ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

MASTER'S DEGREE
IN
TELECOMMUNICATIONS ENGINEERING

INTENT-BASED APPROACH TO VIRTUALIZED INFRASTRUCTURE MANAGEMENT IN SDN/NFV DEPLOYMENTS

Master Thesis
in
Telecommunication Networks Laboratory T

Supervisor
Prof. WALTER CERRONI

Candidate
GIANLUCA DAVOLI

Co-supervisor
Dr. CHIARA CONTOLI

SESSION III
ACADEMIC YEAR 2015/2016

To those who have always supported me,
in every possible way.

Contents

Sommario **vii**

Abstract **ix**

1 Introduction **1**

2 Emerging Software-oriented Network Paradigms **5**
2.1 Software Defined Networking 5
2.1.1 The SDN controller 6
2.1.2 OpenFlow 8
2.2 Cloud Computing 10
2.3 Network Function Virtualization 11
2.3.1 The NVF-MANO framework 14
2.4 Software tools 14
2.4.1 ONOS 15
2.4.2 Mininet 17
2.4.3 OpenStack 18

3 Reference Architecture and Interface Definition **21**
3.1 VIM Northbound Interface 23

4 Specific Deployments **27**
4.1 SFC in the IoT/Cloud deployment 27
4.1.1 IoT SDN Domain 28
4.1.2 OpenFlow and Cloud Domains 31
4.2 Dynamic multi-domain SFC 35

5 OpenFlow/Cloud domain implementation **37**
5.1 Setup of the preliminary local environment 39
5.1.1 Creation of the Virtual Machines 39
5.1.2 ONOS Setup 42
5.1.3 Setup of the IDE 47
5.1.4 How to deploy ONOS again 48
5.2 Setup of the host server 49
5.2.1 Configuration of the ONOS VMs 51
5.2.2 Setup of the Mininet network 52
5.2.3 Configuration of the Web server VM 53
5.2.4 POST message handler and server homepage 55
5.2.5 Additional configurations on `deisnet213` 56
5.3 Modifications on the ONOS application 57
5.3.1 JSON cluster descriptors and requests 57
5.3.2 Waypoint Constraint 58
5.3.3 QoS requirements management 58

6 Performance evaluation **67**
6.1 Measurements of data plane latency 67
6.2 Measurements of control plane delay 70

7 Conclusions **75**

A Additional notes, and code **77**
A.1 SSH on Windows 77
A.2 Python script that builds the Mininet cluster 78
A.3 PHP script implementing the POST handler 87

Acknowledgements **93**

Sommario

Le reti di telecomunicazioni vengono progettate per essere efficienti, trasparenti, e accessibili all'insieme di utenti più vasto possibile. Tuttavia, esse rimangono sistemi complessi, nei quali cooperano un grande numero di componenti, mettendo alla prova la desiderata efficienza, trasparenza e facilità di accesso.

Alcuni paradigmi emergenti, tra cui Software Defined Networking (SDN), Network Function Virtualization (NFV), e Cloud Computing spianano la strada ad un nuovo ventaglio di possibili applicazioni per l'infrastruttura, o a nuove soluzioni per implementare servizi tradizionali, al prezzo, tuttavia, di aggiungere nuovi ambienti necessitanti di coordinazione.

Una delle sfide principali nel fornire concatenazioni end-to-end di servizi distribuiti su multipli domini SDN, NFV e Cloud sta nell'ottenere funzioni di gestione ed orchestrazione unificate. Un aspetto di importanza particolarmente critica è la definizione di una *northbound interface* (NBI) ad accesso aperto, non brandizzata, ed interoperabile, che astragga il più possibile dalle tecnologie di piano dati e di piano di controllo specifiche al dominio, facilitando l'accesso all'infrastruttura sottostante, senza però rinunciare ad un certo grado di flessibilità e libertà nella programmazione della rete.

In questo documento, viene descritta un'architettura di riferimento, ed espansa una NBI basata su *intent* per l'orchestrazione di servizi end-to-end attraverso domini tecnologici multipli. Nello specifico, viene considerato come caso d'uso un deployment di dispositivi per l'Internet of Things (IoT) e i corrispondenti servizi di raccolta, elaborazione e pubblicazione dei dati basati su Cloud, in grado di distinguere multiple classi di Qualità di Servizio (QoS). Infine, viene descritta e riportata una validazione sperimentale su un test-bed SDN eterogeneo e multi-dominio dell'architettura proposta.

Abstract

Telecommunication networks are meant to be efficient, transparent, and accessible to the broadest possible set of users. However, they are very complex systems, in which a large number of components cooperates, posing a challenge to the desired efficiency, transparency and ease of access.

Emerging technological paradigms such as Software Defined Networking (SDN), Network Function Virtualization (NFV) and Cloud Computing open up to a whole new set of possible applications for the infrastructure, or better ways to implement traditional services, but also introduce new environments to be controlled.

One of the main challenges in delivering end-to-end service chains across multiple SDN, NFV and Cloud domains is to achieve unified management and orchestration functions. A very critical aspect is the definition of an open, vendor-agnostic, and interoperable northbound interface (NBI) that should be as abstracted as possible from domain-specific data and control plane technologies, making the underlying infrastructure easier to be accessed, while still allowing a fair amount of flexibility and freedom in programmability of the network.

In this document we describe a reference architecture and expand an intent-based NBI for end-to-end service orchestration across multiple technological domains. More specifically, we consider the use case of an Internet of Things (IoT) infrastructure deployment and the corresponding Cloud-based data collection, processing, and publishing services, differentiating multiple Quality of Service (QoS) classes.

Finally we report the experimental validation of the proposed architecture over a heterogeneous, multi-domain SDN test bed.

Chapter 1

Introduction

In the last few years, telecommunications have become as pervasive as they had never been before. Connectivity is required everywhere, as an ever-increasing number of devices needs to access the Internet to share or retrieve data. However, physical connectivity is just the tip of the iceberg, in the much broader scenario of end-to-end communication of data. In fact, a larger number of connected devices implies a much higher demand for communication resources. Moreover, each application and service running on those devices generates its own flow of data, which need to be handled and delivered by the underlying network infrastructure. This heterogeneity has caused current networks to become very complex to be built and managed, let alone provided with new features. The deployment of such new features is also obstructed by the phenomenon known as *vendor lock-in*, caused by the large-scale deployment of proprietary solutions, which in the long term has led to the ossification of the network [1]. It is clear that, in order to face the ever-increasing needs of the users in an efficient way, the network must evolve to a more flexible, customizable, and cost-efficient system.

Software Defined Networking (SDN) plays a major role in the aforementioned network evolution. SDN decouples the data plane from the control plane, resulting in a more flexible programmability of communication resources. This is achieved by moving the control logic out of the network devices, and into an external entity, referred to as *SDN Controller*, which hosts a running instance of a *Network Operating System*. This way, infrastructures which traditionally suffered from vendor lock-in are turned into communication platforms that are fully programmable via a standardized, open interface

[2].

Network Function Virtualization (NFV) is often complementary to SDN in tackling the same issues. NFV allows network functionalities to be dispatched as software-based building blocks, which can then be used to build complex *Service Function Chains* (SFC) in a completely flexible way. This also greatly simplifies the process of maintaining and upgrading the network's functionality, as it is sufficient to act on the original code of the virtualized network function to have it modified on all of its running instances in the network.

A third and important new paradigm in the current network's evolution is **Cloud Computing**. The paradigm allows service providers to offer network services, computing resources and storage space to users, fitting in the same utility model adopted by electric power or gas suppliers. To this aim, it takes advantage of both hardware and software resources, which are distributed and virtualized in the network. [3]

The joint adoption of SDN and NFV provides enhanced flexibility to service deployment: the SFC, i.e., the sequence of network functions to be applied to data flows exchanged by a given customer (or set of customers), can be dynamically controlled and modified over a relatively small time scale, and with significantly reduced management burden compared to traditional network infrastructures [5]. Moreover, the integration with Cloud solutions, as shown in Figure 1.1, contributes to complete virtualization and scalability with resource sharing and pooling.

The integration of those three paradigms provides unprecedented control and management power over network resources, thus leading to the need of an efficient solution for the orchestration of the whole multi-paradigm system. Moreover, due to the pervasiveness of telecommunication services that we mentioned at the beginning of this Chapter, heterogeneous and multi-domain deployments must be taken into account as typical use cases. It is clear that orchestrating such implementations demands for the development of innovative solutions, based on abstractions allowing, on the one hand, the decoupling from technological details, and, on the other hand, the preservation of expressiveness and efficacy in the way the system is programmed, as well as in the control of the network infrastructures. This approach is known as the *intent-based approach* [6], and it is one of the current top-trending research topics in the field of networking.

The activities of this thesis focus on the description of the development of

an interface for intent-based unified management and orchestration of end-to-end services across heterogeneous, multi-domain deployments.

In Chapter 2 we are going to go deeper in the description of SDN, NFV and Cloud, as well as introducing the main software tools we used for our activities.

Chapter 3 is where we present our reference architecture, and we define the NBI we are going to focus on in the rest of the work.

In Chapter 4 we are going to describe the specific deployments we worked on, while in Chapter 5 we cover the implementation details of a part of such deployments.

Chapter 6 contains the description and results of the measurements conducted for performance evaluation in our test bed.

Finally, in Chapter 7 we state our conclusions on the achieved results and suggest some possible future developments.

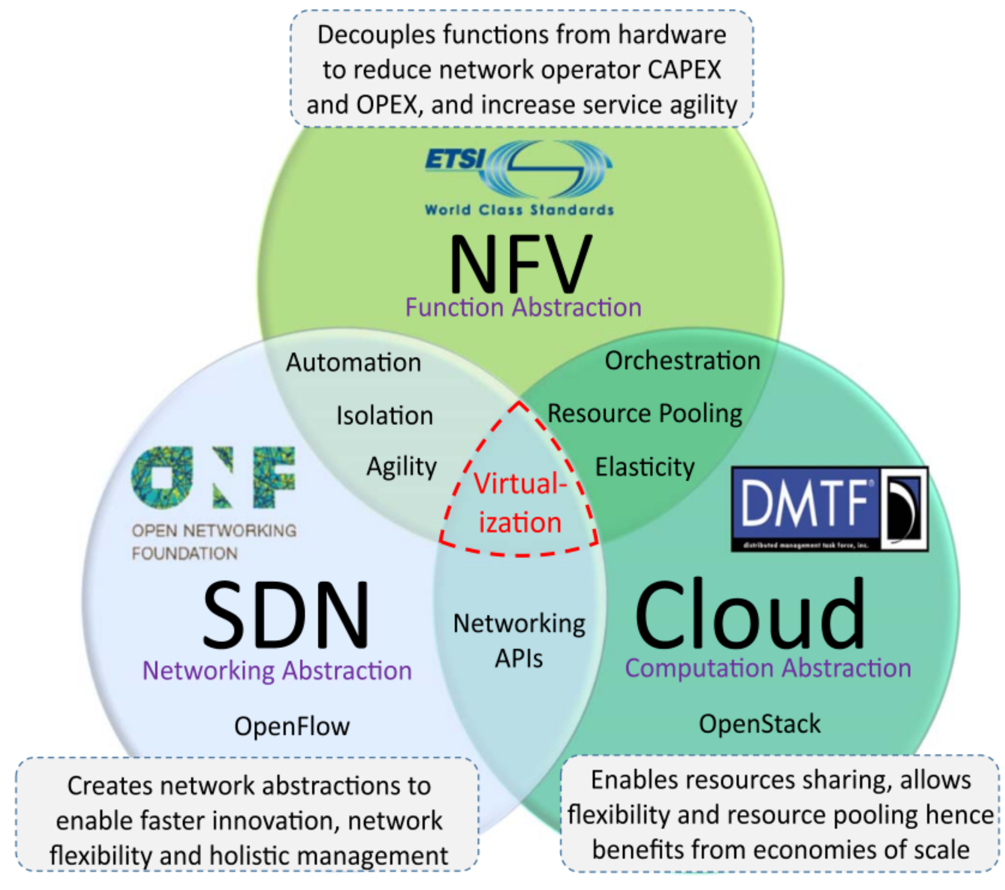


Figure 1.1: The interdependency of SDN, NFV and Cloud [4]

Chapter 2

Emerging Software-oriented Network Paradigms

As summarized in Chapter 1, Software Defined Networking, Network Function Virtualization and Cloud Computing are pushing the world of telecommunication networks to a scenario with an unprecedented central role of software aspects. It is worth examining those models a little further before describing the use we made of them in our activities.

2.1 Software Defined Networking

The SDN paradigm is aimed at supporting the dynamic and scalable computing and storage needs of modern telecommunication environments, by decoupling the control plane (i.e., the part of the system that makes decisions on how to forward packets) from the data plane (i.e., the part of the system that physically receives, stores and forwards the packets) [7].

In most SDN implementations, the communication between the two planes is carried out by means of the **OpenFlow** protocol, which allows remote administration of packet forwarding tables in network devices, by adding, modifying and removing packet matching rules and associated actions. More details on OpenFlow will follow in Section 2.1.2.

In the SDN architecture:

- network control is directly programmable, as it is decoupled from forwarding functionalities, and this abstraction allows administrators to

dynamically adjust network-wide traffic so as to meet the evolving need of the different flows in the network;

- network intelligence is (logically) centralized in software-based SDN controllers that maintain a global view of the network, which appears to applications and policy engines as a single, logical entity;

- network managers can configure, manage, secure and optimize network resources very quickly via dynamic, automated SDN programs, which can be written by the managers themselves, thanks to the open-source nature of the software used;

- network design and operation are simplified because forwarding instructions are provided by SDN controllers instead of multiple, vendor-specific devices and protocols.

SDN was first standardized in 2011 by the Open Networking Foundation (ONF), which is self-defined as *"a user-driven organization dedicated to the promotion and adoption of SDN, and implementing SDN through open standards, necessary to move the networking industry forward."* [8]

ONF is the entity behind the standardization of the OpenFlow protocol, which also inherently standardizes the interface between now-decoupled control and data planes, enabling SDN deployment.

2.1.1 The SDN controller

As previously stated, the SDN controller is a logically centralized, software-based entity that is in charge of controlling network devices operating in the data plane; it takes advantage of the global view it has over the network for running applications aimed at management, security and optimization of the resources it controls.

As it is shown in Figure 2.1, the SDN controller can be logically placed in a *Control Plane* located between the *Data Plane*, where network devices operate the actual packet forwarding, and the *Application Plane*, where SDN applications request specific services to the underlying infrastructure, based on the network state or on specific events. However, in order to communicate with the controller, proper interfaces must be defined. The interface between Application and Control planes is usually referred to as the *Northbound Interface*

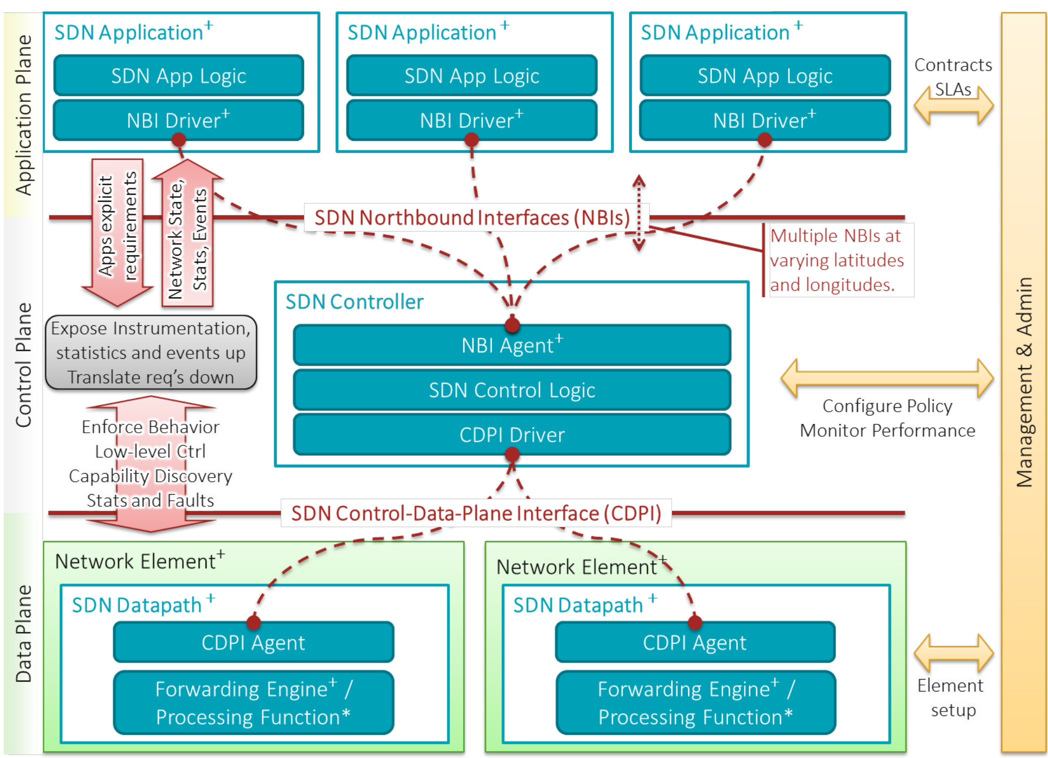


Figure 2.1: The role of the SDN controller in the SDN architecture [9]

(NBI), while the one between Control and Data planes, called SDN Control-Data-Plane Interface in Figure 2.1, can also be referred to as the *Southbound Interface* (SBI). Both interfaces can be specified and designed to use any compatible communication protocol. In practice, the most widely-used protocol for Controller-Device communication through the SBI is the OpenFlow protocol, on which more details will follow in Section 2.1.2.

The activities of our work are focused on the definition of a proper NBI through which high-level orchestration and management entities are allowed to control the underlying NFV and SDN platforms and implement dynamic SFC features [10].

2.1.2 OpenFlow

In the creators' own words, OpenFlow is a communications protocol that provides an abstraction of the forwarding plane of a switch or router in the network [11].

Focusing on its main features, OpenFlow:

- brings network control functions out of switches and routers, while allowing to directly access and manipulate the forwarding plane of those devices;

- specifies basic primitives that can be used by an external software application to actually program the forwarding plane of network devices, just like the instruction set of a CPU would program a computer system;

- works on a per-flow basis to identify network traffic;

- forwards flows according to pre-defined match rules statically or dynamically programmed by the SDN control software.

OpenFlow can be used both in a reactive and in a proactive way. In the former case, whenever an OpenFlow-enabled device receives a data packet it does not know how to handle, it wraps the data packet into an OpenFlow *PacketIn* message, to be sent to the relevant network controller. Upon reception of this message, the controller can analyze the packet and reply to the device it came from with an OpenFlow *FlowMod* message, containing, along with the original data packet that generated the PacketIn event, a set of matching rules and actions to be performed upon reception of data packets with similar characteristics. A scheme of the control message received by the device is shown in Figure 2.2. The device then installs the new *flow rule* into its *flow table*, so that, if it receives a data packet that matches one of the entries of its flow table, it applies the corresponding sequence of actions, without the need of querying the controller again. If the device has to handle a sequence of similar packets, as in a `ping` sequence, the first packet will take a longer time to be forwarded than the following packets in the sequence will. An example of this typical behavior is shown in Listing 2.1, where we have the output of a `ping` session through an OpenFlow/SDN domain.

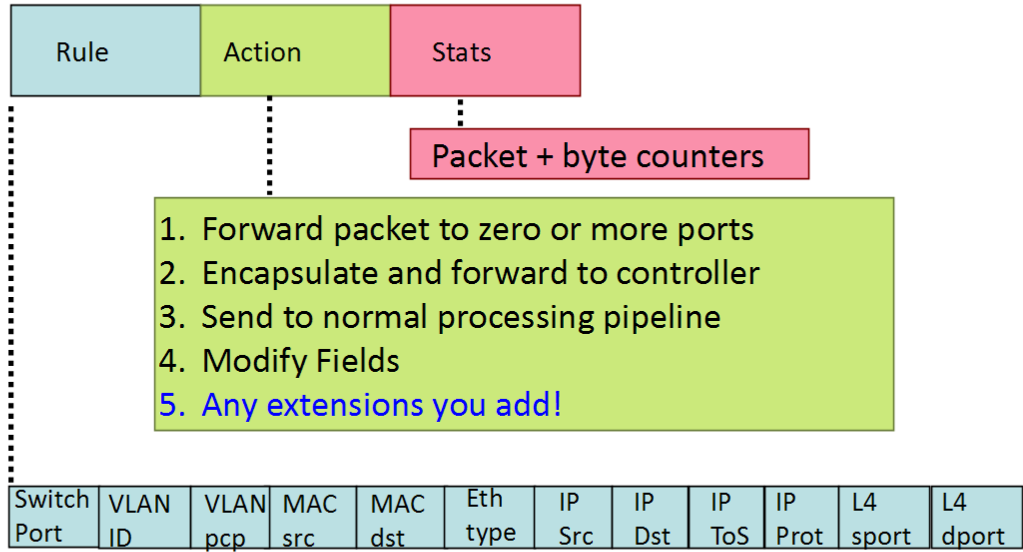


Figure 2.2: The OpenFlow table [12]

```
PING 10.0.0.8 (10.0.0.8) 56(84) bytes of data.
64 bytes from 10.0.0.8: icmp_seq=1 ttl=64 time=29.1 ms
64 bytes from 10.0.0.8: icmp_seq=2 ttl=64 time=0.647 ms
64 bytes from 10.0.0.8: icmp_seq=3 ttl=64 time=0.055 ms
64 bytes from 10.0.0.8: icmp_seq=4 ttl=64 time=0.052 ms
```

Listing 2.1: Start of a `ping` session in a SDN domain with reactive forwarding

We can observe that the first packet of the `ping` sequence has a Round Trip Time (RTT) that is close to 30 ms, while the following packets' RTT is smaller than 1 ms. This is due to the fact that, when the first packet of the sequence traverses each switch, those devices must contact the controller to be instructed on what to do with it, then receive and install new flow rules in their flow table, and finally proceed to forward the packet. When the following packets in the sequence traverse the switches, the devices will not need to contact the controller, as they will act based on the new flow rules, resulting in a much faster forwarding decision phase.

On the other hand, while using OpenFlow in a proactive way, flow rules are

installed before actual traffic reaches the network devices. This approach will obviously enhance data plane latency, as no communication to the controller is needed for the traffic which complies with the filters of the installed flow rules, but this is paid in terms of reduced flexibility. In fact, it is often impossible to install very selective (i.e., precise) and correct flow rules by acting proactively, as in most scenarios many details on the incoming traffic (e.g., client-side TCP port number) are not known a priori. For this reason, when acting in a proactive way, flows are defined with a larger granularity (i.e., a smaller precision) than they would have with a reactive approach.

More details on the OpenFlow protocol and its versions are available on the documents and standards produced by ONF [13].

2.2 Cloud Computing

Cloud Computing, often referred to simply as *Cloud*, is a paradigm that aims at enabling ubiquitous, on-demand access to a shared pool of configurable computing and infrastructure resources [14].

As already mentioned in the Chapter 1, the paradigm allows network service providers to offer their services in the same way as utility services, such as electric power and gas are distributed: the end users pay for what they get. In order to do so, Cloud Computing takes advantage of both hardware and software resources, which are distributed and virtualized in the network [3], and is supported in doing so by the high data rates made available by wide-band connection.

End users expect the resources offered by the Cloud to be instantiated and used in a transparent, seamless way. Those resources, however, may be geographically distributed all over the world, imposing high demands on the interconnecting network, in terms of configuration delay, let alone latency and reliability in the data plane. This is where SDN comes into play, allowing the resources to quickly configure or re-configure in order to match the user's requests. Moreover, thanks to the centralized management approach that the SDN paradigm offers, data flows can be dynamically steered to the best path from the user to the server hosting the resources.

2.3 Network Function Virtualization

NFV is a network architecture paradigm that uses virtualization technologies in order to obtain a new way of designing, deploying and managing network services. [15]

A given service can be decomposed in a set of *Virtual Network Functions* (VNFs) that can be implemented in software and run on general purpose physical servers, without the need of specialized hardware. For example, a single VNF can be implemented as a set of software entities (i.e., different modules of the function), running on one or more Virtual Machines (VMs), hosted by one or more physical servers. The VNFs can be relocated (i.e., migrated) to new network locations, without the need to purchase new hardware.

It is also possible for VNFs to be run on physical machines without virtualization of resources. However, two of the strongest advantages of NFV are flexibility and resource efficiency, which are inherently achievable through resource virtualization.

In general, NFV brings the benefits of the Cloud Computing approach to environment of telecommunication networks. In fact, the advantages introduced by NFV in comparison to the traditional scenario are:

- independence of software from hardware, which allows for separate development and maintenance of the two components;
- flexibility of the services offered by the network, as VNFs can be rearranged and upgraded very rapidly, while maintaining the same hardware platform;
- dynamic and more accurate scaling of the capabilities of the whole service, according to the actual load carried by the network in a given moment.

The NFV architecture has been described by ETSI in [16], and a scheme of the NFV reference architectural framework is shown in Figure 2.4. The identified functional blocks are the following ones:

– Virtualized Network Functions (VNF), which are virtualized versions of network functions in a legacy non-virtualized network, and may include elements of the core network (e.g., the Mobile Management Entity in the 3GPP Evolved Packet Core) as well as elements in a home network (e.g., firewalls);

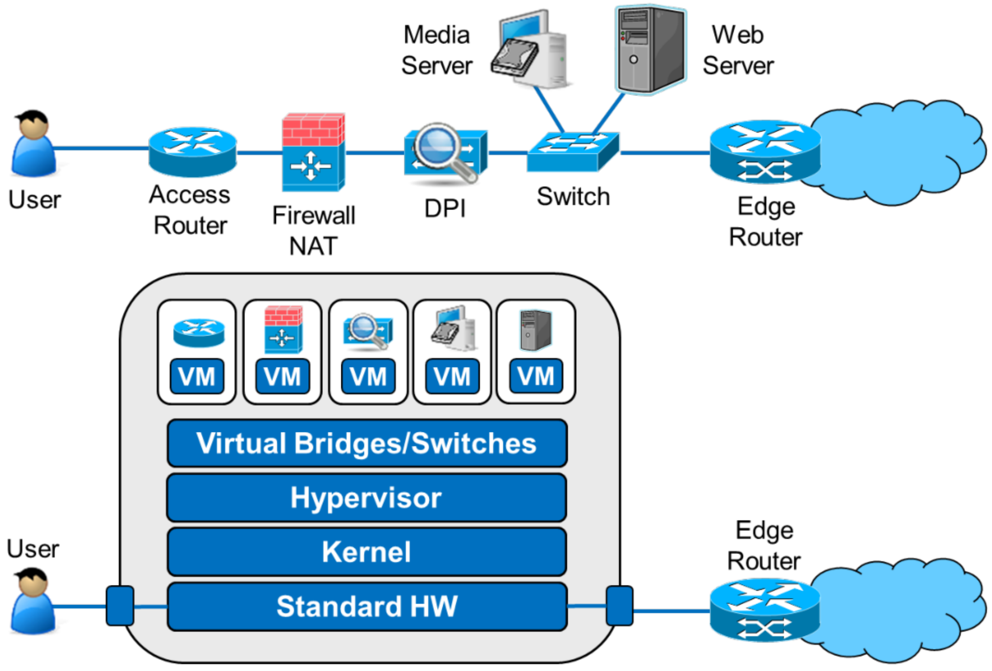


Figure 2.3: Comparison between the traditional approach and the NFV approach, simplified

- Element Management System (EMS), which performs the management functionality for one or multiple VNFs;
- NFV infrastructure, which is the set of all hardware and software components on top of which VNFs are deployed, managed and executed, including:
  - hardware resources, assumed to be Commercial Off-The-Shelf physical equipment, providing processing, storage and connectivity to VNFs through the Virtualization Layer;
  - Virtualization Layer, such as an hypervisor, which abstracts the physical resources, so as to enable the software that implement the VNFs to use the underlying infrastructure, and provide the virtualized resources to the VNF;
- Virtualized Infrastructure Manager(s) (VIM), which comprises the functionalities that are used to control and manage the interaction of a VNF with computing, storage and network resources under its authority, as

well as their virtualization;
- Orchestrator, which is in charge of the orchestration and management of NFV infrastructure and software resources;
- VNF Manager(s), which are responsible for VNF lifecycle management (e.g. instantiation, update, query, scaling, termination;
- Service, VNF and Infrastructure Description, which is a data set that provides information regarding the VNF deployment template, VNF Forwarding Graph, service-related information, and NFV infrastructure information models;
- Operation and Business Support Systems (OSS/BSS), which are used by an Operator to support a range of telecommunication services.

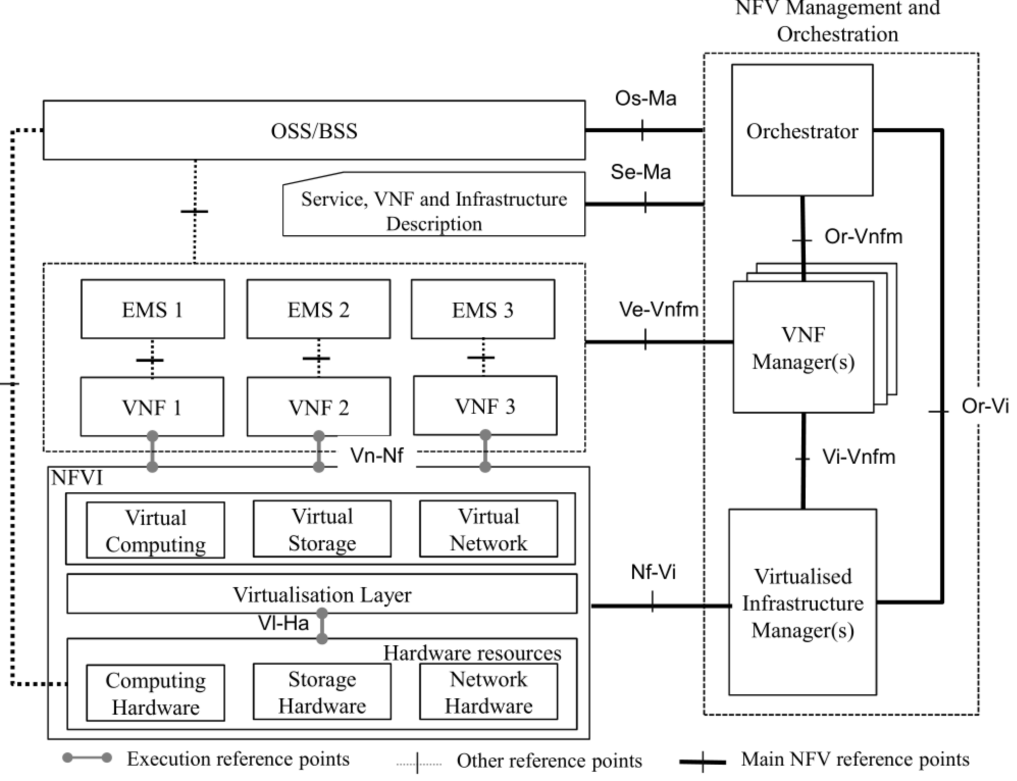


Figure 2.4: NFV reference architectural framework [16]

In Figure 2.4, the main reference points (i.e., the logical interconnections) that are in the scope of NFV are shown by solid lines. In our work, we are

going to focus mostly on the reference points labeled as *Or-Vi* and *Nf-Vi* in the figure. The former interconnection is used for carrying resource reservation and/or allocation requests by the Orchestrator, virtualized hardware resource configuration, and state information exchange (e.g., events). The latter one is used for specific assignment of virtualized resources in response to resource allocation requests, forwarding of virtualized resources state information, and hardware resource configuration and state information exchange (e.g., events).

2.3.1 The NVF-MANO framework

Due to the decoupling of the Network Functions software from the NFV Infrastructure (NFVI), coordination between resources requested by the VNFs is needed. The Network Functions Virtualization Management and Orchestration (NFV-MANO) architectural framework, described by ETSI in [17], has the role of managing the NFVI while orchestrating the allocation of resources needed by the VNFs. A functional-level representation of the MANO framework is shown in Figure 2.5.

The main functional blocks it encompasses have already been described in the previous section. The importance of this framework is in the even more general view it yields over the considered SDN/NFV deployment. A hierarchical scheme of the architecture is shown in Figure 2.6, which highlights the role of SDN Controllers in a multi-domain scenario. In that scheme, we can see all the functional blocks and reference point we are going to focus on in the activities presented in the following Chapters. For instance, the generic deployment we present as our reference architecture in Figure 3.1 is actually a slightly differently characterized version of the one presented in Figure 2.6.

2.4 Software tools

In this section we are going to have an overview on the main software tools that have been used in our activities. We will have a brief look at **ONOS**, the network controller, **Mininet**, the network emulator, and **OpenStack**, the Cloud operating system.

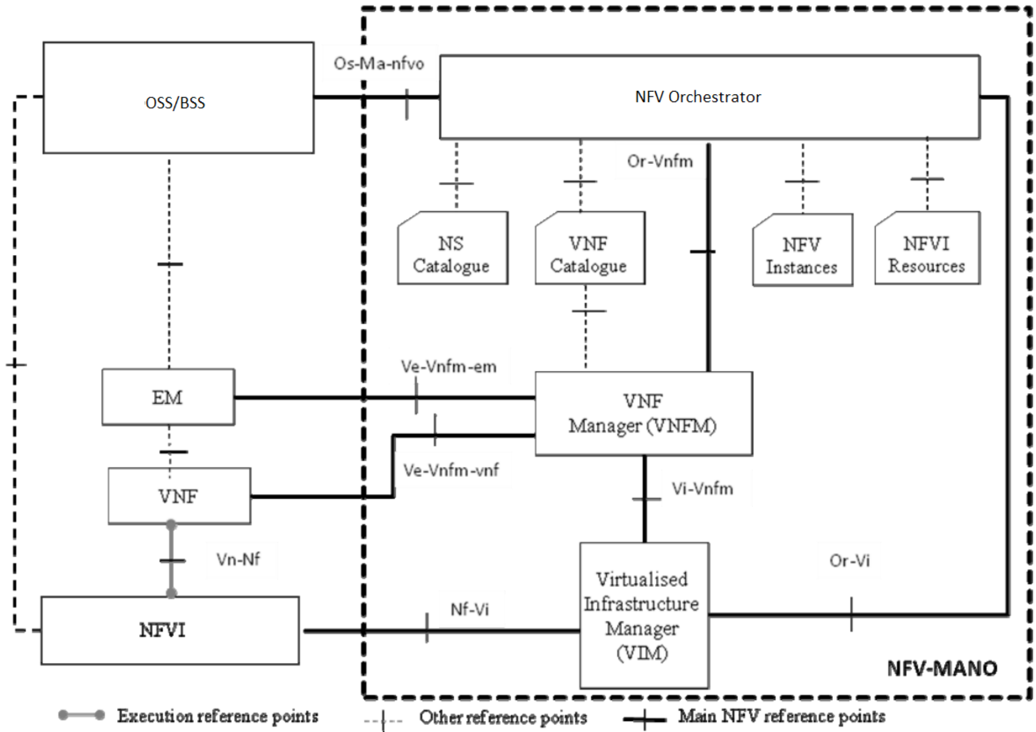


Figure 2.5: The NFV-MANO architectural framework with ref. points [17]

2.4.1 ONOS

ONOS, short for Open Network Operating System, is, as the name suggests, an open-source Network OS (NetOS), developed and maintained as part of the ONOS project, whose mission is *"to produce the Open Source Network Operating System that will enable service providers to build real Software Defined Networks"* [18].

ONOS is able to provide the control plane for a software-defined network, managing network components, such as switches and links, and running software programs or modules to provide communication services to end hosts and neighboring networks [19].

As a NetOS, ONOS aims at:

- providing APIs and abstractions, resource allocation, and permissions, as well as user-facing software such as a CLI, a GUI, and system applications;

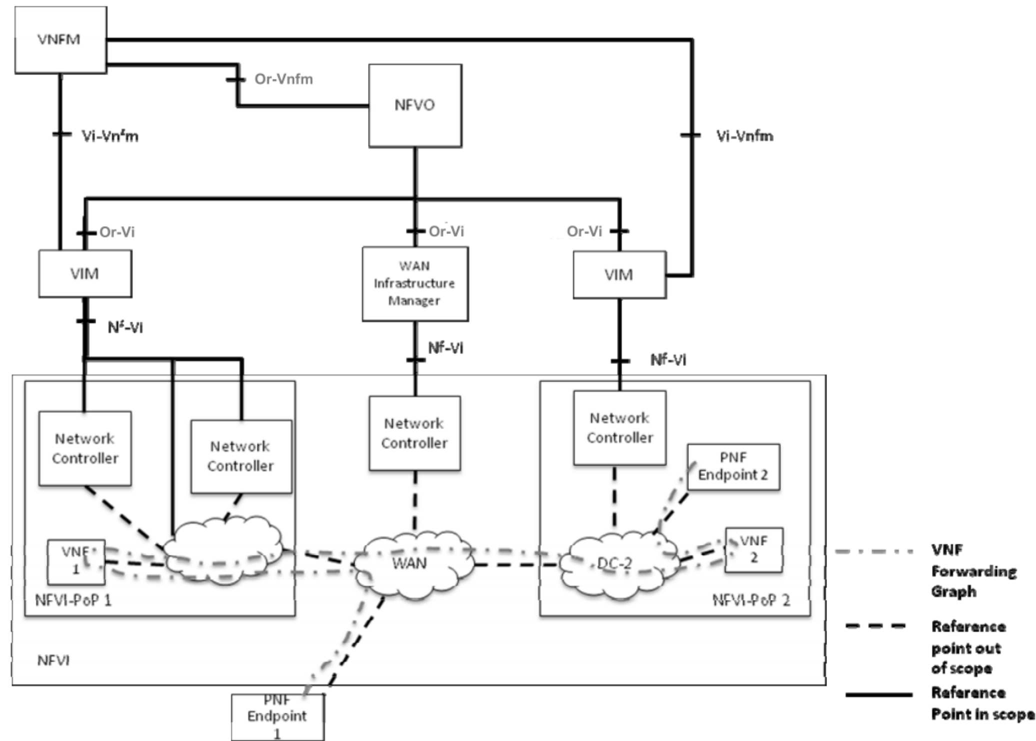


Figure 2.6: Hierarchical view in SDN multi-domain scenario [17]

- managing the entire network rather than a single device, which can dramatically simplify management, configuration, and deployment of new software, hardware, and services;

- acting as an extensible, modular, distributed SDN controller.

ONOS can run as a distributed system across multiple servers, allowing it to use their combined CPU and memory resources while providing fault tolerance in the case of server failure, and potentially supporting live upgrades of hardware and software without interrupting network traffic.

A scheme of the ONOS internal architecture, along with some examples of external agents both towards the Application plane and the Data plane are shown in Figure 2.7.

The ONOS kernel and core services, as well as ONOS applications, are written in Java as bundles that are loaded into the *Karaf OSGi* container.

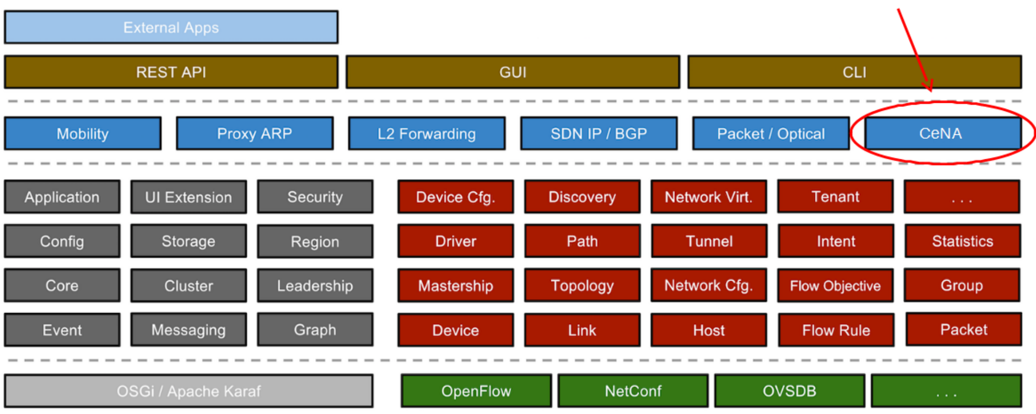


Figure 2.7: Our contribution in the ONOS subsystem

OSGi is a component system for Java that allows modules to be installed and run dynamically in a single Java VM (JVM). Since ONOS runs in the JVM, it can run on several underlying OS platforms.

Our contribution to the ONOS project, the ONOS application implementing the intent-based REST NBI, is circled in Figure 2.7.

2.4.2 Mininet

Mininet is a network emulator which is able to emulate a linked set of virtual hosts, switches, controllers. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and SDN [20].

Mininet supports research, development, learning, prototyping, testing, debugging, and any other tasks that could benefit from having a complete experimental network on a laptop or other PC.

In order to achieve that, Mininet:

- provides a simple and inexpensive network testbed for developing OpenFlow applications;
- enables multiple concurrent developers to work independently on the same topology;
- supports system-level regression tests, which are repeatable and easily packaged;

- enables complex topology testing, without the need to wire up a physical network;
- includes a CLI that is topology-aware and OpenFlow-aware, for debugging or running network-wide tests;
- supports arbitrary custom topologies, and includes a basic set of parametrized topologies which is usable out-of-the-box without programming;
- provides a straightforward and extensible Python API for network creation and experimentation.

In a nutshell, Mininet provides an easy way to get correct system behavior and performance (to the extent supported by the underlying hardware), and to experiment with topologies. Some examples of Mininet's built-in topologies are shown in Figure 2.8.

Mininet networks run "real code", including standard Unix/Linux network applications, as well as the real Linux kernel and network stack. Thanks to this, the 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. Most importantly, this means that a design that works in Mininet can usually move directly to hardware switches for line-rate packet forwarding.

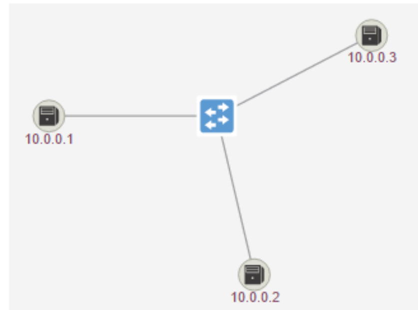
2.4.3 OpenStack

OpenStack [21] is a Cloud operating system that allows for the management of a Cloud platform. Such a platform is a cluster of physical machines which host instances of compute and storage nodes, offered to the user as a service.

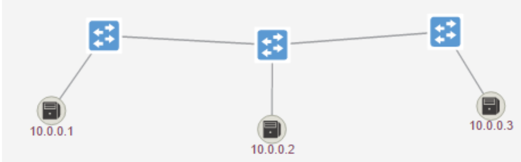
OpenStack provides a convenient GUI in the form of a dashboard, that can be used to simplify the process creation and management of the instances, as well as a CLI, which allows for greater precision in the specifications on the resources to be allocated.

A single user, who represents a tenant, can create a new network in the cluster, and define one or more subnets over it. Then, the user can create a new instance of a VM, placing it on the desired network or subnetwork. From that moment on, the VM will be accessible as a regular machine, through the network it is connected to.

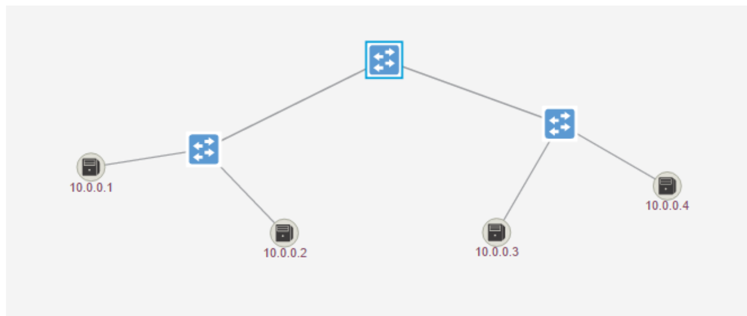
OpenStack also supports multi-tenancy, that is the co-existence of multiple users in the same cluster, which are mutually isolated through the use of separate VLANs and namespaces [3].



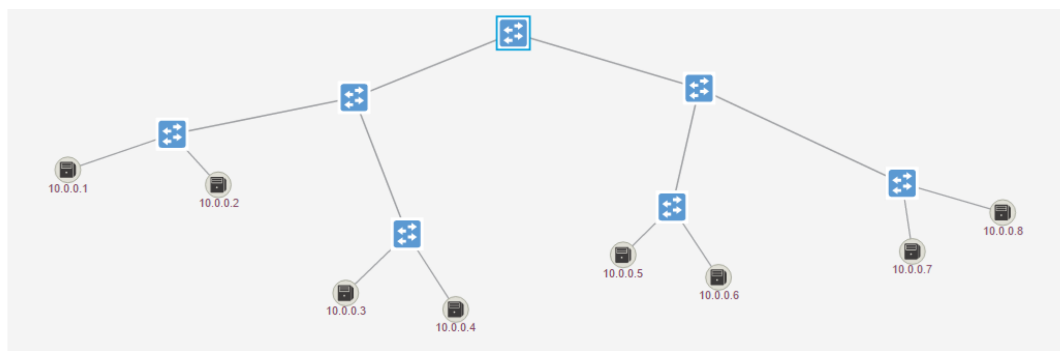
(a) Single (`--topo single,3`)



(b) Linear (`--topo linear,3`)



(c) 2-level tree (`--topo tree,2`)



(d) 3-level tree (`--topo tree,3`)

Figure 2.8: Examples of network topologies emulated in Mininet

Chapter 3

Reference Architecture and Interface Definition

In this chapter we will describe the reference architecture of our scenario, then the proposed NBI, and finally the domains relevant to the activities of this thesis.

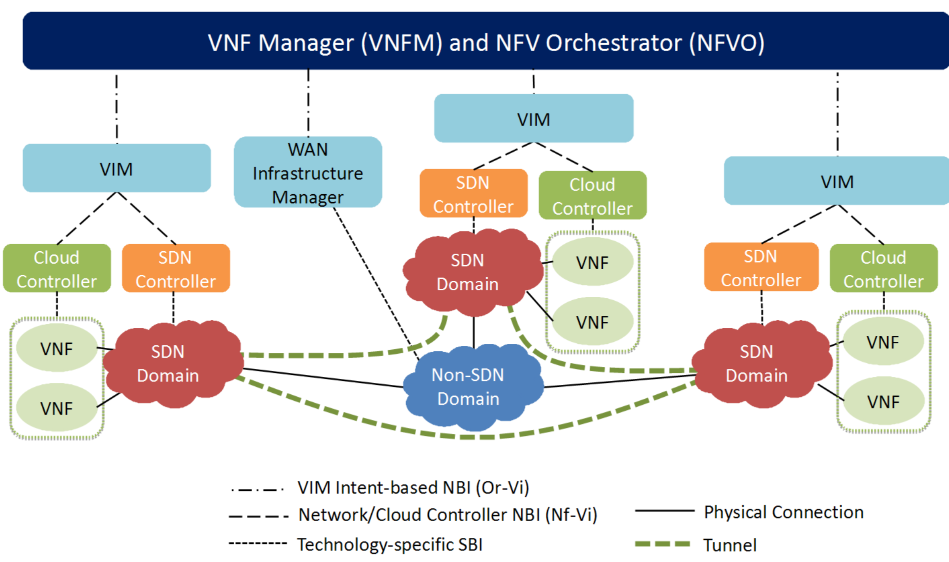


Figure 3.1: Reference multi-domain SDN/NFV architecture, in general

Our reference architecture is shown in Figure 3.1, and it is inspired by the

ETSI NFV specifications, with particular reference to the Management and Orchestration (MANO) framework [17], which has been presented in Section 2.3.1.

This way, the proposed architecture is compliant with the most relevant NFV standard initiative to date, and it can be easily extended to include any further SDN/NFV domain and technology as part of the underlying virtualized infrastructure.

Each of the SDN/NFV domains shown in Figure 4.1 consists of a technology-specific infrastructure, including:

- data plane components, such as IoT nodes and gateways, SDN switches, virtual machines running in Cloud computing nodes, physical and virtual interconnecting links; these components provide the network, compute, and storage resources to be orchestrated;

- control plane components, such as SDN and Cloud controllers with related data stores and interfaces; these components are responsible for proper VNF deployment and traffic steering across VNFs and domains;

- management plane components, such as Virtualized Infrastructure Managers (VIMs) specialized for managing resources in the IoT-based SDN infrastructure, the wired SDN infrastructure, and the Cloud infrastructure; based on the available implementations, some of these components could be in charge of multiple domains, as in the case of the SDN/Cloud VIM in Figure 4.1.

The overarching VNF Manager (VNFM) and NFV Orchestrator (NFVO) components are responsible for programming the underlying VIMs and infrastructure controllers in order to implement and maintain the required service chains in a consistent and effective way, for both intra- and inter-domain scenarios. While technology- and domain-specific northbound (NBI) and southbound interfaces (SBI) are used inside each domain to efficiently control and manage the relevant components, the design of the overarching VNFM and NFVO should be as technology-agnostic as possible, so that a service chain to be deployed can be specified by a customer using a high-level, intent-based description of the service itself. This would also allow the proposed architecture to be more general and capable of being extended to different SDN technologies and domains.

As it is argued in [2], in order to achieve such generality in the high-level management and orchestration components, the act of decoupling service abstractions from the underlying technology-specific resources should be performed mainly by the VIMs. Therefore, the concept of interactions based on intents is extended to the NBI offered by the VIMs, which should be defined as an open and abstracted interface, independent of the specific technology used in the underlying domains.

It is important to outline that this reference architecture considers also the possibility that SDN domains are interconnected through non-SDN domains. This assumption stems from the fact that it appears reasonable that a network operator will deploy SDN technologies mainly within data center infrastructures where the VNF resources will be located e.g., in the operators points of presence or central offices rather than in backbone networks. In this case, traffic flows that must traverse a number of SDN domains can be properly routed by adopting some form of tunneling or overlay network technology across the non-SDN domains, such as the emerging Network Service Header (NSH) [22].

NSH describes a dataplane header used to carry information along a service path, thus creating a transport-independent *service plane*. More specifically, it decouples the service topology from the actual network topology, making each service function an identifiable resource available for consumption from any location in the network. Some more details on NSH is reported in Section 4.2

3.1 VIM Northbound Interface

An intent-based NBI must allow the user to specify policies (i.e., "what to do") rather than mechanism (i.e., "how to do it").

When a given service specification is received, the platform management and orchestration functions must convert that request into a suitable service graph and pass it to the relevant VIMs in charge of the underlying infrastructures and domains involved in the service composition. Then each VIM must coordinate the respective Cloud and network controllers in order to:

- verify availability and location in the Cloud infrastructure of the VNFs required to compose the specified service, instantiating new ones if needed;
- program traffic steering rules in the network infrastructure to deploy a suitable network forwarding path.

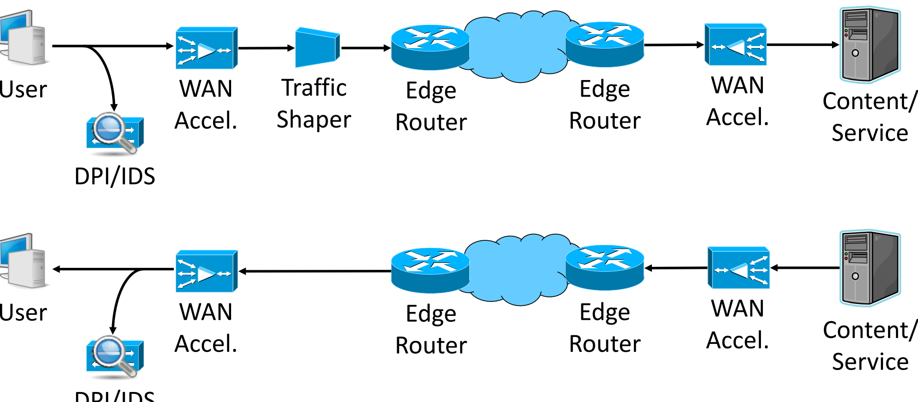


Figure 3.2: Example of a Service Function Chain

In order to provide an abstracted yet flexible definition of the specified service graph, without knowledge of the technology-specific details (such as devices, ports, addresses, etc.), the NBI exposed by the test bed should allow to specify not only the sequence but also the nature of the different VNFs to be traversed, which is strictly related to the service component they implement, as well as other peculiar characteristics of the service itself, such as quality of service (QoS) metrics and thresholds.

As a first attempt to define the NBI, the following abstractions can be considered:

- a QoS feature can be defined in terms of a QoS metric that is relevant to the service specified; in the example discussed above the relevant metric is guaranteed bit rate;
- a QoS threshold can be specified for the QoS metric of interest; in the example, a minimum bit rate value to be guaranteed can be specified;
- a VNF can be terminating or forwarding a given traffic flow; in the example of Figure 3.2, the Deep Packet Inspector/Intrusion Detection System (DPI/IDS) is terminating the flow, whereas the traffic shaper and the WAN accelerator are forwarding it;
- a forwarding VNF can be *port symmetric* or *port asymmetric*, depending on whether or not it can be traversed by a given traffic flow regardless

of which port is used as input or output; in the example, the WAN accelerator is port asymmetric, because it compresses or decompresses data based on the input port used, whereas the traffic shaper can be considered port symmetric, if we assume that the shaping function is applied to any output of the VNF;

- a VNF can be *path symmetric* or *path asymmetric*, depending on whether or not it must be traversed by a given flow in both upstream and downstream directions; in the example, according to the service requirements, the WAN accelerator and the DPI/IDS are path symmetric, whereas the traffic shaper is path asymmetric.

In order to implement the aforementioned abstractions, we define a sort of ETSI MANO deployment template adopting the well-known JSON format. A service chain is therefore defined in the following way:

```
{
  "src": "node_value",
  "dst": "node_value",
  "qos": "qos_type",
  "qos-thr": "qos_value",
  "vnfList": [vnf],
  "dupList": [dup]
}
```

where:

- `src` and `dst` represent the endpoint nodes of the service chain, either global or limited to a given VIM domain;

- `node_value` is a text string that contains a high-level unique identifier of a node known to both orchestrator and VIM;

- `qos` represents the QoS feature to be provided with the service chain, and its value, `qos_type`, is a text string that contains a high-level unique identifier of a QoS metric known to both orchestrator and VIM

- `qos-thr` represents the QoS threshold to be applied to the specified metric, and its value, `qos_value`, is the actual value assigned to the threshold;

- `vnfList` is the ordered list of VNFs to be traversed according to the specified service;
- `dupList` is the list of VNFs towards which the traffic flow must be duplicated.

Each VNF is described in terms of its topological abstractions with the following template:

```
vnf ::= {
  "name": "node_value",
  "terminal": "bool_value",
  "port_sym": "bool_value",
  "path_sym": "bool_value"
} | ϵ
```

where `bool_value` is a text string representing either a Boolean or a null value, and the symbol ϵ indicates the possibility that `vnf` is an empty element. Considering that some network functions (e.g., DPI) require traffic flows to be mirrored, the (possibly empty) list of VNFs towards which the traffic flow must be duplicated is specified with the following template:

```
dup ::= {"name": "node_value"} | ϵ
```

The NBI offered by VIMs can be implemented through the mechanisms of a REST API, and should provide the following methods:

- a method to **define** a new service chain;
- a method to **update** an existing service chain;
- a method to **delete** an existing service chain.

These actions are basically in line with the operations foreseen by the ETSI MANO specifications, with reference to the interface between NFVO and VIM. It is worth highlighting that the NBI description given above is indeed based on the concept of intent. QoS metric, VNFs and service chains are specified in a high-level, policy-oriented format without any knowledge of the technology-specific details. A non-intent-based description of a service chain, e.g. using the OpenFlow expressiveness to steer traffic flows and compose the network forwarding path, would require the customer to specify multiple flow rules in each forwarding device for each traffic direction, involving technology-dependent details such as IP and MAC addresses, device identifiers and port numbers.

Chapter 4

Specific Deployments

The NBI defined in Chapter 3 is used in this Chapter to specify an IoT data gathering service crossing two different SDN domains and an NFV chain, as per the architecture in Figure 4.1.

For the use case considered here, the high-level QoS features offered by the SDN/NFV platform include **minimum latency** and **high reliability** classes, with the possibility to specify a threshold for the relevant metric.

Although the above intent-based NBI definition is common to all VIMs considered in our use case, the orchestrator must specify different content for each VIM depending on the specific resources to be programmed and the specific segment of the service chain to be deployed in each domain.

4.1 Service Function Chaining in the IoT/Cloud deployment

Our aim is to obtain an intent-based, technology-independent, north-bound interface (NBI) for end-to-end service management and orchestration across multiple technological domains, possibly including both SDN and non-SDN domains.

In comparison to Figure 3.1, in Figure 4.1 we present a specialized version of the reference architecture, specific to the use case of IoT data collection and related Cloud-based consumption.

By going further into detail, we obtain the scheme of the actual test bed we developed to demonstrate multi-domain SDN/NFV management and orchestration, shown in Figure 4.2

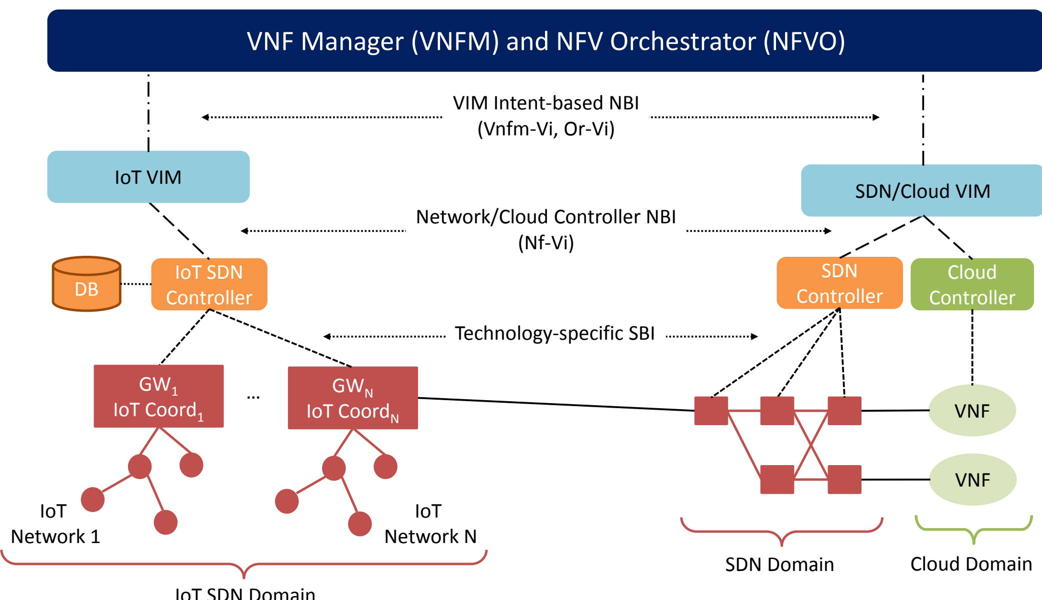


Figure 4.1: Reference multi-domain SDN/NFV architecture, specialized for the presented use case

4.1.1 IoT SDN Domain

The IoT SDN domain included in the architecture of Figure 4.2 is composed of:

- a VIM able to manage components and resources in the IoT domain;
- an IoT SDN controller (IoTC), implementing the software-defined control plane of the IoT domain;
- a set of IoT networks, where different devices send the measured data via multi-hop paths to a coordinator node that forwards them to the final consumer.

Since the different IoT networks will possibly use different technologies (e.g., Zigbee, LoraWAN, 6LowPAN, etc.), each IoT coordinator will be connected to a specific gateway (GW) in charge of forwarding data outside the IoT domain.

When a service request is received from the high-level management and orchestration functions, the IoT VIM gets access to the IoTC. As it is shown in

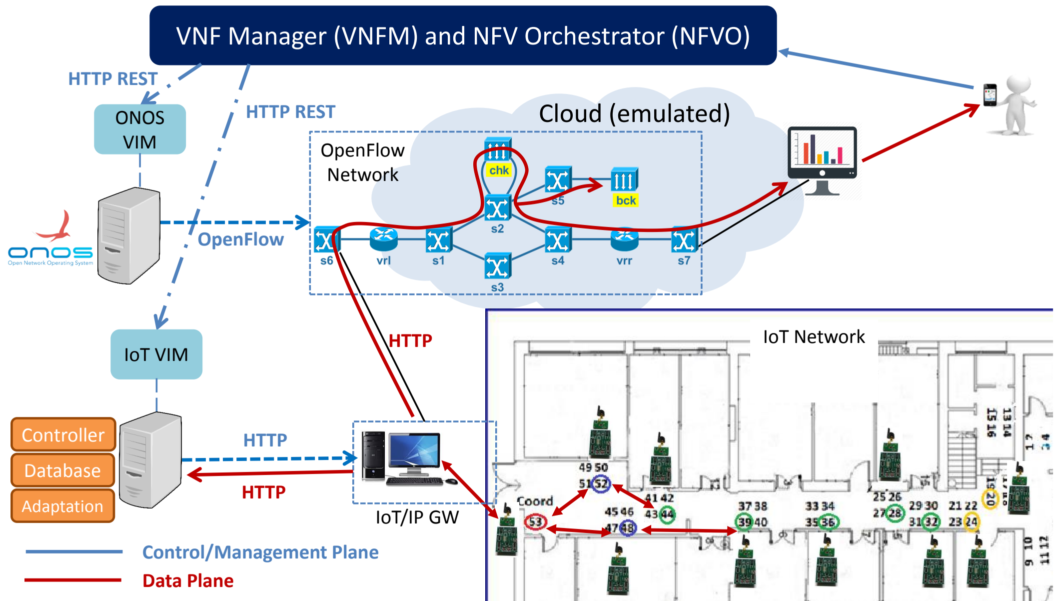


Figure 4.2: the NFV/SDN test bed setup developed

Figure 4.2, one of the components of the controller is a database, which stores information about devices of the different networks, such as how to reach them (i.e., the IP address of the corresponding GW), the service provided, and the related QoS feature that could be guaranteed. The VIM tries to map the incoming request with the resource knowledge available in the database, in order to select the proper IoT device(s) to forward the request to. According to the decision taken, the IoTC will program the selected IoT network(s) to make sure that the requested QoS would be guaranteed, and forward the request to the identified GW(s).

The IoT VIM and database

The VIM is capable of handling requests containing either the particular IoT device to be queried, or a high-level description of the service requested by the customer, together with some other possible specification related to the QoS (in terms of reliability or maximum latency). Let us consider the case of a customer that wishes to periodically collect temperature values in a given room and monitor them by means of a processing/publishing service called

`ServP` running as a VNF in the Cloud domain. Assume that the customer is interested in having a complete record of the measured temperature request, thus requiring a high-reliable service. Then the intent-based request sent to the IoT VIM, expressed according to the JSON format specified in Section 3.1, could be as shown in Listing 4.1.

```
{
  "src": "ServP",
  "dst": "Temperature Room X",
  "qos": "SR",
  "qos-thr": "90%",
  "vnfList": "null",
  "dupList": "null"
}
```

Listing 4.1: Example of request sent to the IoT VIM

In the IoT domain, following the typical IoT device query approach, `src` represents the source of the query, that is the final consumer of the data to be collected. In our example, this is the processing/publishing service in the Cloud. On the other hand, `dst` represents the final endpoint of the query, that could be one or multiple IoT devices. This text string may contain a unique identifier of a specific IoT device, or a high-level intent-based description of the requested service. The second option is used in our example above. The field `qos` represents the requested QoS feature either in terms of latency, expressed as data plane RTT, or reliability, that is the probability of successfully receiving the data from that device. In our IoT VIM implementation, `qos` may assume the values of real time (`RT`), non real time (`NRT`), strictly reliable (`SR`), or loosely reliable (`LR`). If needed, the user may also provide `qos-thr`, representing either the maximum tolerable latency or the minimum requested throughput/reliability. In the example above, `SR` with a 90% threshold is requested. Finally, `vnfList` and `dupList` are not specified in the example because we assume that the orchestrator opted for VNFs located in the Cloud domain.

At this point the VIM checks in the database if the destination the user is looking for is present, and if the requested QoS (if any) could be satisfied.

When the IoTC receives a new measurement from a device, the data is stored in the database, along with the instant in which it was received. Once a new request for the same device arrives the VIM checks the timestamp and decides whether the data needs to be updated or not (if not, the value is immediately returned).

With reference to the QoS, it is important to underline that in case the same device could reach the IoT coordinator via different paths (e.g., having different number of hops), the corresponding QoS values are stored in the database.

The IoT controller and network

The IoT controller is responsible for gathering information from sensor devices, maintaining a representation of the network, and establishing routing paths.

In order to achieve the decoupling of the control plane from the data plane, it is fundamental for each device to be able to discover a path towards the coordinator. This is done during the network initialization phase. Requests coming from the VIM are forwarded by the IoTC to the proper IoT coordinator, along with the information about the selected path connecting the coordinator and the intended device to be setup to guarantee the requested QoS. The details on how this works are covered in [23].

4.1.2 OpenFlow and Cloud Domains

In this section we consider both the wired SDN domain and the Cloud Computing domain depicted in Figure 4.1, assuming that they are managed by a single SDN/Cloud VIM. The data plane topology assumed for the considered use case is shown in Figure 4.3. An OpenFlow-based SDN infrastructure is assumed to be in charge of the connectivity within the Cloud domain as well, thus providing programmable traffic steering functionality to VNF chains. All the switches included in the topology (`s1`, `s2`, . . . , `s7`) are OpenFlow-enabled devices and are governed by an SDN controller (e.g., ONOS), whereas the computing infrastructure is managed through a Cloud platform (e.g., OpenStack).

Switch `s6` is an edge device connecting the IoT gateways in the IoT SDN domain to the Cloud network. Router `vrl` is the (virtual) edge router of the (virtual) tenant network responsible for the connectivity within the Cloud domain of the requested IoT data collection service. Switches `s1` to `s5` are either physical or virtual switches used by the tenant network for VNF connectivity. Two VNFs are deployed in the Cloud: `chk` performs integrity checks on the collected data for improved reliability, whereas `bck` is used to store backup copies of the collected data. Router `vrr` is the (virtual) edge router of the (possibly different) tenant responsible for the IoT data collection, processing,

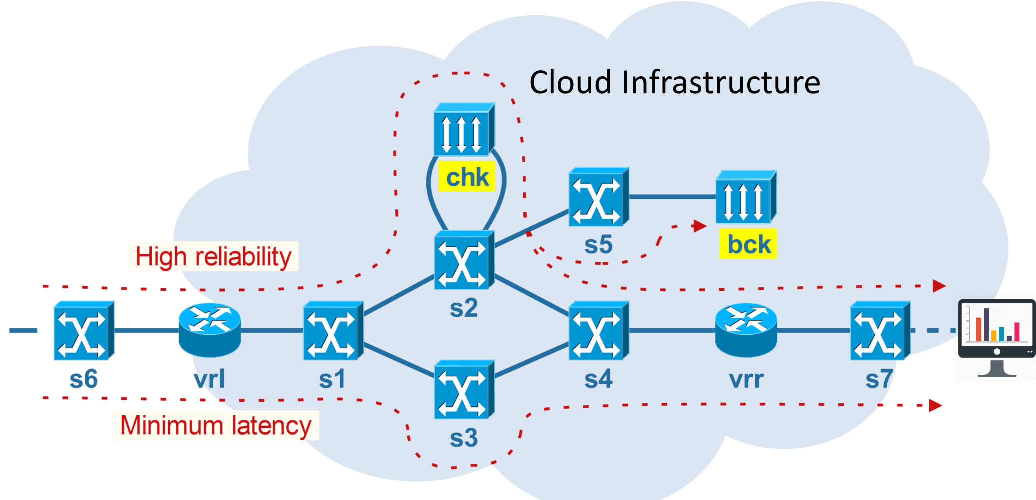


Figure 4.3: data plane topology of the OpenFlow and Cloud domains considered for the use case

and publishing services. Switch `s7` is a (virtual) switch in the latter tenant's network, providing layer-2 connectivity to the server `ServP` where collected data are processed and published.

According to the QoS features of the use case considered here, the connectivity service offers two different paths in the OpenFlow domain. One path is characterized by minimum latency, where switches are configured with small buffers being continuously monitored by the SDN controller for possible congestion, and such that no VNF processing is performed, which could introduce additional delays. The other path is dedicated to highly reliable traffic flows, where switches have large buffers to reduce losses, and data are processed by `chk` and duplicated at switch `s2` in order to be stored in `bck`.

Therefore, depending on the QoS feature requested by the customer, the high level management and orchestration functions can specify two different service chains. Assuming that, based on the interaction between the orchestrator and the IoT VIM, incoming data will be collected from IoT network k and then forwarded to `ServP`, according to the JSON format specified in Section 3.1 the intent-based request to the SDN/Cloud VIM NBI could be the one shown in Listing 4.2 for the minimum latency QoS feature, or the one shown in Listing 4.3 for the high reliability QoS feature. The SDN controller must

implement a data plane monitoring service to make sure that, in the former case, the minimum latency path guarantees the requested maximum delay of 10 ms, whereas in the latter case the VNFs inserted in the service chain and the high reliability path ensure the required 99% accuracy.

For the sake of completeness, we remark that the data paths depicted in Figure 4.2 are those for the case in which the QoS class is set to high reliability.

```
{
  "src": "IoT-GW[k]",
  "dst": "ServP",
  "qos": "Max delay",
  "qos-thr": "10 ms",
  "vnfList": "null",
  "dupList": "null"
}
```

Listing 4.2: Request for minimum latency path

```
{
  "src": "IoT-GW[k]",
  "dst": "ServP",
  "qos": "Reliability",
  "qos-thr": "99%",
  "vnfList": [chk, bck]
  "dupList": [bck]
}

chk ::= {
  "name": "chk",
  "terminal": "false",
  "port_sym": "true",
  "path_sym": "false"
}

bck ::= {
  "name": "bck",
  "terminal": "true",
  "port_sym": "null",
  "path_sym": "false"
}
```

Listing 4.3: Request for high reliability path

We developed the VIM for the SDN/Cloud domains as an application running on top of the ONOS platform. It is worth noting that ONOS already provides a built-in, intent-based NBI that can be used to program the SDN domain and deploy the required network forwarding paths. However, in order to specify the ONOS intents, some knowledge is required of the specific

data-plane technical details, while in our approach we prefer to expose only high-level abstractions to the orchestrator. Therefore, one of the main functions of our VIM is to implement new, more general and abstracted intents that can be expressed according to the NBI specification given above. Then the VIM takes advantage of the network topology features offered by ONOS to discover VNF location in the Cloud and relevant connectivity details, and eventually it is able to compose native ONOS intents and build more complex network forwarding paths.

The VIM can be instantiated as an ONOS service called *ChainService*, which provides the capability of dynamically handling the VNF chains through the abstracted NBI defined in Section 3.1. To achieve extensibility and modularity, the implementation of ChainService is delegated to a module called *ChainManager*, which is in charge of executing all the required steps to translate the high-level service specifications into ONOS-native intents. The input to ChainManager can be given through either the ONOS command line interface (CLI) or a REST API. The latter is preferable because it allows remote applications to use standard protocols (e.g., HTTP) to access resources and configure services. In our implementation, the REST API provides the following service endpoints:

```
POST /chaining/{action}/{direction}
DELETE /chaining/flush
```

In the former endpoint, the `action` variable indicates the operation that the orchestrator means to perform on a specified service chain (`add`, `update`, or `delete`), whereas in case of an update the `direction` variable (`forth`, `back`, or `both`) defines whether the modified chain specification refers to the existing forwarding path from `src` to `dst`, the opposite way, or both directions. The basic operations of this endpoint are described in the following list.

- If the `add` action is given, a new service chain is defined, based on the JSON specification included in the message body. This means that a forwarding path will be created for traffic flowing from `src` to `dst` and another one in the opposite direction. Note that the two paths are not necessarily symmetric, based on the topological abstractions defined by the NBI.

- If the `update` action is given, then the direction is taken into account and the forward path, backward path, or both paths of the specified

existing service chain are changed. In fact, a user may be interested in changing only a segment of the forwarding path and only in one direction, to reduce the control plane latency and limiting the impact that a path change can have on the existing traffic flows;

- If the `delete` action is given, then both forwarding paths of the specified existing service chain are removed.

ChainService provides also the `flush` operation through another endpoint, thus offering the possibility of deleting in a single step the forwarding paths of all the service chains previously created.

4.2 Dynamic multi-domain Service Function Chaining

As previously stated, our goal is to be able to handle seamless communication of data between multiple heterogeneous domains. In order to do that, we can consider using other emerging technologies which implement the concept of SFC. Starting from the reference architecture represented in Figure 3.1, we also developed a second test bed, in which two OpenFlow/SDN domains are interconnected through a non-SDN domain, controlled by a barebone version of a WAN Infrastructure Manager. We wanted the interconnection domain to be SFC-oriented, so we decided to adopt Network Service Header (NSH) [22] as Service Function Chaining resource in that domain.

In a few words, a Network Service Header is metadata added to a packet or frame that is used to create a service plane. The packets and the NSH are then encapsulated in an outer header for transport. The service header is added by a service classification function (i.e., a device or application) that determines which packets require servicing, and correspondingly which service path to follow to apply the appropriate service.

Although many experimental sessions have been run on this test bed, we have not come to a point where we can illustrate the experimental validation of this part of the work in a complete and satisfying way. Therefore we leave the description of its implementation, along with the description of the tests we managed to run on it so far, out of this document, for the benefit of a following work, where this test bed and all related aspects will be thoroughly discussed.

Chapter 5

OpenFlow/Cloud domain implementation

Before proceeding, it is worthy to remark the sequence of actions that our system is designed to handle.

1. The customer requests the service to the high-level management and orchestration functions, specifying the desired QoS feature.

2. The orchestrator forwards the request to the VIM REST NBIs of the relevant domains using the JSON format described in Chapter [**ch:spec-delp**].

3. Each VIM performs the operations required in the respective domain and programs the underlying controllers according to the requested service and QoS feature.

4. Data generated by the IoT devices are sent by the relevant gateway via HTTP POST to the collecting/processing/publishing server in the Cloud, where the customer can retrieve it.

In this chapter, we will cover the implementation details of the OpenFlow/Cloud domain described in Section 4.1.2. First, we will describe the software tools we used, then we will go into details of the setup of a preliminary local environment (i.e., running on a PC, not on the dedicated server we are going to setup later on), and then we will go through the setup of the environment we actually used in the tests carried out for [2].

More specifically, the activities of this thesis are focused on installing and operating the ONOS controller and the OpenFlow/Cloud domains of the test bed shown in Figure 4.3. To this aim, our environment must include four Virtual Machines (VMs), in particular:

- a VM on which to develop ONOS applications;
- a VM on which to deploy the ONOS instance;
- a VM on which to run the Mininet network;
- a VM on which to host the Web server.

These VMs are going to be hosted by a single physical server, located in our lab. A complete scheme of the interconnection of the VMs within the physical server is shown in Figure 5.1.

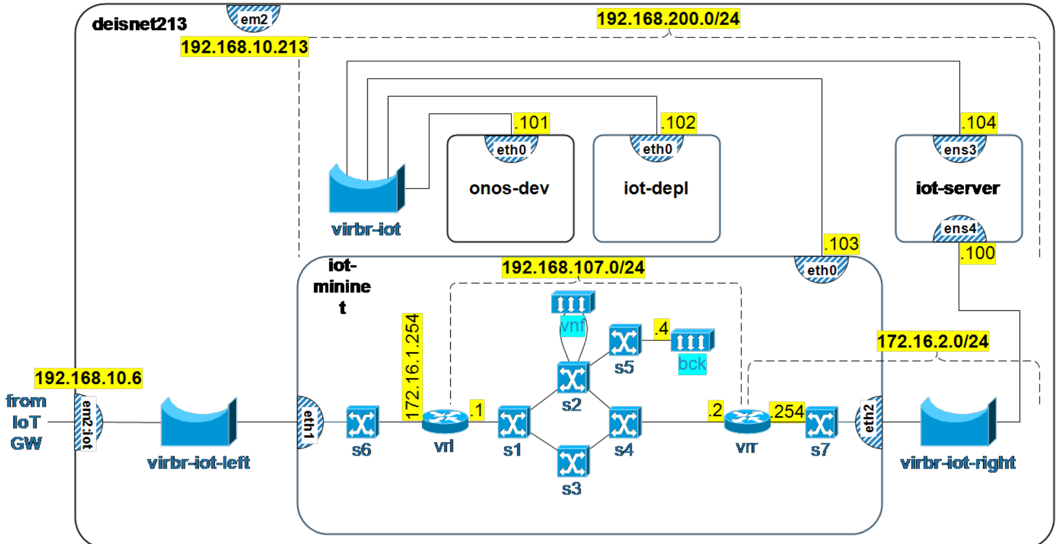


Figure 5.1: Scheme of the physical host server

Before setting up the actual test bed on a server, we simulated part of it locally, so as to then be able to use the virtual hard drives of the trial VMs as starting points for the VMs of the actual development.

Personal note: please take into account that the description of the performed steps is aimed at future students, who might never have used a Linux

system before, as it was for me when I started these activities. For this reason, some of the descriptions included in the rest of the chapter may appear to be over-simplified or over-detailed, to the eye of an experienced user. For the same reason, most of the descriptions of the perfrmed actions is reported in the form of a sort of guide. It was my personal choice to include in the first steps explained every single command to be used to repeat those steps, including the commands needed to change the working directory, so that, if anybody should repeat my activities, they are not required to have any knowledge of a Linux/UNIX system, but hopefully they can start learning something even from here.

5.1 Setup of the preliminary local environment

Before deploying the VMs we are actually going to use for the experimental validation on the server, we created and deployed two of them (those for ONOS development and deployment) on a PC, so as to be able to get familiar with the process of creating and deploying a functional ONOS instance. Therefore, in this section we will go through the steps that are necessary for setting up a basic ONOS cluster on two VMs hosted by a PC.

We will be working on a Santech PC equipped with an Intel i7-4710MQ, 2.50 GHz processor (4 cores, virtualized to 8 logical processors), and 16 GB of RAM, running Windows 10 Pro 64 bits.

5.1.1 Creation of the Virtual Machines

We are mainly going to follow instructions found on *ONOS Wiki* [19], however highlighting the steps which required some troubleshooting.

To begin with, we download and install **Oracle VM VirtualBox** [24].

We are going to create a new VM, which will be running **Ubuntu Server 14.04 LTS 64-bit**. The reason for this choice is that, at the time of writing of this document, the aforementioned OS is the one on which the ONOS team has been developing and testing the version of ONOS we are going to use.

Proceed to download the *.iso file of Ubuntu Server 14.04 LTS 64-bit [25].

Next, we must create the new VM. In the following sequence of steps, the choice of names for the VMs, username, passwords and IP addresses is completely arbitrary. Moreover, if the host machine is less powerful than the

one used for the setup described here, only allocate a quantity of resources that is sustainable for the used host machine.

In VirtualBox (VB), create a new VM called **ONOS_dev**. In the setup wizard, allocate to this VM 4 GB of RAM and 16 GB of hard disk, with fixed-size.

Confirm and wait for allocation to finish, then right click on the newly-created VM, and click on *Settings*, then:

- in the *System* section, under the *Processor* tab, allocate 2 processors to this VM;
- in the *Storage* section, load the *.iso image of the Ubuntu Server installation disc in the virtual disc reader of the VM;
- in the *Network* section, under the tab *Adapter 1*, check the checkbox to enable this adapter, and attach it to *NAT*, then, under the tab *Adapter 2*, check the checkbox to enable the second adapter, and attach it to *Host-only Adapter*.

Confirm the settings, then start the VM. It should boot from the virtual disc of Ubuntu Server.

Proceed in the installation by using:

- Spacebar [ ] to toggle checkboxes,
- Tab [⟷] to highlight *Continue* or a different bottom option,
- Enter [↩] to confirm.

Then:

- select the desired language, then select *Install Ubuntu Server*;
- select the appropriate language, time zone and keyboard layout;
- select **eth0** as primary network interface;
- insert the hostname for this VM. Here we used **onos-dev**;
- insert the username of the first user of the VM. Here we used **developer**;
- insert a password for the new user. Here we used **sonoonos**;
- do not encrypt your home directory;
- confirm or select the correct time-zone;
- install the system to the entire (virtual) disk without setting up LVM, and confirm;
- do not select any proxy;
- select to install security updates automatically;
- choose to install the *OpenSSH* server by checking the corresponding checkbox;

– choose to install *grub* to the Master Boot Record;
– confirm the reboot after the end of the installation with *Continue*.

The VM should then reboot to the login screen of Ubuntu Server. If it launches the installation again, stop the VM and manually unmount the *.iso file from its virtual disk reader, then start it again.

Log in to the VM from the terminal in its window, and install all the available upgrades, and also install the package **git**, with:

```
$ sudo apt-get update && sudo apt-get -y upgrade
$ sudo apt-get -y install git
```

We need to make sure that the network interface on the host-only network will come up at every boot of the VM, and additionally, for ease of use, we need to assign a static IP address to it. In order to choose the IP address, we need to check the network used by the VirtualBox Host-Only Network adapter, by inspecting the network configurations of the host machine. By default, the VirtualBox Host-Only Network adapter should have the IP address 192.168.56.1, which belongs to the network 192.168.56.0/24. If this is the case, we can use for the VM any other IP address of the same network. Here we chose **192.168.56.2**. In order to statically configure the VM's eth1 interface, we need to edit the file `/etc/network/interfaces`. We can do this with the text editor vim:

```
$ sudo vim /etc/network/interfaces
```

The text editor vim starts in *Command Mode*. In order to insert text, it is necessary to press the key I while in Command Mode. Then, after editing, go back to command mode by pressing key Esc . From Command Mode, type `:w` to save and remain in the editor, `:x` to save and exit, or `:q!` to exit without saving.

Add at the bottom of the mentioned file the following lines:

```
auto eth1
iface eth1 inet static
address 192.168.56.2
netmask 255.255.255.0
```

Then reboot the VM with:

```
$ sudo reboot
```

If the configuration was correct, from now on we will be able to access the VM from a SSH client running on the host machine (native in UNIX systems, or easily obtainable in Windows), as we know its IP address on the internal virtual network. Try to access to the VM from a SSH client by using the IP address we just chose. If we succeed in logging in, we can shut the VM down.

At this point, clone the VM, in order to obtain another VM with a clean and upgraded installation of Ubuntu Server 14.04 LTS 64-bit, which we will use as deployment machine in the following. To obtain the clone, just right click on the VM and select *Clone*. We can call the cloned VM **ONOS_depl**.

Wait until cloning is done, then run the VM ONOS_dev again, but this time in headless mode, with right click on it, then select *Start*, then *Headless start*.

Log in to the VM ONOS_dev through a SSH client. Detail on how to use SSH from a Windows system, are reported in Section A.1.

In the user's home folder, which in this case is `/home/developer/`, create two folders, named `Downloads` and `Applications`, with:

```
$ mkdir Downloads Applications
```

Then move into the `Downloads` folder with:

```
$ cd Downloads
```

Download the binaries of **Karaf** and **Maven** with:

```
$ wget http://archive.apache.org/dist/karaf/3.0.5/apache-karaf-3.0.5.tar.gz
$ wget http://archive.apache.org/dist/maven/maven-3/3.3.9/binaries/apache-
    maven-3.3.9-bin.tar.gz
```

Uncompress them in the `Applications` folder with:

```
$ tar -zxvf apache-karaf-3.0.5.tar.gz -C ~/Applications/
$ tar -zxvf apache-maven-3.3.9-bin.tar.gz -C ~/Applications/
```

Next, we need to install **Java**. First off, we need to install its repository, with:

```
$ sudo apt-get -y install software-properties-common -y
$ sudo add-apt-repository ppa:webupd8team/java -y
```

If this step prompts for the installation of *python-software-properties*, agree to install it.

Then, install the actual Java packages, with:

```
$ sudo apt-get update; sudo apt-get install oracle-java8-installer oracle-
    java8-set-default -y
```

Acknowledge the license when prompted.

5.1.2 ONOS Setup

Clone the ONOS repository from `git` in the home directory, with:

```
$ cd; git clone https://gerrit.onosproject.org/onos
```

The preamble of the command (`cd;`) simply makes sure that we are working in the home directory.

We need to add the setting of some environmental variables to the shell profile. To do so, edit the file `~/.bashrc` with:

```
$ sudo vim ~/.bashrc
```

and add at the bottom of the file the following lines:

```
export ONOS_ROOT="/home/developer/onos"
source $ONOS_ROOT/tools/dev/bash_profile
```

Then save and exit. This way the paths required by Java, Karaf, Maven and ONOS will be automatically set at each boot.

Execute the script `~/.bashrc` with:

```
$ source ~/.bashrc
```

Now we can type:

```
$ mvn --version
```

and we should get some information on Maven and Java. In particular, the information on Java should be the same as those given by:

```
$ java -version
```

We will now build ONOS using Maven. Move into the ONOS folder with:

```
$ cd ~/onos
```

and launch the build with:

```
$ mvn clean install
```

or simply:

```
$ mci
```

This operation may take some time. On the VM used in this setup, it took approximately 20 minutes.

During this process, Maven is going to download all the dependencies needed in order to build and install ONOS. For this reason, it is important that the Internet connection of the VM is never interrupted, or Maven will fail the build process.

Once the build has completed, we can use the `onos-package` shell tool to produce an installable compressed *.tar.gz file, which contains ONOS artifacts as well as ONOS-branded distribution of Apache Karaf. Obtain this file with the command:

```
$ onos-package
```

or simply:

```
$ op
```

The *.tar.gz file will be created in the folder `/tmp/`. This folder is cleared every time the machine is turned off. We can create a copy of the .*tar.gz file in order to be able to reuse it, if needed, on successive sessions, without

needing to create it again (refer to Section 5.1.4 for details on re-deployments of ONOS). To this purpose, when the creation of the file is finished, copy the file from the folder `/tmp/` to another folder under the user's home directory. In our setup, we used the folder `~/onos`, and made the copy with:

```
$ cp /tmp/onos-1.8.0.developer.tar.gz ~/onos
```

Then we need to configure on the depolyment VM, ONOS_depl.

First of all, as we obtained this VM as a clone of the other one, we need to change the IP address, to avoid a duplicate address. Edit the relevant file with:

```
$ sudo vim /etc/network/interfaces
```

and modify the address 192.168.56.2 to a different one, in this case **192.168.56.3**.

Now create the same two folders in the home directory as in the other VM, then move into the `Downloads` folder, with:

```
$ cd; mkdir Downloads Applications; cd Downloads
```

and download the binaries for Maven with:

```
$ wget http://archive.apache.org/dist/maven/maven-3/3.3.9/binaries/apache-
    maven-3.3.9-bin.tar.gz
```

then uncompress them in the `Applications` folder with

```
$ tar -zxvf apache-maven-3.3.9-bin.tar.gz -C ~/Applications/
```

Next, we install *Java*, following the same sequence of steps as we did for the other VM:

```
$ sudo apt-get install software-properties-common -y
$ sudo add-apt-repository ppa:webupd8team/java -y
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer oracle-java8-set-default -y
```

We will not install ONOS on this VM, so we cannot rely on its script to initialize the needed environmental variables, therefore we must do it manually. Edit the script `~/.bashrc` with:

```
$ cd; sudo vim .bashrc
```

and add the following lines at its bottom:

```
export JAVA_HOME="/usr/lib/jvm/java-8-oracle"
export MAVEN=${MAVEN:-~/Applications/apache-maven-3.3.9}
```

On this VM we also need to install **Mininet**. To do so, just clone its repository with:

```
$ git clone git://github.com/mininet/mininet
```

Next, choose the version to be installed with:

```
$ cd mininet
$ git tag # list available versions
$ git checkout -b 2.2.1 2.2.1
```

Then, compile and install the chosen version with:

```
$ util/install.sh -a
```

With the `-a` option we are installing every component included in the Mininet repository.

After the installation is finished, we can check its correctness by running a simple test with:

```
$ sudo mn --test pingall
```

This sets up a virtual network composed by two hosts connected to the same switch, and sends a `ping` from each host to the other.

Next, we need to create a new user named **sdn** with:

```
$ sudo adduser sdn
```

Leave all fields to their default values, except for the password, which needs to be **rocks**. Then validate with `y`.

We need the new user to be able to act as superuser without having to insert a password. To enable this, edit the *sudoers* file with:

```
$ sudo visudo
```

and add at the bottom of the file:

```
sdn ALL=(ALL) NOPASSWD:ALL
```

Next, we want the user `developer` on the VM ONOS_dev to be able to log in as the user developer on the VM ONOS_depl, through SSH, without requiring the password at every login. To this purpose, we set up key authorization. From the command line of the VM ONOS_depl, generate a RSA key pair with:

```
$ ssh-keygen -t rsa
```

Leave all fields to their default values. Then, use an utility of ONOS to finalize the authorization mechanism in the other VM, with:

```
$ onos-push-keys 192.168.56.3
```

Confirm by typing `yes` and the password for user `sdn`, that is `rocks`.

We are going to use various ONOS utilities for the act of deployment, and they expect the IP addresses and other parameters to be present in certain environmental variables. We can initialize those variables through the `cell` environment offered by ONOS. We are basically going to create a text file containing a list of environmental variables to be exported, then, with the command `cell`, we can load them all at the same time. In order for the `cell` tool to find the file, we need to create it in the folder `~/onos/tools/test/cells`. So, create the file `mycell` in the mentioned folder and edit it with:

```
$ vim onos/tools/test/cells/mycell
```

Insert the following text in the file:

```
# Controller VM instance OC1 and target Mininet instance OCN
export OC1="192.168.56.3"  # Target
export OCN="192.168.56.3"  # Mininet VM
# for node clustering
export ONOS_NIC= 1 9 2 .168.56.*
# ONOS features to load
export ONOS_FEATURES="webconsole,onos-api,onos-core-trivial,onos-cli"
```

Save and exit. Then load the values with:

```
$ cell mycell
```

and it should display:

```
ONOS_CELL=mycell
OCI=192.168.56.3
OC1=192.168.56.3
OCN=192.168.56.3
ONOS_FEATURES=webconsole,onos-api,onos-core-trivial,onos-cli
ONOS_GROUP=sdn
ONOS_NIC= 1 9 2 .168.56.*
ONOS_SCENARIOS=/home/developer/onos/tools/test/scenarios
ONOS_TOPO=default
ONOS_USER=sdn
ONOS_WEB_USER=onos
ONOS_WEB_PASS=rocks
```

Now, from the VM ONOS_dev, deploy ONOS on the other VM with:

```
$ stc setup
```

The operation may take some time (a couple of minutes, in the case of the PC we used). Moreover, it is acceptable if the last two steps fail. This is because the penultimate step is just a remote check of some installation parameter, and the last step is successful only if all the previous ones are successful. If the process concludes with 8 succesful steps and 2 failed ones, the deployment should work anyway.

From now on we can access the ONOS Graphical User Interface (GUI) at `http://192.168.56.3:8181/onos/ui`.
The username is `onos` and the password is `rocks`.

We are now going to use Mininet on the deployment VM. We want to set up the default topology (2 hosts inter-connected through a switch) and assign the control of the switch to a remote controller, which in this case will be the ONOS instance running on the same VM. The default topology is brought up by the use of the command `mn`, which must be run with superuser privileges. The assignment of a remote controller is made with the option `--controller remote`. It is also possible to specify the IP address and listening port of the controller, but, if not specified, the two parameter will default to 127.0.0.1 (`localhost`) on port 6633, which is actually where our instance

on ONOS is running. So type:

```
$ sudo mn --controller remote
```

Start a `ping` sequence between the two hosts with:

```
mininet> h1 ping h2
```

For the moment it should yield `Destination Unreachable`. This is because the controller does not know how to handle any packets, much less ARP and ICMP packets. We need to activate the proper ONOS application to carry out this task.

Open the ONOS Command Line Interface (CLI) with:

```
$ /opt/onos/bin/onos
```

and run:

```
onos> app activate org.onosproject.fwd
```

Now, in the Mininet window, the ping should start to be successful. If so, we can stop the ping sequence.

We now have everything we need to start creating and developing ONOS apps. ONOS comes with a set of bundled apps, such as the one for simple packet forwarding (`org.onosproject.fwd`) used to test the setup. Those are a good starting point for the development of new applications.

5.1.3 Setup of the IDE

The ONOS project does not enforce the use of a specific Integrated Development Environment (IDE), but rather, a set of guidelines that can be configured in any IDE. However, the examples and documentation focus on **IntelliJ IDEA** [26]. Therefore, we are going to install the mentioned IDE on ONOS_dev.

Move into the `Downloads` folder with:

```
$ cd ~/Downloads/
```

Then download the compressed installable file of the IDE with:

```
$ wget https://download.jetbrains.com/idea/ideaIC-2016.2.4.tar.gz
```

Next, uncompress the file to the `Application` folder with:

```
$ tar -zxvf ideaIC-2016.2.4.tar.gz -C ~/Applications/
```

Now, to run the IDE, just run the script idea.sh located in the folder `~/Applications/idea-IC-162.2032.8/bin` with:

```
$ ~/Applications/idea-IC-162.2032.8/bin/idea.sh
```

The script will try to open the GUI of the IDE, and forward it to the

SSH client. In order for this to work, the VM needs to have the required X11 libraries, and the SSH client must be running a X11 server. The easiest way of making sure that the VM has all the required X11 libraries is to install the package **xorg**, which will proceed to install a X-server on the development VM, plus all the libraries needed for it. We can install it with:

```
$ sudo apt-get -y install xorg
```

It is worthy to notice that the development VM will act as the client in the X11 session, while the machine at the other end, in this case the PC hosting the VM, will act as the X11 server. However, installing the X11 server on the VM automatically installs all the needed libraries for it to act as client as well. Regarding the X11 server, if the machine acting as the SSH client is running Windows, as it is in our case, we need to install and run a X11 server application on it, as this feature is not natively present in Windows. A good choice can be **Xming** [27].

5.1.4 How to deploy ONOS again

It might occur that ONOS needs to be deployed again even on the same deployment machine, due to wrong configurations of the previous deployment, or seemingly unexplicable errors that would require more time for troubleshooting than for a complete re-deployment.

To do that, load the correct cell file used for the first deployment, in this case `mycell`, with:

```
$ cell mycell
```

Then, copy the file *.tar.gz generated with onos-package from the folder where we created a backup copy of it, in this case `~/onos`, back to the folder `/tmp/`, where the utility stc is going to look for it. Superuser privileges are required to modify the content of the `/tmp/` folder, so use:

```
$ sudo cp ~/onos/onos-1.8.0.developer.tar.gz /tmp/
```

Lastly, run:

```
$ stc setup
```

If the setup succedes, the ONOS instance on the deployment machine should be identical to the one deployed the first time.

5.2 Setup of the host server

In this section, we will cover the configuration details of the physical machine that will host all the VMs needed for our test bed.

The machine is provided with a total of 24 virtualized cores, 32 GB of RAM, and 1 TB of storage space.

We performed a clean installation of **CentOS 7** on the machine, and we assign to this machine the hostname `deisnet213` and IP address **192.168.10.213**. Moreover we also create users, giving to at least one of them superuser privileges.

We will use **virt-manager** [28], that is a desktop user interface for managing the virtual machines through *libvirt*.

Install the program from a terminal window with:

```
$ sudo yum install virt-manager
```

Then run it with:

```
$ sudo virt-manager
```

We then need to create a new virtual network for the VMs to connect to. This virtual network is going to be used for management purposes, i.e., to access the VMs on an interface reserved for management. Each VM will also be provided with data interfaces.

In `virt-manager`, double click on the name of the hypervisor (QEMU/KVM), and then, from the *Virtual Networks* tab, create a new virtual network called `iot-net`, having as base network address **192.168.200.0/24**. Disable DHCP and do not activate IPv6 addressing on this network. Connect this network to the physical interface `em2` through NAT, and confirm. Then, again from the Virtual Networks tab, stop the newly created network. Then open a terminal on the host machine, and run:

```
$ sudo virsh net-edit iot-net
```

On the first instance of this command, the system will ask which text editor to use for editing the configuration files. Choose `vim`. Edit the field `bridge-name` so that it contains the name `virbr-iot`. Save the file and exit the editor (using `vim` commands), then, in `virt-manager`, start the virtual network again, and it will be associated to the renamed virtual bridge.

Also, create networks `iot-net-left` (base address: 172.16.1.0/24) and `iot-net-right` (base address: 172.16.2.0/24), which will be used for the communication between the the host server and the Mininet network hosted in VM `iot-mininet`, and the communication between the Mininet network and

the Web server hosted in VM `iot-server`, respectively. By following the procedure for renaming the virtual bridges described in the previous paragraph, associate the former network with interface `virbr-iot-left` (and assigning to it the IP address 172.16.1.253) and the latter one with interface `virbr-iot-right` (and assigning to it the IP address 172.16.2.253).

Using virt-manager's wizards, create the four VMs that we need, namely:

- **onos-dev** (2 cores, 2048 MB of RAM): this will be the VM on which ONOS and ONOS apps are developed and built;
  - 1 network interface on iot-net;
  - username: *developer*, password: *sonoonos*.
- **iot-depl** (2 cores, 2048 MB of RAM): this will be the VM on which ONOS will be deployed, therefore it will host the running instance of the controller;
  - 1 network interface on iot-net;
  - username: *developer*, password: *sonoonos*.
- **iot-mininet** (1 core, 1024 MB of RAM): this VM will host the (virtual) network, obtained with Mininet, whose switches are controlled by ONOS;
  - 3 network interfaces: one connected to iot-net, one bridged to virbr-iot-left, one bridged to virbr-iot-right;
  - username: *mininet*, password: *mininet*.
- **iot-server** (1 core, 1024 MB of RAM): this virtual machine will host a web server, that will serve as end-point of the communication between the IoT network and our system;
  - 2 network interfaces: one connected to iot-net, one bridged to virbr-iot-right;
  - username: *sdn*, password: *rocks*.

The VMs `onos-dev`, `iot-depl` and `iot-mininet` are initialized from pre-existing virtual hard disks. In particular, `onos-dev` uses the virtual hard disk of the VM ONOS_dev created locally in the previous section, and so for `iot-depl`, which uses the virtual hard disk of ONOS_depl, while `iot-mininet` uses the virtual hard disk of a VM previously used in [10].

The VM `iot-server` needs to be created from scratch. The configuration of this VM is covered in Section 5.2.3.

5.2.1 Configuration of the ONOS VMs

As stated in the previous Section, VMs `onos-dev` and `iot-depl` are initialized from the virtual hard disk we created for the preliminary local deployment. Therefore, they need some adjustment in order to let them work in the new deployment they are placed in.

First of all, we need to change the static IP address assigned to the network interfaces of both the VMs. By following the same procedure described in Section 5.1.1, we assigned to interface `eth0` of `onos-dev` the IP address **192.168.200.101**, and to interface `eth0` of `iot-depl` the IP address **192.168.200.102**.

Moreover, in order to allow correct ONOS deployment from the development machine, we need to set the correct environmental variables on `onos-dev`, through the `cell` utility. We created the file `/onos/tools/test/cells/mycellserver`, and inserted the following lines in it:

```
# Controller VM instance OC1 and target Mininet instance OCN
export OC1="192.168.200.102"  # Target
export OCN="192.168.200.103"  # Mininet VM
# for node clustering
export ONOS_NIC=  1 9 2 .168.200.*
# ONOS features to load
export ONOS_FEATURES="webconsole,onos-api,onos-core-trivial,onos-cli"
```

Save and exit. Then, after loading the values with `cell mycellserver`, we have:

```
ONOS_CELL=iot-depl
OCI=192.168.200.102
OC1=192.168.200.102
OCN=192.168.200.103
ONOS_FEATURES=webconsole,onos-api,onos-core-trivial,onos-cli
ONOS_GROUP=sdn
ONOS_NIC=
ONOS_SCENARIOS=/home/developer/onos/tools/test/scenarios
ONOS_TOPO=default
ONOS_USER=sdn
ONOS_USE_SSH=true
ONOS_WEB_PASS=rocks
ONOS_WEB_USER=onos
```

5.2.2 Setup of the Mininet network

As introduced in Section 2.4.2, with Mininet we can emulate all the components of a SDN network, but, in order to do that, we need to properly configure and activate each component.

The data plane topology in Figure 4.3 was built with a customized Mininet script specifying the required OpenFlow switches, as well as routers and VNFs as separated network namespaces. The script has been written in the **Python** language, by taking advantage of the language's integration with the underlying OS, and Mininet's Python APIs [29].

What the script does is, in short, to:

- create all the needed switches, while also disabling the Spanning Tree Protocol (STP) on switches `s1`, `s2`, `s3` and `s4`, so as to prevent broadcast storms due to the ARP requests being flooded in the portion of the network containing a loop;
- create the hosts which will act as virtual routers and virtualized network functions;
- create links between switches and hosts, this way inherently declaring the interfaces of each of them;
- assign MAC addresses to the interfaces of the hosts;
- add (i.e., "connect") each port of the emulated switches to the correspondent Open vSwitch;
- activate the network components;
- delete the default IP addresses and IP forwarding rules from each of the hosts;
- bridge the two interfaces of host `chk` and uses the tool `tc` to configure the introduction of a random delay in the traffic traversing the host;
- assign to each of the virtual routers and host `bck` the proper IP addresses while also setting the required IP forwarding rules;
- set ARP storm avoidance rules on switch `s2`, which is the one having two ports connected to the same host, by installing a proper flow rule in its table;
- bridge switches `s6` and `s7` to the network interface `eth1` and `eth2` of the VM hosting the Mininet network;

- set the parameters for the connection to the remote controller, ONOS, in each switch, and waits for the controller to connect to each of the switches;
- start the CLI of the emulated network;
- stop the emulated network when the user terminates the CLI, and clean the system from any residual Mininet configuration.

The whole Python script, with comments, is shown in Section A.2.

5.2.3 Configuration of the Web server VM

In the design phase, it has been agreed that the data traffic for the two different QoS classes should be sent on a **TCP** connection on different TCP ports, namely **port 8091** for the Minimum Latency class, and **port 8094** for the High Reliability class. Moreover, the gathered data should be accessible to the user through a convenient Web interface.

Therefore, we want to obtain a Web server that is able to listen on multiple TCP ports, specifically:

- port 80, where the server exposes its homepage, displaying tables containing relevant data extracted from the packets received with POST requests,
- port 8091, where the server will receive POST requests of the Minimum Latency class;
- port 8094, where the server will receive POST requests of the High Reliability class.

We will use **Ubuntu 16.04 Server** as the OS of this VM. After downloading its image from [30], we load it in the virtual optical drive and boot the VM, then follow the wizard for the setup of the OS. In virt-manager, connect the primary network interface of this VM to the virtual network `iot-net`, and the second one to the virtual bridge `virbr-iot-right`.

Once the installation of the OS is done, boot the VM for the first time, and install the packages needed to setup an **Apache** Web server, with:

```
$ sudo apt-get update
$ sudo apt-get install apache2 php libapache2-mod-php
```

Now, we need to modify some configuration files which belong to the user `www-data`, which is the user who owns the Apache service, therefore we need superuser privileges to modify the files and folders of that user. Instead of

specifying `sudo` in all of the following commands, we can just start acting as the user `root`, with:

```
$ sudo su
```

We can now move into the parent folder of the default home folder of the Web server, which is `/var/www/`, and create the subfolders which will serve as the home directories of the three Web locations we are configuring:

```
# cd /var/www/
# mkdir domain-80 domain-8091 domain-8094
# mv html/index.html domain-80/
```

Then, we need to make the server listen on the correct ports. In order to do this, we need to modify the file `/etc/apache2/ports.conf` so that it contains the following lines:

```
Listen 80
Listen 8091
Listen 8094
```

Moreover, we need to define virtual hosts in the file
`/etc/apache2/sites-enabled/000-default.conf`, in the following way:

```
<VirtualHost *:80>
        ServerAdmin webmaster@localhost
        DocumentRoot /var/www/domain-80
        ErrorLog ${APACHE_LOG_DIR}/error80.log
        CustomLog ${APACHE_LOG_DIR}/access80.log combined
</VirtualHost>
<VirtualHost *:8091>
        ServerAdmin webmaster@localhost
        DocumentRoot /var/www/domain-8091
        ErrorLog ${APACHE_LOG_DIR}/error8091.log
        CustomLog ${APACHE_LOG_DIR}/access8091.log combined
</VirtualHost>
<VirtualHost *:8094>
        ServerAdmin webmaster@localhost
        DocumentRoot /var/www/domain-8094
        ErrorLog ${APACHE_LOG_DIR}/error8094.log
        CustomLog ${APACHE_LOG_DIR}/access8094.log combined
</VirtualHost>
```

Remark: the symbol " * " before each of the listening port means that the server will listen on that TCP port on any of its IP addresses. This way, we can access to the Web services both from the "management" interface (i.e., the one connected to the virtual network `iot-net`) and the "data" interface (i.e., the one bridged with `virbr-iot-right`).

For the purpose of our activities, we also need to avoid that the TCP connection gets closed by the server after 5 seconds of inactivity, as it would happen by default. The reason for this need is that we don't know the inter-arrival time of two consequent POST requests from the IoT network, and we

want to generate as little overhead as possible, thus we are interested in keeping the connections open for a longer time, and also not to limit the number of times the connection is kept open upon a keep-alive request from the client. Therefore, in file `/etc/apache2/apache2.conf` we modified the values to:

```
KeepAliveTimeout 60
MaxKeepAliveRequests 0
```

At this point, this VM is listening on TCP ports 80, 8091 and 8094. We need to develop and setup an appropriate user interface which will be exposed as the server's homepage, and two suitable POST message handlers.

5.2.4 POST message handler and server homepage

A simple way of obtaining a HTTP POST message handler is by using a **PHP** script. The script will receive the message and extract the JSON carried by the POST request, then store its content in a database. For the sake of simplicity, we decided to simply store the JSON object as a string in a text file, called `database.json`.

We created a PHP script called `postserver.php` for each of the two ports on which the server will listen for POST requests. The two scripts are actually identical, and, according to the configuration described in Section 5.2.3, they must be placed in directories `/var/www/domain-8091` and `/var/www/domain-8094`. The content of the scripts, with comments, is shown in Section A.3.

Without the necessary permissions, the PHP script would not be allowed to write to a file in its folder. That is because the folder does not belong to the user which runs the PHP script, that is the same user that runs the Web server process. By default in this installation, that is the user `www-data`. A simple way to solve the writing permission problem is to change the ownership of the folder `/var/www/html`, and give it to `www-data`, with:

```
$ sudo chown www-data:www-data /var/www/html/
```

This way, any PHP script run by the Web server will be allowed to write in that folder.

We also developed a suitable web interface, reachable on TCP port 80, that is the default port for HTTP connections. This page shows the data received with the POST requests, organized in two tables, one for each class of traffic. The content of the page, as it looks when opened from a Web browser, is shown in Figure 5.2.

5.2.5 Additional configurations on `deisnet213`

For testing purposes (i.e., in order to be able to send forged data packets from the host machine to the Web server through the Mininet/SDN network), it is important to set an IP forwarding rule in the routing table of `deisnet213`. Specifically, all the traffic toward the Web server's data interface (i.e., the one with address 172.16.2.100) must be forwarded to virtual router `vrl` in the Mininet/SDN network. As it is mentioned in Section 5.2.2, the virtual router's external interface is in the same broadcast domain as the host's interface `virbr-iot-left`, therefore we need to steer all the traffic coming from `deisnet213` and directed to the Web server through the mentioned interface. If we don't set this rule, then packets coming from `deisnet213` and directed to the Web server would be delivered directly through interface `virbr-iot-right`.

So, first we need to remove the default rule for reaching network 172.16.2.0/24, then we can add the new rule, with:

```
$ route del -net 172.16.2.0/24
$ route add -net 172.16.2.0 netmask 255.255.255.0 gw 172.16.1.254 dev virbr-
    iot-left
```

The host machine `deisnet213` will also serve as intermediate IP node between the IoT Gateway and the Mininet/SDN network. For this reason, IP forwarding must be enabled on it. To do that, edit the file `/etc/sysctl.conf` with superuser privileges, and add at its bottom the following line:

```
net.ipv4.ip_forward = 1
```

Then apply the new settings with:

```
$ sudo sysctl -p
```

Lastly, we added an alias `em2:iot` to interface `em2` of `deisnet213`, with:

```
$ sudo ifconfig em2:iot 192.168.10.6
$ sudo ifconfig em2:iot up
```

This way, we can assign the address of the alias as next hop towards the Web server to the IoT Gateway, thus avoiding to use as next hop the same address used to access the server `deisnet213` for management purposes (i.e., that of interface `em2`).

5.3 Modifications on the ONOS application

The ONOS application that provided Service Function Chaining across multiple Mininet clusters had been developed for [10]. That application, hereafter mentioned as **CeNA**, made for a perfect starting point for the activities of this thesis. The version of the application we started from already implemented the chaining based on POST requests received on the REST NBI. Then we needed to provide the application with the support for multiple QoS classes, and allow it to operate on our specific set of Mininet clusters.

For the latter aspect, we needed to adapt the JSON files that described the environment which the application was previously designed to work on, as well as the JSON files which contained the description of an example chain to be deployed (details on the JSON files are covered in Section 5.3.1), then we modified the code of the ONOS application, CeNA, in such a way that it could cope with the new working requirements (as covered in Sections 5.3.2 and 5.3.3).

5.3.1 JSON cluster descriptors and requests

CeNA expects to be given the description of the cluster(s) it has to control as JSON files, one for each level-2 broadcast domain. As our Mininet network includes two level-3 devices (i.e., virtual routers `vrl` and `vrr`), it is composed of 3 level-2 broadcast domains. In particular:

- Listing 5.1 shows the description of the part of the network which is used to interface the emulated Cloud domain to the IoT GW; it must contain the reference to that GW, in the form of a host, so that CeNA will then be able to handle traffic coming from it, along with the reference to the network equipment used to interact with it;
- Listing 5.2 contains the description of the main part of the emulated Cloud domain, the one where the actual SFC happens; in fact, it includes the reference to the hosts used to perform the required Virtual Network Functions;
- Listing 5.3 shows the description of the part of the network which is used to interface the emulated Cloud domain to the Web server; it includes the reference to the VM hosting the Web server.

On the other hand, Listings 5.4 and 5.5 show the actual implementation of chain requests that are included in the POST messages sent to the REST

NBI of the ONOS application, one for each QoS class, i.e., minimum latency and high reliability, respectively.

5.3.2 Waypoint Constraint

We modified the function `buildChain` so as to choose deterministically the switch to be traversed in the minimum latency path. Previously, the Service Function Chain (SFC) was installed without specifying any contraints related to the path to be followed between two endpoints of an intent. This way, however, the application tends to always choose to go from `vrl` to `vrr` traversing switches `s1`, `s2` and `s4`. This way, switch `s3` is never traversed, when it could share the traffic going through switch `s2` instead. Moreover, we have a potential overload of switch `s2`.

For these reasons, we added a *Waypoint Constraint* to the intents going from `vrl` to `vrr` and viceversa, forcing the flow rules to be installed in such a way that the traffic which do not need to reach `chk` (i.e., the traffic of the minimum latency QoS class) will not traverse switch `s2` in going from switch `s1` to switch `s3`, but it will traverse switch `s4`, thus distributing the traffic load to both the intermediate switches.

In the portion of code shown in Listing 5.6, we build the lists of `ConnectPoints` and `Constraint` which the intent will be required to use in order to install a path, by means of flow rules, from `vrl` to `vrr` in the case of minimum latency traffic. As the path is symmetric, but the direction of the `ConnectPoints` is not, we need to differentiate the case of the two possible directions of traffic, i.e., from `vrl` to `vrr` or the other way round.

5.3.3 QoS requirements management

We modified the function `addChainIntent` to obtain the two functions `addChainIntentForth` and `addChainIntentBack`, which can be used to install the SFC in both directions, while also including QoS requirements. For the moment, the only specification that is taken into account is the one on the QoS class, that can either be `minlat` or `highrel`, while the specific QoS requirement (e.g., a specific maximum tolerated delay) is ignored. Based on the QoS specification, the intent is given the relevant traffic filter, in the form of a `TrafficSelector`, that filters traffic based on the TCP port, according to the adopted convention of port 8091 for `minlat` traffic and port 8094 for

`highrel` traffic. The portion of code implementing this is shown in Listing 5.7.

For the sake of completeness, it is also worthy to mention that, in order to carry out the performance evaluation of the VIM NBI, as it is presented in Chapter 6, we had to apply a slightly different approach to the creation of the `TrafficSelector`. In fact, as a part of the performance evaluation is focused on measuring the NBI's response time in the event of a sequence of SFC requests for the same class of traffic, and since the requests would all come from the same source (i.e., the host server) and would be directed to the same destination (i.e., the Web server), the VIM would try to install a sequence of intents (one for each request) having the same source IP, destination IP, and TCP port number. However, this would result in an error in the OpenFlow domain, due to the fact that, on each switch, only a single flow rule can be installed for a given set of filters, and further attempts to install a flow rule that is already in the flow table are refused. For those reasons, we decided to allocate a pool of TCP ports to both traffic classes, and assign a different, progressive TCP port number to each subsequent SFC request. This way we allow the VIM to go through the whole process without errors that would invalidate the performance evaluation.

IoT Server

High reliability traffic

Show 10 entries

Search:

NetID	TX-H	TX-L	RX-H	RX-L	Type	TTL	PL0	PL1	PL2	PL3	PL4	PL5	PL6	PL7	PL8	SeqN	RX-t
1	4	19	14	13	114	17	1	40	20	3	4	5	6	7	8	7	1484671788120
1	8	13	19	20	111	10	1	51	59	3	4	5	6	7	8	32	1484671788750
1	10	5	10	11	128	14	1	34	29	3	4	5	6	7	8	25	1484671789261
1	14	16	15	8	117	2	1	33	53	3	4	5	6	7	8	2	1484671789771
1	14	4	7	17	101	10	1	27	30	3	4	5	6	7	8	53	1484671790281
1	4	14	2	9	105	3	1	29	27	3	4	5	6	7	8	36	1484671790792
1	10	14	17	20	125	7	1	22	31	3	4	5	6	7	8	13	1484671791304
1	17	10	2	7	100	12	1	27	47	3	4	5	6	7	8	24	1484671791816
1	4	6	2	9	111	9	1	37	59	3	4	5	6	7	8	19	1484671792328
1	3	19	4	11	125	7	1	32	54	3	4	5	6	7	8	44	1484671792839
NetID	TX-H	TX-L	RX-H	RX-L	Type	TTL	PL0	PL1	PL2	PL3	PL4	PL5	PL6	PL7	PL8	SeqN	RX-t

Showing 1 to 10 of 12 entries

Previous 1 2 Next

Low latency traffic

Show 10 entries

Search:

NetID	TX-H	TX-L	RX-H	RX-L	Type	TTL	PL0	PL1	PL2	PL3	PL4	PL5	PL6	PL7	PL8	SeqN	RX-t
1	20	13	5	20	122	20	1	24	43	3	4	5	6	7	8	27	1484671794372
1	3	10	12	8	119	17	1	60	36	3	4	5	6	7	8	32	1484671794883
1	9	10	3	19	100	2	1	34	28	3	4	5	6	7	8	0	1484671795393
1	18	8	0	17	101	3	1	47	59	3	4	5	6	7	8	41	1484671795903
1	5	11	13	5	122	20	1	36	46	3	4	5	6	7	8	52	1484671796417
1	16	14	7	14	106	4	1	20	32	3	4	5	6	7	8	51	1484671796927
1	6	7	8	1	125	18	1	34	60	3	4	5	6	7	8	60	1484671797437
1	16	15	1	10	121	11	1	41	26	3	4	5	6	7	8	44	1484671797947
1	18	15	6	1	112	19	1	46	45	3	4	5	6	7	8	40	1484671798458
1	20	3	7	15	119	13	1	50	38	3	4	5	6	7	8	17	1484671798968
NetID	TX-H	TX-L	RX-H	RX-L	Type	TTL	PL0	PL1	PL2	PL3	PL4	PL5	PL6	PL7	PL8	SeqN	RX-t

Showing 1 to 10 of 33 entries

Previous 1 2 3 4 Next

Flush DataBases:

- Flush HR DB only
- Flush LL DB only
- Flush both DataBases

Confirm

Figure 5.2: Homepage of the IoT Server

```
{
    "bcastDomain": [
    {
        "id": 1,
        "vnfs": [
        ],
        "hosts": [
        {
            "deviceId": "of:0000000000000005",
            "hostId": "52:54:00:dc:f5:fc/-1",
            "name": "iotgw",
            "location": [
            "of:0000000000000005/2"
            ]
        }
        ],
        "devices": [
        {
            "name": "s6",
            "deviceId": "of:0000000000000005"
        }
        ],
        "gateways": [
        {
            "deviceId": "of:0000000000000005",
            "hostId": "00:00:00:00:00:0A/-1",
            "type": "E",
            "name": "vrl",
            "location": [
            "of:0000000000000005/1"
            ]
        }
        ]
    }
    ]
}
```

Listing 5.1: JSON descriptor of the emulated Cloud domain - east

```
{
    "bcastDomain": [
    {
        "id": 2,
        "vnfs": [
        {
            "name": "vnf",
            "location": [
            "of:0000000000000002/2",
            "of:0000000000000002/3"
            ]
        },
        {
            "name": "bck",
            "location": [
            "of:0000000000000007/2"
            ]
        }
        ],
        "hosts": [
        ],
        "devices": [
        {
            "name": "s1",
            "deviceId": "of:0000000000000001"
        },
        {
            "name": "s2",
            "deviceId": "of:0000000000000002"
        },
        {
            "name": "s3",
            "deviceId": "of:0000000000000003"
        },
        {
            "name": "s4",
            "deviceId": "of:0000000000000004"
        },
        {
            "name": "s5",
            "deviceId": "of:0000000000000007"
        }
        ],
        "gateways": [
        {
            "deviceId": "of:0000000000000001",
            "hostId": "00:00:00:00:00:0B/-1",
            "type": "I",
            "name": "vrl",
            "location": [
            "of:0000000000000001/1"
            ]
        },
        {
            "deviceId": "of:0000000000000004",
            "hostId": "00:00:00:00:00:0F/-1",
            "type": "E",
            "name": "vrl",
            "location": [
            "of:0000000000000004/3"
            ]
        }
        ]
    }
    ]
}
```


Listing 5.2: JSON descriptor of the emulated Cloud domain - center

```
{
    "bcastDomain": [
    {
        "id": 3,
        "vnfs": [
        ],
        "hosts": [
        {
            "deviceId": "of:0000000000000006",
            "hostId": "52:54:00:fb:6c:c1/-1",
            "name": "iotsv",
            "location": [
            "of:0000000000000006/2"
            ]
        }
        ],
        "devices": [
        {
            "name": "s7",
            "deviceId": "of:0000000000000006"
        }
        ],
        "gateways": [
        {
            "deviceId": "of:0000000000000006",
            "hostId": "00:00:00:00:00:0E/-1",
            "type": "I",
            "name": "vrr",
            "location": [
            "of:0000000000000006/1"
            ]
        }
        ]
    }
    ]
}
```


Listing 5.3: JSON descriptor of the emulated Cloud domain - west


```
{
        "src": "iotgw",
        "dst": "iotsv",
        "qos": "minlat",
        "qos-thr": "10 ms",
        "vnfList": "null",
        "dupList": "null"
}
```


Listing 5.4: JSON for request of min lat chain

```
{
        "src": "iotgw",
        "dst": "iotsv",
        "qos": "highrel",
        "qos-thr": "99\%",
        "vnf":
        [{
                "name": "vnf",
                "port_sym": "true",
                "terminal": "false",
                "path_sym": "true"
        }, {
                "name": "bck",
                "port_sym": "true",
                "terminal": "false",
                "path_sym": "false"
        }],
        "dup": [{
                "name": "bck"
        }]
}
```


Listing 5.5: JSON for request of high rel chain

```
List<Constraint> constraintList = new ArrayList<>();
if ( forth ) {
    // Resetting elements to build this extra part
    curIntent = new MyIntent();
    tempCpList = new ArrayList<>();
    tempCpList.add(new ConnectPoint(DeviceId.deviceId("of:0000000000000001"),PortNumber.portNumber(1)));
    tempCpList.add(new ConnectPoint(DeviceId.deviceId("of:0000000000000004"),PortNumber.portNumber(3)));
    constraintList.add(new WaypointConstraint(DeviceId.deviceId("of:0000000000000003")));
    curIntent.setToi(1); // This is a P2P intent
    curIntent.setConnectPoints(tempCpList);
    curIntent.setConstraintList(constraintList);
    intentsList.add(curIntent);
    log.info("[DEBUG-NSE-4] ( gd3 ) Added intents to make the flow forcibly go through s3, forth way");
} else {
    // Resetting elements to build this extra part
    curIntent = new MyIntent();
    tempCpList = new ArrayList<>();
    tempCpList.add(new ConnectPoint(DeviceId.deviceId("of:0000000000000004"),PortNumber.portNumber(3)));
    tempCpList.add(new ConnectPoint(DeviceId.deviceId("of:0000000000000001"),PortNumber.portNumber(1)));
    constraintList.add(new WaypointConstraint(DeviceId.deviceId("of:0000000000000003")));
    curIntent.setToi(1); // This is a P2P intent
    curIntent.setConnectPoints(tempCpList);
    curIntent.setConstraintList(constraintList);
    intentsList.add(curIntent);
    log.info("[DEBUG-NSE-4] ( gd3 ) Added intents to make the flow forcibly go through s3, back way");
}
```

Listing 5.6: Implementation of a Waypoint path constraint

```
TrafficSelector selector = null;
switch (strQos.get(0)) {
    case "minlat":
        selector = DefaultTrafficSelector.builder()
            .matchEthType(Ethernet.TYPE_IPV4)
            .matchIPProtocol( (byte) 6 ) // TCP
            .matchIPSrc(ipSrc)
            .matchIPDst(ipDst)
            .matchTcpSrc(TpPort.tpPort(8091))
            .build();
    break;
        case "highrel":
            selector = DefaultTrafficSelector.builder()
            .matchEthType(Ethernet.TYPE_IPV4)
            .matchIPProtocol( (byte) 6 ) // TCP
            .matchIPSrc(ipSrc)
            .matchIPDst(ipDst)
            .matchTcpSrc(TpPort.tpPort(8094))
            .build();
    break;
    default:
        selector = DefaultTrafficSelector.builder().build();
        log.error("[ERROR] QoS class " + strQos.get(0) + " not
            recognized.");
}
```

Listing 5.7: TrafficSelector for different QoS specifications

Chapter 6

Performance evaluation

6.1 Measurements of data plane latency

We measured the performance within the emulated cloud network when the customer requested the service specifying two traffic classes, according to the QoS features offered by the OpenFlow SDN domain: minimum latency and high reliability. In this case, one-way latency in the emulated cloud network was measured by comparing timestamps of each packet captured at switches `s6` and `s7`. The capture was performed in the server hosting the Mininet virtual machine (i.e., the server `deisnet213`), so that the same reference clock was used for both timestamps.

More specifically, for this performance evaluation:

- the packet sniffer `tcpdump` is set running on the interfaces of `deisnet213` at the two ends of the Mininet network;
- a large number (in this case, 10 000) of POST requests are sent from the IoT Gateway towards the Web server through the Mininet network;
- the captured packets are then post-processed to compute the average data plane latency.

In particular, this captures yield two *.pcap files, which we post-processed in the following way:

- we generated a *.csv file for each *.pcap file by using the tool `tshark`, with a line for each HTTP request packet captured, each line containing

3 fields: the timestamp of the packet as time since Epoch in milliseconds, the TCP Port number as a 16-bit integer number, and the absolute TCP sequence number as a 32-bit integer number;

- we sorted the *.csv files by TCP Sequence number, then compared the obtained files line-to-line: as no packet loss was expected to happen in the Mininet network, we have direct correspondence between the packet captured at the host server's interface and the one captured at the Web server interface;

- by computing the difference between the timestamps of each pair of packets, we obtained the data plane latency undergone by each of them, and we wrote the value to another *.csv file, having on each line the indication of TCP Port, TCP Sequence number, and the value of latency;

- we separated the results of two classes of traffic by discriminating on TCP Port number, and we created another *.csv file for each class of traffic;

- for each class of traffic, we use the set of measurements to compute the statistical mean, standard deviation.

Results are reported in Table 6.1, in terms of average of the data plane one-way latency. They show the correct behavior of the OpenFlow domain with respect to the requested QoS feature: very limited delays (i.e., less than 1 ms) were measured in the minimum latency case, and in the high reliability case the expected delay was measured (i.e., averaging at 30 ms), and `bck` successfully stored a copy of the entire data set transmitted by the IoT Gateway.

QoS feature	**Average lat.**	**St. Dev.**
Min Latency	0.3 ms	0.28 ms
High Reliability	31.7 ms	2.41 ms

Table 6.1: Data plane latency results

We also wanted to test how the emulated network would behave when a sequence of POST requests is sent through it, with a certain average time between consecutive requests. So we sent sequences of 1 000 POST requests from `deisnet213` to the Web server, by making up the data contained in the

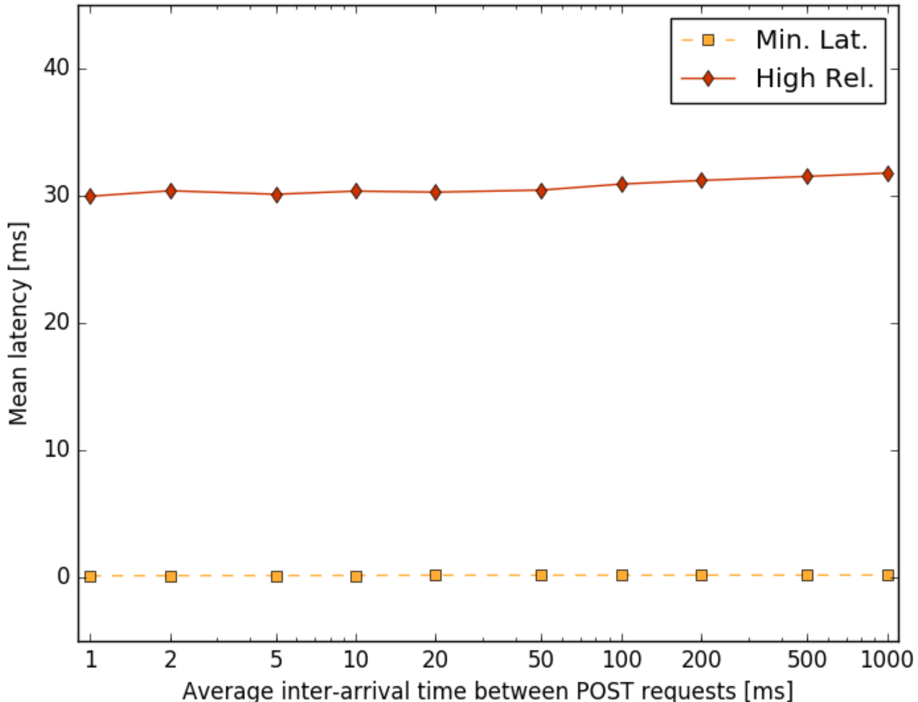


Figure 6.1: Data plane latency

JSON, and sending requests with a certain average inter-arrival time, whose value is sampled every time from a negative exponential distribution, so as to emulate Poisson arrivals. We performed a run for a range of average inter-arrival times, from 1000 ms to 1 ms, then we computed the mean value for each run. The results are shown in Figure 6.1.

As it was expected, the performances remain the same, regardless of the time between two consecutive POST requests. The emulated Cloud network does not show any sign of stress even for average arrival rates of 1 000 requests per second.

6.2 Measurements of control plane delay

We also measured the NBI response time at the VIM implemented in ONOS, i.e., the interval between the instant when a JSON service chain specification is received by the VIM and the successful setup by ONOS of the forwarding rules in the OpenFlow domain. To assess the scalability of the NBI:

- the ONOS instance is cleared of all the installed intents;
- a sequence of POST messages containing JSON service chain specification is sent to the REST interface of the VIM through its NBI;
- the time needed for the installation of the intents is measured as difference between the instant in time on which the request is sent and the instant on which the issuer of the request receives the HTTP OK response;
- the same steps are repeated for an increasing number of requests (from 5 to 200), and each measured response time was obtained as an average over 20 runs with the same number of requests.

Figure 6.2 shows the average NBI response time with 95% confidence intervals. The numbers show that the VIM is very responsive, in the order of tens of milliseconds. The setup of high-reliable service chains takes slightly longer than the minimum latency ones because of the relatively more complex forwarding paths that must be programmed in the switches (traversing `chk`, mirroring to `bck`, as in Figure 4.3).

The measurements reported in Figure 6.2, however, can be broken down into three components:

– the time between the instantiation of the `curl` command generating the POST request and the instant when the POST message is received by the VIM;
– the actual processing time, needed by the VIM to instantiate all the new intents;
– the time between the instantiation of the HTTP OK response by the VIM and the instant when the response is received by the entity who had previously launched the `curl`.

We are mainly interested in the central component of the previously measured time, as the other two depend on the data plane latency in the network. In order to more properly assess scalability of the actual time needed

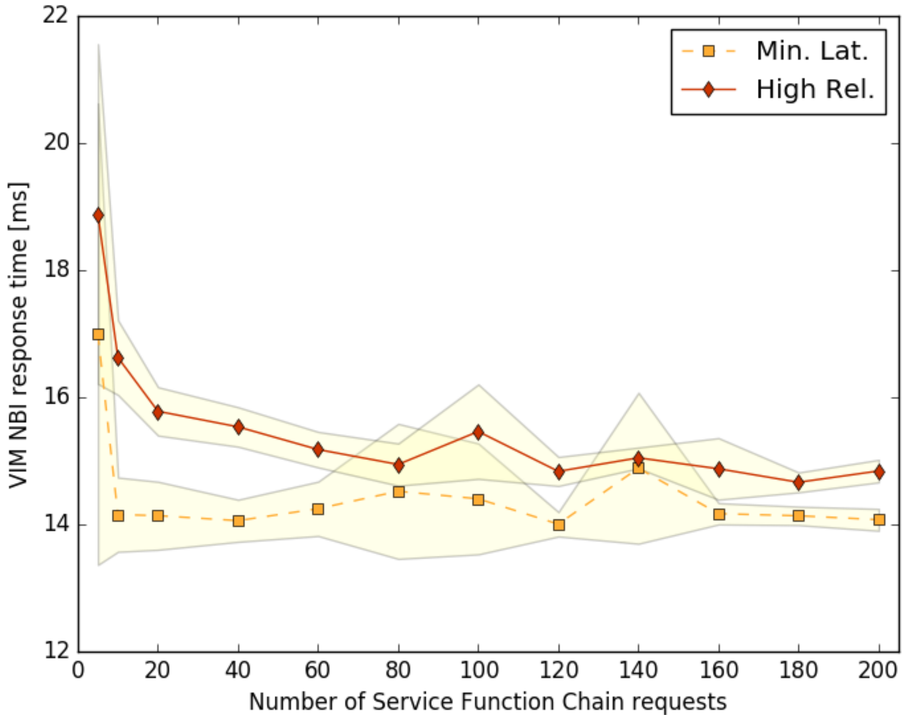


Figure 6.2: Average NBI response time and 95% confidence interval at the SDN/Cloud VIM, with increasing number of SFC requests

to the VIM for instantiating the requested sequence of intents, we proceeded by launching another set of sequences of SFC requests, but this time we saved the content of the ONOS Log after each sequence, in order to then be able to use the timestamp reported in every debug line to measure only the actual processing time. The results of this performance evaluation are reported in Figure 6.3.

It may also be interesting to analyze those results from a different point of view, given by the relative frequency of occurrence of each measured processing time value. The probability mass function of the two distribution, one for each QoS class, is shown in Figure 6.4

As expected, the ideal "bell" which we would obtain by representing an infinite number of measurements is shifted on the horizontal axis toward lower

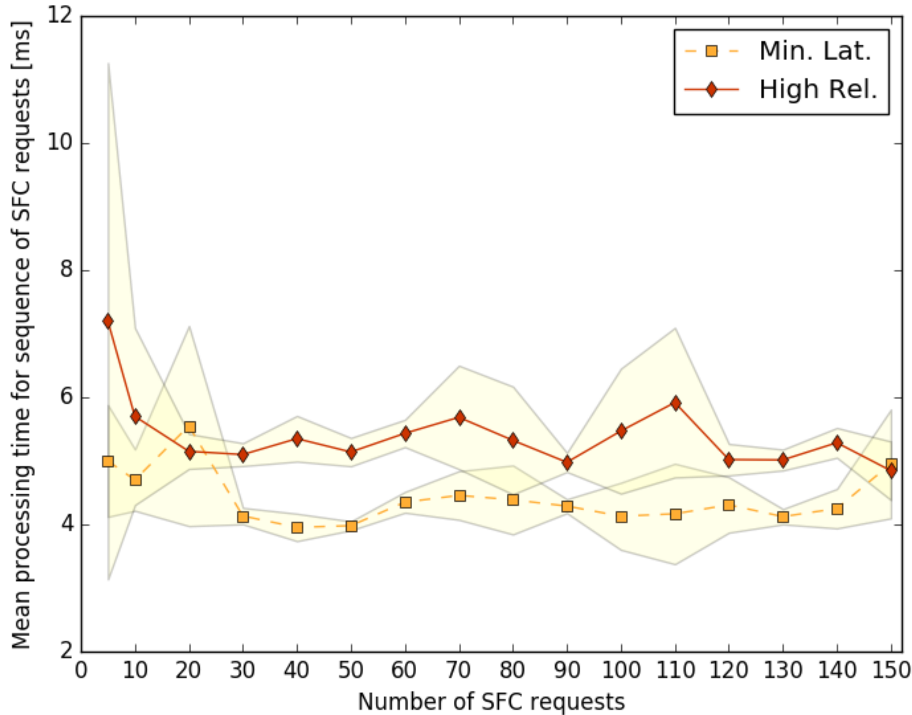


Figure 6.3: Average processing time for each sequence of SFC requests at the SDN/Cloud VIM, with increasing number of SFC requests

values of processing time for the *Min. Lat.* traffic, while it is shifted toward larger values for the *High Rel.* traffic, coherently with the results reported in Figures 6.2 and 6.3.

Although they are difficult to be spotted in Figure 6.4, due to their very low number of occurrences, it is interesting to highlight the presence of some measured processing time values which are much different from the average values. In fact, in sporadic cases we measured a very large processing time, most likely due to a sub-optimal implementation of some part of the code of our application.

By analyzing the ONOS Log, we can further investigate this phenomenon. Its cause appears to be residing in the function `getIpSrcOfCp` of component *ChainManager*, which is in charge of fetching the IP address of the host con-

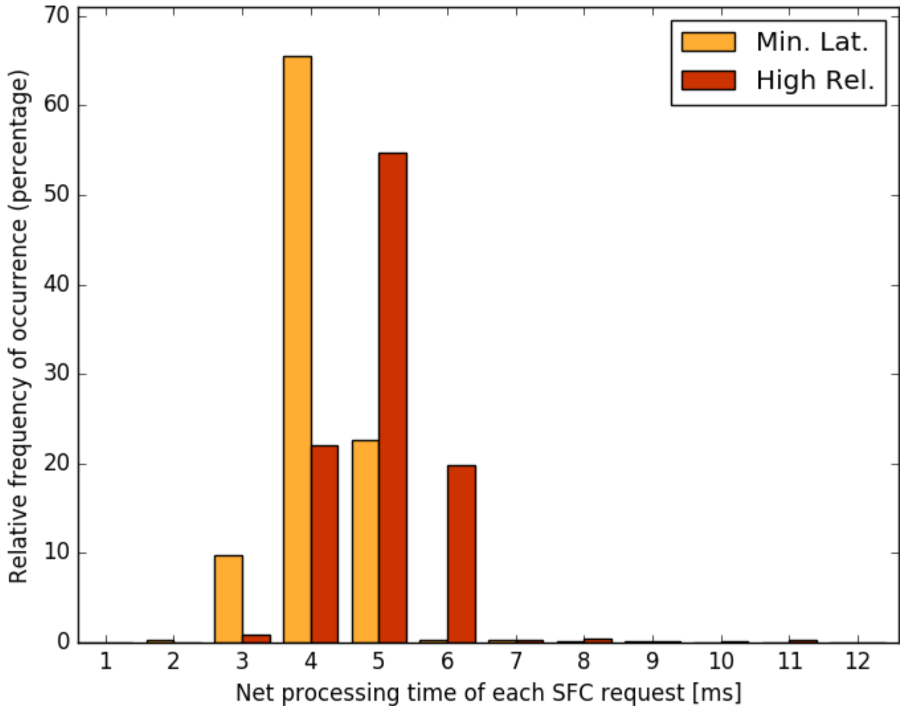


Figure 6.4: Distribution of processing time for each sequence of SFC requests at the SDN/Cloud VIM, with increasing number of SFC requests

nected to a specific *Connect Point*. On average, this function takes less than 1 ms to return, but on some runs it takes a much larger time, in the order of tens of milliseconds. This is probably due to the way the function has been implemented. Finding a solution that limits this unwanted behavior is a starting point for future developments of our ONOS application.

Chapter 7

Conclusions

In the activities of this Master Thesis, after describing the reference scenario and architectures, we worked on a new solution for the declaration of intents for Service Function Chain over a SDN network, and expanded the implementation of this solution to also include differentiation among multiple Quality of Service classes, presenting an experimental validation of that extension of the application over a test bed.

The main focus of this work has been on the OpenFlow/Cloud domain, which we had to design and build by taking concerted decisions on the resources to use, and configuring the hardware and software necessary to obtain a working instance of it. We also focused on a non-SDN domain interconnecting multiple SDN domains while achieving Service Function Chaining, obtaining promising results that will be the main point of future activities.

The experimental results show that our implementation works as it is designed to, presenting the expected latency in the data plane, and a reasonable behavior in terms of control plane delay, in the instance that more complex SFCs require a longer time to be deployed than simpler ones do.

All things considered, we worked on a heterogeneous and multi-domain Software Defined Network with Network Function Virtualization, in which we implemented an innovative way of obtaining Service Function Chaining by using intents, whose declaration no longer needs knowledge of technological details, but only of the technology-independent symbolic name of the points we wish to reach and connect to.

Appendix A

Additional notes, and code

A.1 SSH on Windows

Download **Portable PuTTY** and put the executable file in a folder, for example `C:\PortableApps`. Also, create a new folder where we will store scripts, for example `C:\Scripts`.

Add this folder to the environmental variable **PATH**. In Windows 10, this can be done by opening System, by right clicking on the Windows logo in the application bar or by pressing the combination of keys [Win] + [Pause], then open *Advanced Settings*, then *Environmental Variables*, then edit the variable Path and add to the already existing sequence the path of the folders *PortableApps* and *Scripts*.

Create a new Batch script (that is, a simple text file with extension "bat") called *ssh.bat*, put it in the folder *Scripts*, and edit it so that it contains the following code:

```
@echo off
REM ssh command
REM use like this: ssh [username@]host
if %1.==. goto no1
REM if 1 or more arguments are specified
set params=%1
:loop
shift
if [%1]==[] goto afterloop
set params=%params% %1
goto loop
```

```
:afterloop
start putty -ssh %params%
goto end
:no1
REM No arguments specified
echo No host specified, use this command as:
echo ssh [username@]host
echo To simply open PuTTY, type:
echo putty
:end
```

This way, in order to open a new SSH session from the Windows command line, we can type:

```
> ssh 192.168.56.2
```

and reach the login screen of the VM *ONOS_dev* (provided that the VM is running).

We can also use PuTTY's options syntax to provide additional login information, with:

```
> ssh developer@192.168.56.2 -pw sonoonos
```

and immediately log in to the VM as the user *developer* using the password *sonoonos*.

In case we need to to open a X11 session from the SSH session we are initiating, we also need to specify the -X option. The complete command for this purpose would then be:

```
> ssh -l developer -pw sonoonos -X 192.168.56.2
```

A.2 Python script that builds the Mininet cluster

```
'''
TOPOLOGY USED IN CLUSTER-IOT
'''

from mininet.net import Mininet
from mininet.node import Node
from mininet.node import Host
from mininet.link import TCLink
from mininet.link import Intf
```

```
10 from mininet.log import setLogLevel, info
11 from mininet.cli import CLI
12 from mininet.node import Controller
13 from mininet.node import RemoteController
14 from mininet.util import quietRun
15
16 from time import sleep
17
18 import os
19 import sys
20
21 '''
22 This function sends a single ARP request to a non-existing IP
     address
23 without even waiting for a response (which would not come anyway).
24 This is useful for the controller to notice all the Mininet hosts
25 connected to one of the switches it controls.
26 '''
27 def arpingone(net):
28   for host in net.hosts:
29     for iface in host.intfs.values():
30       host.cmdPrint('arping -c 1 -w 1 -I ' + '%s' % iface + '
         1.2.3.4')
31
32 '''
33 This function defines then starts a Mininet network, yielding the
     Miniet CLI at the end.
34 When the CLI is terminated with [Ctrl]+[D], the function performs
     a cleanup of the virtual equipment.
35 '''
36 def startNetwork():
37   # these two variables must contain the controller's IP address
       and TCP port, as strings
38   controller_ip_address = '192.168.200.102'
39   controller_tcp_port = '6633'
40
41   # process the user arguments given to the script (if any)
42   if len(sys.argv) < 2: # no user arguments given (the first
       argument is always present and it contains the name of the
       script)
43     # we will assume that user requests no debug, no additional
         hosts and default bridging
44     debug = False
45     addhosts = False
46     bridgephysical = True
```

```
47    elif len(sys.argv) == 2: # only one user argument given
48      # use the argument
49      if sys.argv[1] == '0':
50        debug = False
51      elif sys.argv[1] == '1':
52        debug = True
53      else:
54        sys.exit('Usage: cluster-iot.py debug=0|1')
55      # we will assume that user requests no additional hosts and
          default bridging
56      addhosts = False
57      bridgephysical = True
58    elif len(sys.argv) == 3: # two user arguments given
59      # use first argument
60      if sys.argv[1] == '0':
61        debug = False
62      elif sys.argv[1] == '1':
63        debug = True
64      else:
65        sys.exit('Usage: cluster-iot.py debug=0|1')
66      # use second argument
67      if sys.argv[2] == '0':
68        addhosts = False
69      elif sys.argv[2] == '1':
70        addhosts = True
71      else:
72        sys.exit('Usage: cluster-iot.py debug=0|1')
73      # we will assume that user requests and default bridging
74      bridgephysical = True
75    else: # there are at least 4 arguments (1st is name of script,
        others are user input)
76      # use first argument
77      if sys.argv[1] == '0':
78        debug = False
79      elif sys.argv[1] == '1':
80        debug = True
81      else:
82        sys.exit('Usage: cluster-iot.py debug=0|1')
83      # use second argument
84      if sys.argv[2] == '0':
85        addhosts = False
86      elif sys.argv[2] == '1':
87        addhosts = True
88      else:
89        sys.exit('Usage: cluster-iot.py debug=0|1')
```

```
90         #sys.exit('Usage: cluster-iot.py [debug=0|1 [hosts=0|1 [
               bridgephysical=0|1]]]')
91       # use third argument
92       if sys.argv[3] == '0':
93         bridgephysical = False
94       else:
95         bridgephysical = True
96 
97     # define the Mininet network without building it
98     net = Mininet(controller=RemoteController, link=TCLink, build=
         False)
99 
100    info('*** Create an empty network and add switches and nodes to
         it *** \n')
101 
102    # BUILDING CLUSTER-IOT
103    # adding switches
104    info('\n*** Creating switch sl *** \n')
105    sl = net.addSwitch('s1')
106    sl.cmd( 'ovs-vsctl del-br ' + sl.name )
107    sl.cmd( 'ovs-vsctl add-br ' + sl.name )
108    sl.cmd( 'ovs-vsctl set Bridge ' + sl.name + ' stp_enable=false'
         ) # Disabling STP
109    info('\n*** Creating switch su *** \n')
110    su = net.addSwitch('s2')
111    su.cmd( 'ovs-vsctl del-br ' + su.name )
112    su.cmd( 'ovs-vsctl add-br ' + su.name )
113    su.cmd( 'ovs-vsctl set Bridge ' + su.name + ' stp_enable=false'
         ) # Disabling STP
114    info('\n*** Creating switch sd *** \n')
115    sd = net.addSwitch('s3')
116    sd.cmd( 'ovs-vsctl del-br ' + sd.name )
117    sd.cmd( 'ovs-vsctl add-br ' + sd.name )
118    sd.cmd( 'ovs-vsctl set Bridge ' + sd.name + ' stp_enable=false'
         ) # Disabling STP
119    info('\n*** Creating switch sr *** \n')
120    sr = net.addSwitch('s4')
121    sr.cmd( 'ovs-vsctl del-br ' + sr.name )
122    sr.cmd( 'ovs-vsctl add-br ' + sr.name )
123    sr.cmd( 'ovs-vsctl set Bridge ' + sr.name + ' stp_enable=false'
         ) # Disabling STP
124    info('\n*** Creating switch sb *** \n')
125    sb = net.addSwitch('s5')
126    info('\n*** Creating switch sel *** \n')
127    sel = net.addSwitch('s6')
```

```
128     info('\n*** Creating switch ser *** \n')
129     ser = net.addSwitch('s7')
130     info('\n*** Creating chk *** \n')
131     chk = net.addHost('chk')
132     info('\n*** Creating vrl *** \n')
133     vrl = net.addHost('vrl')
134     info('\n*** Creating vrr *** \n')
135     vrr = net.addHost('vrr')
136     info('\n*** Creating bck *** \n')
137     bck = net.addHost('bck')
138 
139     if addhosts == True:
140       iotgw = net.addHost('iotgw')
141       iotsv = net.addHost('iotsv')
142 
143 
144     info('\n*** Creating links on Cluster-IOT *** \n')
145     # IMPORTANT: the order in which links are created IS RELEVANT!
146     # For switches, this is because the first link will be assigned
           to port 1
147     # the second to port 2 and so on.
148     # For hosts, it is because the first link will be assigned to
           hostname-eth0
149     # the second to hostname-eth1 and so on.
150     net.addLink(sel, vrl, bw=100)
151     net.addLink(vrl, sl, bw=100)
152     net.addLink(sl, su, bw=100)
153     net.addLink(sl, sd, bw=100)
154     net.addLink(su, chk, bw=100)
155     net.addLink(su, chk, bw=100)
156     net.addLink(su, sr, bw=100)
157     net.addLink(sd, sr, bw=100)
158     net.addLink(vrr, ser, bw=100)
159     net.addLink(sr, vrr, bw=100)
160     net.addLink(su, sb, bw=100)
161     net.addLink(sb, bck, bw=100)
162 
163     if addhosts == True:
164       net.addLink(sel, iotgw, bw=100)
165       net.addLink(ser, iotsv, bw=100)
166 
167     # Trying to assign MAC address to each node of the cluster
168     vrl.setMAC('00:00:00:00:00:0A', vrl.name + '-eth0')
169     vrl.setMAC('00:00:00:00:00:0B', vrl.name + '-eth1')
170     chk.setMAC('00:00:00:00:00:0C', chk.name + '-eth0')
```

```
171     chk.setMAC('00:00:00:00:00:0D', chk.name + '-eth1')
172     vrr.setMAC('00:00:00:00:00:0E', vrr.name + '-eth0')
173     vrr.setMAC('00:00:00:00:00:0F', vrr.name + '-eth1')
174     bck.setMAC('00:00:00:00:00:1A', bck.name + '-eth0')
175
176     if addhosts == True:
177       iotgw.setMAC('00:00:00:00:00:01', iotgw.name + '-eth0')
178       iotsv.setMAC('00:00:00:00:00:FF', iotsv.name + '-eth0')
179
180     # Disabling IPv6 on all hosts
181     for host in net.hosts:
182       print 'Going to disable IPv6 on ' + host.name
183       host.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
184       host.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
185       host.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
186
187     # add each port of the switches to the corresponding virtual
            switch
188     for switch in net.switches:
189       for iface in switch.intfs.values():
190         switch.cmd( 'ovs-vsctl add-port ' + switch.name + str(iface)
                )
191         print 'Added port ' + str(iface) + ' to virtual switch ' +
                switch.name + '\n'
192
193     nhosts = len(net.hosts)
194     nswitches = len(net.switches)
195     print 'Total number of hosts: ' + str(nhosts) + ' - Total number
            of switches: ' + str(nswitches) + '\n'
196
197     net.start()
198     info('\n*** Taking down default configuration ...\n')
199     info('\n*** ...and creating Linux bridge on chk, as well as
            configuring interfaces \n')
200
201     for host in net.hosts:
202       print 'Deleting ip address on ' + host.name + '-eth0 interface
              '
203       host.cmd('ip addr del ' + host.IP(host.name + '-eth0') + '/8
              dev ' + host.name + '-eth0')
204       print 'Deleting entry in IP routing table on ' + host.name
205       host.cmd('ip route del 10.0.0.0/8')
206       print 'Configuring new IP'
207       if host.name == 'chk': # VNF case
208         print 'Host with 2 L2 interfaces: ' + host.name
```

```
209         host.cmd('brctl addbr br-' + host.name)
210         host.cmd('brctl addif br-' + host.name + ' ' + host.name + '
                -eth0')
211         host.cmd('brctl addif br-' + host.name + ' ' + host.name + '
                -eth1')
212         host.cmd('ip addr add 192.168.107.3/24 dev br-' + host.name)
213         host.cmd('ip link set br-' + host.name + ' up')
214         print 'LB configured!'
215         host.cmd('sysctl -w net.ipv4.ip_forward=1')
216         print 'IP Forwarding enabled!'
217         host.cmd('ip link set dev ' + host.name + '-eth0 mtu 1400')
218         host.cmd('ip link set dev ' + host.name + '-eth1 mtu 1400')
219         print 'MTU configured on each interface of ' + host.name
220         host.cmd('tc qdisc del dev chk-eth1 root')
221         # introduce a random delay, uniformely distributed between
                25ms and 35ms
222         # and each random sample is 30% correlated with the previous
                one
223         host.cmd('tc qdisc add dev chk-eth1 root netem delay 30ms 5
                ms 30%')
224         print 'tc started on ' + host.name
225       elif host.name == 'vrl' or host.name == 'vrr':
226         if host.name == 'vrl':
227           host.setIP('172.16.1.254', 24, 'vrl-eth0')
228           host.setIP('192.168.107.1', 24, 'vrl-eth1')
229           print 'Interfaces of ' + host.name + ' have been
                  configured!'
230           host.cmd('ip link set dev vrl-eth0 mtu 1400')
231           host.cmd('ip link set dev vrl-eth1 mtu 1400')
232           print 'MTU configured on each interface of ' + host.name
233         elif host.name == 'vrr':
234           host.setIP('172.16.2.254', 24, 'vrr-eth0')
235           host.setIP('192.168.107.2', 24, 'vrr-eth1')
236           print 'Interfaces of ' + host.name + ' have been
                  configured!'
237           host.cmd('ip link set dev vrr-eth0 mtu 1400')
238           host.cmd('ip link set dev vrr-eth1 mtu 1400')
239           print 'MTU configured on each interface of ' + host.name
240         host.cmd('sysctl -w net.ipv4.ip_forward=1')
241         print 'IP Forwarding enabled on ' + host.name
242       elif host.name == 'bck': # BACKUP HOST case
243         host.setIP('192.168.107.4', 24, 'bck-eth0')
244         host.cmd('ip link set dev bck-eth0 mtu 1400')
245         print 'IP and MTU configured on ' + host.name
246
```

```
        print '\n'

    # Configure routing tables on the virtual routers
    for host in net.hosts:
        if host.name == 'vrl':
            host.cmd('route add -net 172.16.2.0 netmask 255.255.255.0 gw
                192.168.107.2')
            host.cmd('route add -net 192.168.10.0 netmask 255.255.255.0
                gw 172.16.1.253')
        elif host.name == 'vrr':
            host.cmd('route add -net 172.16.1.0 netmask 255.255.255.0 gw
                192.168.107.1')
            host.cmd('route add -net 192.168.10.0 netmask 255.255.255.0
                gw 192.168.107.1')

    # Configuring debug hosts
    if addhosts == True:
        for host in net.hosts:
            if host.name == 'iotgw':
                host.setIP('172.16.1.10', 24, 'iotgw-eth0')
                host.cmd('ip link set dev iotgw-eth0 mtu 1400')
                host.cmd('route add -net 192.168.107.0 netmask
                    255.255.255.0 gw 172.16.1.254')
                host.cmd('route add -net 172.16.2.0 netmask 255.255.255.0
                    gw 172.16.1.254')
            elif host.name == 'iotsv':
                host.setIP('172.16.2.10', 24, 'iotsv-eth0')
                host.cmd('ip link set dev iotsv-eth0 mtu 1400')
                host.cmd('route add -net 192.168.107.0 netmask
                    255.255.255.0 gw 172.16.2.254')
                host.cmd('route add -net 172.16.1.0 netmask 255.255.255.0
                    gw 172.16.2.254')

    for switch in net.switches:
        if switch.name == 's2':
            # ARP storm avoidance rules on the switch with 2 ports
                connected to the same host
            # ATTENTION: the field dl_type cannot be specified as a
                string, but it must be specified as in the following
                command:
            switch.cmd('ovs-ofctl add-flow ' + switch.name + ' in_port
                =2,dl_type=0x0806,dl_dst=FF:FF:FF:FF:FF:FF,actions=drop')
                 # ARP -> EthType 0x0806
            switch.cmd('ovs-ofctl add-flow ' + switch.name + ' in_port
                =3,dl_type=0x0806,dl_dst=FF:FF:FF:FF:FF:FF,actions=drop')
```

```
278      else:
279        #switch.cmd('ovs-ofctl add-flow ' + switch.name + ' priority
             =1,dl_type=0x0806,actions=normal')
280        # bridge the two "external" switches s6 and s7 to the
             network interfaces of the VM iot-mininet
281        if bridgephysical == True:
282          if switch.name == 's6':
283            print 'Bridging ' + switch.name + ' with interface eth1
                 of VM iot-mininet'
284            switch.cmd('ovs-vsctl add-port ' + switch.name + ' eth1'
                 )
285          elif switch.name == 's7':
286            print 'Bridging ' + switch.name + ' with interface eth2
                 of VM iot-mininet'
287            switch.cmd('ovs-vsctl add-port ' + switch.name + ' eth2'
                 )
288
289    # Set the controller for the switches
290    for switch in net.switches:
291      switch.cmd('ovs-vsctl set-controller ' +  switch.name + ' tcp:
           ' + controller_ip_address + ':' + controller_tcp_port) #For
           ONOS connection
292      info('\n*** Waiting for switch to connect to controller')
293      while 'is_connected' not in quietRun('ovs-vsctl show'):
294        sleep(1)
295        info('.')
296      info('\n')
297
298    if debug == True:
299      print '**** DEBUG ACTIVE ****'
300      info('\n*** INFO ABOUT HOSTS \n')
301      for host in net.hosts:
302        host.cmdPrint('ifconfig')
303        host.cmdPrint('route -n')
304        host.cmdPrint('cat /proc/sys/net/ipv4/ip_forward')
305        if host.name == 'chk':
306          host.cmdPrint('tc -s qdisc')
307
308      info('\n*** INFO ABOUT SWITCHES \n')
309      count = 1
310      for switch in net.switches:
311        switch.cmdPrint('ovs-ofctl dump-flows ' + switch.name)
312        switch.cmdPrint('ovs-ofctl show ' + switch.name)
313        if count == nswitches:
314          print ('List of switches: \n')
```

```
315             switch.cmdPrint('ovs-vsctl show')
316           count = count + 1
317
318         arpingone(net)
319
320     info('... running CLI ***\n')
321     CLI(net)
322     info('\n')
323     info('... stopping Network and cleaning up ***\n')
324     net.stop()
325     os.system('sudo mn -c')
326
327 #Main
328 if __name__ == '__main__':
329     setLogLevel('info')
330     startNetwork()
```

Listing A.1: Python script that builds the Mininet cluster

A.3 PHP script implementing the POST handler

```
1  <?php
2  function isJson($string) {
3      json_decode($string);
4      return (json_last_error() == JSON_ERROR_NONE);
5  }
6  // Make sure that it is a POST request.
7  if(strcasecmp($_SERVER['REQUEST_METHOD'], 'POST') != 0) {
8      echo "Request method must be POST!\n";
9      throw new Exception('Request method must be POST!');
10 }
11 // Make sure that the content type of the POST request has been
     set to application/json
12 $contentType = isset($_SERVER["CONTENT_TYPE"]) ? trim($_SERVER["
     CONTENT_TYPE"]) : '';
13 if(strcasecmp($contentType, 'application/json') != 0) {
14     echo "Content type must be: application/json\n";
15     throw new Exception('Content type must be: application/json');
16 }
17 // Receive the RAW post data.
18 $content = trim(file_get_contents("php://input"));
```

```
19 // Check if content is valid JSON
20 if(!isJson($content)) {
21     echo "Received content contained invalid JSON!\n";
22     echo '|' . $content . '|' ;
23     throw new Exception('Received content contained invalid JSON!'
       );
24 } else {
25     echo "Received valid JSON. Adding its content to database.\n";
26 }
27 // Remove newlines and carriage returns
28 preg_replace( "/\r|\n/", "", $content );
29 // Define the name of the database file
30 $db_file = 'database.json';
31 // Read the current database
32 $fh = fopen($db_file, 'r+') or die("Can't open file " . $db_file);
33 $stat = fstat($fh);
34 ftruncate($fh, $stat['size']-2);
35 fclose($fh);
36 // if it is only initialized but still empty, just append the new
     data
37 if($stat['size'] == strlen('{"data":[]}')) {
38     $data = $content . ']}';
39 }
40 // if it already contains some data, append a comma then the new
     data
41 else {
42 $data = ',' . $content . ']}';
43 }
44 // write the new database to the same file
45 file_put_contents($db_file,$data, FILE_APPEND | LOCK_EX);
46 ?>
```

Listing A.2: PHP script implementing the POST handler

Bibliography

[1] F. Callegati, W. Cerroni, and C. Contoli. “Virtual Network Performance in OpenStack platform for Network Function Virtualization”. In: *Journal of Electrical and Computer Engineering* (2016).

[2] W. Cerroni, C. Buratti, S. Cerboni, G. Davoli, C. Contoli, F. Foresta, F. Callegati, and R. Verdone. “Intent-Based Management and Orchestration of Heterogeneous OpenFlow/IoT SDN Domains”. In: *NetSoft17*. 2017.

[3] F. Foresta. “Composizione Dinamica di Funzioni di Rete Virtuali in Ambiente Cloud”. 2015.

[4] *NFV, Cloud and SDN*. URL: `https://networkbuilders.intel.com/docs/Genband_nfv_whitepaper.pdf`.

[5] F. Callegati, W. Cerroni, C. Contoli, R. Cardone, M. Nocentini, and A. Manzalini. “SDN for dynamic NFV deployment”. In: *IEEE Communications Magazine* 54.10 (2016), pp. 89–95. ISSN: 0163-6804. DOI: `10.1109/MCOM.2016.7588275`.

[6] *Intent NBI - Definition and Principles*. Tech. Rec. The Open Networking Foundation (ONF), 2016. URL: `https://www.opennetworking.org/sdn-resources/technical-library`.

[7] F. Hu, Q. Hao, and K. Bao. “A Survey on Software-Defined Network and OpenFlow: From Concept to Implementation”. In: *IEEE Communications Surveys Tutorials* 16.4 (2014), pp. 2181–2206. ISSN: 1553-877X. DOI: `10.1109/COMST.2014.2326417`.

[8] Open Networking Foundation. “Software-Defined Networking: The New Norm for Networks”. In: (2012).

[9] *SDN Architecture Overview*. Tech. Rec. The Open Networking Foundation (ONF), 2013. URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf.

[10] F. Callegati, W. Cerroni, C. Contoli, and F. Foresta. "Performance of Intent-based Virtualized Network Infrastructure Management". 2017.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. "OpenFlow: Enabling Innovation in Campus Networks". In: *White paper*. 2008.

[12] S. Seetharaman. *OpenFlow/SDN tutorial*. OFC/NFOEC. 2012. URL: http://www.slideshare.net/openflow/openflow-tutorial.

[13] *OpenFlow*. URL: https://www.opennetworking.org/sdn-resources/openflow.

[14] Q. Hassan. "Demistifying Cloud Computing". In: *The Journal of Defense Software Engineering* (2011).

[15] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. "Network Function Virtualization: State-of-the-Art and Research Challenges". In: *IEEE Communications Surveys and Tutorials, Vol 18, No. 1* (2016).

[16] *Network Functions Virtualisation (NFV); Architectural Framework*. The European Telecommunications Standards Institute (ETSI). 2013. URL: http://www.etsi.org/technologies-clusters/technologies/nfv.

[17] *Network Functions Virtualisation (NFV); Management and Orchestration*. The European Telecommunications Standards Institute (ETSI), 2014. URL: http://www.etsi.org/technologies-clusters/technologies/nfv.

[18] *ONOS: Open Network Operating System*. URL: http://onosproject.org.

[19] *ONOS Wiki*. URL: https://wiki.onosproject.org/.

[20] *Mininet: An Instant Virtual Network on your Laptop*. URL: http://mininet.org.

[21] *OpenStack: Open Source Cloud Computing Software*. URL: https://www.openstack.org.

[22] P. Quinn and U. Elzur. "Network Service Header". In: *IETF Internet-Draft draft-ietf-sfc-nsh-10* (2016).

[23] S. Cerboni. *Software Defined Networking for the Internet of Things*. 2016.

[24] *Oracle VirtualBox*. URL: `https://www.virtualbox.org/`.

[25] *Ubuntu 14.04*. URL: `http://releases.ubuntu.com/14.04/`.

[26] *IntelliJ IDEA*. URL: `https://wiki.onosproject.org/display/ONOS/IDE+Setup`.

[27] *Xming*. URL: `http://www.straightrunning.com/XmingNotes/`.

[28] *Virtual Machine Manager*. URL: `https://virt-manager.org/`.

[29] *Mininet Python API Documentation*. URL: `http://mininet.org/api/index.html`.

[30] *Ubuntu 16.04*. URL: `http://releases.ubuntu.com/16.04/`.

Acknowledgments

This Master Thesis contains the results of my work in the last few months, but it can only serve as a symbol of the true, most important achievement of the last period for me: I got the chance to meet, get to know, and work with some of the most kind, humane, and professionally skilled people I have ever met.

First and foremost, I would like to thank Prof. Walter Cerroni, for his invaluable representation of the values I have mentioned above. He gave me the opportunity to spend my Master Thesis period with him and the research group he works into, and it has been an opportunity I will always be glad I have had.

Without loss of importance, I also wish to thank Dr. Chiara Contoli, on whose work I have based mine, and who has always been available for confrontation and help, starting when she still was thousands of kilometers away for her Ph.D period abroad.

Third, but equally important (you know my affection for number 3, so you can understand that this is a very honorable position for me to put you in), I would like to thank my captain Francesco Foresta, who literally welcomed me with a hug on my first day in Net2Lab, and kickstarted my activities, by sharing with me his vast practical knowledge on how to configure stuff, how to run it, where to eat the best kebab and where to buy the best Chinese trinkets online. Joking aside, there is hardly anything I can say about him that would encompass the entirety of what he shared with me throughout these past few months. I will try doing it with a hug.

I wish to thank Prof. Carla Raffaelli, as well as Prof. Franco Callegati, both of whom have always made me feel welcome in Net2Lab and more generally in the research group they share.

I also wish to thank Prof. Chiara Buratti, as well as my fellow colleague and friend Simone Cerboni, for their tireless work on the IoT part, and their availability to a fruitful collaboration on the activities presented in this thesis.

I would like to dearly thank Federico, Andrea and Bahare, whom I shared so many moments of fun with, and played a key role in my decision of remaining to work there after my graduation. Thank you guys, it has been my honor and privilege to work alongside all of you.

Special thanks go to Silvia, who started from being a great university colleague and became a dear every-day friend. Friendship can rise above neighborhood battles, and we have the power to prove it. There is much more to say about her, too, but I think I will resolve in giving her a hug as well.

I would also like to thank il mio caro collega Shaham, the man with the coolest mustache ever, who was the first one to show me that Iranian people are much more similar to Italians than we think, and gave me crash courses in the Persian language. I am very glad that the two of us graduate on the same day. Khasteh nabashi, dooste man.

My thanks also go to my fellow course mates and friends Giammarco ("Roccia", "Bomber", "Troppo Grosso") and Matteo ("Maestro", "Mitico"), who provided these last two years with invaluable rumagnòl spontaneity, thanks to which I now no longer hang out with people, but I vado oltre and faccio bagarre with them.

I also want thank Bob, my first real English teacher, who honored me with his friendship long ago, and offered his opinion on this thesis to me, which has been my first major work written in English. A big hug to Gabrielle, too, for always being so smiling. Cheers 'n' beers!

Best for last, I wish to thank my family, who has never left me alone and has always cheered for me.

Mamma Terry, Babbo Gianni, Nonno Marcello, Zia Lalla: grazie, vi voglio tantissimo bene.

E Teresa, ti amo.

Casalecchio di Reno, Bologna - 8 March 2017