[25] *XOS and NEM.* https://wiki.opencord.org/display/CORD/XOS+and+NEM+Background+Information (cit. on pp. 39, 41).

[26] *SEBA-in-a-Box.* `https://guide.opencord.org/profiles/seba/siab.html` (cit. on pp. 44, 49).

[27] *Mininet Overview.* `http://mininet.org/overview` (cit. on p. 45).

[28] *Mininet in SiaB.* `https://guide.opencord.org/charts/mininet.html` (cit. on p. 45).

[29] *AT&T Workflow.* `https://docs.google.com/document/d/1nou2c8AsRzhaDJmA_eYvFgd0Y33KiCsioveU77A0VCI/edit#heading=h.x73smxj2xaib` (cit. on p. 46).

[30] *CoreDNS Pods in Kubernetes.* `https://kubernetes.io/docs/tasks/administer-cluster/coredns` (cit. on p. 53).

[31] *VLAN Cross Connect.* `https://wiki.opencord.org/display/CORD/VLAN+Cross+Connect` (cit. on p. 60).

[32] *Fabric Crossconnect Service.* `https://guide.opencord.org/fabric-crossconnect` (cit. on p. 60).

[33] *Introduction to gRPC.* `https://grpc.io/docs/what-is-grpc/introduction` (cit. on p. 71).

Acknowledgements

Mi sembra impossibile siano già passati due anni dall'ultima volta che mi sono ritrovato qui, alla fine di una tesi e di un percorso universitario che mi ha portato tanto, a scrivere i ringraziamenti.

Desidero ringraziare i miei relatori, prof. Fulvio Risso e Guido Marchetto che non solo mi hanno accompagnato in questo percorso di tesi ma mi hanno fatto appassionare alle reti e al cloud computing, temi centrali di questo elaborato e del mio futuro lavorativo. Assieme a loro voglio ringraziare l'ing. Roberto Morro di TIM e Raffaele Trani, che mi hanno supportato in ogni singola fase di questa tesi, soprattutto nei temi più tecnici.

Il grazie più importante va a mamma Alessandra e papà Enrico, che non hanno mai smesso di sostenermi e di spronarmi a dare il meglio di me in qualunque situazione, rendendomi una persona migliore sotto tutti i punti di vista. Grazie anche a tutta la mia famiglia: nonni, cugini e zii, perché in questi anni ho avuto modo di vedere molte famiglie diverse e mi sono reso conto di quanto io sia fortunato ad avere voi al mio fianco.

Grazie a Ilaria, sei stata la mia coinquilina, la mia migliore amica e ora sei molto di più. Mi sei stata vicina in ogni momento di questa tesi, credendo in me ed aiutandomi in tutti i modi possibili, il tuo supporto è stato essenziale.

Infine, ma non certo per ordine di importanza, voglio ringraziare tutti gli amici che mi hanno accompagnato in questi due anni, rendendo Torino la mia seconda casa. Grazie quindi a Simone e Riccardo, siete la mia seconda famiglia e due dei più cari amici che io abbia mai trovato, vi voglio un bene dell'anima. Grazie a Enrico, Giulia e Daniele che, assieme a Riccardo e Simone, hanno reso questi due anni indimenticabili, pieni di gioie, risate e serate passate tutti assieme. Un grazie speciale poi a tutte le amiche di UniTo, che hanno portato altra gioia e bei momenti in quest'ultimo anno assieme. Senza di voi non avrei mai potuto superare tutti gli ostacoli che mi si sono posti davanti, e ve ne sarò per sempre grato.

Francesco