Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2015

Rui Nuno Carrulo Brás

Gestão integrada de cloud usando redes definidas por software

Integrated Cloud management using software defined networks

Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2015

Rui Nuno Carrulo Brás

Gestão integrada de cloud usando redes definidas por software

Integrated Cloud management using software defined networks

"*The only easy day was yesterday.*"

— United States Navy Seals Motto

Universidade de Aveiro
2015

Departamento de Eletrónica,
Telecomunicações e Informática

**Rui Nuno
Carrulo Brás**

Gestão integrada de cloud usando redes definidas por software

Integrated Cloud management using software defined networks

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Daniel Corujo, Investigador Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Pedro Gonçalves, Professor Adjunto da Escola Superior de Tecnologia e Gestão de Águeda.

Texto Apoio financeiro do POCTI no âmbito do III Quadro Comunitário de Apoio.

Texto Apoio financeiro da FCT e do FSE no âmbito do III Quadro Comunitário de Apoio.

Dedico este trabalho a toda a minha familia, especialmente aos meus pais, Nuno e Beatriz, e também à Filipa por estarem sempre presentes e pelo seu apoio incondicional.

agradecimentos / **acknowledgements**

Gostaria de agradecer aos meus orientadores, Daniel Corujo e professor Pedro Gonçalves, pela sua paciência, apoio e conhecimento transmitido.
Gostaria também de agradecer à minha família, namorada e amigos por todo o apoio e incentivo que me deram durante esta importante etapa da minha vida.

Palavras Chave

Redes Definidas por software (SDN), Gestão de redes, cloud, Openflow, Qualidade de Serviço, Balanceamento de carga

Resumo

Nestes últimos anos tem-se vindo a registar um grande aumento no número de dispositivos ligados à Internet a partir de praticamente qualquer lugar. Assim, para manter-se par com a procura, os *Internet Providers* têm de melhorar sua rede através da aquisição de mais dispositivos de rede, o que por sua vez aumenta o custo da infra-estrutura. Além disso, isso dificulta ainda mais a gestão das redes devido à sua abordagem centrada no *hardware* que requer a configuração manual de cada um dos dispositivos de rede. Uma mudança no paradigma atual tem sido estudado e evoluindo ao longo de décadas, mostrando, nos últimos anos, que ele pode realmente ser a futura direção para a rede.

SDN é um novo paradigma de rede que facilita a gestão da mesma ao permitir que esta se torne programável. Isso é feito principalmente através da separação dos planos de dados e controlo e também através da criação de abstrações que tornam a rede mais flexível e escalável. O que por sua vez, permite a inovação e gestão mais simples de redes de *data center* em ambientes de *cloud*.

O objetivo principal desta dissertação é implementar e avaliar uma solução que facilita a gestão, em ambientes de data center, usando este novo paradigma, SDN. Ela fornece configuração transparente e automática da rede subjacente, a fim de permitir a comunicação entre os nós com requisitos de largura de banda especificados. Além disso, também realiza a monitorização de balanceamento de carga para optimizar o uso de tráfego na rede.

Esta dissertação apresenta a solução desenvolvida que utiliza um controlador de SDN e OpenvSwitch. A solução faz uso de ambos os protocolos *OpenFlow* e *OVSDB*, bem como os módulos do controlador OpenDaylight. A interacção com o controlador é realizada através do uso do *REST APIs* fornecidas pelo controlador acima mencionado.

Durante a fase de avaliação, vários cenários de teste foram executados a fim de avaliar a correção e o desempenho do sistema que interage com a rede. A aplicação comportou-se razoavelmente bem por ser capaz de aplicar a largura de banda especificada on-demand (QoS) de uma maneira simples e sem falhas. Do mesmo modo, o balanceamento de carga foi também aplicado com sucesso, sem perder a comunicação entre os nós. Tudo isto foi realizado com *overhead* moderado (em termos de tempo de instalação e da quantidade de dados enviados para gerir a rede). Em conclusão, a solução mostra-se promissora pela sua facilidade na gestão de redes em *data centers*.

Keywords Software-Defined Networks, Network management, cloud, Openflow, Quality of Service, Load balance.

Abstract

These past few years we have been experiencing a large increase in the number of devices connected to the Internet from almost anywhere. Thus, to keep up with the demand, Service Providers have to upgrade their network by acquiring more network devices, which greatly increases the infrastructure's cost. In addition, this hampers even more the management of the networks due to their hardware centric approach which requires manual configuration of each of the network devices. A change in the current paradigm has been studied and evolving for decades, showing, in the last years, that it can actually be the future direction for networking.

SDN is a modern networking paradigm that eases network management by enable the network to be programmable. This is done mainly through the separation of data and control planes and also through the creation of abstractions that make the network more flexible and scalable. Which, in turn enables innovation and simpler management to data center networks in cloud environments.

The main objective of this dissertation is to implement and evaluate a solution that eases the management in data center environments using this new paradigm, SDN. It provides seamless and automatic configuration of the underlying network in order to allow communication between nodes with on-demand bandwidth requirements. In addition, it also performs load balance monitoring to optimize the traffic usage on the network.

This dissertation presents the developed solution which uses a SDN controller and OpenvSwitch. The solution makes use of both OpenFlow and OVSDB protocols as well as OpenDaylight controller's modules. The interaction with the controller is performed through the use of the REST APIs provided by the above mentioned controller.

During the evaluation stage, several test scenarios were executed in order to evaluate the correctness and performance of the system interacting with the network. The application behaved reasonably well by being able to apply the specified bandwidth on-demand (QoS) in a simple manner without flaws. Similarly, the load balance was also successfully applied without loosing communication between the hosts. All this was accomplished with moderate overhead (in terms of time of installation and quantity of data sent to manage the network). In conclusion, the solution shows to be promising for the ease of management in data center networks.

Contents

List of Figures

List of Tables

Glossary

AD-SAL	API-Driven SAL
API	Application Programming Interface
BGP	Border Gateway protocol
CAPEX	Capital Expenditure
CBQ	Class-based Queueing
CRUD	Create, Read, Update, Delete
FIFO	First in First out
HTB	Hierarchical Token Bucket
IETF	Internet Engineering Task Force
IP	Internet Protocol
JSON	JavaScript Object Notation
MD-SAL	Model-Driven SAL
MLU	Maximum Link Usage
MSS	Maximum Segment Size
NB	Northbound
NBI	Northbound API Interfaces
NBI-WG	Northbound Interface Working Group
NETCONF	Network configuration protocol
NIB	Network Information Base
NOS	Network Operating System
ODL	OpenDaylight
ONF	Open Networking Foundation
OPEX	Operational Expenditure
OSGi	Open Services Gateway iniciative
OVS	Open vSwitch
OVSDB	Open vSwitch Database Management Protocol
QoS	Quality of Service
RED	Random Early Detection
REST	Representational State Transfer
RTT	Round-trip Time
SAL	Service Abstraction Layer
SB	Southbound
SDN	Software Defined-Networks
SFQ	Stochastic Fair Queueing
SLA	Service Level of Agreement
SNMP	Simple Network Management Protocol
SOA	Service-oriented Architecture
TCP	Transmission control protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UUID	Universally Unique Identifier
VLAN	Virtual Local Area Network
VM	Virtual Machine
XML	Extensible Markup Language
YANG	Yet Another Next Generation

CHAPTER 1

Introduction

1.1 MOTIVATION

Networks are stuck in the past, routing algorithms change very slowly, network management is extremely primitive. The infrastructure so far works, but only to our ability to master complexity, which is both a blessing as well as a curse. Future networking [1], is about extracting simplicity, which implies separating problems and define abstractions, instead of mastering complexity in order to enable progress on networking area.

In traditional Internet Protocol (IP) networks, depicted in Figure 1.1, the control and data planes are tightly coupled, embedded in the same networking devices, and the whole structure is highly decentralized. Despite this approach has been quite effective in terms of network performance, the outcome is a very complex and relatively static architecture, as has been often reported in the networking literature (e.g., [2], [3], [4] and [5]). It is also the fundamental reason why traditional networks are rigid, and complex to manage and control. These two characteristics are largely responsible for a vertically-integrated industry where innovation is difficult. For example, a misconfigured device may cause undesired and unexpected network behaviour which may compromise the correct operation of the network for some time. To foreharm it, some vendors offer proprietary solutions on specialized hardware, operating systems, and control programs to support the network management. However the cost of building and maintaining a networking infrastructure is significant, with long return investments which prevents innovation and new features to be provided (load balancing, energy efficiency, traffic engineering).

SDN's paradigm, depicted in Figure 1.1, simplifies the network management, allowing network innovations, evolution and is directly programmable. Computer networks can be divided in three planes of functionality: the management, control and data planes. Briefly, network policy is defined in the management plane, the control plane enforces the policy, and the data plane executes it by forwarding data accordingly. This brings important advantages, especially at the management level of networks such as:

- Provides faster, easier, more reliable and automated network operation and configuration.
- A simpler and less error-prone way to modify network policies using high level languages compared to low level device specific configurations.

- Centralization simplifies the development of sophisticated features, services and applications.
- Interoperation with non-SDN networks.

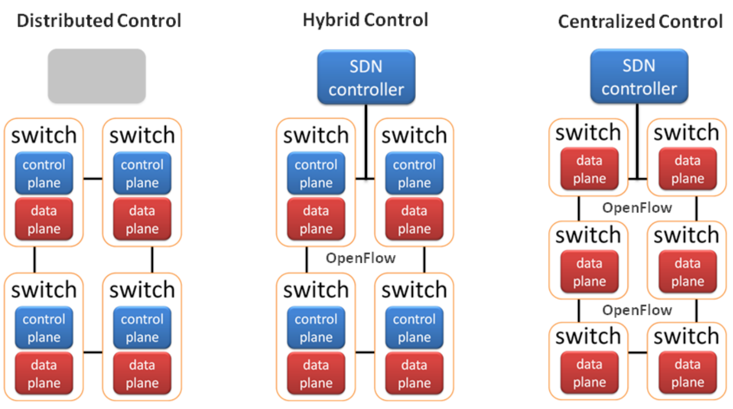


Figure 1.1: Illustration of Network Paradigms: Traditional Network (left), SDN (center and right). Source: `http://www.themetisfiles.com/2012/10/the-future-of-the-network-is-software-defined/`

Thus, enterprises and carriers gain unprecedented programmability, automation, and network control, enabling them to build innovative, highly scalable, flexible networks that readily can adapt to the change of their business needs. In addition, through the use of OpenFlow, the first standard interface designed specifically for SDN, enterprises and carriers gain vendor-independent control over the entire network from a single logical point.

Ultimately SDN will enable in conjunction with network virtualization that such enterprises to reduce costs by optimizing their data center networks as well as reduce its complexity due to the ease of management this paradigm introduces. They will then be able to focus on providing better and more efficient services, which will in turn provide them higher revenues. The ease of management and the introduction of new services is facilitated by the development of new applications that maintain a global view of the network and can provide a wide variety of functionality, either in terms of network management Quality of Service (QoS) on-demand, Load balance, firewall, fault-tolerance mechanisms and many others).

1.2 OBJECTIVES

The main objective consists on providing a generic network application that could simplify the management of a network using this new network paradigm called SDN.

The application is able to reconfigure datacenter networks on demand, applying QoS according to reservation parameters. The reservations can be created either by a network administrator or, more importantly, a cloud orchestration domain like OpenStack [1], OpenNebula [2] or CloudStack [3]. Upon

[1]https://www.openstack.org/
[2]http://opennebula.org/
[3]https://cloudstack.apache.org/

reservation the application would instantly react and automatically install the necessary rules on the network switches.

In addiction it makes use of an internal monitoring tool which provides information for the load balancing optimization mechanism to act automatically. The load balancing mechanism prevents the existence of links with high load due to some burst traffic the network might have by ofloading the flow through other path less loaded.

The key advantage is the fact of being a generic application capable of being used by any cloud orchestration domain which in conjunction with the standardization that Open Networking Foundation (ONF) pretends to achieve would make the application to work with any Network Operating System (NOS).

This way, the application aims to provide an abstraction layer that can be used for an orchestration domain to provide the network QoS in a simply manner through the usage of high level commands.

1.3 CONTRIBUTIONS

The work presented in this dissertation is a management application that provides on demand-bandwidth and load balance optimization with a certain level of automation. The source code of the implemented application is available online in [4].

Community support to other developers through OpenDaylight's ask [5], which is a forum where developers aid each other. By participating on it, a reputation (karma) of 235 was achieved, which corresponds to approximately 15 correct answers to other community members working with Opendaylight.

1.4 STRUCTURE

This document is divided in six chapters. The current chapter discusses the motivation for the work of this dissertation, on providing QoS in SDN-enabled networks using OpenFlow as well as information about the contributions made to the community. The remaining chapter provide information as follows:

- Chapter 2 (State of the Art) - Presents the current work and efforts related to SDN technologies. A description about the necessity of such paradigm, the benefits in comparison with traditional networking and the challenges it faces. It also provides a description of the key components that form SDN architecture such as the controllers, OpenFlow protocol and forwarding devices. Finally, it describes some existing applications related to QoS and management in SDN-enabled networks.
- Chapter 3 (Solution Description) - Presents the selected components in a SDN architecture, describing briefly the key features of the architecture of each component that influenced the selection decision. It also describes the assembling details of the different architectures. Finally, presents the description of the system developed.

[4]https://github.com/nunocarrulo/QLAMES.git
[5]https://ask.opendaylight.org/questions/

- Chapter 4 (Evaluation) - Illustrates and describes different test scenarios performed in order to evaluate several aspects of the solution.
- Chapter 5 (Conclusion) - Contains an overall apreciation of the solution developed, the main results obtained and an indication of a direction for future work.

CHAPTER 2

State of the Art

In traditional networks there are three basic components of a telecommunications architecture. The data, control and management planes, as depicted in Figure 2.1:

The *management plane* contains the protocol implementations to be used by other planes as well as the custom policies for decision-making in the control plane.

The *Control plane* is responsible for the decisions about where the traffic and for the configuration and management of the node. The Control plane is then considered as the network inteligence or the networks' brain. The resulting paths are then pushed down to the *Data Plane*.

The *Data plane*, also known as the Forwarding Plane, is the part of a network that carries user traffic. forwards the traffic along the paths specified by the control plane.

Unfortunately, this legacy network infrastructure – primarily a mixed bag of vendor, platform and protocol solutions, each initially deployed in response to short-term challenges – makes reaching the ultimate goal of an integrated, orchestrated and automated network ecosystem a long and difficult process for many organizations.

The distributed control and transport network protocols running inside the routers and switches are the key technologies that allow information, in the form of digital packets, to travel around the world. Despite their widespread adoption, traditional IP networks are complex and hard to manage [6].

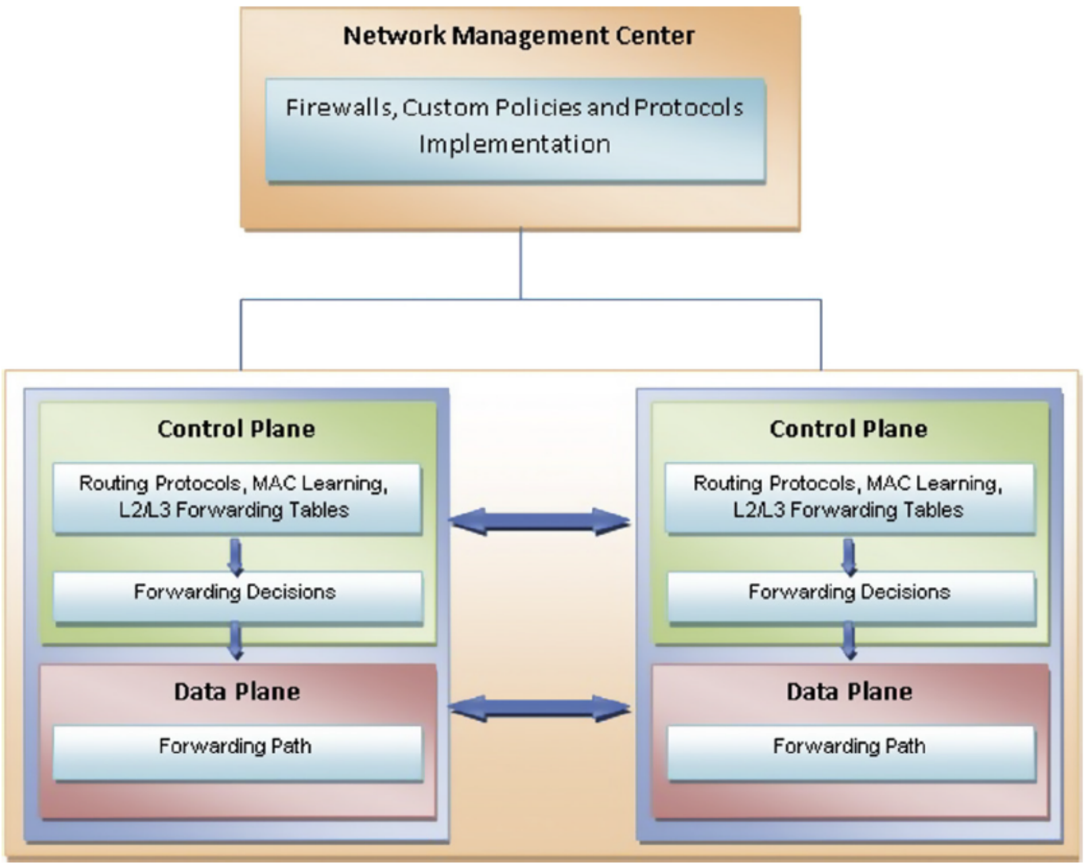

Network Management Center
Firewalls, Custom Policies and Protocols Implementation
Control Plane
Routing Protocols, MAC Learning, L2/L3 Forwarding Tables
Forwarding Decisions
Data Plane
Forwarding Path
Control Plane
Routing Protocols, MAC Learning, L2/L3 Forwarding Tables
Forwarding Decisions
Data Plane
Forwarding Path

Figure 2.1: Legacy networks infrastructure. Source : [7]

Experience has shown that the high complexity underlying the design and configuration of enterprise networks generally leads to significant manual intervention when managing networks. In turn, manual intervention leads into a more prone to failures and configuration errors environment, which difficults the upgrade (for example, the addition of new network devices) and management (for example, the adjustment of a network policy) of the networks even more [3]. Due to this fact, automatic reconfiguration and response mechanisms are virtually non-existent in current IP networks. Enforcing the required policies in such a dynamic environment is therefore highly challenging.

To make it even more complicated, current networks are also vertically integrated. The control and data planes are embedded in the same network node, which means that the data paths and the decision-making processes for switching or routing are collocated on the same device. This reduces the flexibility and hinders innovation and evolution of the networking infrastructure [3], [6].

The management of such complex environments using traditional networks is time-consuming and expensive, especially in the case of Virtual Machine (VM) migration and network configuration. In addition, enterprises are continuously deploying new application and upgrading others for easing the management process on their networks.

Today's Internet applications require the underlying networks to be fast, carry large amounts of traffic, and to deploy a number of distinct, dynamic applications and services. Network operators thus require new network solutions to tackle the increasing demands of its clients. In response, most modern-day vendors use control-plane software to optimize data flow to achieve high performance and competitive advantage [8]. Despite of their effort to enable data flow efficiency accross the network, the rigid structure of legacy networks hampers programmability to meet the variety of client requirements, sometimes forcing vendors into deploying complex and fragile programmable management systems. Due to this, a huge number of network administrators teams are employed to deal with that complexity and manually make thousands of changes to the network components [8], [9]. As the number of network applications and devices connected to the Internet grows, so does the demand for services and network usage. Although growth drivers such as video traffic, big data, and mobile usage increase their revenues, they pose significant challenges for network operators, being the most important challenge the management of the network [10]. According to an Enterprise Strategy Group (ESG) study [11] the deployment of new applications or the upgrade of existing ones is essential for powering up IT companies business, takes longer that it was expected. For new applications, around two thirds reported that it would take at least one month to be deployed, while for the upgrade of the applications only 50% reported that it would also take at least one month. The research also points out that in many environments, the ability to deploy applications quickly is hampered not only by application issues (e.g compatibility or performance problems), but also by lack of flexibility in legacy networks that require too much time to provision services and can not scale quickly to meet demand. Thus they state that, the creation of a highly flexible, scalable and agile infrastructure is of major importance and will enable organizations to respond more efficiently to the changing of the market dynamics. Nowadays many enterprises are well down the path to virtualizing their environments and are even creating private clouds. According to Interoute [12], "a private cloud is a model of cloud computing that involves a distinct and secure cloud based environment in which only the specified client can operate". Basically it provides a pool of resources such as storage, computing power or networking as a service in its virtualized environment that can only be accessed by a single organisation, providing that organisation with greater control and privacy.

So, in order to tackle all these challenges and thus improve the network infrastructure, organizations need to look for network technologies that will enable much higher levels of flexibility, efficiency and scalability, without the need to hire more network staff or to deploy more network hardware. This way, new approaches must be taken into action in order to enable flexibility, efficiency and scalability.

Taking into account all these challenges a new network paradigm has emerged, called Software-defined Networks (SDN) being a network paradigm that can provide a solution (directly and indirectly) to several of the challenges described above. In brief, SDN decouples the data from the control plane enabling network programmability, which in turn eases the management process of the network.

The idea of a programmable network initially took shape as active networking [13] from the mid-1990s to the early 2000s. This concept shares many of the same visions as SDN, however it lacked both a clear use case and an incremental deployment path. After the era of active-networking research projects, the tendency changed from vision to pragmatism, in the form of separating the data and control planes to make the network easier to manage.

SDN has a central software program, the controller or NOS, which has a global view of the network and dictates the overall behaviour of the network. This, in conjunction with open interfaces between the devices in the control plane, those in the data plane and the network applications, in the management plane, brings a wide variety of benefits. SDN enables : the ability to make the network programmable and thus supporting the dynamic nature of future network functions; the simplification and automation of the network management; the innovation of the network; lower the cost of managing enterprise and carrier networks; and many others. These, in turn, increase the flexibility, interoperability or scalability of the networks.

Obviously SDN also faces several challenges, mentioned later in Subsection 2.1.3 that could hinder its performance and implementation in cloud networks. In order to speed up innovation in the networking area using SDN, a new organization called ONF [1] was created.

ONF is a non-profit organization, funded by prominent companies such as Deutsche Telekom, Facebook, Google, Microsoft, Verizon, and Yahoo. Currently, it is also backed up by the major companies in the networking and virtualization fields, such as Juniper, Brocade, Cisco, NEC, VMware and others. This group is dedicated into improving networking by the development and standardization of SDN, which includes the OpenFlow protocol, described in detail in Section 2.2. In order to accomplish that, ONF organized "Specification Areas" or "Working Groups". According to [14], the "Specification Area" is responsible for publishing all ONF's technical specifications. Examples of "Specification Areas" are *Extensibility* [15], *Forwarding Abstractions* [16] or *Testing and Interoperability* [17]. All these areas have its specific objective, for example, the first is responsible for developing OpenFlow extensions, the second describes a framework in which controllers request switch forwarding behavior at runtime and the last ensures standardized development through testing and certification.

Beyond these, more groups are being formed, each one with its own purpose, but bearing in mind the goals of the ONF organization. One other important group founded is Northbound Interface Working Group (NBI-WG) [18] with the purpose of increasing portability of software designed to interact with SDN by providing extensible, stable Northbound API Interfaces (NBI) Application Programming Interface (API)'s to controller and network applications. This working group has the objective of developing data models for NBIs, prototyping and after obtaining end-user feedback on selected examples with real code, being able to standardize various SDN controller NBIs [19]. Basically they aim at evaluate existing Northbound (NB) interfaces and standardize a NBI which enables any application to interact with any controller through the same NB APIs.

A detailed description of this new paradigm, called SDN, its benefits and challenges will be discussed in Section 2.1.

2.1 SDN

2.1.1 ARCHITECTURE OVERVIEW

SDN is an emerging networking paradigm that promises to change the state of affairs of current networks, by breaking vertical integration. This is accomplished by separating the network's control logic from the underlying forwarding devices (routers and switches), promoting (logical) centralization of network control, and introducing the ability to program the network. In addition, by breaking the network control problem into modular pieces, SDN makes it easier to create and introduce new abstractions in networking, simplifying network management and facilitating network evolution.

Figure 2.2 compares both paradigms (the traditional and SDN paradigm) extolling the importance and advantages brought by this separation. The Figure in question suggests that through that separation there is no longer any need for deploying dedicated hardware in the network. We can simply perform those functions by network applications, sparing for example the costs of buying and adding new hardware into the network.

[1] https://www.opennetworking.org/

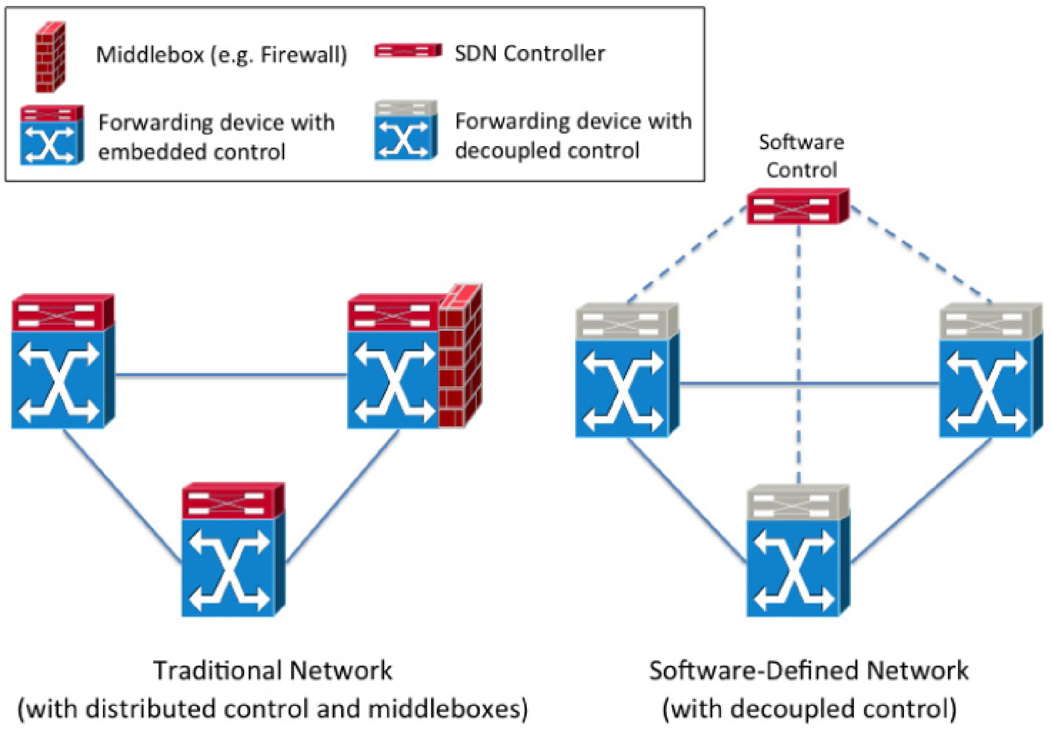

Middlebox (e.g. Firewall)
SDN Controller
Forwarding device with embedded control
Forwarding device with decoupled control
Software Control
Traditional Network (with distributed control and middleboxes)
Software-Defined Network (with decoupled control)

Figure 2.2: Traditional Network Infrastructure and SDN Architecture focusing the decouple of control and data planes. Source : [20]

As it was mentioned in Chapter 1 and according to the SDN concept described in [1] the key point of future networking is all about extracting simplicity, instead of mastering complexity. This enables the progress in the networking area and is, in part, achievable by the definition of abstractions. The SDN concept relies on three fundamental abstractions, present in Figure 2.3:

- Forwarding abstraction - Any forwarding behavior desired by the network application (the control program) should be allowed while hiding details of the underlying hardware. OpenFlow, discussed later in Section 2.2, is one realization of such abstraction.
- Distribution abstraction - Should have shield mechanisms due to the vagaries of distributed state in order to provide a centralized global network view to network applications. It is also responsible for installing the control commands on the forwarding devices.
- Specification abstraction - A network application should be allowed to express the desired network behavior without being responsible for implementing that behavior itself.

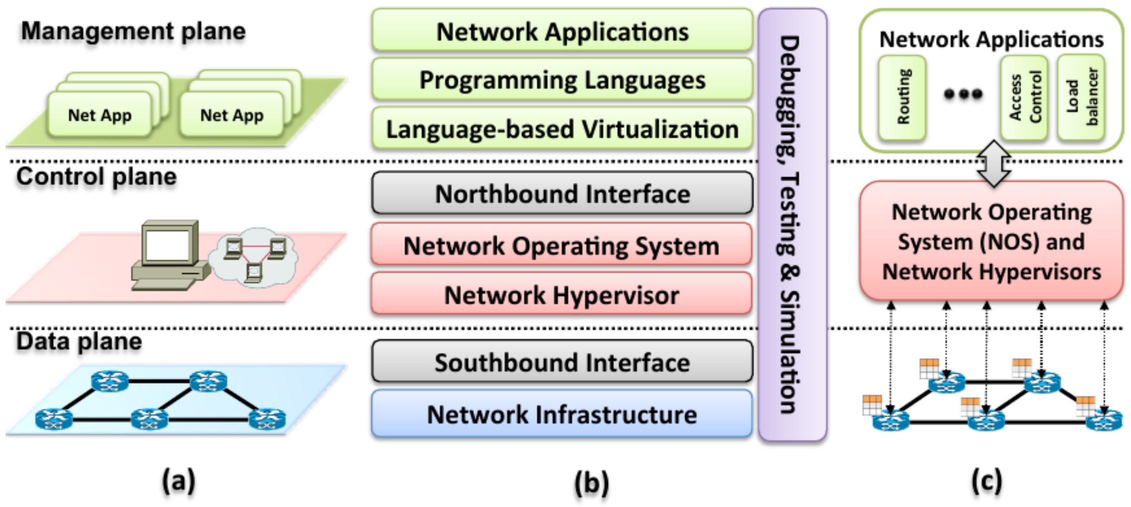


Figure 2.3: SDN architecture divided by (a) planes, (b) layers and (c) system design architecture. Source : [6]

The realization of such abstractions are responsible for the above mentioned advantages that SDN would bring to the networking area. According to [21], ”In the SDN architecture, the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications.”

The key architectural features underlying SDN, depicted in Figure 2.3, are in line with the abstractions above mentioned. They are the following :

- The control and data planes are decoupled. Data plane becomes simpler with network devices just being responsible for forwarding the traffic. Meanwhile, the decisions and policies appliance (network’s brain) moves to the control plane.
- The ”brain” of the network, generally called the ’SDN Controller’ or NOS is the central unit of the architecture. NOS is a software platform that provides a centralized view of the whole network and instructs the data plane how to forward the traffic on the network.
- The creation of API’s to communicate with the NOS. On one hand, the Northbound (NB) APIs allow network applications to communicate with the NOS. In the other hand, the Southbound (Southbound (SB)) APIs allow the communication between the NOS and the forwarding devices. Finally the East/West Interfaces that allow communication between several NOS instances (Cluster model followed by several NOS such as ONOS [22] [2], Onix [23] or OpenDaylight [24] [3]). The different NOS instances seek a consistent view of the whole network and each one handles a subset of networking devices. This model (cluster) is depicted in Figure 2.4.

[2]http://onosproject.org/
[3]http://www.opendaylight.org/

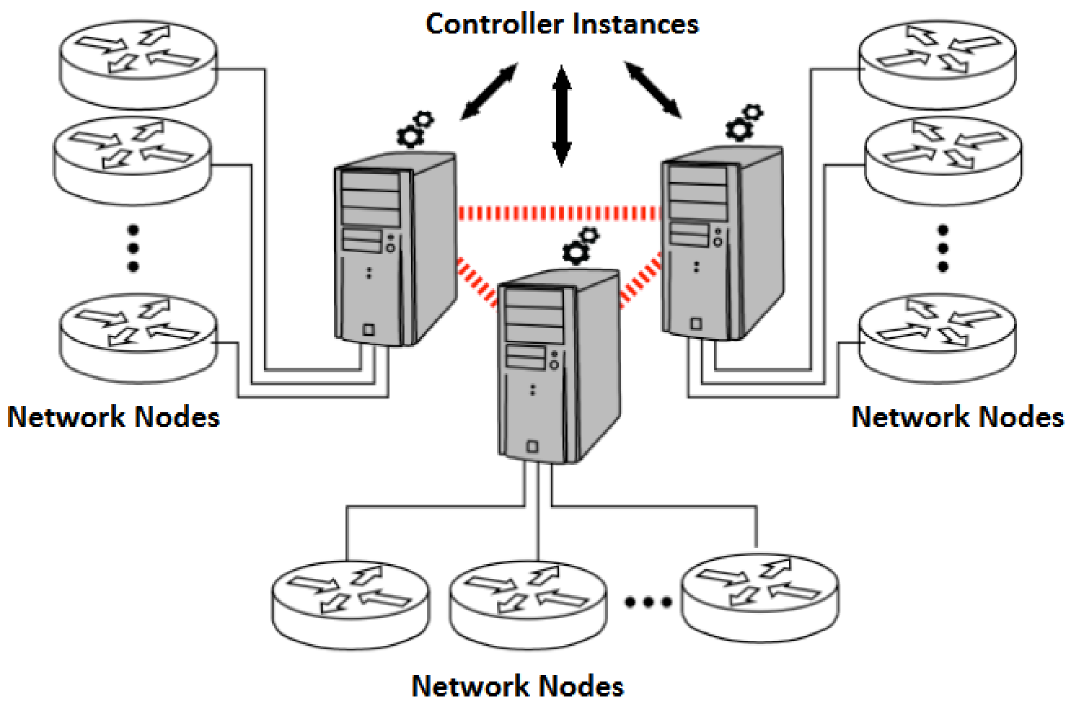


Figure 2.4: Distributed controller architecture (Cluster model). Source : [25]

The three planes (*Data*, *Control* and *Management*) are populated by components with their own purpose and functionality on the architecture, that rely and provide information to each other, which in turn enriches the network into a more functional environment to work with.

The Control Plane, is formed by three main components. The SB interface is composed by a set of protocols such as OpenFlow (standard SB protocol), Open vSwitch Database Management Protocol (OVSDB) [26], Simple Network Management Protocol (SNMP) [27] and others, where OpenFlow stands as it fundamental piece, present in all controllers. It is the bridge between the controller and the forwarding devices being responsible for the interpretation of the commands provided by the controller and the communication bridge between the controller and the forwarding devices. Basically it is the component that translates the 'orders' given by the controller and deploys it on the network devices (switches, routers) in such a way they understand how to handle the packets. The controller (NOS) is the main piece of the environment being internally programmable by modules that belong to the controller. These modules, depending on their purpose, instruct the controller into performing some network functionality such as retrieving information about the network, allowing/denying traffic with certain patterns and others. Last but not least, the NB interfaces have a huge importance on this architecture. They allow that external network applications to instruct the controller into performing some kind of behaviour on the network. Note that, not all the NOS have west/east interfaces, it depends on its nature (centralized/distributed). The distributed ones have these interfaces in order to allow communication between several NOS instances. This aims at providing a solution in order to overcome the problem of 'single point of failure' inherent to this architecture [25].

Finally the *Management plane* is where the external network applications lie. They communicate with the controller through mainly Representational State Transfer (REST)ful API's provided by the

NOS. So, instead of having specialized hardware into performing a task on the network such as load balance, mobility management or traffic monitoring, all this can be accomplished by one or several network applications. Many applications can be created with a wide variety of functionality, bearing in mind the limits of the API's provided by each controller.

2.1.1.1 NORTHBOUND INTERFACES

The NB interfaces, as mentioned in previous Subsection 2.1.1 are one of the key abstractions of the SDN ecosystem. On the contrary to the so-called SB APIs that already have a primary protocol to communicate with the underlying network devices, the NB API are always referred in a more vague terms.

The standardization of the NBI are, lately, a subject much discussed by the SDN community ([28], [29], [30], [31]). Essentially, applications are the exciting part of an SDN environment, enabling everything from network services, such as QoS, monitoring, load balancing and others. Even though it offers an enormous contribution to the SDN stack, the common consensus is that is too early to develop a single standard at this moment. Some argue that it is preferable to not develop a standard right now because it would cause more harm than good if it was somehow wrong or incomplete. Others claim that the community should focus on bringing a more stabilized and mature environment before proceeding to the development of such standards.

However, most believe that, as SDN evolves, the standardization process regarding NBI will eventually occur. As mentioned in the beggining of Chapter 2, the ONF's NBI-WG is looking into standardizing the NBI. They think that the participation of non-ONF-members while developing such standards is critical as they want it to be an open community effort instead of a vendor driven activity [19]. According to this group "the argument that it is 'too early' to standardize APIs contradicts the facts that network operators are clearly concerned about the lack of such standards and see it as an obstacle to deployment".

The ultimate goal is not to be tied to a single vendor's controller API. However, existing controllers such as OpenDaylight, Floodlight [4] or Onix have already proposed and defined their own NB APIs. The existence of different northbound APIs causes incompatibility issues for applications designed for a specific controller. This means that an application is designed to interact with a particular controller by complying with its NB API into receive/transmit information to the controller. Thus, in order to use that application with another controller, another version would have to be created to properly communicate with that other controller. Having a single standard NBI would allow the portability and interoperability of an application among the different controllers.

The southbound protocols, mainly OpenFlow protocol, and the different existing controller will be furtherly discussed in Sections 2.2 and 2.3 respectively.

2.1.2 BENEFITS

SDN-enabled networks bring several important advantages, some of them mentioned in the previous Subsection 2.1.1. In this Subsection, we will enumerate several important benefits [32], [33] that SDN brings into the networking area (including IT companies), more precisely in data center and

[4]http://www.projectfloodlight.org/

cloud environments. We will also briefly point out which architectural changes were responsible for the given benefit.

2.1.2.1 VENDOR AGNOSTIC

A vendor agnostic solution could be implemented with any off-the-shelf hardware. OpenFlow delivers a uniform, vendor agnostic interface between control and data planes where any manufacturer of network assets (such as switches) can support it. In fact, many proeminent switching vendors already support OpenFlow [34], [35] such as Cisco, Brocade, IBM, NEC and many others.

This way, no longer exists the necessity to administer the network through proprietary, vendor-specific tools/protocols. Instead, that can be provided by an open source controller. These open source solutions using OpenFlow may empower network administrators to create their own SDN solutions.

The abstraction of the control plane is the reason why network administrators are able to programmatically manage network provisioning, configuration, and traffic. OpenFlow protocol allows the communication between any OpenFlow-enabled switches with an open source controller which in turn may be programmed into performing some particular functions into the network.

2.1.2.2 UNNECESSARY SPECIALIZED HARDWARE

We are used now into having some kind of middlebox performing a particular function on the network such as traffic inspection (firewall), load balance or QoS. All these tasks may be performed by network applications on top of the controller (in the *Management Plane*). They interact with the controller by providing it with instructions such as: how to treat a certain traffic profile according to some high-level policies; or simply collecting data for billing, monitoring or internal processing purposes.

The underlying architecture structure of SDN and its abstractions, mentioned in the previous Subsection 2.1.1 allows the controller to be instructed by external network applications and effectively make the specified changes on the underlying network.

This is accomplished mainly by the planes separation abstraction which are backed up by mainly two interfaces (REST APIs on the NB and OpenFlow on the SB) that allow bottom-up communication between the several components of the SDN stack. In this case, this allows that network application instructions to be effectively transmitted along SDN stack to the network (routers, switches).

2.1.2.3 ENABLE PROGRAMMABLE NETWORKS

Obviously a consequence of the abstractions of the SDN concept. The controller and network applications are pieces of software created into managing in a central way the underlying network. These pieces are fundamental and have the ability to shape and control data traffic which is a big advantage that provides increased flexibility, scalability, performance and a way easier way to manage the network.

2.1.2.4 INFRASTRUCTURE SAVINGS

Network infrastructure refers to the hardware and software resources of an entire network that enables network connectivity, communication, operations and management of an enterprise network. This includes the network devices (switches, routers, middleboxes) as well as the personnel responsible for configuring and maintaning those devices. This new approach helps lowering the capital and operational expenditures (Capital Expenditure (CAPEX) and Operational Expenditure (OPEX)) thus improving efficiency of the network.

On one hand, lowering the CAPEX is accomplished by avoiding the necessity of buying that many extra network devices to cope with the demand of services and customers as well as the growing number of devices connected to the network. The main reason behind it is the virtualization of the network. Virtualizing the network enables the execution of several virtual machine instances into a single physical node. This approach has several advantages including better utilization of the datacenter servers [36].

On the other hand, lowering the OPEX is also achievable by having a better resource utilization also mentioned in the CAPEX case, but also for avoiding the necessity of having such huge teams of management personnel. This means that, service providers would no longer need to hire such a high amount of administrators to configure the network due to the ease of management that the SDN approach brings to the table. In addition, configuring manually the network, which is a tedious and error-prone process, could origin configuration errors that may lead the network to misbehave and thus causing disruption of services.

2.1.2.5 IMPROVED AVAILABILITY

This benefit is a direct consequence of the immediatly previous one, which by eliminating manual intervention, enables the reduction of configuration and deployment errors that can impact and reduce the network availability. Note that, in this particular case we are refering to availability at the network level, not at the controllers level. Obviously, if the controller fails it will also impact the network's availability or if some network device or link fails it will impact in the network's reliability. However, some controllers possess mechanisms that help mitigate these challenges. We will discuss the above mentioned challenges in Subsubsection 2.1.3.2.

2.1.2.6 CENTRAL NETWORK CONTROL

The Central Network Control is basically the so-called SDN controller that provides a central view of the network and the ability to control the underlying network devices from different vendors through the OpenFlow protocol.

This piece of software allows a more granular network control with the ability to apply comprehensive and wide-ranging policies at the session, user, device and application levels. In addition, it can accelerate service delivery and provide more agility in provisioning both virtual and physical network devices from a central location.

2.1.2.7 ENABLE INNOVATION

SDN alongside OpenFlow protocol enables organizations the ability to create new types of applications, services, and business models [37], but also enables researches to test and deploy the ability new protocols in production networks to evaluate more accurately its behaviour in a real scenario.

2.1.2.8 CLOUD ABSTRACTION

Cloud computing is based on the concepts of computing research areas such as Service-oriented Architecture (SOA) [38], distributed and grid computing, and virtualization.

Cloud computing can be considered a new computing paradigm that enables companies to provide their computing infrastructure resources (processing time, storage, network connectivity) to customers as a service/utility [39]. Consequently, several business models rapidly evolved to embrace this technology by providing software applications, programming platforms, data storage, computing infrastructure and hardware as services.

In addition, cloud computing is evolving into a unified infrastructure [40], [39]. By abstracting cloud resources using software defined networking, it's easier to unify cloud resources. The networking components that make up massive data center platforms can all be managed from the SDN controller allowing the creation of such an application that, for example, would automatically make changes on the network according to the customers demand and Service Level of Agreement (SLA). SLA is a service contract agreement between the customer and the service provider specifying all the service parameters such as the contracted delivery time, amount of resources currently rented, the rent costs and others.

Once again, the separation of duties achieved by the abstraction layers and the ability to program the network allow the creation of such applications that can centrally coordinate the network according to the demands required by the service provider which in turn provides its infrastructure resources dynamically to its customers. Such applications, as the one mentioned before, can improve user experience by giving them the ability to ask on-demand resources. Therefore, customers, which can be other companies or simple users, may construct their own private cloud according to their needs or their application needs.

2.1.2.9 SECURITY

The SDN Controller provides a central point of control to distribute security and policy information consistently throughout the enterprise. Centralizing security control into one entity, like the SDN Controller, has the disadvantage of creating a central point of attack. However, SDN can effectively be used to manage security throughout the enterprise if it is implemented securely and properly.

2.1.2.10 EXTENSIBILITY

Since SDN is software-based, it is easy to use SDN API references for vendors to extend the capabilities of an SDN solution either by developing/improving open protocols (OpenFlow, OVSDB [26], NETCONF [41] and others) or by developing applications (internal or external to the controller) that express a particular function in the network (QoS, load balance, firewall, intrusion detection, network monitoring, etc). These applications may rely on other applications functionality, thus not having to develop everything internally and might contribute to the improvement of already existent modules.

2.1.2.11 ISOLATION AND TRAFFIC CONTROL

Cloud service providers can benefit from centralizing the networking control using a central management tool. OpenFlow-enabled SDNs enable the process of traffic isolation of tenants by programming flow entries that would distinguish the tenants (using for example Virtual Local Area Network (VLAN)s). That can be achieved by defining the traffic rules using the SDN controller's modules or applications, which helps in providing full control over the network traffic.

2.1.2.12 EFFICIENT NETWORK UTILIZATION

Existing capacity and performance management tools perform best when demand increases at a linear or predictable rate. Capacity planners are able to leverage the data to monitor, plan, or resolve additional capacity needs. For example, network fault, load balance and performance monitoring applications can provide notification when a threshold has been breached and, if that's the case, some may perform automatically changes on the network without service interruptions and balancing the use of resources on the network.

The real-time monitoring of resource utilization is essential to those automation tools that pretend to take advantage of SDN to drastically reduce provisioning time while increasing overall service performance and availability. Basically, SDN provides new opportunities to simplify the monitoring, management, and re-provisioning processes.

2.1.3 SDN MAJOR CHALLENGES

Even though SDN is a promising solution for IT and cloud providers and companies, it faces certain challenges that concern the community, delaying its adoption and implemention in real production environments.

In this section, we will briefly describe some key challenges that SDN faces.

2.1.3.1 LEGACY INTEROPERABILITY

As it has been mentioned in Subsection 2.1.2, SDN, despite its challenges, adds a lot of value to current networks. However, its transition must occur incrementally due to the vast installed base of networks that support many vital systems and businesses [42]. It is not possible to simply build a new infrastructure and 'move' our networks to this infrastructure overnight. The transition must occur slowly which will require backwards compatibility with legacy equipment. This implies a hybrid SDN infrastructure, in which SDN-enabled and legacy equipment may coexist while the transition to SDN occurs.

The interoperability required should be reached by introducing new protocols that could help achieve a softer transition to SDN by reducing the costs, risks and disruption of production networks. Organizations like Internet Engineering Task Force (IETF) and ONF have already proposed some protocols that help with the transition by decoupling the data and control planes. Forwarding and Control Element Separation (ForCES) is an IETF's working group that defines an architecture which includes a protocol, transport and model [43] that aim at separating the control and data planes. On the other hand, OpenFlow a protocol originally developed by Stanford and now managed by ONF is the first standard communications interface defined between the control and forwarding layers of an SDN architecture. In addition, the Migration Working Group [44] in [45] specifies two migration approaches : Direct upgrade (Greenfield deployment) or Phased Migration (Mixed and Hybrid Network deployments). The first means that the legacy network is upgraded to become OpenFlow enabled and the control machine is replaced with an OpenFlow controller (Greenfield). The second approach may occur either using Mixed or Hybrid deployments. A Mixed deployment means both the OpenFlow controller and traditional devices coexist in the network and need to exchange routing information via a legacy control machine. This mixed approach requires companies to replace portions of their network with SDN equipment. In the Hybrid deployment both mechanisms are "installed" in the same device being capable of communicating with the OpenFlow Controller as well as with the legacy control machine.

There are several controllers (for exameple OpenDaylight [46]) that can integrate non-SDN technologies (SNMP [27], Border Gateway protocol (BGP) [47], PCEP [48], Network configuration protocol (NETCONF) [41]) with interfaces like OpenFlow or OVSDB.

In general such approaches seem to bring short-term advantages such as reducing costs by only upgrading a couple devices and using mechanism that enable the network to maintain functional. Despite this short-term advantages it should not be considered a 'best of both worlds' mechanism to be used for the future, but simply serve has a means to an end, which is the change the whole infrastructure to fully support the SDN paradigm.

2.1.3.2 AVAILABILITY AND RELIABILITY

Achieving resilient communication is a top requirement of networking. This way, SDNs are expected to achieve, at least the same levels of availability as legacy networks in order to prove as a real new alternative technology. In SDN environments two main problems regarding reliability are usually addressed [49], [7]. First, the necessity of quickly recovering from a link or device failure by rerouting that traffic through an alternative path preventing manual configuration error, so the connectivity is not lost and avalability is increased. Second, the nature of a centralized controller, despite the many advantages it brings is still considered a concern because it presents a single point of failure to the network.

In legacy networks, when a network device fails, the network traffic passing through those devices is quickly rerouted through an alternative path in order to maintain connectivity. So, in case of link/device failure SDN controller should have the ability to rapidly recover from that situation by using some mechanism internal/external mechanism. For example, an internal module of the controller or an external network application that performs monitoring in the network and acts upon that information to establish an alternative path in case of failure or creates *a priori* redundant paths.

In case of controller failure the whole network may collapse, which would interrupt all network traffic and flow requests [50]. To address that problem, it is important that the controller enables clustering of two or more SDN controller in active stand-by mode. It is thus essential to maintain a consistent state between the stand-by controllers.

To tackle this problem several distributed SDN controllers such as Onix [23], ONOS [22] or OpenDaylight [24] provide the ability to create a cluster [25] in order to mitigate the reliability issue in case of controller failure.

2.1.3.3 SCALABILITY

Despite the decoupling of the control and data plane may present huge advantages in the networking field as mentioned in Subsection 2.1.2, it also has drawbacks through which scalability limitations may arise.

The first drawback is the latency introduced by exchanging network information between multiple nodes and a single controller. The increasing number of nodes in a network may generate a high amount of flow requests that will be issued to the controller, which may not handle them all turning the SDN controller into a key bottleneck [51]. Each controller has different performance levels, which affect their ability to handle a great amount of flow requests. Usually each controller has an estimated maximum capacity on the amount of flows they can handle at each second (for example, NOX's average capacity is 30K flows/s [52], while OpenDaylight's average capacity is around 75K flows/s [53]). The capacity of each controller is influenced by the capacity of the machine where the controller is running and the testing environment (number of threads and switches). Section 2.3 will provide more details about the capacity of several controllers, namely Beacon, Floodlight, Maestro among others. Usually this is enough for campus or small enterprise networks, but is a bottleneck for large data center networks containing a much higher number of network devices. In addition, [52] estimates that a large data center network may have 2 million virtual machines, which in turn may generate 20 million flows/sec.

The existence of distributed controllers may help mitigate this problem by having one or more physically distributed controllers managing the network such as the previously mentioned Onix [23], ONOS [22] or OpenDaylight [46]. Onix [23] is a distributed (proprietary) control platform that helps address the scalability issue through its multiple instances. Onix maintains a Network Information Base (MIB) containing all network information and makes sure that its state is consistent among all instances using a distribution import/export module (East/Westbound API). It also allows partitioning of the network by enabling configuring a particular instance to only contain an up-to-date subset of the Network Information Base (NIB) in memory.

2.1.3.4 PERFORMANCE

In the SDN paradigm the processing of packet is not performed via some distributed routing protocols which only forward traffic based on the destination address. In SDN, the forwarding decisions are flow-based, instead of destination-based. A flow is broadly defined by a set of packet field values acting as a match (filter) criterion and a set of actions.

Because SDN is a flow-based technique, its performance is measured based on two metrics: flow-setup time, and the number of flows per second that the controller can handle (controller's capacity) [49]. If the capacity is exceeded more controllers must be inserted to cope with the demands of the network. There are two approaches to insert flows into the switches: proactive or reactive.

The first means that the flow setup process is performed before the packet arrives at the switch. This way the switch already knows how to treat the packet before it is received in the switch. This method implies negligible delay and removes the limits on the number of flows per second that can be handled by the controller.

The second occurs when the switch receives the packet and does not find any flow match in its table. The switch may either send the entire packet or just some packet headers to the controller in order to be processed. The controller informs the switch how to handle the packet and for how long it shall be kept in the switch's table before expiring. The flow setup time in this approach includes: the time it takes to send the packet to the SDN controller; the packet processing time at the SDN controller; the time it takes to send the answer to the switch populating its flow table and informing how to treat the packet.

Therefore, flow initiation adds overhead that limits network scalability and introduces reactive flow-setup delay [54], [55]. In data centers, a reactive setting of flow rules can lead to an unacceptable performance when only a low amount of switches are handled by one OpenFlow controller. Such approach would cause an immense amount of data to be sent to the controller which could hamper the effectiveness of the controller and slow down the routing process of the network. This means that large-scale SDN deployments should not rely on a purely reactive approach, but rather on a combination of proactive and reactive flow setup. In general, the SDN controller fills the flow table with the maximum number of possible flows.

This suggests that the controller's performance, its capability of processing the flow requests from the switches directly affects the whole system performance. Each controller is designed differently to the others and some of their design characteristics impact directly in its performance and thus in networks performance. According to [49], [56] the language in which they are written (C, Java, Python), the processing type (centralized, distributed) or the techniques used by them (packet batching, task batching) are all relevant factors for the performance of the controller. All these characteristics impact on the metrics mentioned before: the flow setup time and the amount of flows a controller can handle for second. In addition, the number of controllers needed to manage a network and its placement also makes an impact in the network's performance, but usually one controller is enough [57].

Recent studies also show that the SDN control plane cannot be fully physically centralized due to availability, reliability and scalability metrics [58]. Therefore, distributed controllers are the natural choice for creating a logically centralized control plane, due to their ability of being capable of coping with the demands of large scale networks. However, these controllers bring additional challenges, such as the consistency of the global network view, which can significantly affect the performance of the network if not carefully engineered. It is obvious that fundamental trade-offs must achieved in order to improve the SDN environment.

2.1.3.5 SECURITY

As the interest on SDN by companies increases, an important issue comes up, security. Companies need to know if SDN products are not going to introduce vulnerabilities to their infrastructure if they move forward into adopting SDN [59]. A greater focus on security is therefore required if SDN is going to be acceptable in broader deployment. A security study [60] shows that about 32% of the 250 business technology professionals that participated think that networks will become somewhat more secure, while only 12% think that they will become less secure. The lack of integration with existing security technologies is one of the main security problems pointed.

In fact, potencial security vulnerabilities exist across a SDN platform. For example, at controller-application level proper authentication and authorization should be performed in order to access the controller. The controllers are a particularly attractive target for attack in the SDN architecture open to unauthorized access and exploitation because they are the main component that controls the whole network. With the proper knowledge, an attacker could quickly and easily subvert the network operations into its own benefit [42]. According to [59], the security of SDN elements may come down to OS hardening where all best practices for hardening would be applicable thus contributing for the security of the environment.

Although some controllers already provide a simple mechanism of authentication (such as OpenDaylight with a simple password-based authentication) those mechanisms should go even forward by providing different privileges to different network applications or a role-based access control (RBAC) for controller administrators.

At the data-controller level, OpenFlow's current specifications describe that Transport Layer Security (TLS) may be used for mutual authentication between switches and the controller [21]. However, the use of TLS is not mandatory thus requiring a full security specification between the controller and the switch for the sake of the protection of the transmitted data [42].

Despite the above mentioned challenges, the SDN architecture offers an effective alternative to operate highly secure networks. From the security perspective SDN can support [61], [42]:

- **Network forensics** - facilitate quick and straightforward, adaptive threat identification and management through a cycle of harvesting intelligence from the network, analyzing it, updating policy, and then reprogramming to optimize from network experience.
- **Security policy alteration** - Ease in policy management and enforcement, allowing the insertion of a security policy into all network elements, reducing the risk of misconfiguration between the devices and conflicts due, for example, to the use of applications like FortNox which handles conflicts between policies.
- **Security services/applications** - enable the easy introduction of security applications to supervise the network's traffic such as firewalls, intrusion detection system (IDSs).
- **Agility on mitigation** - Programmability enables automation and adaption enabling a quickly reaction upon threats thus mitigating risks.

2.2 OPENFLOW

We verified that SDN is a major innovation bringing many advantages, as mentioned in Subsection 2.1.2 to the networking area. Many of these advantages are due to OpenFlow protocol, specially for its ability to decouple the data and control planes that is achieved. This is due to Openflow protocol. OpenFlow is needed to move network control out of the networking switches to logically centralized control software.

Openflow [62] was originally developed by Stanford University in 2008. Even though SDN and OpenFlow started as academic experiments [62], they gained significant importance on the last few years. Nowadays, most vendors include OpenFlow API in their equipment. The momentum was so strong to make some huge companies, like Google, Facebook, Microsoft, Verizon to fund ONF in order to promote and adopt SDN through open standards development. OpenFlow latest version is 1.5, however most of the existent controllers support at most version 1.3. Each version is an evolution of the previous one by inserting new capabilities such as more possible matching fields, actions, instructions or even the change from a single flow table to multiple flow tables. Table 2.1 depicts the differences between all the relevant versions which are the evolution between version 1.0 and 1.3. In Subsection 3.1.2 are described in more detail the main differences between versions 1.0 and 1.3.

OpenFlow enables SDN due to its ability to decouple the data and control planes. This allows the controller to exercise direct control over the data plane elements via a well-defined API such as OpenFlow [62], [64]. In other words, OpenFlow enables controller entities to dynamically program heterogeneous forwarding devices, something difficult in traditional networks, due to the large variety of proprietary and closed interfaces and the distributed nature of the control plane. OpenFlow is the most widely accepted and deployed open southbound standard for SDN. These standards promote interoperability, allowing the deployment of vendor-agnostic network devices.

OpenFlow switches consist in three parts : A Flow Table, a secure channel and the OpenFlow protocol.

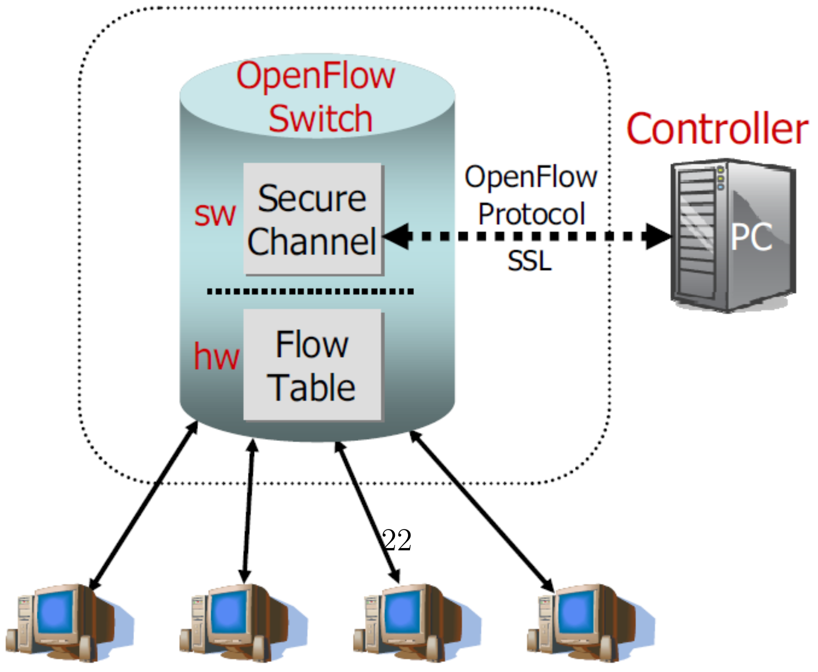
1. The **flow table(s)** responsible for maintaining the flow entries also called rules. Each rule matches a subset of the traffic and performs certain actions (dropping, forwarding, modifying, etc.) on the traffic.

2. The **secure channel** is a communication channel responsible for the exchange of all the information between the controller and the switch.

3. The **OpenFlow protocol** provides an open and standard way enable communication between the SDN controller and the switch. Using this interface, the controller configures and manages the switch.

OpenFlow version	Match Fields	Statistics	Actions		Instructions	
			Req	Opt	Req	Opt
1.0	• Ingress port • Ethernet type, source and destination • VLAN ID and priority • IP protocol, source, destination and TOS bits • TCP/UDP source and destination ports	• Per table (3) • Per flow (4) • Per port (2) • Per queue (3)	• Forward (All, Controller, Local, Table, In-Port) • Drop	• Forward (Normal, Flood) • Enqueue	• Apply-actions	
1.1	• Metadata • MPLS Label • MPLS Traffic Class • ICMP type • ICMP Code • ARP Code	• Per Group (3) • Per Bucket (2)	• Output instead of Forward (All, Controller, Table, In-Port) • Group	• Output (Local, Flood, Normal) • Set-Queue (former Enqueue) • Push-pop tag (VLAN, MPLS) • Set-field (former Modify-field)	• Write-Action • Goto-table	• Write-Metadata • Clear-Actions • Apply-Actions
1.2	• Open Extensible Match (OXM) • IPv6 Source, Destination, protocol number, traffic class, flow label • ICMPv6 code • IPv6 neighbour discover header fields			• Change/-Copy TTL (MPLS, IP)		
1.3	• IPv6 Extension header	• Per Meter (5) • Per Meter Band (2)		• Push-pop tag (PBB header)		• Meter

Table 2.1: OpenFlow evolution between 1.0 and 1.3 versions. Required (Req) and Optional (Opt) capabilities. Source: [63]



A flow entry is composed by seven fields which are :

- **Matching fields (Header fields)** are used to match the arriving packets based on their headers' values such as: Ingress port, Ethernet source/destination addresses, Ethernet type, VLAN, IP source/destination addresses, source and destination ports or more recently metadata among others.
- **Statistical counters** are used for statistics purposes. There are twenty-two counters in existence, including: Packet Matches per Table, Received Packets and Duration (nanoseconds) per Flow, Received Packets, Receive Errors, Collisions per Port, and Transmit Packets per Queue.
- **Actions** are encapsulated in instructions (since version 1.1) and inform the switch how to process the matching packets that arrive on the switch. There are three required actions available such as *Output* (sends a packet into a logical or physical port), *Drop* (drops the packet), *Group* (process the packet through the specified group), which are required actions. Examples of optional actions, also depicted in Table 4.1, are *set-queue*, *push/pop tag* or *set-field* which allows to modify a specified header field of the packet.
- **Instruction** appeared first in version 1.1 and allows the control of the pipeline processing. It allows directing the packet to another table or communicate information between tables in the form of metadata. In addition, it also allows the merge/delete the current action set or to apply the actions specified immediately. All of these instructions are depicted in Table 4.1.
- **Priority** is a parameter associated with each flow entry that differenciates their order of look-up. The flow entries with higher priority within each flow table are processed first.
- **Timeouts** are fields that specify how long the entry will be cached in the switch. There are two types of timeouts, hard and soft. The hard timeout defines how many seconds a flow entry remains in the switch before expiring, while the idle timeout defines how many seconds it can remain on the switch on the absence of traffic.
- **Cookie** is a value chosen by the controller, used to distinguish flow entries.

Using OpenFlow, the controller may configure and manage the network through the secure channel that exists between them. This channel may optionally be encrypted using TLS or just operate directly over Transmission control protocol (TCP). In turn, the switch is responsible for forwarding the traffic according to the flow entries installed by the controller reactively or proactively.

The OpenFlow switch works as follows:

- When a packet arrives at the switch it performs a look-up on its flow tables. The first OpenFlow version only had a single table, but the following versions already supported several flow tables, which required a new pipeline. The new OpenFlow pipeline is depicted in Figure 3.8. The look-up procedure is performed in each table sequentially. Note that, the flows with higher priority are inspected first. The switch will move on to the next table if a match was not found in the previous one. By match we mean that a flow entry was found that its header fields match the packets header fields.

- When a match is found, the switch executes the instruction(s) contained in that flow entry. Such actions can be : forward a packet to a specific port (logical or physical), move on processing another flow table, changing a packet field and many others.

- If a match is not found after processing all the flow tables, the switch sends a *packet-IN* message to the controller through their communication channel. This message may include the whole packet or just some packet fields. The controller processes the flow request and then informs the switch how the packet should be handled. This is performed by sending a *packet-OUT* message or, alternatively, by also installing a flow entry into the switch through a *flow-MOD*. This way, the switch knows how to treat future similar packets that arrive to it. After that, the switch handles the packet according to the action specified by the *packet-OUT* message or the newly flow installed through the *flow-MOD* message. Finally the statistical counters of that particular flow entry are updated depending on the rules installed by a controller application, an OpenFlow switch can – instructed by the controller – behave like a router, switch, firewall, or perform other roles (e.g., load balancer, traffic shaper, and in general those of a middlebox).

2.3 CONTROLLERS

The Controller or NOS stands as the main component in the network architecture. It supports the communication with both Management and data planes. The communication with the network devices is performed via several SB protocols where OpenFlow protocol stands out as the first standards in communication between control and data planes enabling commands to be interpreted by the switches while all the complexity is managed by the controller. In addition it allows that several network applications may express the desired behaviour on the network by communicating with the controllers through their NB API. These applications may perform a wide range of functionality and thus may replace specialized hardware that perform a specific functionality in the network such as firewall, monitoring, QoS or load balance.

The overall architecture of existing NOS consists of five different hierarchical layers:

- **Network applications layer -** Layer containing applications developed by programmers to give more functionalities to the SDN;
- **Northbound layer -** Offers an API for the applications so they can communicate with the NOS core and instruct the controller on performing a certain functionality;
- **Core layer -** This layer contains modules that wrap up the overall information about the topology. Basically, this layer provides the necessary abstractions that enable developers to configure the network without knowing the low-level details of connecting and interacting with the network devices. The interaction with the network devices is performed via the SB protocols, such as OpenFlow, while the interaction with network applications is performed via the NB APIs or the controller's internal API;
- **Southbound layer -** Offers an API to the lower and higher layer so the core can communicate and control the hardware from the physical topology;
- **Southbound applications layer -** Provides applications to control the physical hardware via management protocols. These applications do not need to be OpenFlow, there are other protocols that can be used to configure the physical forwarding devices on this layer such as:

OVSDB [26], NETCONF [41], SNMP [27], among others. Their existence is due to legacy hardware and they cannot provide the same functionalities offered by OpenFlow.

Table 2.2 shows several OpenFlow-based controllers and their characteristics such as : the processing type; the northbound interfaces supported; the southbound protocols supported; the OpenFlow version supported; the language they were written and the average amount of flows they can handle at each second (average capacity). As we previously mentioned in Subsubsection 2.1.3.3, the controller's capacity depends largely on the capacity of the machine where the controller is running as well as the testing environment. We will discuss about the capacity of several controllers at the end of this section, resorting to several studies existent in the literature ([51], [53], [55], [65], [66] and [67]).

The type of processing is a major factor that impacts on the controllers availability, scalability and performance as it was discussed in Subsubsections 2.1.3.2, 2.1.3.3 and 2.1.3.4 respectively.

A centralized controller is a single entity that manages all forwarding devices of the network. Naturally, it represents a single point of failure and may have scaling limitations.

An example of centralized controllers are as NOX [69] [5], NOX-MT [55] [6], POX [67] [7], Maestro [70] [8], Beacon [65] [9], Floodlight [71] or Ryu [72] [10]. Most of them (with the exception of NOX, POX and Ryu) are multi-threaded systems designed to explore the parallelist of multi-core computer architectures. NOX-MT is an evolution of the first SDN controller, NOX [11]. It solves several limitations that NOX controller had such as the fact of being single-threaded and not optimized for performance. NOX-MT is multi-threaded and uses well-known optimization techniques (I/O batching) to improve its performance.

POX is NOX's sucessor and was built as a friendlier alternative. Compared to NOX, POX has an easier development environment to work with and a reasonably well written API and documentation. It also provides a web based GUI and is written in Python.

Maestro is a Java-based controller that, in addition of having a multi-threaded structure, it also looks into distibuting evenly the work among the available cores in order to maximize the throughput. It also uses some optimization techniques such as I/O batching to improve the efficiency of the system.

Beacon is cross-platform, modular, event-based OpenFlow controller written in Java. Beacon introduced the ability to load core bundles dynamically in runtime, making the controller more OS-like, enabling thus new use cases such as start and stop applications at runtime that can be used to debugging or extend another module functionality. The ability to load bundles at runtime was performed by using Equinox [73] an implementation of the Open Services Gateway iniciative (OSGi) [74] core framework specification. Despite being wrote in Java and thus considered less efficient than controller written in C/C++ the goal was to use a more friendly language (namely Java) for the developers while avoiding some common problems of the C/C++ language such as long compilation times and manual memory errors.

Floodlight is a Java-based OpenFlow controller forked from the Beacon controller, originally developed by David Erickson at Stanford. Floodlight supports a broad range of both hypervisor-based virtual switches like Open vSwitch. It is easy to set up with minimal dependencies and user friendly (

[5]https://github.com/noxrepo/nox-classic
[6]https://github.com/noxrepo/nox
[7]https://github.com/noxrepo/pox
[8]http://zhengcai.github.io/maestro-platform/
[9]https://openflow.stanford.edu/display/Beacon/Releases
[10]http://osrg.github.io/ryu/
[11]https://github.com/noxrepo/nox

it contains both a web based and Java based GUI). In addition, most of its functionality is exposed through a REST API enabling the use of network applications to interact with the network.

Ryu is an SDN controller fully written in Python that supports various protocols for managing network devices, such as OpenFlow, NETCONF, SNMP, and others. Both Ryu and Floodlight have integration with OpenStack project. In addition, Ryu also has integration with Snort [12], an Intrusion Detection System (IDS).

On the other hand, a distributed controller environment consists on several controller instances that communicate to each other via their east/west API's in order to achieve a consistent and centralized view of the network. The distributed controller can be either a centralized cluster of nodes or a physically distributed set of elements. While the first alternative offers a high throughput, the second offers a more resilient enviroment regarding logical and physical failures. These controllers, as mentioned in Subsubsections 2.1.3.2 and 2.1.3.3, help mitigate the scalability problem inherent to centralized controllers. Such examples are Onix [23], ONOS [22] or OpenDaylight [75].

Onix is a distributed system which runs on a cluster of one or more physical servers, each of which may run multiple Onix instances. Onix maintains a Network Information Base (MIB) containing all network information and makes sure that its state is consistent among all instances. It also allows partitioning of the network by enabling configuring a particular instance to only contain an up-to-date subset of the NIB in memory. However, according to Martin Casado et al. in [23], "Onix provides general tools for managing state, but it does not magically make problems of scale and consistency disappear."

ONOS is a scalable and distributed controller platform that targets service provider networks and service provider's requirements, such as policy-driven network programmability and being operator-friendly. ONOS can also be deployed as collection of controller-servers (cluster) that coordinate with each other to achieve resiliency, fault-tolerance, and better load management.

Finally, the Opendaylight project is a collaborative open source project hosted by the Linux Foundation. This project is supported by many important companies in the networking and virtualization field such as Cisco, NEC, Brocade, RedHat and many others. Opendaylight covers the entirety of the SDN stack, including network applications, orchestration, a controller, NB APIs and a SB abstraction layer. It also features a new user interface, called 'Dlux', and a more simplified and customizable installation process, due to the use of the Apache Karaf container, a lightweight OSGi container. Opendaylight's latest stable release called 'Helium' has deeper integration with OpenStack, including improvements in the Open vSwitch Database Integration project.

OpenDaylight supports the OSGi framework and bidirectional REST for the NB API, which can be used by network applications. It also provides integration with OpenStack via Neutron REST API. The OSGi framework enables the load, at runtime, of modules/bundles running the same address space as the controller, providing modularity and extensibility. It basically enables a development model where modules (plugins) can be installed, started, stopped, updated, and uninstalled into the controller, without requiring a reboot from the controller. Meanwhile the REST API is used for external applications that may not be even running in the same machine in the controller. The business logic and algorithms reside in the external applications and internal modules. They use the controller to gather network intelligence, run algorithms to perform analytics, and then use the controller to orchestrate the new rules, if any, throughout the network.

In addition, the Opendaylight's platform also provides multi-protocol southbound support and

[12]https://www.snort.org/

abstracts its specifics from the applications by using a Service Abstraction Layer (Service Abstraction Layer (SAL)). Such southbound protocols are OpenFlow, OVSDB, SNMP and others. It is one of few controllers with support for OpenFlow recent versions(namely 1.3 version)

Last but not least, Opendaylight also provides high availability by enabling clustering[24]. Opendaylight is described in more detail in Subsection 3.1.1.

It is worth mentioning that, despite some controllers being written in solely one language (e.g. Opendaylight is only written in Java and Ryu is only written in Python) they provide a REST NB API, which enables an application to be written in any other language as long as it can communicate with the controller via REST.

There are also a huge variety of SDN controllers in the commercial space such as Big Switch Big Cloud Fabric [13], Plexxi Big Data Fabric [14], Brocade Vyatta Controller [15], HP Virtual Application Networks (VAN) [16], Juniper Contrail [17], Cisco Application Centric Infrastructure (ACI) [18], amongst others [76]. It is clear that there are several different SDN approaches, some more similar to each other than others. The ideas themselves are currently in development and many of these products are either in beta testing or have recently been released.

In these last years, several performance studies were made to analyze the efficiency indexes of several OpenFlow-enabled controllers, namely the throughput, which is essentially the capacity of the controller. In 2012, Voellmy et al. [51] performed a simple throughput test using a slightly modified version of Cbench benchmark tool [77]. The controller was executed on a DELL Poweredge R815 server with 48 cores of AMD Opteron 1.7GHz 6164 processors and 64 GB memory, using eight 1Gbps and two 10Gbps NICs. The results showed that Beacon could handle up to 10 million flows/sec using around 40 cores, while a multithreaded version of NOX handled about 1 million using the same number of cores. Also in 2012, Casado et al. [55], presented a new version of NOX, called NOX-MT, while comparing it with three other controllers, more precisely NOX (single threaded), Beacon and Maestro. The controllers were executed in an eight core machine with a 2 GHz CPU and a 4GB RAM. The environment was constituted by 2 servers, the first running the controller and the second running Cbench benchmarking tool. The testing results showed that the average maximum throughput of NOX-MT was about 1.6 million, while Maestro, Beacon and NOX (single-threaded) reached 300k, 500k and 30k, respectively.

In 2013, Shalimov et al. [66] made another study to analyze the performance of several controllers, such as Beacon, Maestro, Floodlight, among others. The testing environment consisted in 2 servers, connected to each other through a 10Gbps link. Each server has two processors Intel Xeon E5645 with 6 cores (12 threads), 2.4GHz and 48 GB RAM. Both servers are running Ubuntu 12.04.1 LTS with default network settings. The experimental study was performed using Cbench tool and showed that, using the maximum number of threads (a total of 12 threads), Beacon, Maestro and Floodlight were able to process a total of 7, 3.8 and 1.6 million flows/sec, respectively. Also in 2013, David Erickson in [65] presented the architectural decisions of the controller he created, called Beacon. In addition, he

[13]http://www.bigswitch.com/sdn-products/big-cloud-fabrictm

[14]http://www.plexxi.com/products/big-data-fabric/

[15]http://www.brocade.com/en/products-services/software-networking/sdn-controllers-applications/sdn-controller.html

[16]http://www8.hp.com/us/en/networking/sdn/

[17]http://www.juniper.net/us/en/products-services/sdn/contrail/

[18]http://www.cisco.com/c/en/us/solutions/data-center-virtualization/application-centric-infrastructure/index.html

also evaluated Beacon in comparison to other controller such as Maestro, NOX (multi-threaded), Pox and Floodlight. The tests were run on Amazon's Elastic Computer Cloud [19] using a Cluster Compute Eight Extra Large instance, containing 16 physical cores from 2 x Intel Xeon E5-2670 processors, 60.5GB of RAM, using a 64-bit Ubuntu 11.10 VM image. The experimental study also used the Cbench tool connected to each controller over loopback, and it was given a dedicated CPU. The evaluation encompassed two types of throughput testing scenarios: Single-threaded and Multi-threaded. In the first testing scenario Beacon, Maestro, NOX and Floodlight achieved a throughput of 1.35M, 420K, 828K and 135K responses/sec, respectively. Pox and Ryu, both Python-based controllers run significantly lower serving 35 and 20 thousands responses/sec. Meanwhile, in the second testing scenario, higher throughputs were achieved, depending on the number of threads, until a maximum of 12 threads. Beacon, Maestro, NOX and Floodlight achieved a maximum throughput of 12.8, 3.5, 5.3 and 1.2 million of responses/sec, respectively. Finally, OpenDaylight recently executed performance tests, comparing *Helium* and *Lithium* versions. The controller was executed in a machine with 4 cores and 16GB of RAM. The tests were also performed using the Cbench tool that was running on a similar separate VM and connected via internal vSwitch. The results showed that in *Helium* version, the controller was able to process around 75K flows/sec, while *Lithium* was capable of handling around 65K of flows/sec.

[19]http://aws.amazon.com/pt/ec2/

Controller	**Processing Type**	**Northbound**	**Southbound**	**OpenFlow version**	**Language**	**Flows/s**
NOX	Centralized	Internal	OpenFlow	v1.0	C++	30k
NOX-MT [55]	Centralized w/ parallel computation	Internal	OpenFlow	v1.0	C++	1.6M
POX	Centralized	Internal, REST	OpenFlow	v1.0	Python	35k
OpenDaylight	Distributed	Openstack Neutron, REST, Internal	OpenFlow, OVSDB, SNMP, NETCONF, BGP, PCEP	v1.3	Java	100k
Floodlight	Centralized	Openstack Neutron, REST	OpenFlow	v1.0	Java	600k
Onix	Distributed	Internal	OpenFlow, OVSDB, NIB	v1.0	C++, Java, Python	-
Maestro	Centralized w/parallel computation	Internal	OpenFlow	v1.0	Java	3.8M
ONOS	Distributed	Internal	OpenFlow	v1.0	Python	-
Ryu [68]	Centralized	Openstack Neutron, REST, Internal	OpenFlow, OVSDB, NETCONF	v1.4	Python	20k
Beacon [65]	Centralized w/parallel computation	REST, Internal	OpenFlow	v1.0	Java	7M

Table 2.2: SDN Controllers Comparison

2.4 OPENFLOW FORWARDING DEVICES

Data centers are an infrastructure for storing large volumes of data and hosting large-scale applications. According to [78], huge companies like (Google, Yahoo, Facebook, and others) very often use data centers for storage, Web search, and large-scale computations. In addition with cloud computing growing at a fast pace, data centers play an essential role in the Information Technology (IT) industry. Thus their infrastructure should be as cost-effective as it can be. However, data centers current physical infrastruture has several weaknesses such as [79]:

- High operational cost.
- Poor server utilization and high operational cost due to the use of dedicated servers to run applications .
- Slow provisioning of new applications and services.
- Limited mobility of applications, due to physical constraints.
- Hardware dependent on type of server, which is inflexible.
- Intensive operations to maintain or change the network.

The situation improves with the use of virtualization technologies (e.g. VMware [20], XEN [21], and others). Such technologies allow the co-location of several virtual machines into the same physical machine. This results in a virtualized data center which consists on a collection of virtualized hardware such as switches, servers, routers and links. A virtualized data center brings many advantages like reduced costs, easier migration to cloud, no vendor lock-in, and others [80].

Virtualization is the future in data centers helping to tackle many of the issues mentioned before, and with this, software switches are also emerging being considered a promising solution for data center and their virtualized network infrastructures [79]. Virtual switches can simplify the provisioning of IT resources and applications, and make them available for consumption in minutes. In addition, they can implement security policies, virtual private networks, and VM mobility, by themselves. This allows the core infrastructure to remain very simple, focused only on providing connectivity between relatively static hosts [81].

The market offers several OpenFlow-enabled forwarding devices both commercial and open source. We may also distinguish them as of hardware or software. Examples of software-based OpenFlow switch implementations include ofsoftswitch13 [82] and Open vSwitch [81]. The whole idea is to migrate towards SDN environments enabling the use of commodity switches, usually called bare-metal or white-box switches. Usually those terms are used interchangeably however there is a slight difference between them. Basically bare-metal are switches from original design manufacturers (ODMs) with no network operating system loaded on them. In turn, white-box switches are commodity-based bare-metal switches with a network operating system (installed either by an original equipment manufacturer (OEM) or the end user). These switches are relatively inexpensive, easy to obtain and are emerging as an option for specialized networking devices proving to be a cheap and flexible solution with an acceptable performance.

CPqD's ofsoftswitch13 is a user-level software switch that supports OpenFlow v1.3 and implements almost all the features described in its specification [63]. It is destined for experimentation purposes.

On the other hand, Open vSwitch is a virtual switch used as the network switching component in

[20]http://www.vmware.com/
[21]http://www.xenproject.org/

the hypervisor. Open vSwitch maintains the logical state of a virtual machine's network connection across physical hosts when a virtual machine is migrated. It can be managed and monitored by standard protocols such as OpenFlow, NetFlow [83], sFlow [84], SPAN/RSPAN (port mirroring). Open vSwitch can currently run on any Linux-based virtualization platform, including: KVM, VirtualBox [22], Xen, Xen Cloud Platform [23], XenServer [24] [85].

OpenvSwitch consists of two components: a kernel-resident "fast path" and a userspace "slow path". The fast path implements the forwarding engine which is responsible for per-packet lookup, modification and forwarding. It also maintains counters for each forwarding table entry. The majority of the functionality is implemented within the slow path, which is intended to run within the VM management domain. The slow path implements the forwarding logic, the remote visibility and configuration interfaces, such as OpenFlow, and other remote management protocols.
OpenvSwitch is most commonly used as a virtual switch in large cloud deployments (many thousands of servers) for automated VLAN, policy, and tunnel management. However, it can be also used as a simple OpenFlow switch, or a more sophisticated programmatic switch to control hardware environments.

Despite both switches (CPqD's ofsoftswitch13 and Open vSwitch (OVS)) support OpenFlow v1.3, Open vSwitch can be used as a replacement for the default Linux bridging module, while CPqD's switch can not. In contrast, OpenvSwitch does not implement all OpenFlow v1.3 features, more precisely *meters* [85], [63].

As we mentioned before in this Section, the market place also offers many commercial solutions.

An interesting observation is the number of small, startup enterprises devoted to SDN, such as Big Switch, Pica8, Cyan, Plexxi, and NoviFlow. This seems to imply that SDN is springing a more competitive and open networking market, one of its original goals. Other effects of this openness triggered by SDN include the emergence of so-called "bare-metal switches" and "white-box switches", where the software and hardware are sold separately and the end-user is free to load an operating system of its choice [86].

2.5 QUALITY OF SERVICE IN SDN

Nowadays, providing users with a guarenteed QoS meeting SLA is extremely important. However, implementing such QoS system is challenging in the current Internet due to the complexity of proposed QoS solutions and largely manual per-device configuration by network administrators [87]. In result, two commonly adopted approaches are physical network isolation and overprovisioning. However, these solutions lead to a significant installation, operational and management costs increasing even more the complexity of the network [88].

Application running on virtual machines in a data center require performance guarantees not only from compute and storage layers, but also from the network layer. The traffic from different tenants (group of users who share a common access with specific privileges to the software instance) may have to be treated differently and according to the SLA established between the client and the Service

[22]https://www.virtualbox.org/
[23]http://www-archive.xenproject.org/products/cloudxen.html
[24]http://xenserver.org/

Provider. This way, QoS guarantees stands as an important function in a data center environment.

SDN is a promising concept enabling better control of networks, particularly in data centers, through the use of OpenFlow protocol that decouples the data and control planes. With the advent of SDN, a new wave of interest is exploring the possibility of deploying a simple but effective QoS mechanism. There are a huge amount of traffic engineering applications that take advantage of technologies such OpenFlow or OpenvSwitch to enable QoS in SDN networks.

Table 2.3 depicts several existing solutions that enable traffic engineering in a SDN-enabled network. More particularly these applications focus on managing the network by providing load balance and QoS enforcement functions to the network.

Application	**Main Purpose**	**Controller**	**Southbound Protocol**
FlowQos [89]	Traffic classification and rate limiting	POX	OpenFlow
MicroTE [90]	Traffic Engineering approach to achieve minimal overhead	NOX	OpenFlow
OpenQoS [91]	Dynamic QoS routing for multimedia apps	Floodlight	OpenFlow
QoS for SDN [92]	QoS enforcement	Floodlight	OpenFlow, OVSDB
QoS framework [88]	QoS enforcement	NOX	OpenFlow
QoSFlow [93]	QoS enforcement with packet schedulers	-	OpenFlow
QueuePusher [94]	Queue management tool for QoS enforcement	Floodlight	OpenFlow, OVSDB
QoSMeter Framework [95]	QoS enforcement	NOX	OpenFlow

Table 2.3: Traffic Engineering Network Applications. Source : [6]

FlowQos firstly performs traffic classification based on the protocol and ports. After that to enforce rate limit between the different flows (previouly differentiated) it is created an auxiliary virtual switch where each link has a traffic shaper (using tc command) and has a correspondent traffic class. The traffic is identified on the first switch, which according to the flow class redirects to the appropriate link to the next switch. Each link was already configured to limit the rate using the previously mentioned tc command.

MicroTE is a system that adapts to traffic variations by leveraging the short term and partial predictability of the traffic matrix. It relies on a central controller to track which ToR pairs have predictable traffic at a fine granularity, and routes them optimally first. Optimal routing is computed assuming perfect knowledge of the traffic matrix every second with the objective of minimizing the Maximum Link Usage (MLU). The remaining unpredictable traffic is then routed along weighted equal-costmultipath routes, where the weights reflect the available capacity after the predictable traffic has been routed. The monitoring is performed in two ways : through a set of servers that perform measurement of the traffic sent/received informing the controller of the demands; or through the network controller that periodically polls the switches, using OpenFlow API, to retrieve flow statistics. Careful end-host modifications coupled with an upgrade to the firmware in existing data center switches to use OpenFlow are performed in order to ensure minimal modifications.

OpenQos is an extension of the standard OpenFlow controller which provides multimedia delivery with QoS. It exploits OpenFlow's flow-based forwarding paradigm in order to differentiate data and multimedia traffic through the inspection of some packet header values (e.g. source and destination ports and type of service). It collects the up-to-date network state information and according if the packet is a multimedia packet it calculates two paths, an optimized path and the shortest path. The optimized path is calculated using Lagrangian Relaxation Based Aggregated Cost (LARAC) [96]. If the packet is not multimedia it simply uses the shortest path.

QoS for SDN implements a QoS framework using vendor-agnostic interfaces of SDN technologies such as OpenFlow and OVSDB protocols. The OVSDB protocol is used to configure the three default queues (control traffic, high-priority traffic and best-effort traffic). The traffic is classified using the type of service field (TOS) and routed into the appropriate queues according to it. The high-priority traffic queues are configured to have a maximum rate according to the SLA negotiated between the bandwidth broker and a business customer. The bandwidth broker is responsible for checking the availability of the network resources. A failure recovery mechanism was also added onto it, but high-priority traffic always get a higher precedence over best-effort traffic, even after a failure.

QoS framework aims at providing automated QoS through the usage of eight static priority queues at each switch's port. The network administrator only needs to define high level specifications for services (or customers), then the controller automatically 'reserves' network resources. The system automatically manages the mapping of the flows to the correspondent queues focusing on maximizing a new flow's performance requirements while minimizing the number of rejected flows through the use of heuristics and optimization models. Basically it tries to find the appropriate queue, at each hop, in order to satisfy the new flow requirements without violating the requirements of the existing flows. In addition it adapts QoS configuration based on the network state in order to achieve a better utilization of the network resources.

QoSFlow module adds extensions to Openflow v1.0 datapath. It provides differentiated traffic through the use the creation of queues specifying the rate limits of such queues and associating each queue to a flow. The QoS requirements are then enforced by using the packet schedulers (Hierarchical Token Bucket (HTB), Stochastic Fair Queueing (SFQ) and Random Early Detection (RED)) of the Linux kernel.

The QueuePusher is an extension to Floodlight controller that provides Create, Read, Update, Delete (CRUD) API exposed through REST. It allows the creation of queues through the mentioned API, that in turn uses the OVSDB protocol on OpenvSwitch switch.

Finally, QoSMeter application offers QoS support at a per-flow level on the network through the usage of meters, added in version 1.3 of OpenFlow. A meter may be considered as a rate limiter. It is associated with a flow, and defines a limited number of 'meter bands' along with specific rates. Associated with each band is an action on the matched packets (of a flow).

Most of the referred applications are embedded with the respective controllers or a module that can be installed in the respective controller using the controller's internal API to manage the network and perform their function. The adaption to other controllers may involve considerable changes according to what each controller can do and the way the controller does it. No to mention that different controllers may be written in different programming languages which makes the transition of the application to other controllers quite difficult. So, despite their useful and functional mechanisms most of them faces an interoperability problem between the other SDN controller. This means that they offer a solution but only within the controller used. A more generic solution could be achieved, if the communications were performed through REST APIs to the controller, which could be a major evolution after the NB API of SDN controllers is standardized. This would allow any application to work with any SDN controller offering interoperability of any application with any SDN controller.

All of the above mentioned applications provide only one kind of functionality, either load balance or QoS enforcement, but never both. As opposed our application is able to perform both of those functionalities, although in different ways. We do not consider monitoring and routing functionality because most of these applications perform it. The majority of the QoS enforcement applications use queues however they are not able to apply fine-grained flow-based QoS enforcement. The exception is QueuePusher and QoSMeter, while the first configures queues using the OVSDB protocol the second uses *meters* introduced in version 1.3 of OpenFlow protocol. The remaining QoS enforcement applications basically perform the differentiation of traffic through the usage of pre-defined priority queues.

In short, none of the above mentioned application is so complete like our solution. In addition of providing a load balance mechanism (monitoring and enforcement), our solution is able to provide fine-grained QoS enforcement on a flow-basis through the usage of high level commands.

2.6 MANAGEMENT IN SDN

In the previous section, several network applications were discussed, being created with specific goals like QoS enforcement, monitoring, load balancing and/or routing. However, neither of the mentioned applications were able to configure the network based on reservations that, in addition on providing on request connectivity of the network also provide some of the functionality mentioned previously like bandwidth on-demand and load balance.

There are several protocols that can be used both in SDN and legacy networks that aim at easing the management process of networks such as NETCONF, RESTCONF and SNMP protocols or modeling languages such as Yet Another Next Generation (YANG). In addition, there are also several programming languages for the NB layer that aim at abstracting network applications from specific details of the controller functions such as Flow Management Language (FML) [97], Frenetic [98] or Procera [99]. These programming languages allow the creation of event-based applications and policies to express network behaviour in an easier way. However the objective is to integrate everything underneath an SDN approach which the above mentioned protocols do not have. As for the programming languages, most of the controllers have already defined their own REST API, being their prevalent choice for the NB API and thus enabling developers to write their applications in any programming language.

This way we discuss about some applications and systems that enable the management of the network in a SDN environment. The applications can be both external to the controller, or simply modules/plugins integrated in a particular SDN controller that by leveraging their internal API perform some specific functionality in the underlying network. Note that, this internal plugins may be able to communicate with external applications by exposing an API through the NB. In fact, most of the existing controllers (e.g. OpenDaylight and Floodlight) provide the ability to use and configure this module's functionalities in order to express a desired behaviour in the underlying network.

An example of such environment is the Openstack's Modular Layer 2 (ML2) plugin, depicted in Figure 2.6. Neutron (formerly known as Quantum) is the core part of OpenStack that provides an API to dynamically request and configure virtual networks. The Neutron API supports extensions to provide advanced network capabilities (e.g. QoS, ACLs, network monitoring, etc). The ML2 plugin [25] is one of its extensions and has several drivers depending on the controller that is aimed to work with (e.g. OpenDaylight, Floodlight and Ryu). In turn, each of these controllers have the Openstack Service plugin integrated in order to communicate with the ML2 plugin at OpenStack.

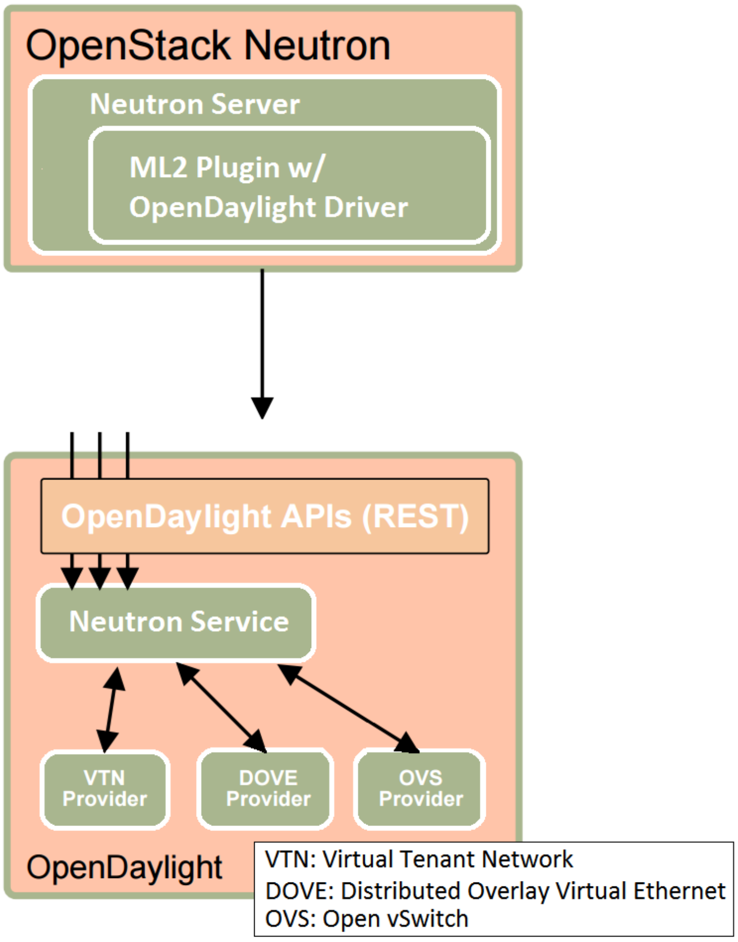


Figure 2.6: OpenStack and OpenDaylight Integration. Source : `https://www.openstack.org/assets/presentation-media/osodlatl.pdf`

The main objective of such plugins is to perform some specific behaviour on the network and expose a specific perfectly adapted API to a particular external application use it via the NB of a particular controller. In this particular case, the ML2 plugin at Openstack is basically a REST proxy that communicates with the OpenStack Service at Opendaylight, pushing a huge amount of complexity to the controller side. It enables a variety of layer 2 networking technologies to be created and managed

[25]https://wiki.openstack.org/wiki/Neutron/ML2

with less effort compared to standalone plugins. Currently it works with Open vSwitch, Linux Bridge, and Hyper-v L2 agents. The ML2 framework contains two types of drivers, type drivers and mechanism drivers. Type drivers maintain any needed type-specific network states, perform provider network validation and tenant network allocation. Mechanism drivers ensure the information established by the type driver is properly applied.

Basically, the ML2 plugin is capable of creating networks, subnets, bridges or ports in the underlying network, ensuring isolation between networks of different tenants (clients). It is able to create one or more overlay network for each tenant according to their demands. The connectivity is assured by the Openstack Service at the controller side and the segmentation of the network is performed the use of VLANs (trunked ports) or by the configuration of tunnels in OpenvSwitch using GRE [100] or VXLan [101] technologies.

Regarding QoS, two blueprints were already filed, approved and are currently under development. The first, called *LinuxQoSBridge* [26] is basically a rate limiting extension. However it only provides per-port basis traffic shaping. The second called *QoS* [27] aims at providing QoS per network, for example all the VMs/hosts belonging to a particular tenant or per port. They enable traffic categorization and to define a policy to be applied for that category. In addition to being currently under development, both of these blueprints are still unable to provide fine-grained and flow-based QoS to the network.

In short, ML2 plugin from Neutron manages the network, however in a different way. They focus on the virtualization of the network, while our application abstracts all of that, just needing to know what hosts have to communicate being the remaining necessary information retrieved from the controller. Neutron's ML2 plugin creates overlays networks for their tenants to communicate, distinguishing them by VLAN in trunk ports. However they are unable to provide QoS service to the network, being the above mentioned blueprints still under development or needing code review. In turn, our application is such an automation mechanism that given a specific tuple (IP source and destination and the QoS parameters) provides connectivity between the necessary machines on the network, a QoS on-demand and load balance optimization service abstracting whoever uses it from the complexity of how it is done and what commands are sent to the controller. It is worth to mention that the QoS enforcement is flow-based which allows a better and more fine-grained control and management of the underlying network.

CloudNaas [102] is a cloud network platform that aims at providing flexible and fine-grained control of the network by leveraging techniques such as SDN. Its architecture consists of two basic primary components, cloud controller and the network controller. The main focus is the network controller part that is responsible for monitoring and managing the configuration of the network devices according to the policies defined by the clients as well as defining the positioning of the VM within the cloud. These policies are defined using CloudNaas policy language that is used by the clients to specify the network services required by their application. They establish connectivity for the client's network through the segmentation and allocation of the network, ensuring that the client's demands are fulfilled. In addition, CloudNaas also provides a bandwidth reservation service that may be requested by the client through the definition of the policies to be applied to the network. Although it is not perfectly clear we assume that the QoS performed by the network controller of CloudNaas is flow-based. That is due to its capability of distinguishing the traffic of different VMs in the network instead of the port-based approach that Neutron QoS APIs specify in their blueprint. However, given the information available by CloudNaas in [102], we assume that it has not a "reservation" approach where clients could easily

[26]https://wiki.openstack.org/wiki/Neutron/QoSforLinuxBridge#Blueprint
[27]https://wiki.openstack.org/wiki/Neutron/QoS

reserve resources for a future period of time and their demands would be enforced correctly, during that period, in the network.

There are also several proprietary applications that provide a certain level of automation and management of the network. For example, ”Switch Light” [28] from Big Switch, ”Application Centric Infrastructure” [29] from Cisco or ”NEC ProgrammableFlow” [30] from NEC. These applications provide VM management, network configuration and management, policy-based routing among other services. However, it requires the use of their hardware and software which are not free unlike our application, or the software mentioned above and in the previous Section 2.5.

[28]http://www.bigswitch.com/products/switch-light

[29]http://www.cisco.com/c/en/us/solutions/data-center-virtualization/application-centric-infrastructure/index.html

[30]https://www.necam.com/docs/?id=67c33426-0a2b-4b87-9a7a-d3cecc14d26a

CHAPTER 3

Solution Design

The technologies and software described in Section 2 basically represents different implementations to reach a given objective. This way, it is important to corretly select the technologies in order to achieve the specific objective of this dissertation, mentioned in Section 1.2.
The objective is to develop a network application that could ease up the management process of a network (more precisely a datacenter network) in a new network paradigm, the SDN paradigm. In addition, the network application, called QLAMES (QoS and Load Balancing Automated ManagEment System), should be generic. This means that it should allow any authenticated orchestration tool or administrator to create reservations. A reservation is basically an entry which provides information to the application about: the hosts that are allowed to communicate; the QoS parameters (priority and minimum and maximum rates) and the period of time they must be active, defined by its start and end date. Each reservation would translate into the creation of flows and queues on the network, automatically. By automatically we mean: interpret the topology, find a path between the hosts specified in the reservation using a Shortest Path Algorithm (namely, Dijkstra's); configure the QoS parameters for the given reservation on the switches, using queues for that effect; and establish the communication between those hosts, using flows for that effect. All that without any outside intervention.

According to each reservation parameters, the flows are created to establish the communication between the specified hosts and the queues are created to ensure that the specified QoS parameters are enforced. After the reservation expires (end date is exceeded) the application is responsible for tearing down the communication and deleting all the flows and queues created upon the installation of that reservation. The installation of each reservation will be performed as soon as the start date is reached.

Communication between QLAMES and Opendaylight controller is done via REST requests and the network is emulated using Mininet with OpenvSwitch as its virtual switch. The architecture described above is shown in Figure 3.1.

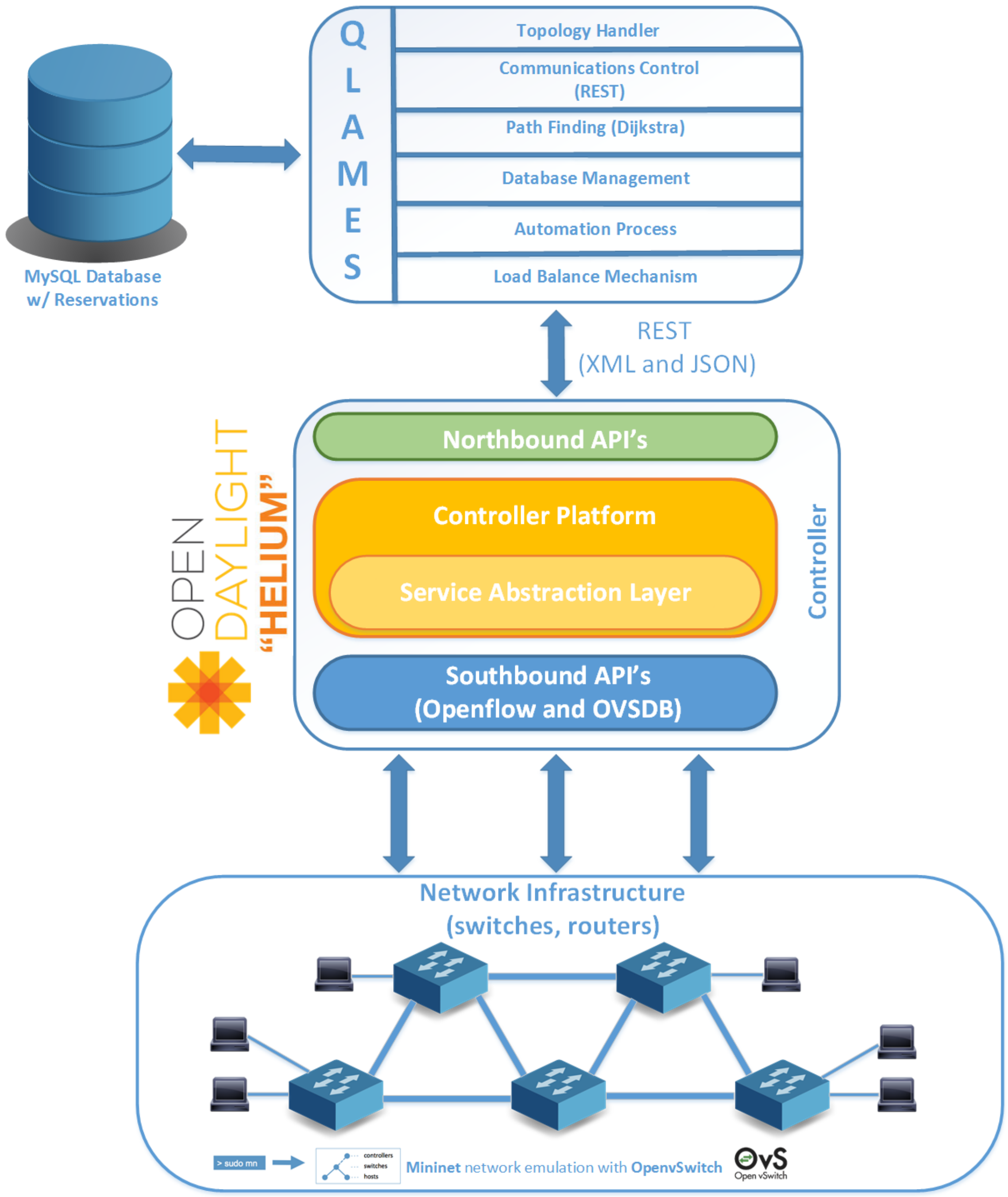


Figure 3.1: System Architecture

QLAMES eases the process of network management by interpreting the network and establishing the connection by itself without the intervention of an administrator. It just requires that the administrator or any outside tool to access its database and create the reservations, according to their needs.

The effort of NBI-WG, mentioned in Chapter 2 introduces significant application portability, which is a huge advantage [103]. When that moment arises, it makes possible the use of a network application with every SDN controller. This means that an external application such as QLAMES, would not only work with any orchestration tool, but also with every SDN controller making it even more valuable and generic.

In the course of this chapter, we will describe in more detail the technologies used as well as the implementation details of the network application, QLAMES. The rest of the chapter is organized as follows.
In Section 3.1 we will explain the decisions behind the selected technologies such as the controller (Opendaylight) and the virtual switch (OpenvSwitch). We will also briefly describe the OpenDaylight (ODL) controller, Openvswitch virtual switch and the protocols, Openflow and OVSDB.

In Section 3.2 we will briefly describe which ODL modules were installed and why. We will also describe the taken setup procedures on both ODL and Mininet. Lastly, Section 3.3 describes the several components that compose QLAMES as well as its database data model.

3.1 TOOLS OVERVIEW

3.1.1 OPENDAYLIGHT

In Section 2 several NOS were described as part of the SDN literature. However, between those controllers, Opendaylight's controller stood out. During this subsection we will briefly describe some key aspects of Opendaylight, which also serve as justification for its selection.

Opendaylight's platform was built to meet the following key requirements:

- **Flexibility** : The controller must be able to accommodate a variety of diverse applications; at the same time, controller applications should use a common framework and programming model and provide consistent APIs to their clients. This is important for troubleshooting, system integration, and for combining applications into higher-level orchestrated workflows.
- **Scale the development process** : The architecture must allow for plugins to be developed independently of each other and of the controller infrastructure, and it must support relatively short system integration times.
  OpenDaylight projects are largely autonomous, and are being developed by independent teams, with little coordination between them.
- **Run-time Extensibility**: The controller must be able to load new protocol and service/application plugins at run-time. Run-time extensibility allows the controller to adapt to network changes (new devices and/or new features) and avoids the EMS/NMS approach where each new feature in a network device results in a manual change of the device.
- **Performance and scale** : controller must be able to perform well for a variety of different loads/applications in a diverse set of environments. However, performance should not be achieved at the expense of modularity. The controller architecture should allow for horizontal scaling in clustered/cloud environments.

OpenDaylight is an open platform for network programmability to enable SDN and Network-Function Virtualization (NFV) for networks at any size and scale. Founded by industry leaders (Cisco, Brocade, NEC, IBM and others) and open to all, the OpenDaylight community is developing a common, open SDN framework consisting of code and blueprints. It was originally inspired by Beacon [65], which introduced OSGi. OSGi is a set of specifications that define a dynamic component system for Java [74]. Opendaylight's first release called 'Hydrogen' used *Equinox*, an implementation of the

OSGi core framework specification. Opendaylight's second and current stable release is called 'Helium' and the third release is 'Lithium', which has its formal release marked to June 2015.

We will focus about the second release 'Helium', because it is the more recently stable version of Opendaylight and the one used in this architecture. On the 'Helium' release, Apache Karaf's OSGi container was used. Apache Karaf [104] is a small OSGi based runtime which provides a lightweight container onto which various components and applications can be deployed. In addition of having Equinox as its OSGi core, Apache Karaf also provides a centralized logging system, remote access via SSH [105] dynamic configuration through property files and others as Figure 3.2 shows. And more importantly, it allows the user to choose which features he wants to install. This allows users to customize Opendaylight according their needs.

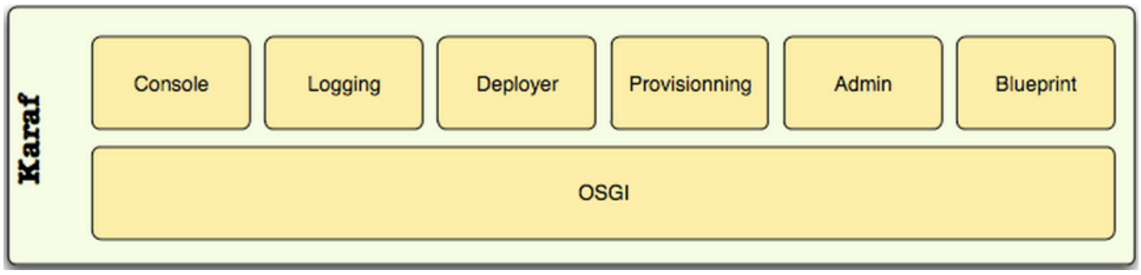


Figure 3.2: Karaf's Architecture

Similarly to other controllers, Opendaylight provides a REST NB API that establishes the connection between network applications and the controller. This way, network applications are able to manage the network through the controller according to their will and/or needs. The requests sent to the controller contain information to be stored and interpreted in order to make the appropriate changes to the network.
However, when most of the controllers operate using only OpenFlow protocol as its southbound protocol, Opendaylight's is one of the few that has a multiprotocol southbound, as it was mentioned in Section 2.3. It allows more than one protocol interface with network elements with diverse capabilities southbound from the controller, such as OpenFlow, OVSDB, BGP, SNMP and others. This aspect is particularly useful and advantageous to QLAMES, because the insertion of the flows on the switches is done using the OpenFlow protocol, while the queue configuration to enforce QoS is done via the OVSDB protocol. OVSDB protocol will be discussed in the next section.

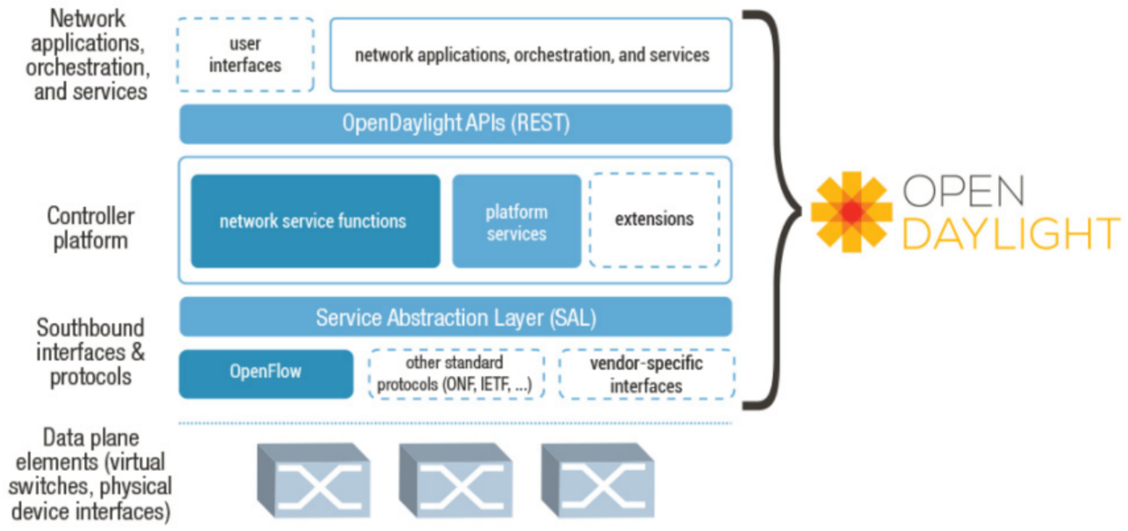

Network applications, orchestration, and services
user interfaces
network applications, orchestration, and services
OpenDaylight APIs (REST)
Controller platform
network service functions
platform services
extensions
Southbound interfaces & protocols
Service Abstraction Layer (SAL)
OpenFlow
other standard protocols (ONF, IETF, ...)
vendor-specific interfaces
Data plane elements (virtual switches, physical device interfaces)
OPEN DAYLIGHT

Figure 3.3: Opendaylight's Project Framework

This ability is due to the major innovation introduced by ODL : the Service Abstraction Layer (SAL). SAL allows ODL to support multiple protocols on the Southbound and providing consistent services for modules and applications (where the business logic is embedded) [106]. As it is shown in Figures 3.1 and 3.3, SAL separates the southbound protocol plugins from the northbound service/application plugins [75]. As far as the SAL is concerned, there is really no north or south, only producers and consumers. It is the key design that enables the abstraction of services between the services' consumers and producers. SAL can be seen as a large registry of services advertised by various modules and binds them to the applications that require them. Modules providing services (producers), can register their APIs with the registry. When an application (consumer), requests a service via a generic API, SAL is responsible for assembling the request by binding producer and consumer into a contract, brokered and serviced by SAL [107]. Basically it is a data exchange and adaptation mechanism between plugins. SAL has two architecturally different ways of implementing this registry: the Application-Driven SAL (AD-SAL) and the Module-Driven SAL (MD-SAL).

AD-SAL architecture, shown in Figure 3.4, was designed to provide abstraction through the use of a generic set of APIs that provide all device functions. In AD-SAL there is a dedicated REST API for each NB/SB plugin [108].

First a producer (normally a SB plugin) would register its services/data in order to expose its API for consumer plugins. The consumers could request the services through the SAL. The routing request between consumers and providers and data adaptations are all statically defined at compile/build time. The request routing is based on the plugin type: the SAL knows which node instance is served by which plugin, and when an NB Plugin requests an operation on a given node, the request is routed to the appropriate plugin which then routes the request to the appropriate node [108]. Optionally, SAL also provides service adaptation if a NB plugin API is different from its corresponding SB API.

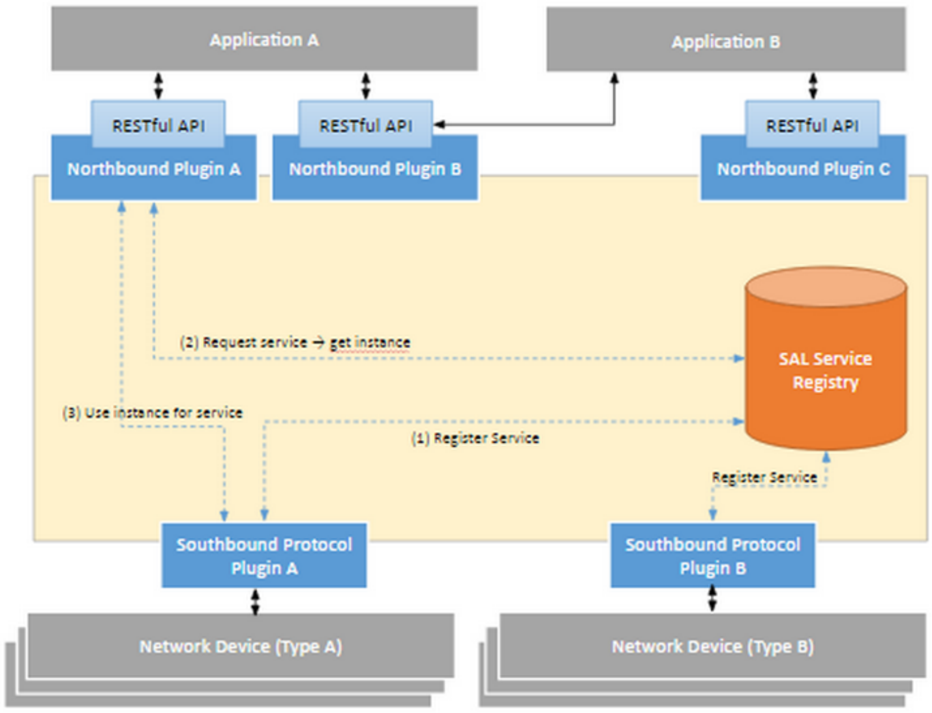


Figure 3.4: AD-SAL Architecture

The key differences lie in how the APIs are used by the provider and consumer plugins. In the case of MD-SAL, shown in Figure 3.5, the providers (generally southbound plugins) create a model containing the description of the data or services they expose. The models are in form of YANG definitions.

YANG [109] is tree-structured rather than object-oriented; data is structured into a tree and it can contain methods and properties inherited as well as complex types, such as lists and unions. In addition to data definitions, YANG supports constructs to model Remote Procedure Calls (RPCs) and Notifications.
Thereafter, a YANG compiler is used on the models created to generate the APIs. Those APIs are now part of the plugin and can now be used by every consumer plugin. MD-SAL provides a common REST API to the access of data and functions defined in the YANG models.

Last but not least important, in this architecture ODL is used as a centralized controller which may become an unique point of failure. However, ODL is a distributed controller, which allows fault tolerance in case a controller goes down, due to their consistent clustering solution. Clustering gives fine-grained redundancy and scales out while insuring network consistency. Even though in this architecture we are using ODL has a centralized controller, in the future efforts may be made to make it a distributed controller and overcome the 'unique point of failure' problem faced in this particular situation.

As we see, ODL is an SDN controller that has all the key requirements mentioned at the beggining of this subsection. For that and for all the reasons mentioned throughout this subsection Opendaylight's controller was selected to be used with our application, QLAMES.

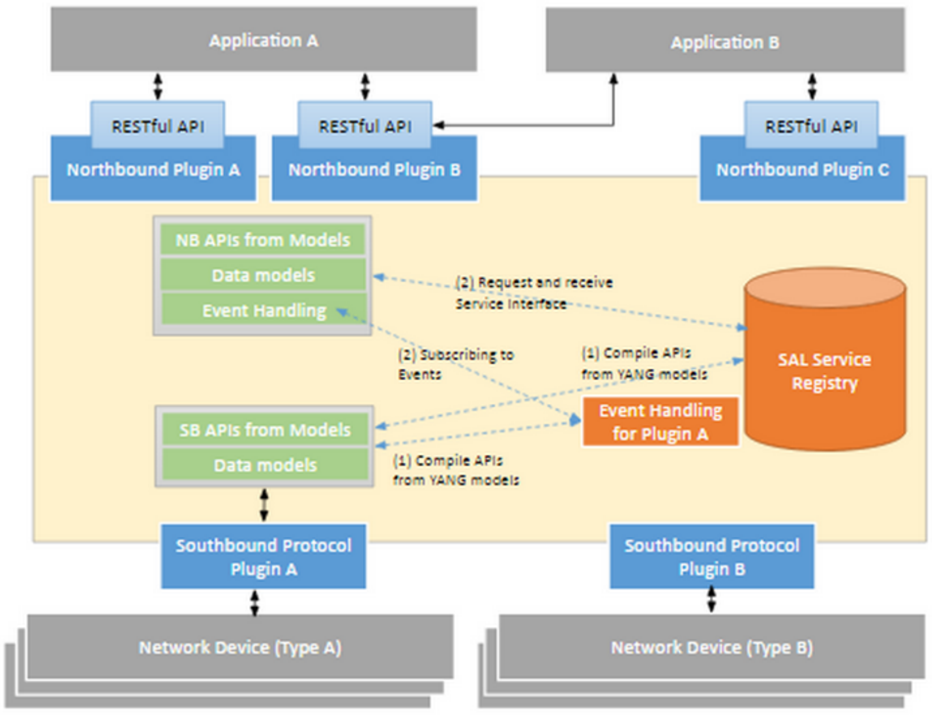


Figure 3.5: MD-SAL Architecture

3.1.2 OPENFLOW VERSIONS

The OpenFlow protocol was already described in Subsection 2.2. We will just describe very shortly the main differences between OpenFlow1.0 and OpenFlow1.3 and explain why version 1.3 was chosen. As it is shown by Figures 3.6 and 3.7 there are several differences between OpenFlow1.0 and OpenFlow1.3 versions.

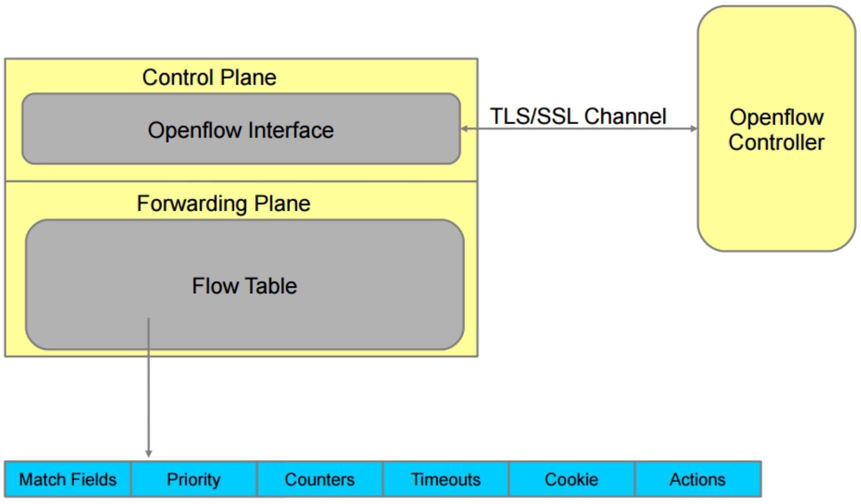


Figure 3.6: OpenFlow1.0 Forwarding Plane

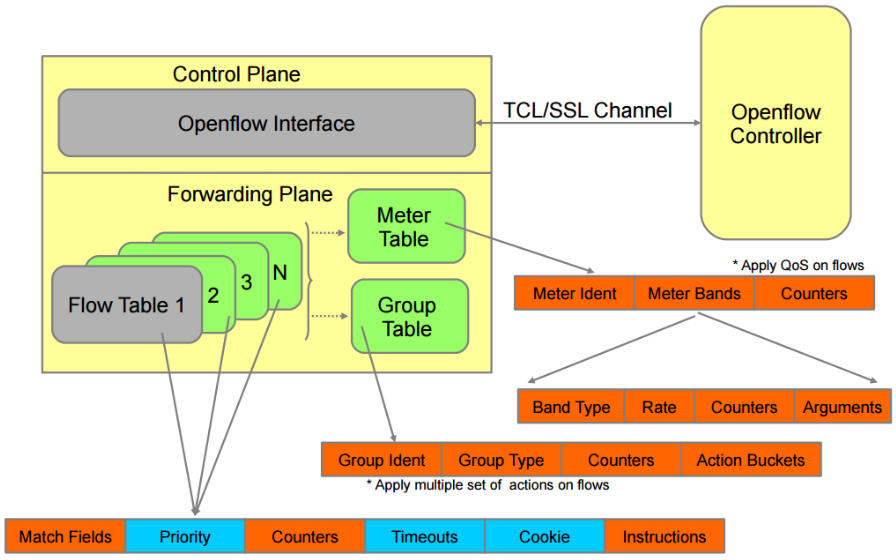


Figure 3.7: OpenFlow1.3 Forwarding Plane

Firstly the forwarding plane is quite more elaborated. The major added features in relation to the 1.0 version were the inclusion of meters, groups and several flow tables. A *Meter* defines a per-flow meter. A per flow-meter allows OpenFlow to implement various simple QoS operations, such as rate-limiting on the network [63]. On the other hand, *Groups*' purpose is to further process the packets and assign a more specific forwarding action to them. *Groups* also provide an efficient way to direct that the same set of actions must be carried out on multiple flows. In turn, as shown in Figure 3.8, the addition of several flow tables changes the current pipeline into a similar but more elaborated one. Now, instead of simply matching the packet with the highest flow entry and executing the correspondent instructions we also do the same to the other flow tables, by order. That happens unless an instruction arises with an action *go-to-table*, that may 'send' the switch to process another flow table, which implies that the flow tables with an inferior ID won't be processed.
For example if table 0 is being processed and the correspondent action for a matched packet is a *go-to-table* action which sends the switch to table 3, this means that tables 1 and 2 won't be processed trying to find a match for that packet.

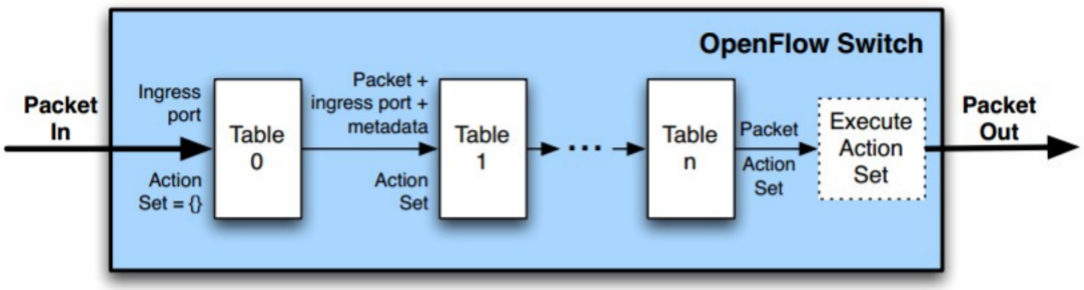
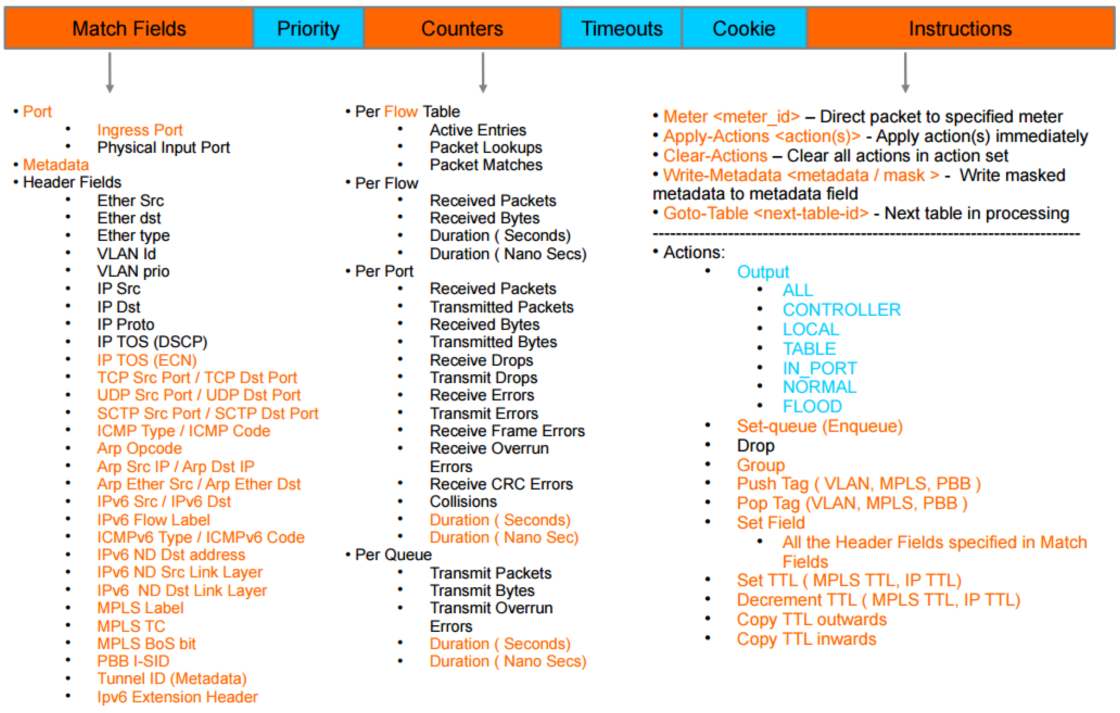


Figure 3.8: OpenFlow1.3 Pipeline

Furthermore, as shown in Figure 3.9, it includes a more extensive list of matching fields (such as TCP and User Datagram Protocol (UDP) ports) and adds the possibility to match packets also based on its metadata. In addition, it also has more detailed statistical counters, better organization of instructions as well as more instructions/actions. It is now possible to include meters, write metadata to be further used for matching packets, change all fields specified in 'Match Fields', use the group action and others.

Note that the differences shown in Figure 3.9 of OpenFlow1.3 in relation to the OpenFlow1.0 are highlighted. Orange means new additions while blue means amendments.



Match Fields	Priority	Counters	Timeouts	Cookie	Instructions

- Port
  - Ingress Port
  - Physical Input Port
- Metadata
- Header Fields
  - Ether Src
  - Ether dst
  - Ether type
  - VLAN Id
  - VLAN prio
  - IP Src
  - IP Dst
  - IP Proto
  - IP TOS (DSCP)
  - IP TOS (ECN)
  - TCP Src Port / TCP Dst Port
  - UDP Src Port / UDP Dst Port
  - SCTP Src Port / SCTP Dst Port
  - ICMP Type / ICMP Code
  - Arp Opcode
  - Arp Src IP / Arp Dst IP
  - Arp Ether Src / Arp Ether Dst
  - IPv6 Src / IPv6 Dst
  - IPv6 Flow Label
  - ICMPv6 Type / ICMPv6 Code
  - IPv6 ND Dst address
  - IPv6 ND Src Link Layer
  - IPv6 ND Dst Link Layer
  - MPLS Label
  - MPLS TC
  - MPLS BoS bit
  - PBB I-SID
  - Tunnel ID (Metadata)
  - Ipv6 Extension Header

- Per Flow Table
  - Active Entries
  - Packet Lookups
  - Packet Matches
- Per Flow
  - Received Packets
  - Received Bytes
  - Duration (Seconds)
  - Duration (Nano Secs)
- Per Port
  - Received Packets
  - Transmitted Packets
  - Received Bytes
  - Transmitted Bytes
  - Receive Drops
  - Transmit Drops
  - Receive Errors
  - Transmit Errors
  - Receive Frame Errors
  - Receive Overrun Errors
  - Receive CRC Errors
  - Collisions
  - Duration (Seconds)
  - Duration (Nano Sec)
- Per Queue
  - Transmit Packets
  - Transmit Bytes
  - Transmit Overrun Errors
  - Duration (Seconds)
  - Duration (Nano Secs)

- Meter <meter_id> – Direct packet to specified meter
- Apply-Actions <action(s)> - Apply action(s) immediately
- Clear-Actions – Clear all actions in action set
- Write-Metadata <metadata / mask > - Write masked metadata to metadata field
- Goto-Table <next-table-id> - Next table in processing

- Actions:
  - Output
    - ALL
    - CONTROLLER
    - LOCAL
    - TABLE
    - IN_PORT
    - NORMAL
    - FLOOD
  - Set-queue (Enqueue)
  - Drop
  - Group
  - Push Tag (VLAN, MPLS, PBB)
  - Pop Tag (VLAN, MPLS, PBB)
  - Set Field
    - All the Header Fields specified in Match Fields
  - Set TTL (MPLS TTL, IP TTL)
  - Decrement TTL (MPLS TTL, IP TTL)
  - Copy TTL outwards
  - Copy TTL inwards

Figure 3.9: OpenFlow1.3 Flow Entry

In short, OpenFlow1.3 enables a more fine-grained control of the network. Even though this version was not absolutely necessary for development of this application it may be improved in the future and use this new features mentioned above in order to expand its functionality.

3.1.3 OPENVSWITCH AND OPENVSWITCH DATABASE MANAGEMENT PROTOCOL

After selecting the controller (ODL) to be used, the time to chose the more appropriate openflow-enabled switch implementation arose. Bearing in mind that there was a need to implement QoS on the network, we had to choose the switch according to that fact.There were two possibilities on the table: OpenvSwitch [110] or Ofsoftswitch13 [82].

OpenvSwitch is a production quality open source openflow capable software switch designed to be used as a vswitch in virtualized server environments. OpenvSwitch is tipically used with hypervisors to interconnect virtual machines within a host and virtual machines between different hosts across networks. Ofsoftswitch13 is an OpenFlow 1.3 compatible user-space software switch implementation.

Both of these switches have the possibility to incorporate QoS on the networks. The first, OpenvSwitch could accomplish it by the use of linux queues (further explained in this Subsection), while the second could accomplish it by using the meters mentioned in the previous Subsection 3.1.2.

OpenvSwitch is the software switch more widely used by the comunity. It has great performance because it provides, not only an userspace mode but also a kernel mode, as it is shown in Figure 3.10, being favourable to be used in production environments. It includes many features such as fine-grained QoS control, IPv6 support, multiple tunneling protocols, VLANs, and others. In addition, it can also replace Linux bridge.

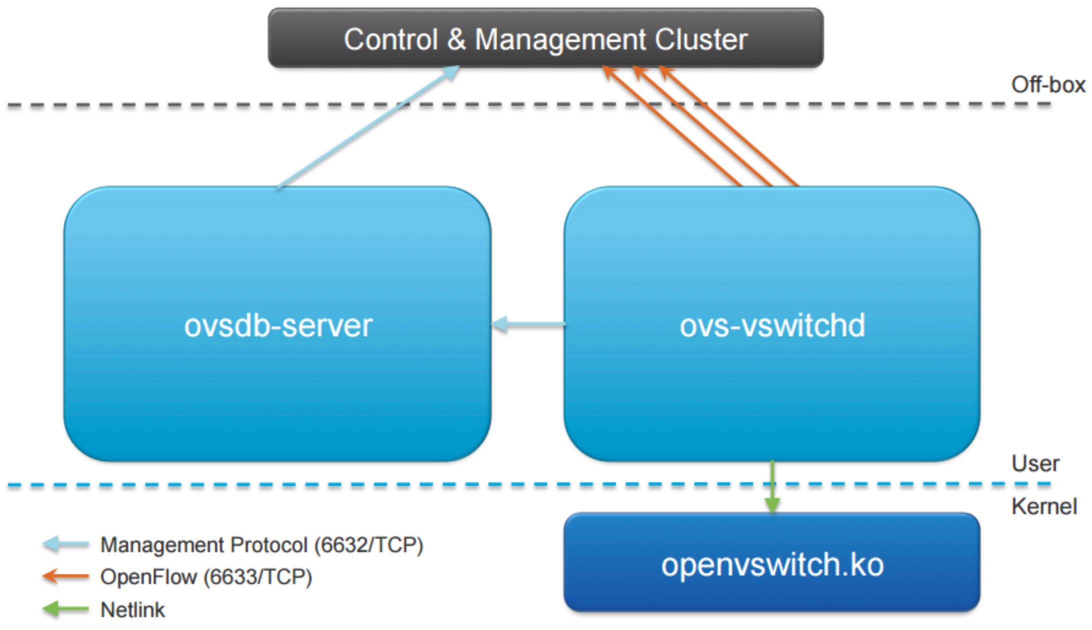


Figure 3.10: OpenvSwitch Architecture

Hypervisors need the ability to bridge traffic between VMs and with the outside world. On Linux-based hypervisors, this means the Linux bridge, which is fast and reliable. However Open vSwitch is aimed at multi-server virtualization deployments, a scenario for which the Linux Bridge is not well suited. These environments are often characterized by highly dynamic end-points, the maintenance of logical abstractions, and (sometimes) integration with or offloading to special purpose switching hardware. [111].

OpenvSwitch's ability to replace Linux Bridge is due to some of its characteristics and design considerations mentioned in [111], such as:

- **The mobility of state -** OpenvSwitch has support for configuring and migrating all network state associated with a network between different hosts. This may include a L2 learning table, L3 forwarding state, ACLs, QoS policy, monitoring configuration, etc.
- **Responding to networks dynamics -** OpenvSwitch has several features such as OVSDB and Openflow which allow it to monitor link-state information and quickly cope with those changes using OVSDB support for remote triggers.

- **Maintenance of logical tags -** OpenvSwitch has the capability to specify, maintain and store the tagging rules in a correct and optimized form. All the rules are accessible to a remote process for orchestration.
- **Hardware integration -** OpenvSwitch's forwarding path (the in-kernel datapath) is designed to be able to "offload" packet processing to hardware chipsets, which brings the advantage of a more performant switch but also a central mechanism for automated network control.

Even though Ofsoftswitch13 implements almost all OpenFlow1.3 features (unlike OpenvSwitch that for example does not support meters) it has a more limited number of features, is less performant (only has userspace mode) and cannot replace Linux Bridging. In addition, meters can only define a maximum rate, while Linux's queues can define a minimum rate, a maximum rate and a priority as well. For all the reasons mentioned above, OpenvSwitch was the selected virtual switch to be used on this architecture.

After the selection of the switch, the time now arises to explain how the so mentioned 'Linux queues' will be configured on the switches. The answer is simple, by using OVSDB protocol available on ODL as well as mentioned in previous Subsection 3.1.1. OpenvSwitch's architecture apart from the kernel module OpenvSwitch's implementation also contains a daemon (*ovs-vswitchd*), a database server (*ovsdb-server*) and a *Control and Management Cluster*.

The first, *ovs-vswitchd*, is the core component of the system. It communicates with the outside world (Controller) using OpenFlow, with the *ovsdb-server* module using management protocol (OVSDB), with the kernel over netlink and with the system through the netdev abstract interface. Basically it is a daemon that manages and controls any number of Open vSwitch switches on the local machine. Meanwhile, the second is a JavaScript Object Notation (JSON) database that holds switch level configuration such as queue configuration and the last contains managers and controllers that use the OVSDB Protocol to supply configuration information to the switch database server.

Using the OVSDB Protocol, managers (for example, ODL controller) can specify the number of individual virtual bridges within an Open vSwitch implementation and create, configure and delete ports and tunnels from a bridge through JSON-RPC (remote procedure calls). A manager can also create, configure and delete queues. So, OVS will apply associated actions to packets using the *vswitchd* daemon, which in turn reads the configuration parameters from OVS database *ovsdb-server*. This means that queues can be configured by writing its configuration, in the appropriate format, in the OVS database. Which, in turn, means that because ODL supports the OVSDB protocol (due to its OVSDB southbound plugin), it can communicate with the OVS database and provide it the correspondent information of the queues configuration. Lastly but not least, because Opendaylight has a REST NB API for OVSDB, an external network application, such as QLAMES, can send the queue configuration information all the way back into OVS database. Basically, in an abstract way, the queue configuration information goes all the way down from QLAMES into OVS database to be later enforced by OVS core module *vswitchd*.

Note that, before a queue can be associated with a port, a *QoS* row in OVS database has to be created. Each *QoS* row corresponds to a given port in a particular switch. In turn, a QoS row may contain several queues.

In short, when a *QoS* row is created, it is associated with a particular port of a particular switch. Then, one or more Queues are created and finally associated with a correspondent *QoS row.* We have now a queue configured on a particular port of a given switch.

Note that all *Port*, *QoS* or *Queue* rows have an unique identifier associated. Also 'Queue' or 'Queue row' are used interchangeably. Finally, only remains associate the queues with the correspondent flows using the *set-queue* action, available in OpenFlow1.3, as it is shown in Figure 3.9 in the previous Subsection 3.1.2. Further details will be discussed respectively on Subsection 3.3.3.

It is worth to mention that, the queueing discipline used is the Hierarchical Token Bucket. A queue discipline (qdisc) is a scheduler, which in turn arranges and/or rearranges packets for output. Every output interface needs a scheduler of some kind, and the default scheduler is a First in First out (FIFO). Other classfull qdiscs are available under Linux such as Class Based Queuing (Class-based Queueing (CBQ)), *PRIO* a priority scheduler and Hierarchical Fair Service Curve (HFSC).

Both CBQ and HTB help you to control the use of the outbound bandwidth on a given link. Both allow you to use one physical link to simulate several slower links and to send different kinds of traffic on different simulated links.
Unlike CBQ, HTB shapes traffic based on the Token Bucket Filter algorithm which does not depend on interface characteristics and so does not need to know the underlying bandwidth of the outgoing interface [112]. In one hand as HTB's configuration gets more complex, its configuration scales well. On the other hand, CBQ it is already complex even in simple cases.

HTB is well suited for setups where you have a fixed amount of bandwidth which you want to divide for different purposes, giving each purpose a guaranteed bandwidth, with the possibility of specifying how much bandwidth can be borrowed. Finally, according to D. Gabriel et al. [113], HTB 'provided eloquent results which can prove that HTB implementations can fully satisfy any complex QoS requirements'. For the reasons meantioned above, HTB was the chosen queuing discipline.

3.1.4 MININET

Mininet was used as the environment for prototyping our system. Mininet [114] is a system for rapidly prototyping networks for Software-defined networks on a single machine. It is basically a network emulation software that allows the launch of a virtual network with switches, hosts and a SDN controller all with a single command.

Mininet was chosen due to the following advantages [115]:

- **Flexibility** - custom topologies and functionality can be defined by software, using familiar languages.
- **Interactivity** - managing and running the network occurs in real-time, as if interacting with a real network.
- **Scalability** - the prototyping environment enables the creation of dozens or hundreds of switches on a single machine.
- **Ease of use** - easy creation and interaction with the network using the Mininet CLI (and API).

In addition, Mininet already had the necessary tools for interoperating with the controller and already had OpenvSwitch installed. Finally, using its Python API [116] we can create a customized topology in order to test our application.

3.2 SETUP CONFIGURATION PARAMETERS

In order to assemble together all the modules from the project's architecture, shown in Figure 3.1, some configurations must take place in some modules such as Opendaylight, OpenvSwitch and Mininet. Figure A.1 of Appendix-Sequence of installation procedures depicts the sequence of installation procedures to setup the architecture scenario.

3.2.1 OPENDAYLIGHT

Since the 'Helium' version does not include all packages by default, as it was mentioned in the previous Section 3.1.1, we have to choose and install which modules we want the controller to have. The installation process is done via karaf's CLI, (see Appendix-ODL's module installation). The modules/packages were chosen and installed following 'Opendaylight's Installation Guide' [117]. The modules/packages installed were the following:

- **RESTCONF API Support (*odl-restconf*)** - Enables REST API access to the MD-SAL including the data store.
- **OpenFlow Flow Programming (*odl-openflowplugin-flow-services-ui*)** - Enables discovery and control of OpenFlow switches and the topology between them.
- **L2switch (*odl-l2switch-switch*)** - Provides L2 (Ethernet) forwarding across connected OpenFlow switches and support for host tracking.
- **DLUX (*odl-dlux-core*)** - Automatically installed with 'L2Switch' or 'OpenFlow Flow Programming'. Enables a web interface to draw information from the 'OpenFlow Flow Programming' and 'L2 Switch' components to display information about the topology of the network, flow statistics, host locations and others.
- **OVSDB (*odl-ovsdb-all*)** - Enable OVS management using OVSDB plugin and its associated OVSDB NB APIs.
- **Documentation (*odl-mdsal-apidocs*)** - Enables access to API documentation of MD-SAL.

After that, since we do not want the network to have connectivity, until explicit flows are installed due to reservations we have to change two *L2Switch* configuration files. The configuration files are located in 'etc/opendaylight/karaf'. To disable the communication at start we have to change *arphandler* and *l2switch* main configuration files. In the first file we have to disable the *is-proactive-flood-mode* flag in order to avoid the installation of ARP flows. That's because it would not install flows matching only ARP packets which would flood the traffic through the network. In the second file we have to enable the *is-learning-only-mode* flag in order to the controller not install MAC-To-MAC flows automatically.

As we mentioned in the previous Subsection 3.1.2 the OpenFlow version selected was the 1.3 version. This way we have to instruct both ODL and OVSDB plugins that the language to be spoken is OpenFlow1.3. This is done separately and in different ways.

OVSDB plugin is informed through a configuration file where we specify that the OVSDB protocol will use OpenFlow protocol version 1.3 (refer to Appendix-OVSDB's Configuration to OpenFlowv1.3). Finally, the controller is instructed that it will use version 1.3 of the OpenFlow protocol, by adding the flag 'of13' at the time the launch (refer to Appendix-Controller's Configuration to OpenFlowv1.3).

3.2.2 MININET

After the controller is launched, it is time to load the topology in Mininet. Due to the fact that the topology is not a standard, pre-defined one from Mininet's libraries we need to go through two more steps.

First, we need to design the topology of our network. This is done using Mininet's Python API available in [116]. The topology created, that was also used in the test scenarios Chapter 4, was defined by a Python program define in Appendix-Mininet's Custom Topology.

Second and last, since it was a custom topology, the launch command was slightly different. In addition the command also informs Mininet if the controller is local or remote, if it is remote specifies its IP address and listening port, which switch is using and which OpenFlow version is being used (refer to Appendix-Topology's Launch command). Further details on the command will be described on Chapter 4. Note that, Mininet's Virtual Machine already includes a version of Openvswitch (2.0.3). Due to some problems with the configuration of the queues that version was uninstalled. The problem was that the queue was being configured but it wasn't being applied, which means that all the bandwidth was being given for all flows. Basically we were having a best effort network instead of a network with differentiated traffic between the some hosts.
For that reason, version 2.3.1 of OpenvSwitch was installed, which solved the problem just mentioned.

3.3 SYSTEM STRUCTURE

The system (QLAMES), as we may see in Figure 3.1, is essentially composed by six main modules:

- Topology Handler
- Database Management
- Communication Control (REST)
- Path Finding Dijkstra
- Automatization Process
- Load Balance mechanism

Each one of these will be respectively explained in detail throughout this section.

3.3.1 TOPOLOGY HANDLER

First of all, bearing in mind that all configuration details are handled, in order to start building the application it is essential to extract information about the topology and store it into the appropriate data structure. After that is accomplished, we need to make sure that the data structure has the proper methods to insert/retrieve information about the topology. Thenceforth we are able to work with that information in order to accomplish the application's goals, such as enable communication between hosts and provide QoS to the network.
The first point is being able to extract and interpret information. ODL's controller has a wide variety of NB API's that enable the insertion/retrieval of information to/from the controller to, for example,

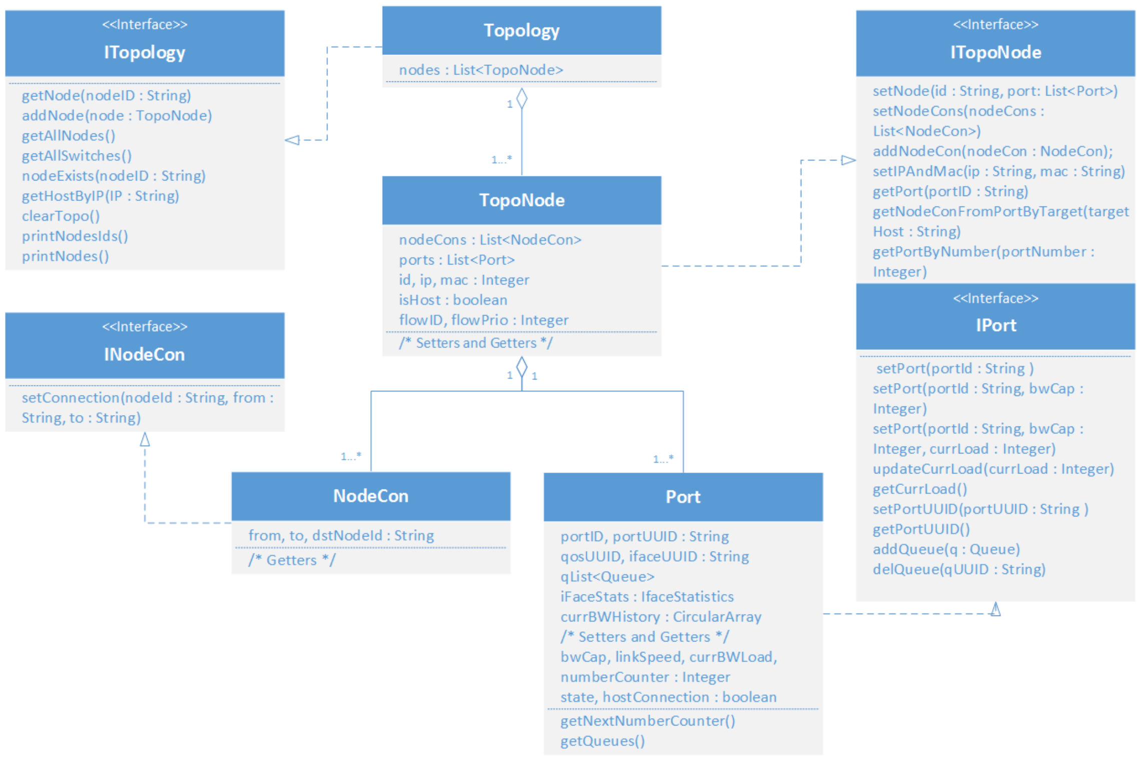
an outside application, such as QLAMES. This way, we issued an authenticated REST request to the controller in order to retrieve the topology's information from the operational data store.

In this case, the controller would accept a request in either Extensible Markup Language (XML) or JSON. The XML format was chosen for REST requests regarding topology and flow operations with the controller. In order to issue the request, a class, named *myXML*, was created with the objective to handle those requests. By handle we mean several operations regarding XML REST requests since creating the request, using the appropriate REST URL according to the situation in question, open the communication channel and sending the request. All these details regarding REST communication will be further explained in Subsection 3.3.3.

Basically, as far as this module is concerned, it only asks about topology information, parses the received information and stores the interpreted information into its data structure. The process of retrieving the topology information performed by the *Topology Handler* module is also described in Appendix-Topology Handler.

To request the topology information it uses the *sendGet* method from *myXML* class identifying what it is requesting (in this case, the topology) using a constant for that effect. The controller then answers with an XML response containing the topology's information. A parser was created for the purpose of decoding and storing the interpreted information into the our topology data structure. Note that, the first thing done here is to request information about the topology, interpret it and store it to be later used by QLAMES.

At this point, the topology's data structure has already been defined, as well as some basic



<<Interface>>
ITopology
getNode(nodeID : String)
addNode(node : TopoNode)
getAllNodes()
getAllSwitches()
nodeExists(nodeID : String)
getHostByIP(IP : String)
clearTopo()
printNodesIds()
printNodes()
Topology
nodes : List<TopoNode>
1
1...*
TopoNode
nodeCons : List<NodeCon>
ports : List<Port>
id, ip, mac : Integer
isHost : boolean
flowID, flowPrio : Integer
/* Setters and Getters */
<<Interface>>
ITopoNode
setNode(id : String, port: List<Port>)
setNodeCons(nodeCons : List<NodeCon>)
addNodeCon(nodeCon : NodeCon);
setIPAndMac(ip : String, mac : String)
getPort(portID : String)
getNodeConFromPortByTarget(target Host : String)
getPortByNumber(portNumber : Integer)
<<Interface>>
INodeCon
setConnection(nodeId : String, from : String, to : String)
1
1
1...*
1...*
NodeCon
from, to, dstNodeId : String
/* Getters */
Port
portID, portUUID : String
qosUUID, ifaceUUID : String
qList<Queue>
iFaceStats : IfaceStatistics
currBWHistory : CircularArray
/* Setters and Getters */
bwCap, linkSpeed, currBWLoad, numberCounter : Integer
state, hostConnection : boolean
getNextNumberCounter()
getQueues()
<<Interface>>
IPort
setPort(portId : String)
setPort(portId : String, bwCap : Integer)
setPort(portId : String, bwCap : Integer, currLoad : Integer)
updateCurrLoad(currLoad : Integer)
getCurrLoad()
setPortUUID(portUUID : String)
getPortUUID()
addQueue(q : Queue)
delQueue(qUUID : String)

Figure 3.11: Topology data structure class diagram

methods on the data structure in order to create the graph that would contain all necessary topology information. Figure 3.11 shows the topology's data structure class diagram.

As we see, the *Topology* is basically a list of nodes (switches and hosts), which in turn, contain a list of node connections (*nodeCon*), a list of ports and some additional attributes to identify the node and provide some extra relevant information about it.

A node connection is associated with a topology node, but is basically a link. It has information about the source port, destination port and destination node to where it is connected.

A *Port* has all the relevant information about port, QoS and interface *Universally Unique Identifier (UUID)s*; The first two relevant for defining queues and the last for retrieving statistical information about an interface such as the number of transmitted or received bytes and others. It also has the mapping of all queues associated to its own port as well as some extra fields relevant later for load balance mechanism such as the *linkSpeed*, *hostConnection* flag, or the *currBWHistory* containing the transmit rate of the previous moment of monitoring.

All these classes have associated with it several operations defined by its interfaces. The operations may be simple basic methods or more particular and elaborated methods. The first ones are essential methods, useful to create the topology graph, like add nodes to the topology, add node connections or ports to the topology nodes. The latter ones are more precise and particular methods created to provide some sort of particular information about the network's topology for decision making in some other part of the application, this means in some other module. For example, find a particular node given its ID to thenceforward obtain information about to where it is connected or its identification. It also can be used to find a particular port of a given node in order to add a queue due to a reservation that came into effect, and many other examples.

In short, an appropriate data structure for the topology was created, as well as all the necessary methods to insert/retrieve information from it. This information can and will be later used for other modules for their own particular objectives.

3.3.2 DATABASE MANAGEMENT

A central piece in our application is the database. There it is stored all the relevant information that allows the application to identify which hosts have to communicate, its QoS parameters and the period of time they are authorized to communicate.
Through these entries, for example a network administrator, can provide the application with enough information in order to automatically enable the communication between the specified hosts; the communication is only allowed inbetween the stipulated dates and the communication will be provided with the required QoS parameters specified on the correspondent entry. All this important information mentioned above is stored in a table called *Reservation*.

A set of other tables are also part of the database but its functionally is different. Those tables are the *FlowMap* and *QoSMap*. The database's structure is depicted in Figures 3.12 and 3.13.

So, basically, the database has 3 entities. The first and most important, the *Reservation* entity providing the application with all the decision and configuration parameters needed. The decision parameters are the dates in between the reservation must be applied. Meanwhile, the configuration parameters are the parameters needed to configure the flows that enable communication and the queues that enable differentiated service. For example the hosts IPs are useful for identifying the hosts

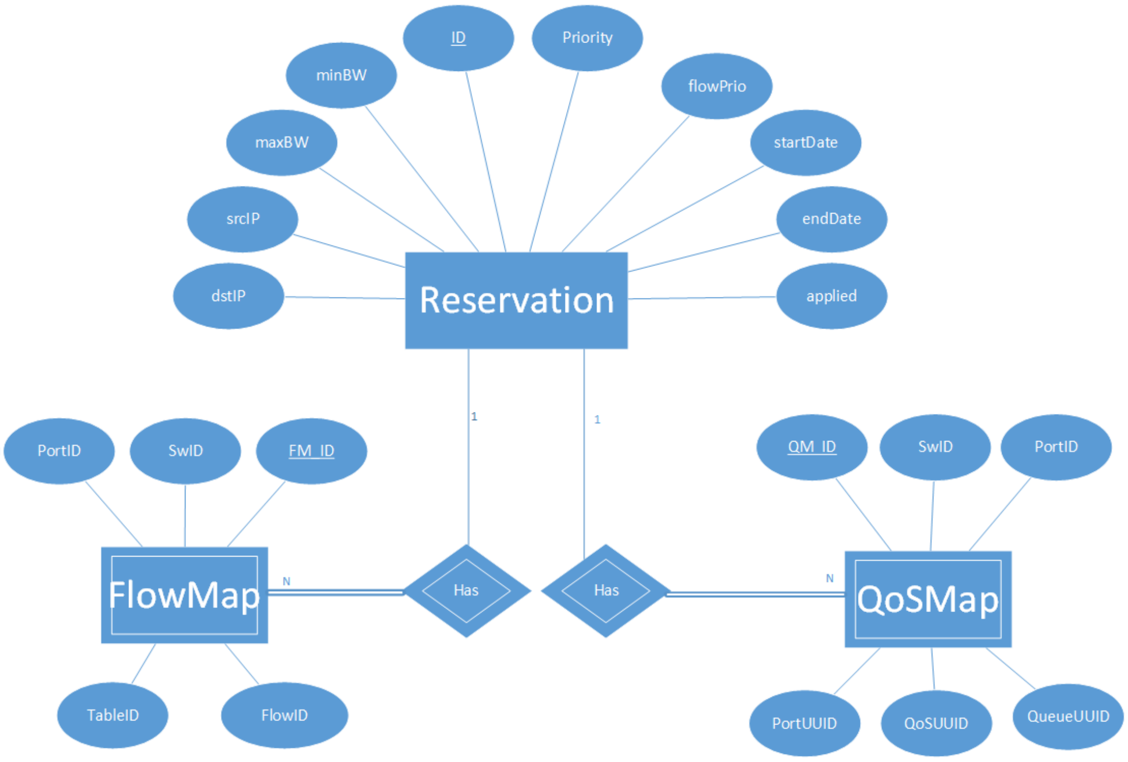


Figure 3.12: Database Entity Relation Diagram

that have to communicate and the minimum rate, maximum rate and priority are useful to configure the queues.

The other two entities *FlowMap* and *QoSMap* were created with the objective of simplifying the deletion process of the flows and queues from the switches, respectively.

The first stores all the information needed to delete a flow, for example in which switch it is installed, what is its table and flow IDs and to which reservation it is associated. This way, deleting a flow is simple as assembling this information into a proper DELETE REST request and send it to the controller, by using *MyXML* class furtherly explained in SubSection 3.3.3.

The same goes for the latter, *QoSMap*. However, to delete a queue more information is needed. In addition to the switch, port identification and associated reservation ID, it is also required to know the *UUIDs* of the *port*, *qos* and *queue* rows. The need for extra information is due to the fact that here we are using OVSDBs NB API. As we will see further in Subsection 3.3.3 all the aspects related to the queues we will use a different NB API where the REST requests' format are in JSON.

Considering this, when a reservation expires the process of obtaining the information required for the deletion of the flows and queues is trivial. By using the ID of the expired reservation, we obtain all the flow information from the *FlowMap* table which is required to delete all the flows on the switches related to that reservation. The same goes for the queues, using the expired reservation's ID, we can obtain all the necessary information in order to delete all the queues associated with that reservation from the network switches. After all queues and flows are deleted, the associated information and the reservation itself is deleted from the database.

Finally, the database if constantly polled in order to keep the network always up-to-date. This

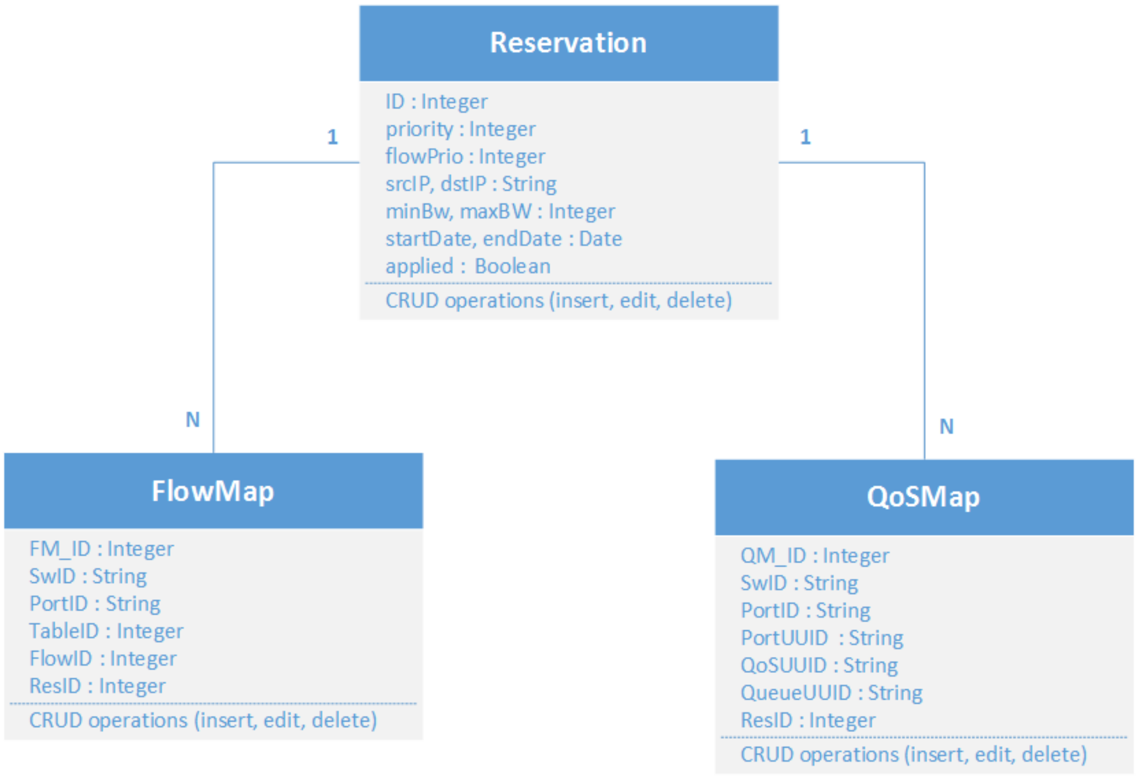


Figure 3.13: Database Class Diagram

means deleting the entries that are no longer valid and process the new ones with the minimum delay possible.
The workflow of an entry to be deleted, also mentioned above in this Subsection, is also depicted in Appendix-Database Management.

3.3.3 COMMUNICATION CONTROL

The communications between the external application, QLAMES, and the ODL controller will be all performed by REST. Through the usage of REST requests the application will be able to retrieve and insert information from/to the controller, as mentioned in the beggining of the Chapter 3. This module is essentially divided into two classes :

The first is related to the *MyXML* class, that is responsible for constructing the XML requests for flow related operations. Operations like retrieve the topology information or add/remove a given flow.

The second is related to the *MyJson* class, that is responsible for all the operations related with OVSDB. This means operations like adding a qos row, associating/dissociating a queue to/of a *qos row* or creating/deleting a queue (this means inserting into *ovsdb-server* the queue's configuration information).

Before passing to the description of both of these cardinal classes that compose the *Communication Control* module there are also two auxiliary classes that simplify the action of these two classes. They are the *Constants* file and *REST URLs* file.

The first provides to the whole application with a pre-defined set of constants. This constants identify HTTP error codes, the types of XML and JSON requests, ethernet types and OpenFlow1.3 logical ports, (refer to Appendix-Communication Control).

The second provides *MyXML* and *MyJson* classes the pre-defined REST URLs for all the operations required by these classes. It has the necessary REST URLs for deleting (a flow, a table, a queue, a qos row), for inserting (a flow, a queue, a qos row, and others) and for retrieving information (topology, table, flow, interface information and others). Note that, all these requests have to be furtherly changed in order to execute the operation to the correct entity. For example, if a flow is to be inserted, the *nodeID (switchID)* where it has to be added as well as the *tableID* must be correctly specified and included on the URL. The process is the same for the addition of a *queue*, however the link is obviously different and we have only to specify the OVSID.
The OVSID is an field that informs where the connection manager of OVS is running, this means its IP address and port number (for example, '192.168.57.102:54582').

Since the REST URLs are generic, a set of methods were created in order to modify those *Strings* and insert the proper information into the correct place. Obviously, dependending on the request type (*putFlow*, *getInterfaces*, *putQos*, refer to Appendix-class 'REST URLs') and its input parameters the methods vary.

This auxiliary classes are an important support for the cardinal ones as they abstract and simplify the work for them. Basically provide them a modular functionality that simplifies the code.

This Subsection was not divided into two because, despite of being different, both of these cardinal classes have the same abstract objective and workflow. The differences don't impact on the workflows followed by the two of them.

The most notorious difference, apart from the fact that one is in JSON format and the other in XML, is the fact that one has the objective to communicate with a NB API that provides topology information and means to insert/remove flows. The other communicates with another NB API (OVSDB's API) that allows the retrieval of interface information (for statistical effects) and the insertion/deletion of queues which may be used to enforce QoS on the network. However that separation is already abstracted by *BaseURLs* and *Constants* classes that according to the constant type identifies which type of information is to be inserted/requested/deleted. The operation to be performed is identified by the method, this means that each method is associated with a particular operation to be performed (GET, PUT, DELETE or POST).

There are also the methods for constructing the REST request. In spite of some operation like GET OR DELETE not requiring a body other operation like PUT or POST do require. For example, in the case of *MyXML* a *createFlow* method is required in order to construct the REST request to add the flow to the network. This method acts the same way the the ones that replaced some information from the *BaseURLs* class. The base structure of the request is already pre-defined and in the arguments a data structure named *FlowConfig* is passed containing all the necessary information needed to compose the REST request correctly. The same goes for the methods *createQueue* or *createQos* present in *MyJson* class. However on this case the QosConfig data structure is used, because as we mentioned before on this subsection the information required for queues is different to the one required for flows.

It is noteworthy to mention that, before any operation is performed, the credentials are stored in its internal data structure in order to authenticate correctly every REST request sent to the controller.

In Figure A.5 of Appendix-Communication Control a class diagram is depicted identifying the methods used by the cardinal classes. In turn, Figure A.4 of Appendix-Communication Control shows the data structures *FlowConfig* and *QosConfig* also used by the above mentioned classes.

In Figure A.6 of Appendix-Communication Control the general workflow for adding a flow/queue (*putFlow*/*putQueue*) requested by the *Automation Process* module is depicted. We used the same example for adding a Queue and a Flow because apart from the fact that both will communicate with different NB API from the controller and send different information the workflow is basically the same as it as previously mentioned on this subsection.

3.3.4 PATH DISCOVERY - DIJKSTRA

Another fundamental piece of our application is the *Path Discovery.* We are provided, in each reservation, with the information of IP addresses that have to communicate. In spite of knowing where the hosts are we have to find a path between them and automatically create the flows on the appropriate switches that in turn will enable the communication. That is done using a Dijkstra algortihm, to find the shortest path between the specified hosts. The base of the algorithm was obtained here [118]. That algorithm was then slightly modified to fit to our application's needs.

The data structure of the Dikjstra module is divided in two parts :

- The data structure, constituted by classes *Vertex* and *Edge*, (refer to Appendix-Path Discovery (Dikjstra), Figure A.7)
- The operations performed to the data structure, such as *createGraph*, *computePaths* and others, (refer to Appendix-Path Discovery (Dikjstra), Figure A.8).

The first thing that needs to be done is to create the Dikjstra's graph using the topology information. The method responsible for doing that is method *createGraph.* It basically runs across all nodes, adding the topology nodes has *Vertices.* Thereupon it runs accross all the node connections of each node and add it into the graph as the *Edges.*

After the creation of the graph, we are now able to find the shortest path between two hosts using the method *findPath.* On that method the two hosts are searched in the data structure. Thereupon being found, all the paths from the source node are calculated and then the shortest path for the destination node is retrieved. This is done by using the internal methods *computePaths* and *getShortestPathTo* respectively.

Finally the paths obtained are decoded into a data structure for simplifying the process of creating the flows on the network. This is done using method *decodeVertexPath* that received the path and the source and destination nodes and generates two customized lists, the *parsed path* and the *reversed parsed path* so the communication can be done both ways. This customized lists are in actually a new data structure created, (refer to Appendix-Path Discovery (Dikjstra), Figure A.8).

As it was shown in the previous referenced figure, each object of the list only contains the *switchID* and the *portID* to where the traffic must be redirected. This eases the process of creating the flows because we only need to iterate on these lists and send the PUT REST requests with that information and the flows are inserted to the controller and thus into the network switches.

Note that, everytime a path is calculated for the first time it is stored in an auxiliary data structure called *DC_Paths*, (refer to Appendix-Path Discovery (Dikjstra), Figure A.8). This is important because, before calculating a new path we first check if that path was already calculated. If it was, we will use it, otherwise we will calculate it. This mechanism may make a difference in terms of performance, specially if the network is composed by a high number of switches with many operational links.

Another point worth to mention is that when we are parsing the paths, the end nodes are excluded. The reason behind that is because they are hosts, and flows are only inserted to switches.

3.3.5 AUTOMATION PROCESS

Bearing in mind all the modules presented before, the main process of automation becomes substantially simpler. Basically it is an orchestration module that invokes each one of the others when it is required.

Firstly we retrieve the reservations from the database to be processed. Note that the list of reservations obtained is already ordered by starting date (ascendent). After that, we process each reservation obtaining some of its information (for example its ID or the IP addresses of hosts) for further processing. Next step, we verify if the load balance mechanism is to be applied. This mechanism will be described later in Section 3.3.6.

Thereupon we check the reservation's dates to verify if we have to install or delete flows. If the reservation is not yet to be applied (because it is still too early), we move on to next reservation; if the reservation already expired it will be deleted from the network and the database; if the reservation's dates are with the currently date then the application proceeds to the installation of the queues and flows in the network.

We check now if the reservation was already applied. If it was we moved on to the next reservation. Otherwise we proceed to the next step. The next step includes obtaining the path between the hosts specified in the reservation. Note that, as mentioned in the previous Subsection 3.3.4 we verify if a path has already been calculated between those two hosts. If it was, we use that path, otherwise we calculate it using the *Path Discovery* module, already described in previous Subsection 3.3.4.
We will then proceed to the installation of the queues and flows on the upstream and downstream paths retrieved from the *Path Discovery* module. The upstream path is an unidirectional path between host A and host B (A to B) with the information of the switches and the output ports of those switches on that path. The downstream path is basically the same, the difference is that specifies the unidirectional path between host B and A (B to A).

We then iterate on the path (first the upstream path and then the downstream path). The process is the following:

1. Get necessary information from the reservation (priorities, rates, IP addresses).
2. Create the queue and send the request with the correspondent information.
3. Store the previous step information in the database (*QosMap* table).
4. Create the flow and send the request with the correspondent information.
5. Store the previous step information in the database (*FlowMap* table).

After the paths were processed we store those paths in an internal data structure, which is basically a list of reservation paths created. The data structure is called *ReservPath*. Its functionality and data structure will be later described in Subsection 3.3.6.

Finally we update the status of the reservation ($applied = true$).

The main workflow above mentioned is described in Appendix-Automation Process, Figure A.9.

3.3.6 LOAD BALANCE MECHANISM

The load balance mechanism has the objective of rerouting a reservation's traffic through another path, if possible. The mechanism is activated if the load on a given link exceeds a given threshold, for example 90%. The mechanism consists in two subsystems : Monitoring and Enforcement, described in the next subsections.

3.3.6.1 MONITORING

Due to the fact that we cannot obtain the current transmit rate at any moment directly from the controller we had to obtain the necessary network statistics and use that information to calculate it.

Using the OVSDB's NB API we are able to obtain, from a single request, some information related to every interface on the network, such as the amount transmitted/received bytes, number of collisions and others. By using the transmitted bytes information (cumulative field) and by monitoring those information every 5 seconds (default value was used because of performance concerns) we are able to calculate the transmitted rate at any moment.

The process consists on a thread that requests the interfaces information every 5 seconds, parses the retrieved information and stores it in an internal data structure called *Ifacestatistics* (refer to Appendix-Load Balance Mechanism, Figure A.10). While it is parsing and storing the information of all relevant interfaces, it is also calculating the current transmit rate and verifying if the threshold was exceeded. The threshold verification is done by comparing the current calculated transmit rate value with the transmit rate value from the previous moment. If a given link is 'overloaded' it enables a flag, warning that there is an 'overloaded' link on the network and stores in an internal data structure the ID of the overloaded port. Otherwise it removes the port from the overloaded links data structure, if it wasn't already deleted. By relevant interfaces we mean the interfaces only between switches. The others are not relevant because we can not load balance a path of a link connected between a host and a switch. Also, a link is considered 'overloaded' when it surpasses the pre-defined threshold.

The workflow of this process is depicted in Appendix-Load Balance Mechanism, Figure A.11.

3.3.6.2 ENFORCEMENT

The enforcement subsystem uses the information obtained from the Monitoring subsystem 3.3.6.1 and before a reservation is to be processed it verifies two things. First, if it has to apply load balance, and second if there is an 'overloaded' link in the network. That verification is done by checking the warning flag mentioned previously in Subsubsection 3.3.6.1 is enabled. If it is, it retrieves the first entry from the *ReservPath* list data structure to identify the port that is 'overloaded', (refer to Appendix-Load Balance Mechanism, Figure A.12). The next step is to select a reservation that contains that port in its path. The reservations selection is performed taking into consideration two parameters:

- The reservation's priority. This means that we will reroute first the affected reservations with highest priority.
- The reservation's lower max rate. This means that we will reroute first the ones that have a smaller amount of traffic first. This allows a better resource utilization but may require more replacements of paths.

Then, we iterate over the list of reservation paths (*ReservPath* list) in search for a reservation that has the 'overloaded' link in its path. When a reservation's path includes the 'overloaded' port, we change the weight of the correspondent Dijkstra's graph into a very large value and then proceed to the discovery of a new path. If that new path exists and does not have on its path any 'overloaded' link, it is then considered valid. We then reestablish the port's correspondent edge weight to its previous value and proceed to the path's replacement for that reservation. Otherwise we continue looking for another reservation that fits.

The replacement of the path's reservation is done using the approach make-before-break. This means that we create a new path, with a higher flow priority so the oldest path is not picked. After the new path has been created we tear down the other. All of this is done the same way we specified in Subsection 3.3.5 on the 'Within' and 'After' cases respectively.

Finally the link is removed from the overloaded links list and if the list is empty the warning flag stating there is a link overloaded is disable.

The workflow of when a link is 'overloaded' is depicted in Appendix-Load Balance Mechanism, Figure A.13.

3.4 UTILIZATION OBJECTIVES

SDN is such a paradigm that brings many benefits, as mentioned in 2.1.2, into the networking field. It enables carriers to reduce costs and the complexity of their networks and opens the door to new applications/services to be developed. Such services will enable for example the automation of the network which will greatly simplify the management of such large networks.

The application was designed to be applied specifically to data center networks. It leverages the fact that centrally programmable OpenFlow controllers can substantially optimize network operations by reducing management overhead, boosting scalability, and reducing interoperability issues. The global objective is to provide a set of mechanisms that work together into forming a solution to several of data center use cases such as:

- The first is automation on demand, basically management simplification. SDN enables the ability to programatically manage the network through the creation, modification or removal of routing rules (flow rules) while maintaining a global view of the network through the controller. This simplifies the management of the network. It is also faster than manual configuration, allowing behaviour change upon external factors. Particularly, QLAMES provides connectivity between hosts in a pre-defined period of time, with minimal interaction from the administrator. It only requires the insertion of reservation entries into the application's database while the application performs the changes on the underlying network.

- The second is bandwidth on demand, the ability to provide quality of service to the network. Once again, the ability to program the network also enables an easier and more fine-grained control of the whole network that can be used to request for a given amount of bandwidth on carrier links. This increases agility on the provisioning process in comparison with manual provisioning. In addition, it is a less error-prone approach, where an error could violate service contracts agreed with clients. Particularly, QLAMES establishes the communication for each reservation ensuring that their QoS requirements (minimum and maximum bandwidth and priority) are respected.

- Finally it provides a certain level of data center optimization by load balancing the traffic. SDN controller is able to provide a global view of the network, which can be used in this particular case to monitor the whole network and act upon the information retrieved. Basically this optimization mechanism tries to reroute traffic through other paths when a given link is near its maximum capacity. This allows a better global resource utilization while avoiding bottlenecks in the network. In addition, as network traffic is somewhat unpredictable it provides roominess in case of burst ocurrence, which would help in avoiding congestion problems in the network. Particularly, QLAMES monitores the whole network verifying the existence of links that are reaching its maximum capacity (defined by a threshold, usually 90%). In such cases it selects and reroutes their paths until the threshold is no longer surpassed.

SDN is changing the way we manage our networks bringing countless advantages, particularly to data center networks which due to their nature are quite complex to manage. This new paradigm proves to be valuable and enables the creation of such mechanisms that solve many of the current networking problems. QLAMES is a network management application that implements several mechanisms that aim at providing a solution the challenges above mentioned, that exist in data center networks.

CHAPTER 4

Evaluation

In this section the performance of the QLAMES application will be evaluated according to several parameters, such as its installation delay or overhead. The correctness and effectiveness of the Quality of Service and Load Balance mechanisms will also be verified and evaluated.

The testing scenarios are constituted by an OpenDaylight controller [1], an Oracle VM VirtualBox application with a virtual machine running Ubuntu 14.04 LTS with Mininet tools and an external network application, QLAMES.

The QLAMES application is a Java application and is already connected to a MySQL database that will contain the reservations as well as the flows' and queues' corresponding information. The reservation table stores a source and destination IP address, QoS parameters (queue priority, minimum and maximum rate), flows' priority and period of time (start and end date) that defines the period of time the actions have to be applied.

All the components mentioned above are running in the same machine. That machine is a laptop running the Linux Mint 17.1 Rebecca operating system. The laptop has a 5th generation processor Intel® Core™ i7-5500U, a 256GB solid-state drive, 8 GB of RAM and a clock speed of 2.40/3.00 GHz.

The OpenDaylight controller version is Helium SR2. Through a configuration file we specify that the OVSDB protocol will use OpenFlow protocol version 1.3 (refer to Appendix-OVSDB's Configuration to OpenFlowv1.3). The controller is also configured to use version 1.3 of the OpenFlow protocol using for that effect the flag 'of13' when launching the controller, (refer to Appendix-Controller's Configuration to OpenFlowv1.3.

The virtual machine has 2GB of RAM and was assigned 1 CPU from the processing node. The virtual topology used in all the tests was created on the virtual machine. The topology was created using mininet Python API and launched using the following command, (refer to Appendix-Topology's Launch command).

First we specify that we want to create a custom topology defined by the python file 'testScenario1', (refer to Appendix-Mininet's Custom Topology). Then we connect to the controller that is listening on port 6653 with IP address 192.168.57.1. Finally we specify openvSwitch as our Openflow switch and that the protocol version of OpenFlow is 1.3. It is worth mentioning that the signalling and data datapaths are separated. This means that while flow rules and queues are being installed, the throughput of links is not being affected. In addition, Controller is connected to every OpenFlow

[1] `http://www.opendaylight.org/software/downloads`

switch on the network via a control channel. Those connections were not included on the topology figure in order not to degrade the legibility it.

The topology is shown in Figure 4.1:

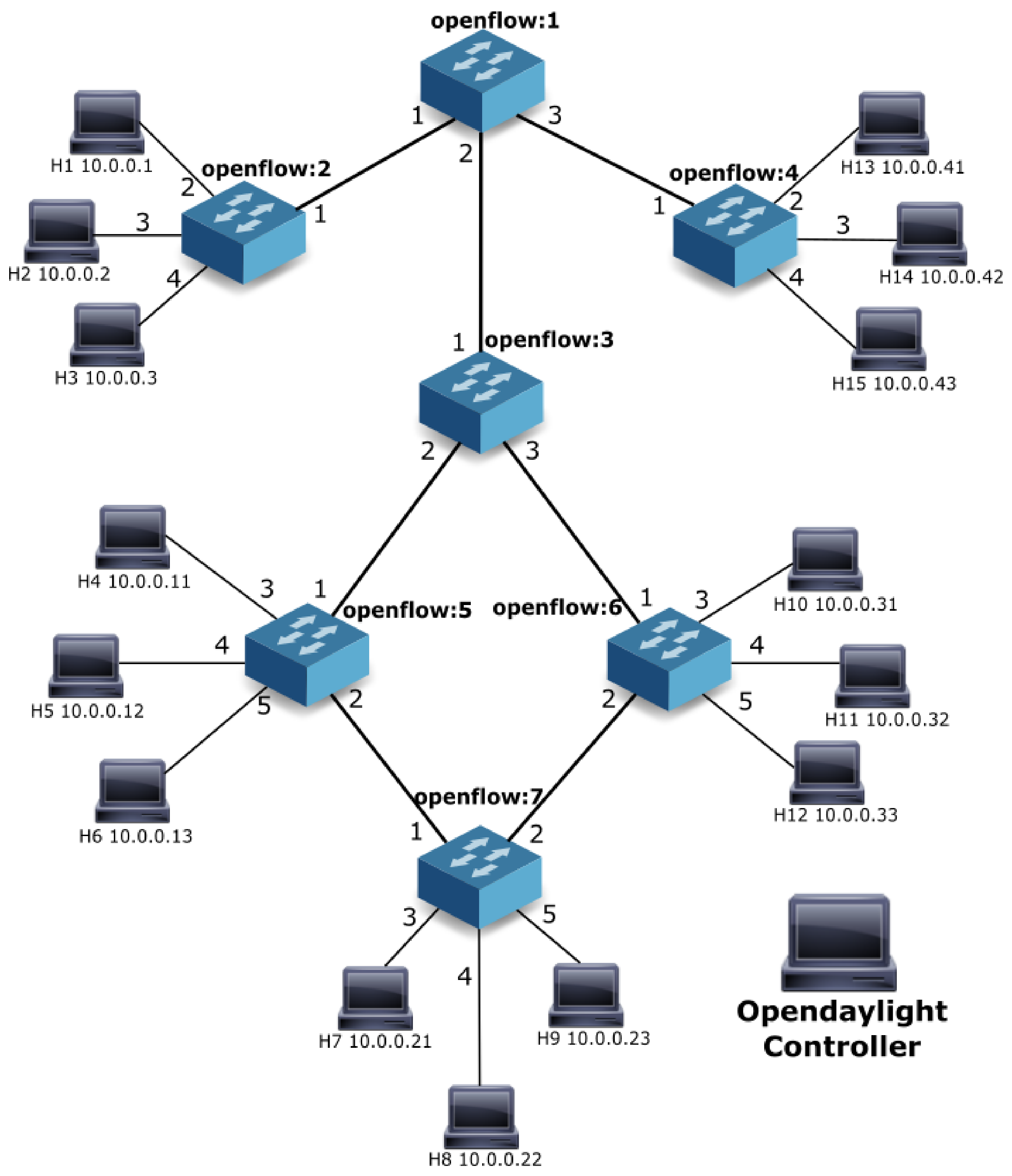


Figure 4.1: Network topology

The HostTracker module is responsible for informing the Opendaylight controller on where each host is connected, this means specify which host is connected to which switch on which port. HostTracker won't show any hosts until they are learned. It listens to the Address Tracker events and identifies hosts connected on network. In turn, Address tracker internally listens to packet in events by default ARP packets. For that reason, a ping command was issued from one host to all the others in order to the Opendaylight controller recognize the hosts and their location on the network. Thus, the topology is fully interpreted by the application.

We will have four scenarios to test and measure the application performance:

- Bandwidth on Demand - to verify if QoS is being applied correctly and if the existence of other queues associated to the same ports, or the existence of flows in the same switches impacts the bandwidth.
- Installation delay - to measure the time spent, by each task, inserting a given number of flows and, inherently, queues.
- Installation overhead - to measure the amount of data that are sent to the controller upon the insertion of a given number of flows and, inherently, queues.
- Load Balance application - to verify the correct change of path, without any packet being lost on the process. The objective is also to measure the amount of time necessary to modify a path, in a simple environment, with different selection parameters such as, changing the more prioritised reservations first or the ones with lower maximum rate first.

4.1 BANDWIDTH ON DEMAND

4.1.1 DESCRIPTION

In this scenario, two reservations for flows were submitted.

The first is between nodes 'H1' and 'H7' which resulted on path openflow:2-openflow:10-openflow:3-openflow:6-openflow:7. The minimum rate was set to 40 Mbps and the maximum rate was set to 50 Mbps with a priority of five.

The second is between nodes 'H15' and 'H7' which resulted on path openflow:4-openflow:10-openflow:3-openflow:6-openflow:7. The minimum rate was set to 90 Mbps and maximum rate was set to 100 Mbps with a priority of five.
Node 'H1' is connected to switch 'openflow:2' while node 'H7' is connected to switch 'openflow:7' and node 'H15' is connected to switch 'openflow:4'. Two types of tests were made: one with both nodes ('H1' and 'H15') communicating with the same server ('H7') at the same time (Case A) and the other is similar except the communications are separately (Case B).
The paths created upon processing the reservations are depicted in Figure 4.2.

In order to measure the bandwidth the Iperf [2] tool, version 2.0.5, was used. Iperf is available on mininet virtual machine to exchange TCP packets. In addition to the bandwidth, the amount of data transferred on these tests was also included in the results for a more detailed analysis. Adjustments were made on Iperf parameters, namely the window size, in order to follow the *bandwidth* × *delay* product rule.

To achieve high rates over paths with a large bandwidth-delay product, it is often necessary to select a TCP window size larger than the operating system default. The window size of the TCP measurement should follow the *bandwidth* × *delay* product rule, and should therefore be set to at least the measured Round-trip Time (RTT), multiplied by the path's bottleneck speed. Every link in

[2] https://iperf.fr/

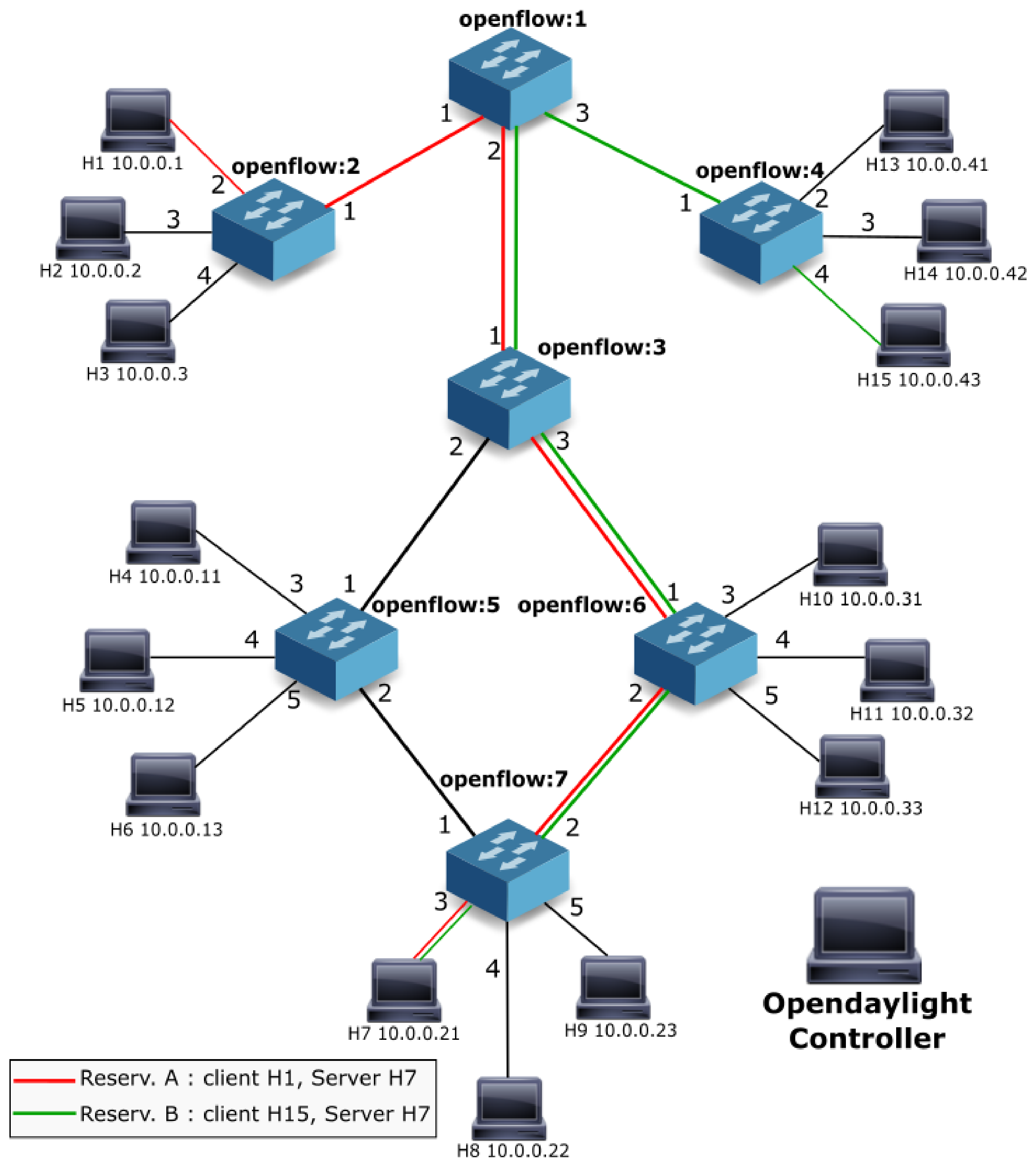


Figure 4.2: Network topology with the paths created

our network has an interface of 10 Gbps and thus we assume 10Gbps is the path's bottleneck speed. Despite this network's delay not being large, ultra-high speed networks may fall into this category on account of their extremely high bandwidth.

To estimate the RTT between hosts 'H1' and 'H7', and hosts 'H15' and 'H7', the ping command was used. The test was executed for 60 seconds and we have considered the higher value of the average of both host tests : 0.114 msec for the RTT between 'H1' and 'H7' and 0.125 msec for the RTT between

’H15’ and ’H7’). The window size is then given by

$$\begin{aligned} WindowSize &= RTT \times BottleneckLinkSpeed \\ &= 0.125 \times 10^{-3} \times 10 \times 10^{9} \\ &= 1.25Mbps \simeq 0.16MB \end{aligned} \tag{4.1}$$

It’s noteworthy to mention that, all results presented on this section were calculated with a 90% level of confidence, using a custom matlab function, (refer to Appendix-Script to calculate confidence intervals from Bandwidth on Demand scenario).

4.1.2 RESULTS

4.1.2.1 CASE A

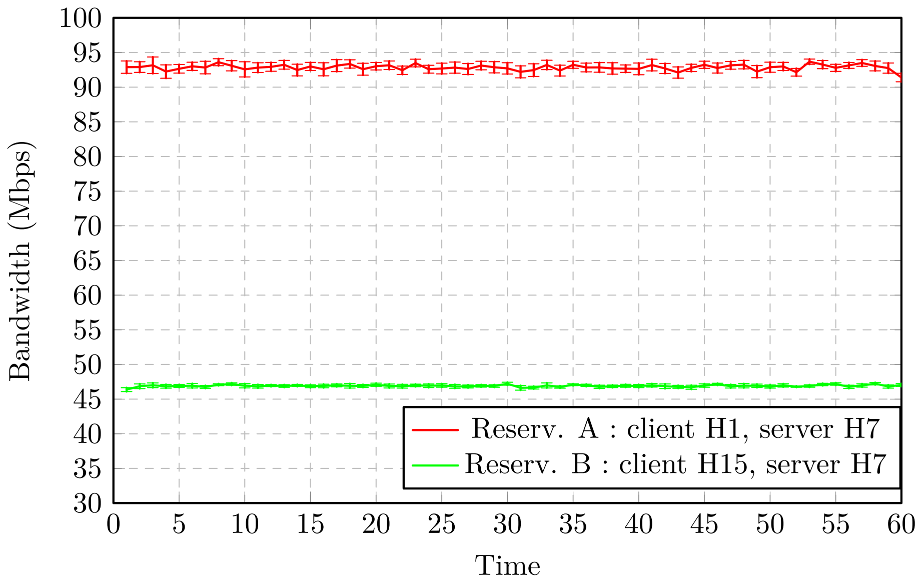


Figure 4.3: Simultaneous communication : Bandwidth

During this trial, as shown in Figure 4.3, both hosts (’H1’ and ’H15’) are communicating with the same server (’H7’) using the Iperf tool for a duration of sixty seconds. The bandwidth for reservation A is restricted to a maximum of 100 Mbps whilst reservation B is restricted to a maximum of 50 Mbps.

It is clearly noticeable that the maximum bandwidth obtained for each reservation never exceed the maximum rate, although being close to it. The results obtained are quite accurate, with reservation A having 46 Mbps as average bandwidth, while reservation B has around 91 Mbps, with very small confidence intervals for both reservations.

While the Iperf test was executed, the amount of data that was received (in megabytes) by the server at each moment was also retrieved and is depicted in Figure 4.4. The same happens on case B, where the amount of data (in megabytes) received by the server is depicted in Figure 4.6.

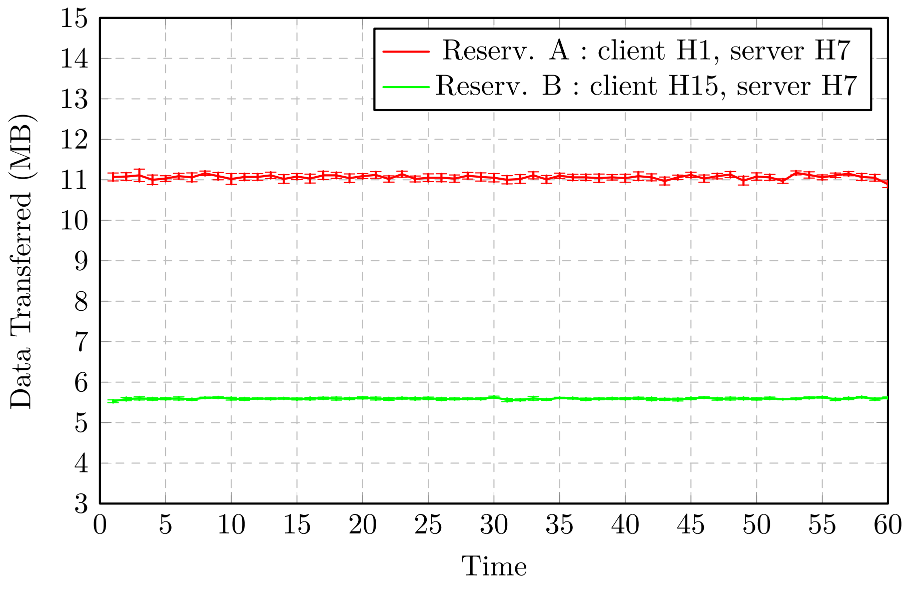


Figure 4.4: Simultaneous communication : Data Transfer

4.1.2.2 CASE B

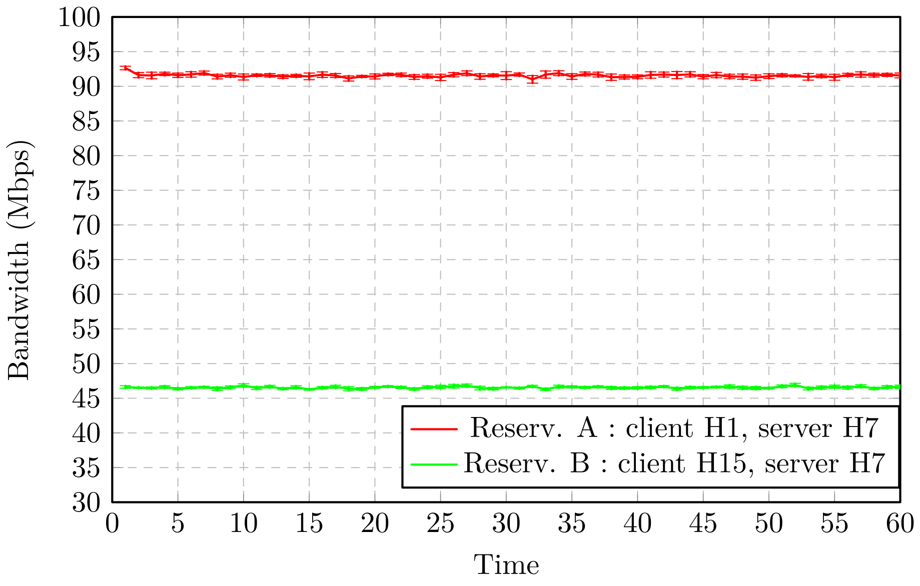


Figure 4.5: Separate communication : Bandwidth

During this trial, as shown in Figure 4.5, both hosts ('H1' and 'H15') are communicating separately. Firstly, 'H1' communicated with 'H7'; then, 'H15' communicated with the server. Apart from that, the same procedure was applied with the same reservation parameters. During the execution of the test, the value for the bandwidth was still within the parameters, and the confidence intervals remained very small, as in case A.

Table 4.1 shows the average values of the bandwidth, the data transfer as well as the average on their confidence intervals. These results are also depicted in Figures 4.3, 4.4, 4.5 and 4.6 in a more

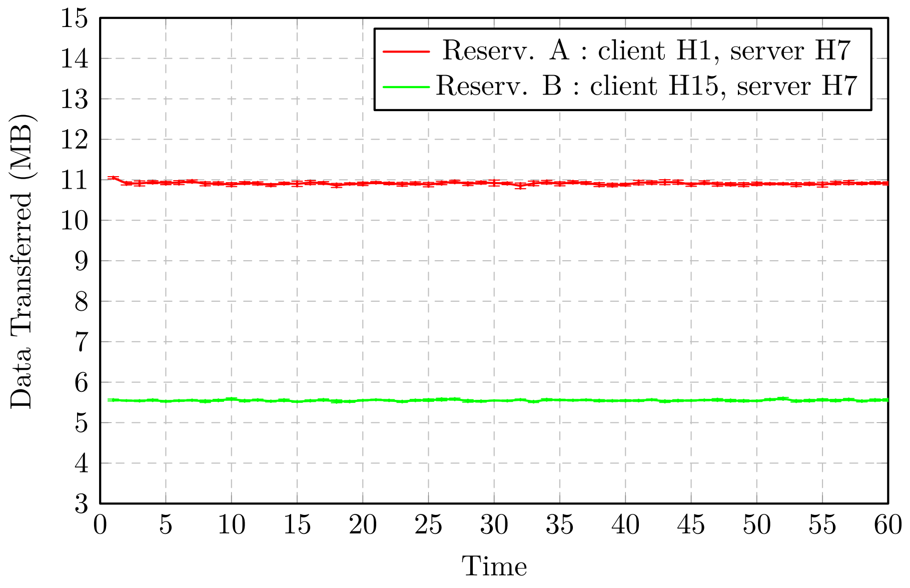


Figure 4.6: Separate communication : Data Transfer

Case	Bandwidth		Data Transfer	
	Flow A	Flow B	Flow A	Flow B
A (Simultaneous)	46.91 ± 0.23	92.85 ± 0.70	5.60 ± 0.03	11.06 ± 0.09
B (Separated)	46.56 ± 0.19	91.56 ± 0.34	5.55 ± 0.02	10.91 ± 0.04

Table 4.1: Comparison in terms of bandwidth and data transfer between Flows A and B

fine-grained form. The differences between Cases A (Simultaneous) and B (Separated) are very small.

4.1.3 CONCLUSION

The results in fact prove that the Quality of Service is being correctly applied according to the reservation parameters, having no difference if one or more flows are communicating or not at the same time, as long as there's enough bandwidth to accommodate them.

It is logical that the server received packets at a rate lower than the maximum rate specified on the queues because Iperf only measures the throughput of the payload. Ethernet's Maximum Transmission Unit (MTU) equals 1500 bytes; however, the Maximum Segment Size (MSS), which is the maximum amount of data that can be sent in a packet, equals 1448 bytes. The difference between MTU and MSS is due to headers (20-byte IP header + 20-byte TCP header) and timestamps (12-byte option), making a total of 52 bytes. That is the reason for the throughput measured to be lower than the maximum rate defined.

Despite some subtle value deviations, the overall bandwidth in each case, for each reservation, was in between the established parameters.

4.2 INSTALLATION DELAY

4.2.1 DESCRIPTION

On this scenario, a number of reservations were defined, which would equal a given number of flows to be inserted at the same time (worst case scenario). This means that the reservations' start date would be the same. A total of six tests were performed, each for a given number of flows: 100, 250, 500, 1000, 1500 and 2000. Each test was executed ten times in order to reach more precise values. In each test, we separated the delay values for the task in order to understand more accurately how long each task takes.

After all the flows were inserted and the values stored, the reservations' end dates were changed in such a way as to simulate that they were already expired. With this, we wanted to measure how long it would take to tear down all the flows and queues previously inserted. The execution time of the tear down process is also split by tasks, namely the delay from the database as well as the delay from the 'delete' REST requests made to the controller.
The load balance monitoring is performed once every five seconds, collecting data from all switch ports on the network.

Immediately after a flow is inserted, it is stored on the config data store. In order for the given flow to be stored on the switches, it has to be on the operational data store. The Opendaylight controller, by default, verifies every three seconds if there are any flows on the config data store. If so, it puts them on the operational data store. On the worst case scenario, a flow takes an extra three seconds to pass from the config data store into the operational one. The reason behind that is because the 'polling' interval to check if new flows were inserted in the config data store is three seconds; And some flows may have been inserted during the three second period and do not have to wait the whole time to be inserted on the switches.

A table and a graphic will be presented on Subsection 4.2.2 with the processing times of each task, either from QLAMES and Opendaylight, upon the insertion of a given number of flows. The time a flow takes to be transferred from the config data store into the operational data store was not included on the ’ODL Flow’ column because of the confidence intervals, but it must be taken into account. The default time was kept because, if the time was reduced, too many messages would be traded, which could compromise the performance of the controller.

Tests were not made for a higher number of flows because of a bug[3] that exists on the OpenDaylight OVSDB subsystem, found while testing the application. The switch is configured to connect to the controller when the controller starts up. After the controller is connected to the switch, the reservations start getting processed and, thus, start adding a lot of queues to OVS database. Eventually, when the amount of information is too large (around 100,000 bytes) and the OVSDB subsystem tries to pull information from the switch’s database, the Opendaylight controller immediately closes the OVSDB connection. After that, the switch attempts to reconnect repeatedly. The bug was already confirmed, but at the time of writing this document, there were no solutions available. This adds a limitation to QLAMES, which is currently using the Opendaylight’s OVSDB subsystem to configure the queues.

It’s noteworthy to mention that all results presented on this section were calculated with a 90% level of confidence, using a custom matlab function (refer to Appendix-Script to calculate confidence intervals from Installation delays scenario).

4.2.2 RESULTS

4.2.2.1 SETUP

In this test we notice that, by observing Table 4.2 and Figure 4.7, installation delay exists. Although it is not severely high for this number of flows, it is still clearly noticeable, which may lead to a scalability problem, which in turn translates into a limitation of the application.

[3]`https://bugs.opendaylight.org/show_bug.cgi?id=2732`

Flow Number	QLAMES					ODL		Total
	Flow	QoS	Database Interactions	Load Balance Monitoring	Dijkstra	ODL Flow	ODL QoS	
100	404.2 ± 7.1	187.7 ± 3.4	1394.9 ± 14.1	0.3 ± 0.0	6.0 ± 0.6	328.2 ± 14.4	1205.24 ± 39.5	3543.8 ± 59.7
250	770.3 ± 5.7	253.9 ± 5.2	3174.0 ± 177.9	1.4 ± 0.0	15.5 ± 0.9	594.3 ± 9.5	2059.3 ± 27.6	6894.6 ± 186.5
500	1203.0 ± 38.9	552.6 ± 17.1	5794.2 ± 248.0	2.5 ± 0.4	21.5 ± 1.4	1393.5 ± 80.66	5292.0 ± 304.34	14313.22 ± 492.23
1000	2040.5 ± 76.1	1201.0 ± 49.0	10553.7 ± 234.1	3.7 ± 0.6	31.1 ± 1.8	2459.3 ± 95.5	8623.4 ± 330.8	25007.74 ± 729.9
1500	2488.2 ± 105.5	1579.3 ± 40.7	14148.3 ± 567.4	4.3 ± 0.1	35.9 ± 1.2	3215.6 ± 93.5	11832.48 ± 358.38	33426.61 ± 1091.79
2000	2947.2 ± 35.7	2108.7 ± 39.4	18033.2 ± 350.0	5.6 ± 0.2	38.5 ± 1.2	4113.4 ± 36.5	16304.1 ± 223.4	43707.5 ± 567.3

Table 4.2: Installation delay by task (msec)

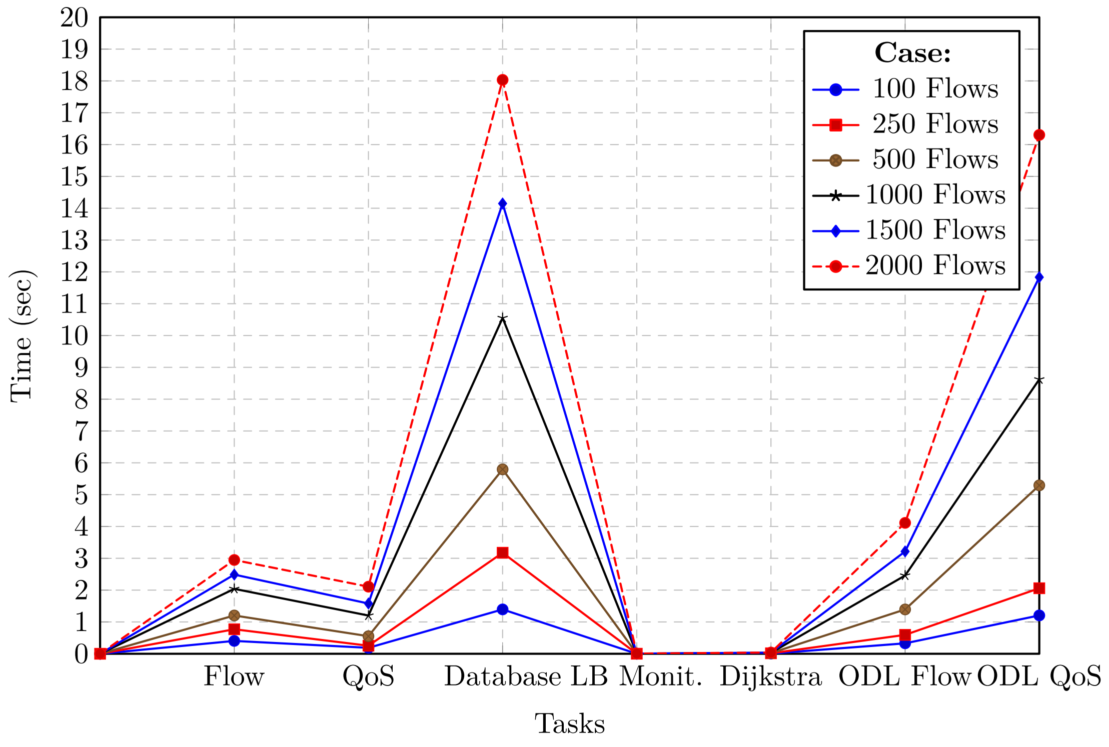


Figure 4.7: Installation Delay by tasks

The 'QLAMES' column contains the amount of time that each task took to be processed on the application. The 'ODL' column contains the time a REST request takes to be accepted by the controller. This column basically contains the amount of time a flow takes to be stored in the config data store, from the moment the REST request was issued by the application. In turn, column 'QoS' contains the amount of time a queue takes to be stored OVS database instead of being stored on the config data store of Opendaylight like flows were.

As an integral part of the QLAMES application, we notice that load balance monitoring and Dijkstra were the less burdensome tasks. Load balance monitoring takes a short time because, as we stated in Subsection 4.2.1, it only monitors once every five seconds. It only retrieves, parses and stores the measures from all interfaces of the network, such as the amount of transmitted and received data, the number of lost packets, etc. Dijkstra does not take a long time either, firstly because the topology is not very large or dense, and secondly because once a path between two hosts is calculated, it is stored and used in further cases. Basically, if a given path between two hosts was already found, it will be used instead of recalculating the path.

For Flow and QoS columns, the ODL part is more time-consuming than the application processing task. QLAMES, after obtaining the path, has to prepare and send all the REST requests for all the switches included on the path. In turn, ODL has to verify if they are correct and then submit them to the correct database (config data store for flows and OVSDB for queues). The 'travel time' of the REST request is considered as being part of the Opendaylight Controller because application has already issued the request. This justifies why the delay from the Opendaylight controller with respect to flow and queue processing is generally higher than the QLAMES application.
It is also noticeable that QoS delay grows faster than the Flow delay. QoS will take more time as more flows are installed on the network, because in addition to sending the queue configuration information to the controller, we also have to map the queues into the correct ports.

Last, but not least, database interactions are the most time-consuming task. In this task interactions like pull, add or update information are included. The database contains all the reservations and its parameters' data. After a given reservation is processed, all the data related to that reservation is stored in the database as well. That includes the *UUIDs* for the *queue*, *qos* and *port* rows as well as the *switch*, *table* and *flow* IDs.

4.2.2.2 TEAR DOWN

On the tear down scenario, we only have three relevant parameters in comparison to the setup scenario. These parameters are the delays from the database interactions, from the flows and from the QoS processing. Other parameters are not considered relevant because they are not used in this process. Table 4.3 and Figure 4.8 depict the values obtained tearing down the flows inserted on the setup case.

Flow Number	Database Interactions	Flow	QoS	Total
100	1368.8 ± 158.2	215.2 ± 11.7	1092.0 ± 52.0	2676.3 ± 136.5
250	3080.1 ± 48.5	418.5 ± 24.3	1974.2 ± 33.1	5473.8 ± 91.7
500	6085.3 ± 309.5	1050.1 ± 97.8	5553.5 ± 354.5	12752.9 ± 767.5
1000	9957.8 ± 244.8	1466.6 ± 83.2	8694.7 ± 489.1	20126.5 ± 693.6
1500	14373.3 ± 248.3	1942.1 ± 50.1	12942.0 ± 797.4	29263.6 ± 1017.4
2000	19351.8 ± 353.4	2457.7 ± 33.6	18423.0 ± 592.6	40239.4 ± 603.7

Table 4.3: tear down delay by task (in msec)

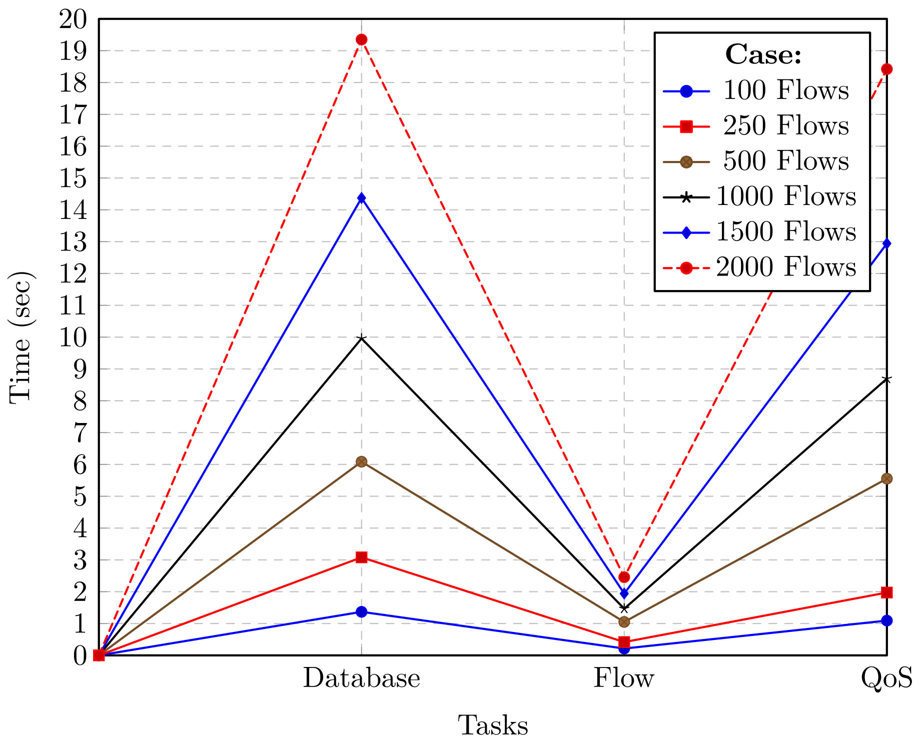


Figure 4.8: Installation Delay by tasks

We've noticed some similarities between this case and the setup case, which are: the database interactions is the most burdensome, followed by QoS which its delay increases with the number of flows installed and finally the flow delay which increases almost linearly with the number of flows installed. The explanation on these, despite being very similar to the setup case has the opposite objective which is to delete instead of insert. In addiction to that, it is clear that overall, the tear down process takes less time than the setup process. Although Database and QoS delays are very much the same as their counterparts (database slightly higher, QoS slightly lower) the flow tear down delay is clearly lower.

The delay obtained from database interactions is quite similar to its counterpart on the setup case. This is because when reservations are processed, all related information is added to the database (*switch*, *port* and *flow* IDs, *qos* and *queue UUIDs*, among other parameters) and associated to the correspondent reservation. That eases the process of tearing down any reservation by pulling all the necessary data related to that reservation. Simply because, that information is all the required information to remove the correspondent flows and queues from the controller, and thus from the switches. After pulling the necessary information, all data related to that reservation is deleted, including the reservation itself.
Basically, what distinguishes setup from tear down is that on the first, we add information while on the latter we remove information.

A simple test was executed to verify the amount of time MySQL would take to insert and delete a given number of entries. According to table 4.4, with the increase of the number of entries in a MySQL database, removing, on average, takes a little more time than adding entries to it. This explains why the delays from the database increase when the number of flows to delete also increases. Which means that, in total, the database delay on the tear down scenario is higher than its counterpart on the setup scenario.

Number of entries	Insertion delay	Deletion delay
100	210.77	188.98
250	472.07	478.29
500	1166.12	941.64
1000	1851.27	2001.33
1500	2938.49	3267.45
2000	3843.97	4066.08

Table 4.4: MySQL Database performance test : Comparison between insertion and deletion times

The QoS delay values are also quite similar, but slightly lower. What makes up for the difference is the fact that, when we insert queues, we have to formulate the request (REST URL + request body with the queue configuration information). Meanwhile, to remove it, it is only necessary to identify which queue (using its *UUID*) is to be deleted on the REST URL. Regarding the queue mapping, the process is very alike. On setup, queues are added, one by one, while on tear down, queues are removed from the *QoS* row, one by one.

Finally, the flows' tear down delays are clearly lower than the setup delays. This is due to the fact that there's no need to construct the request (REST URL + body with the flow configuration) as there was on the setup case. It is merely necessary to fetch the switch, table and flow ID from the database, add them to the REST URL and send the request to the controller. For that reason, it requires far less time to remove flows from the controller and, thus, from the switches.

4.2.3 CONCLUSION

In conclusion, it is obvious that, with the increase of the number of flows to be processed, the longer each task will take, either to setup or to tear down the network. That increase is generally linear, however, the *QoS* row, both from QLAMES and ODL, increases at a higher rate. That is due to the fact that, in addition to processing the configuration parameters for a queue, QLAMES also has to map that queue into the proper port. Each port has a list of queues associated with it, and with the increase of the number of flows (and thus the number of queues), each port possesses more information that has to be resent to the controller every time we want to add a new queue. That is why it will exponentially take more and more time, according to the number of queues associated with each port, which is directly affected by the number of flows.

Summarising, the higher the number of flows installed, the higher the number of queues. Thus, with more flows, more queues will be mapped per port, which increases both the setup and tear down delays as well as the installation overhead for QoS related activities. The increase of the number of entries on the database also makes the deletion process slower. In turn, the flow deletion process is clearly shorter simply because it already has all the information retrieved from database and just has to send a simple empty-body REST request to delete it from the network.

4.3 INSTALLATION OVERHEAD

4.3.1 DESCRIPTION

This scenario occurred at the same time the previous one was being executed. During those tests, the application was already storing the data that were being sent to the controller. This data came from the REST requests being made to insert the flows and the queues.

In this case, it was only contemplated the setup of flows and not the tear down because the delete REST requests for the flows and the queues have no body and thus its overhead is not relevant. Respectively to the queues, the queue mapping is also sent but the process is the reverse to the one where we would add the queues, and thus it was considered not relevant.

4.3.2 RESULTS

Table 4.5 depicts the impact of the application in terms of overhead, also obtained when the setup case was executed. As the number of flows to insert increases, so does the QoS overhead increase. Like we explained on Section 4.3.1, the delay increased because it was necessary to resend the information from all other queues to add/remove one queue, which obviously has repercussions, in the same way, on the overhead. In contrast, the flow overhead is on average steady across bearing in mind the increase on the number of flows that has been installed.

Flow Number	Flows	Overhead per flow	QoS	Overhead per Queue	Total
100	76162	761.62	42495	424.95	118657
250	190559	762.24	131585	526.34	322144
500	381268	762.54	386064	772.13	767332
1000	763153	763.15	1349361	1349.36	2112514
1500	1145205	763.47	2895261	1930.17	4040466
2000	1527355	763.68	5026399	2513.20	6553754

Table 4.5: Installation overhead (in bytes)

4.3.3 CONCLUSION

As we may notice, the flow overhead for a different number of flows is almost the same, unlike the QoS overhead, which increases with the number of flows installed. That is due to the fact that when a Queue is inserted, we also have to map it to a given QoS row that belongs to a given port/interface. Every time a queue is inserted, the configuration information for that queue must be sent. It must also be mappped to the correct QoS row that belongs to the port we want to attach the queue. That is why, when a higher amount of flows is installed on the network, the amount of information to send to the controller via REST requests is higher.

It is noteworthy to mention that a reservation translates into multiple flows and every flow will have a queue.

The flow overhead has a very small deviation. The reason behind it is that, even though the base format of the request is the same, its parameters (like the flow ID, the IP addresses, the port numbers or the priority) may vary in size. The variation in terms of size is minimal, however, it exists and it's noticeable. For example, the flow ID may be '7' but some other flow may have an ID of '34' or '102', depending on how many flows are inserted on that table, which adds an extra character to the request.

In conclusion, it is considered that the overhead observed is not high enough to become an obstacle, taking into consideration the benefits that an external application may bring to the table in the future, such as the capability of being used for any controller. That, of course, bearing in mind the effort from NBI-WG to standardize SDN Controller NBI.

4.4 LOAD BALANCE APPLICATION

4.4.1 DESCRIPTION

In this scenario, three reservations were submitted. The first (Flow A) allows the communication between host 'H1' and host 'H7' with a minimum rate of 30 Mbps, a max rate of 40 Mbps and a priority of 1 (highest priority). The path established due to that reservation was 'openflow:2-openflow:10-openflow:3-openflow:6-openflow:7'. The second (Flow B) allows the communication between host 'H15' and host 'H7' with a minimum rate of 25 Mbps, a maximum rate of 35 Mbps and a priority of 3. The path established due to that reservation was 'openflow:4-openflow:10-openflow:3-openflow:6-openflow:7'. The third and last (Flow C) allows the communication between host 'H10' and 'H7' with a minimum rate of 20 Mbps, a maximum rate of 30 Mbps and a priority of 2. The path established due to that reservation was 'openflow:6-openflow:7'.
The paths before the application of the load balance mechanism are depited in Figure 4.9.

For testing purposes we established a controlled environment where we assume that every interface has a link speed of 100 Mbps instead of 10 Gbps. Note that, a link is considered overloaded when it exceeds 90% of the link's usage. In that moment the load balance mechanism will be triggered. Also take into account that link monitoring will be executed every five seconds.

The Opendaylight controller and QLAMES application were iniciated and flow tables were filled with the necessary flows so the communication would be possible. In this case we identify the switch and then the port, for example if we specify 'openflow:2:1' it means that it is openflow switch number 2, port 1.
As shown in Figure 4.10, all flows are using link 'openflow:6:2' to reach switch 'openflow:7' which makes up about 100 Mbps. This way, the total maximum rates will exceed the 'openflow:6:2' bandwidth usage threshold which activates the load balance mechanism.

The load balance mechanism chooses an already applied reservation that contains the overloaded link. By applied we mean that all flows and queues were already installed on the network. The reservation is chosen based on its priority ('Priority Case') or based on the lower maximum rate ('Rate Case'), depending on the administrator's choice. In 'Priority Case', reservations with the highest priority (lower values of priority) are chosen first; In 'Rate Case', reservations with lower maximum rate will be chosen first.

The first allows us to change the most prioritised traffic first into a link that is not overloaded. Meanwhile, the second allows a better resource utilization in terms of bandwidth. However, it may require a higher number of replacements (changing reservations paths), which increases the delay caused by this mechanism. That delay may be higher depending on how many reservations have to be changed in order to drain the extra traffic from the overloaded link.

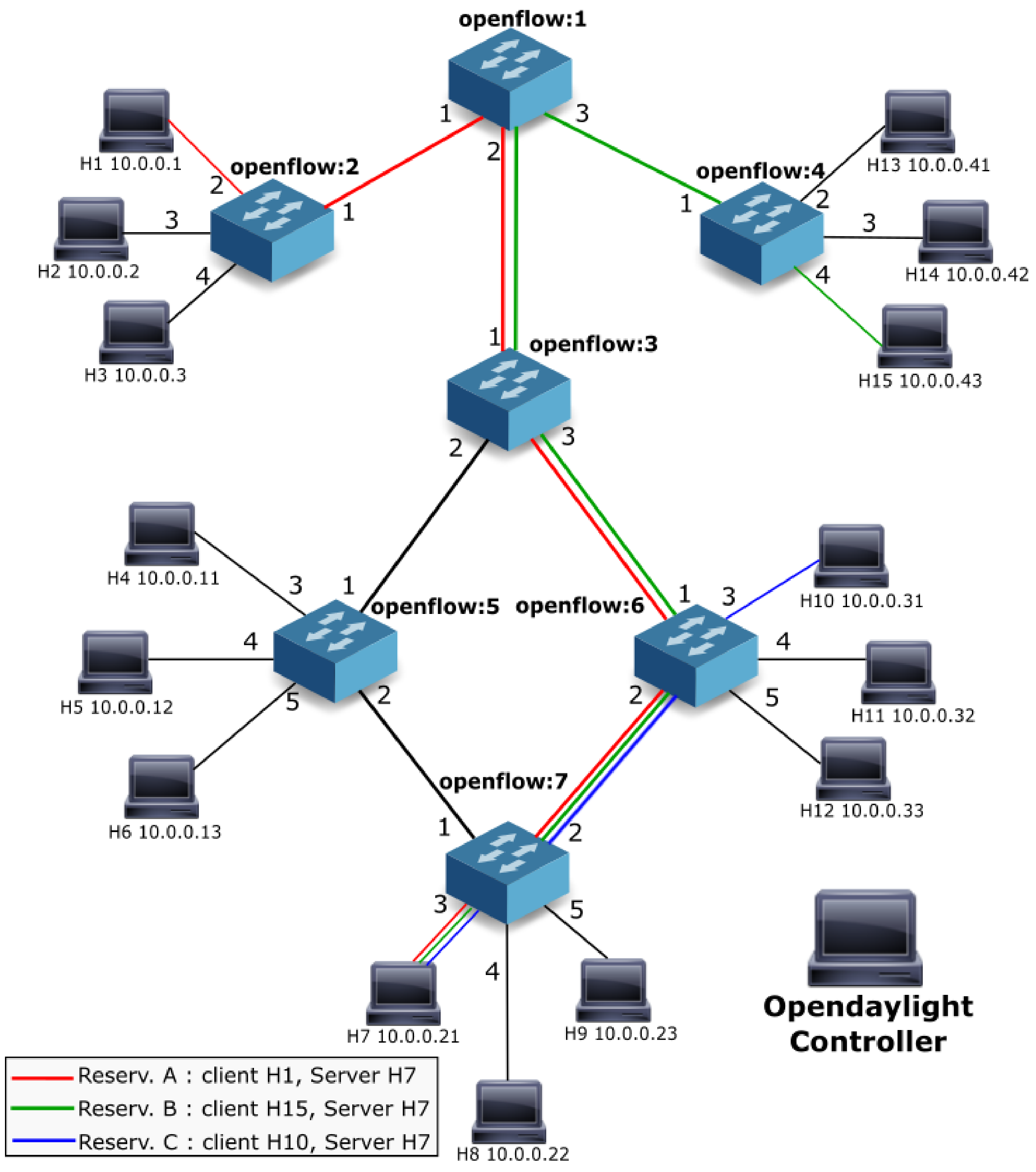
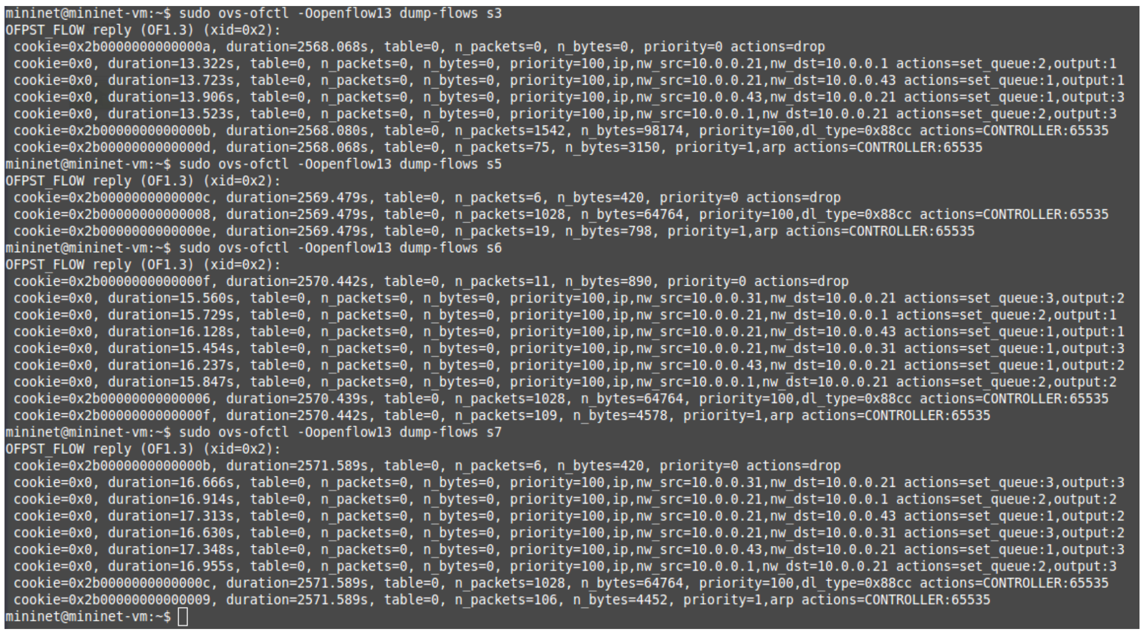


Figure 4.9: Network topology with the paths created before the application of the load balance mechanism

It is worth pointing out that the changes in a reservation's path will only be performed if a new path is possible and, if in that new path there is no overloaded link. Otherwise, the reservation's path in question will remain the same and the mechanism proceeds to the next reservation.

4.4.2 RESULTS

After the flows are inserted for the correspondent reservations mentioned in Subsection 4.4.1, the flow tables from all switches were filled. Figure 4.10 depicts flow tables of switches 'openflow:3' (s3),



```
mininet@mininet-vm:~$ sudo ovs-ofctl -Oopenflow13 dump-flows s3
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b0000000000000a, duration=2568.068s, table=0, n_packets=0, n_bytes=0, priority=0 actions=drop
 cookie=0x0, duration=13.322s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.1 actions=set_queue:2,output:1
 cookie=0x0, duration=13.723s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.43 actions=set_queue:1,output:1
 cookie=0x0, duration=13.906s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.43,nw_dst=10.0.0.21 actions=set_queue:1,output:3
 cookie=0x0, duration=13.523s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.1,nw_dst=10.0.0.21 actions=set_queue:2,output:3
 cookie=0x2b0000000000000b, duration=2568.080s, table=0, n_packets=1542, n_bytes=98174, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b0000000000000d, duration=2568.068s, table=0, n_packets=75, n_bytes=3150, priority=1,arp actions=CONTROLLER:65535
mininet@mininet-vm:~$ sudo ovs-ofctl -Oopenflow13 dump-flows s5
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b0000000000000c, duration=2569.479s, table=0, n_packets=6, n_bytes=420, priority=0 actions=drop
 cookie=0x2b00000000000008, duration=2569.479s, table=0, n_packets=1028, n_bytes=64764, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b0000000000000e, duration=2569.479s, table=0, n_packets=19, n_bytes=798, priority=1,arp actions=CONTROLLER:65535
mininet@mininet-vm:~$ sudo ovs-ofctl -Oopenflow13 dump-flows s6
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b0000000000000f, duration=2570.442s, table=0, n_packets=11, n_bytes=890, priority=0 actions=drop
 cookie=0x0, duration=15.560s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.31,nw_dst=10.0.0.21 actions=set_queue:3,output:2
 cookie=0x0, duration=15.729s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.1 actions=set_queue:2,output:1
 cookie=0x0, duration=16.128s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.43 actions=set_queue:1,output:1
 cookie=0x0, duration=15.454s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.31 actions=set_queue:1,output:3
 cookie=0x0, duration=16.237s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.43,nw_dst=10.0.0.21 actions=set_queue:1,output:2
 cookie=0x0, duration=15.847s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.1,nw_dst=10.0.0.21 actions=set_queue:2,output:2
 cookie=0x2b00000000000006, duration=2570.439s, table=0, n_packets=1028, n_bytes=64764, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b0000000000000f, duration=2570.442s, table=0, n_packets=109, n_bytes=4578, priority=1,arp actions=CONTROLLER:65535
mininet@mininet-vm:~$ sudo ovs-ofctl -Oopenflow13 dump-flows s7
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b0000000000000b, duration=2571.589s, table=0, n_packets=6, n_bytes=420, priority=0 actions=drop
 cookie=0x0, duration=16.666s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.31,nw_dst=10.0.0.21 actions=set_queue:3,output:3
 cookie=0x0, duration=16.914s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.1 actions=set_queue:2,output:2
 cookie=0x0, duration=17.313s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.43 actions=set_queue:1,output:2
 cookie=0x0, duration=16.630s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.31 actions=set_queue:3,output:2
 cookie=0x0, duration=17.348s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.43,nw_dst=10.0.0.21 actions=set_queue:1,output:3
 cookie=0x0, duration=16.955s, table=0, n_packets=0, n_bytes=0, priority=100,ip,nw_src=10.0.0.1,nw_dst=10.0.0.21 actions=set_queue:2,output:3
 cookie=0x2b0000000000000c, duration=2571.589s, table=0, n_packets=1028, n_bytes=64764, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000009, duration=2571.589s, table=0, n_packets=106, n_bytes=4452, priority=1,arp actions=CONTROLLER:65535
mininet@mininet-vm:~$ 
```

Figure 4.10: Both Cases : Flow Tables before Load Balance

'openflow:5' (s5) and 'openflow:6' (s6) before load balance mechanism being applied. In turn, Figure 4.9 already shows the path established for each reservation's flow. The *actions* specify how a packet should be processed. The *set_queue* action sends a packet into a queue and the *output* action sends a packet into a switch port. According to the paths specified in Subsection 4.4.1 and the topology depicted in Figure 4.1 it is possible to verify the correctness of these flow tables.

The amount of time that each task of Load Balance mechanism took is depicted in Figure 4.11.

The monitoring task is executed once every 5 seconds and it is executed concurrently. The 'Path Discovery' task is the process of discovering a new valid path for a given reservation. On 'Priority case', reservations with highest priority are processed first while on the Rate case, reservations with highest maximum rate are processed first. The 'Path Replacement' task happens when a valid path is found. It consists on creating all new flows and queues for the new path and, after that, deleting the hold flows and queues from the network's switches. That includes sending all REST requests to the controller in order to delete and create the flows and queues. It also includes the creation of new entries on the database to store all the parameters for the new reservation as well as deletion of the parameters of the old reservation.

It is clear that the 'Priority Case' took a higher amount of time than the 'Rate Case' due to the 'Path Replacement' task. That is due to the fact that it was needed to add and delete more flows and queues than the reservation picked from the 'Rate Case'. For example, the reservation replaced on 'Priority Case' had a total of ten flows and ten queues. While replacing the path, a total of ten flows and queues had to be created and then the other ten flows and queues had to be deleted. Meanwhile, the reservation replaced on 'Rate Case' had to create 8 flows and queues but only had to delete 4 flows and queues.

The reasons mentioned above explain the gap between those values otherwise they would be close.

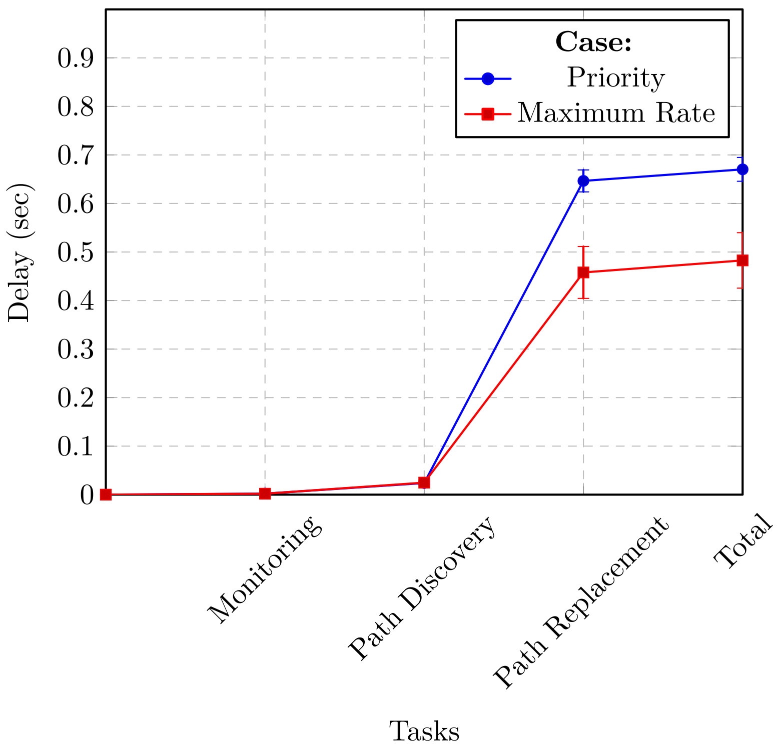
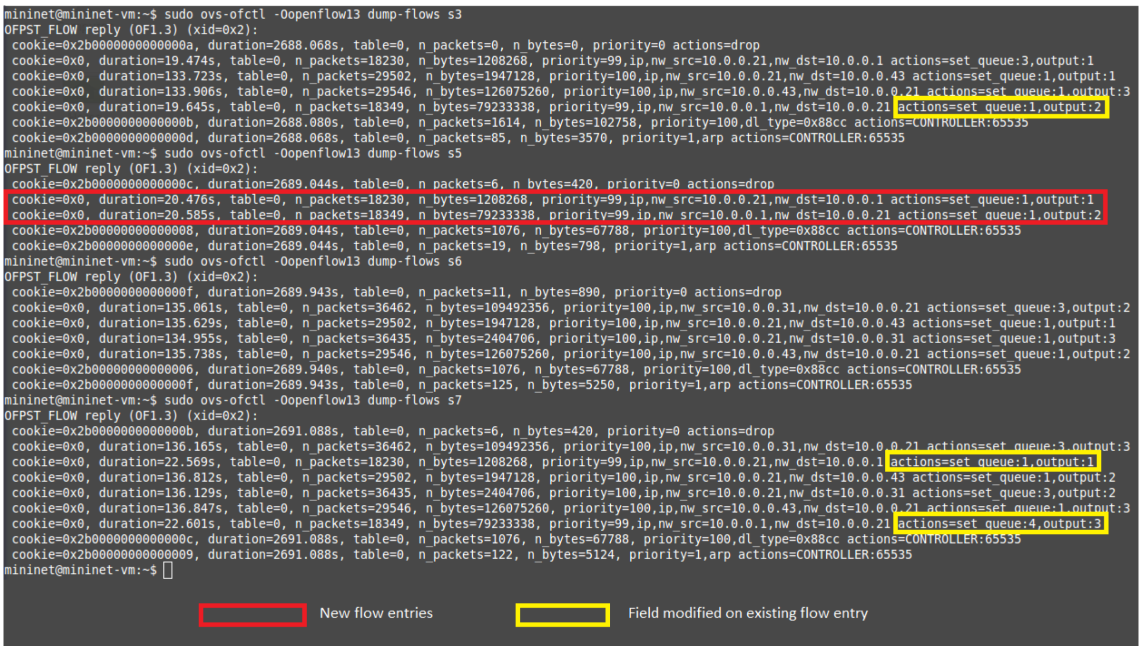


Figure 4.11: Load Balance Delay by tasks

4.4.2.1 PRIORITY CASE

The reservation with highest priority is the one between hosts 'H1' (10.0.0.1) and 'H7' (10.0.0.21). An alternate path, that would not cross any overloaded link, was searched. In this case, an alternate path was found that avoided the overloaded link. The new path established was 'openflow:2:1-openflow:10:2-openflow:3:2-openflow:5:2-openflow:7:3' as shown in Figure 4.13. By observing Figure 4.12 it is noticeable that switch 'openflow:6' no longer has entries for host 'H1' and that 2 new entries were added in switch 'openflow:5' precisely for host 'H1'. The flow entries for host 'H1' still remain on the switch with the difference that they now redirect the path from 'H1' to 'H7' through port 2 and not port 3 of switch 'openflow:3'. The flow entries added on switch 'openflow:5' are inside a red box and the field changed on a existing flow entry on switch 'openflow:3' is inside a yellow box on Figure 4.12.

Through Figure 4.14 and also Figure 4.15 we verify that no connection was lost during the path replacement process. In addition, we can also verify that no packet was lost during the process because the number of packets dropped before and after the Iperf test is the same. This is depicted by observing the 'drop' flow rule in Figures 4.10 (before) and 4.12 (after in Priority case). This is due to the make-before-break (MBB) approach taken by the load balance mechanism of QLAMES. The correspondent bandwidth and RTT from each reservation remained quite the same during the whole process.



```
mininet@mininet-vm:~$ sudo ovs-ofctl -Oopenflow13 dump-flows s3
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b000000000000000a, duration=2688.068s, table=0, n_packets=0, n_bytes=0, priority=0 actions=drop
 cookie=0x0, duration=19.474s, table=0, n_packets=18230, n_bytes=1208268, priority=99,ip,nw_src=10.0.0.21,nw_dst=10.0.0.1 actions=set_queue:3,output:1
 cookie=0x0, duration=133.723s, table=0, n_packets=29502, n_bytes=1947128, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.43 actions=set_queue:1,output:1
 cookie=0x0, duration=133.906s, table=0, n_packets=29546, n_bytes=126075260, priority=100,ip,nw_src=10.0.0.43,nw_dst=10.0.0.21 actions=set_queue:1,output:3
 cookie=0x0, duration=19.645s, table=0, n_packets=18349, n_bytes=79233338, priority=99,ip,nw_src=10.0.0.1,nw_dst=10.0.0.21 actions=set_queue:1,output:2
 cookie=0x2b000000000000000b, duration=2688.080s, table=0, n_packets=1614, n_bytes=102758, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b000000000000000d, duration=2688.068s, table=0, n_packets=85, n_bytes=3570, priority=1,arp actions=CONTROLLER:65535
mininet@mininet-vm:~$ sudo ovs-ofctl -Oopenflow13 dump-flows s5
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b000000000000000c, duration=2689.044s, table=0, n_packets=6, n_bytes=420, priority=0 actions=drop
 cookie=0x0, duration=20.476s, table=0, n_packets=18230, n_bytes=1208268, priority=99,ip,nw_src=10.0.0.21,nw_dst=10.0.0.1 actions=set_queue:1,output:1
 cookie=0x0, duration=20.585s, table=0, n_packets=18349, n_bytes=79233338, priority=99,ip,nw_src=10.0.0.1,nw_dst=10.0.0.21 actions=set_queue:1,output:2
 cookie=0x2b0000000000000008, duration=2689.044s, table=0, n_packets=1076, n_bytes=67788, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b000000000000000e, duration=2689.044s, table=0, n_packets=19, n_bytes=798, priority=1,arp actions=CONTROLLER:65535
mininet@mininet-vm:~$ sudo ovs-ofctl -Oopenflow13 dump-flows s6
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b000000000000000f, duration=2689.943s, table=0, n_packets=11, n_bytes=890, priority=0 actions=drop
 cookie=0x0, duration=135.061s, table=0, n_packets=36462, n_bytes=109492356, priority=100,ip,nw_src=10.0.0.31,nw_dst=10.0.0.21 actions=set_queue:3,output:2
 cookie=0x0, duration=135.629s, table=0, n_packets=29502, n_bytes=1947128, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.43 actions=set_queue:1,output:1
 cookie=0x0, duration=134.955s, table=0, n_packets=36435, n_bytes=2404706, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.31 actions=set_queue:1,output:3
 cookie=0x0, duration=135.738s, table=0, n_packets=29546, n_bytes=126075260, priority=100,ip,nw_src=10.0.0.43,nw_dst=10.0.0.21 actions=set_queue:1,output:2
 cookie=0x2b0000000000000006, duration=2689.940s, table=0, n_packets=1076, n_bytes=67788, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b000000000000000f, duration=2689.943s, table=0, n_packets=125, n_bytes=5250, priority=1,arp actions=CONTROLLER:65535
mininet@mininet-vm:~$ sudo ovs-ofctl -Oopenflow13 dump-flows s7
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b000000000000000b, duration=2691.088s, table=0, n_packets=6, n_bytes=420, priority=0 actions=drop
 cookie=0x0, duration=136.165s, table=0, n_packets=36462, n_bytes=109492356, priority=100,ip,nw_src=10.0.0.31,nw_dst=10.0.0.21 actions=set_queue:3,output:3
 cookie=0x0, duration=22.569s, table=0, n_packets=18230, n_bytes=1208268, priority=99,ip,nw_src=10.0.0.21,nw_dst=10.0.0.1 actions=set_queue:1,output:1
 cookie=0x0, duration=136.812s, table=0, n_packets=29502, n_bytes=1947128, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.43 actions=set_queue:1,output:2
 cookie=0x0, duration=136.129s, table=0, n_packets=36435, n_bytes=2404706, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.31 actions=set_queue:3,output:2
 cookie=0x0, duration=136.847s, table=0, n_packets=29546, n_bytes=126075260, priority=100,ip,nw_src=10.0.0.43,nw_dst=10.0.0.21 actions=set_queue:1,output:3
 cookie=0x0, duration=22.601s, table=0, n_packets=18349, n_bytes=79233338, priority=99,ip,nw_src=10.0.0.1,nw_dst=10.0.0.21 actions=set_queue:4,output:3
 cookie=0x2b000000000000000c, duration=2691.088s, table=0, n_packets=1076, n_bytes=67788, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b0000000000000009, duration=2691.088s, table=0, n_packets=122, n_bytes=5124, priority=1,arp actions=CONTROLLER:65535
mininet@mininet-vm:~$
```

Figure 4.12: Flow Tables after Load Balance enforcement

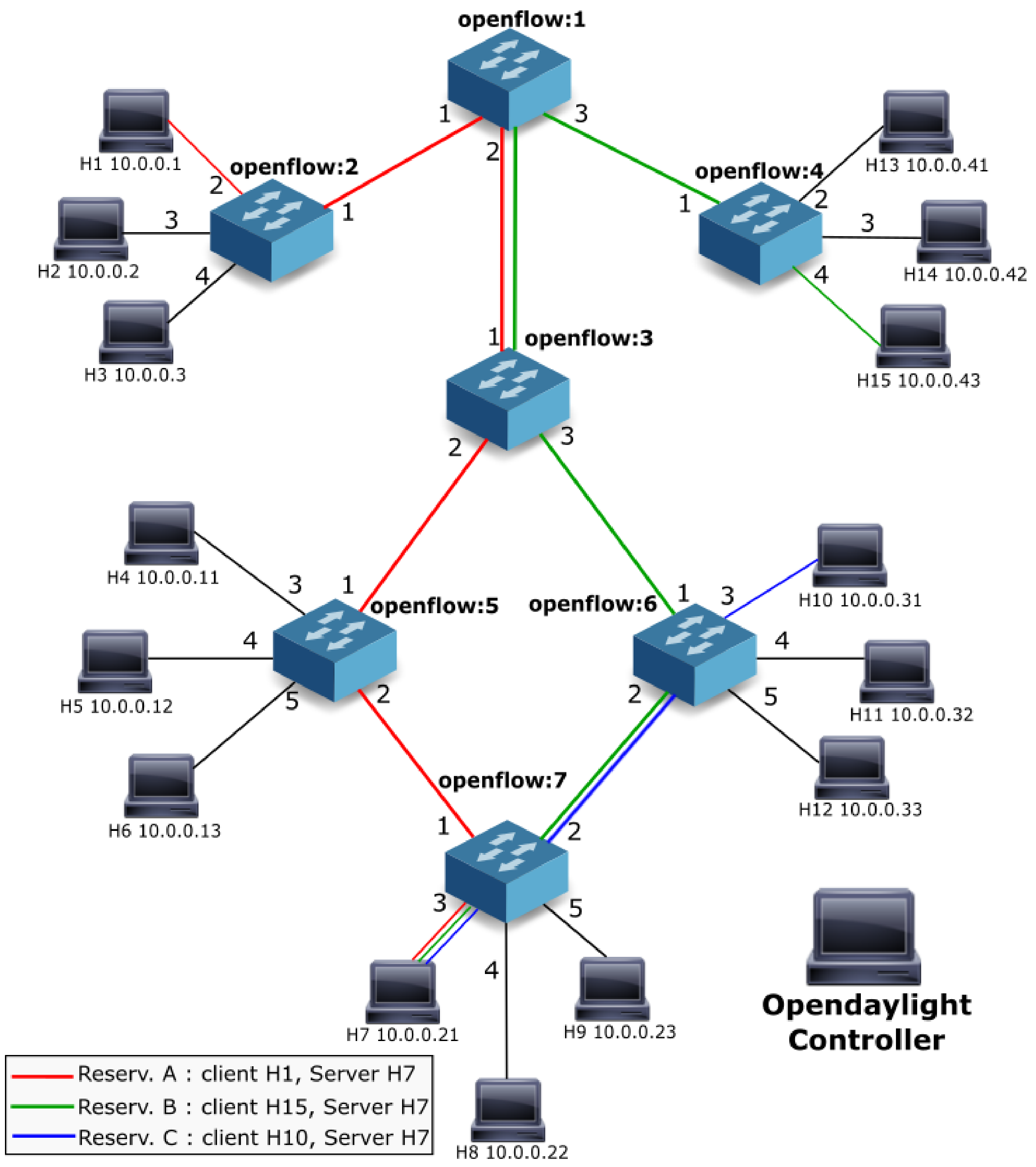
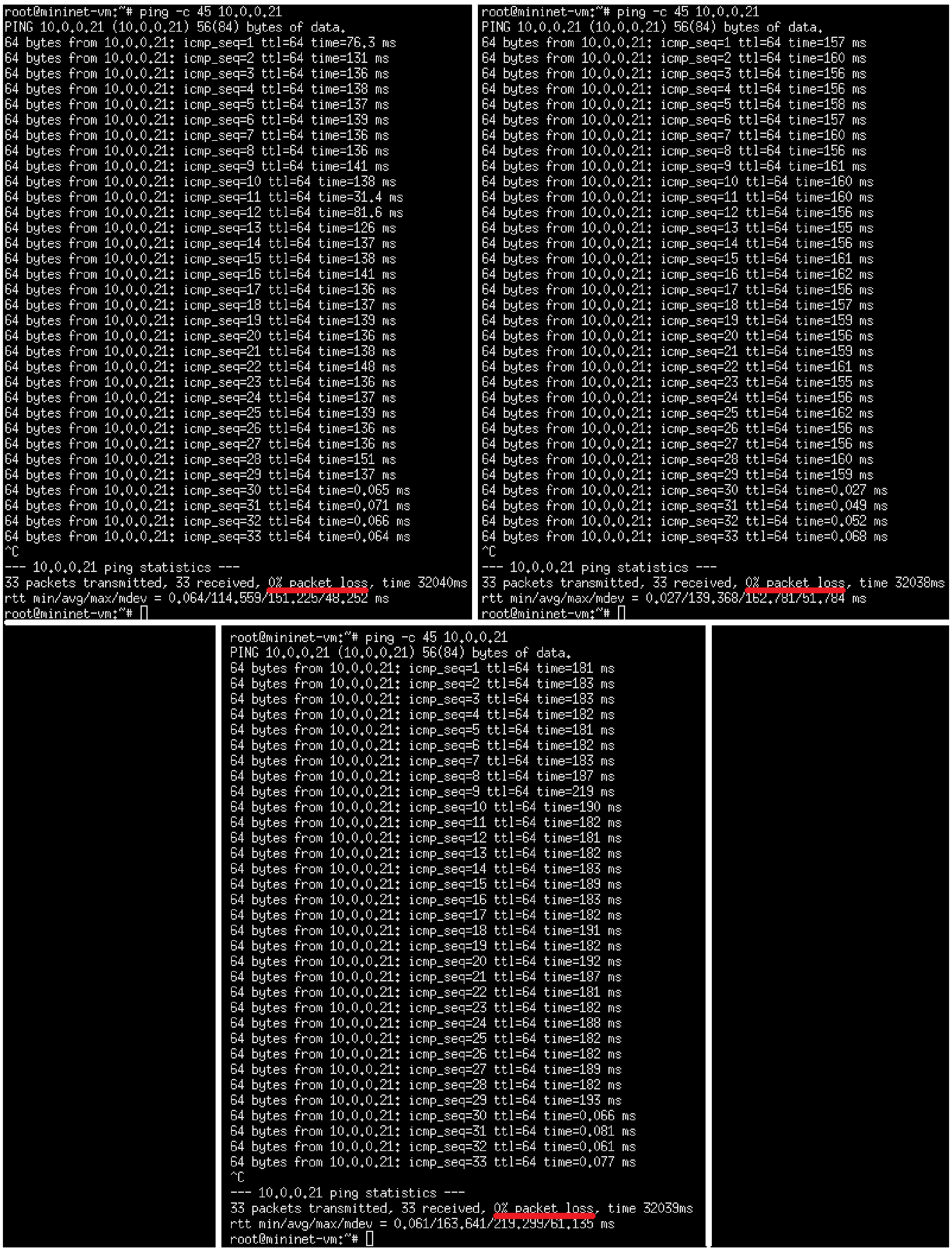


Figure 4.13: Network topology with the paths, after the application of the load balance mechanism with the highest priority parameter



```
root@mininet-vm:~# ping -c 45 10.0.0.21
PING 10.0.0.21 (10.0.0.21) 56(84) bytes of data.
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=76.3 ms
64 bytes from 10.0.0.21: icmp_seq=2 ttl=64 time=131 ms
64 bytes from 10.0.0.21: icmp_seq=3 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=4 ttl=64 time=138 ms
64 bytes from 10.0.0.21: icmp_seq=5 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=6 ttl=64 time=139 ms
64 bytes from 10.0.0.21: icmp_seq=7 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=8 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=9 ttl=64 time=141 ms
64 bytes from 10.0.0.21: icmp_seq=10 ttl=64 time=138 ms
64 bytes from 10.0.0.21: icmp_seq=11 ttl=64 time=31.4 ms
64 bytes from 10.0.0.21: icmp_seq=12 ttl=64 time=81.6 ms
64 bytes from 10.0.0.21: icmp_seq=13 ttl=64 time=126 ms
64 bytes from 10.0.0.21: icmp_seq=14 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=15 ttl=64 time=138 ms
64 bytes from 10.0.0.21: icmp_seq=16 ttl=64 time=141 ms
64 bytes from 10.0.0.21: icmp_seq=17 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=18 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=19 ttl=64 time=139 ms
64 bytes from 10.0.0.21: icmp_seq=20 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=21 ttl=64 time=138 ms
64 bytes from 10.0.0.21: icmp_seq=22 ttl=64 time=148 ms
64 bytes from 10.0.0.21: icmp_seq=23 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=24 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=25 ttl=64 time=139 ms
64 bytes from 10.0.0.21: icmp_seq=26 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=27 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=28 ttl=64 time=151 ms
64 bytes from 10.0.0.21: icmp_seq=29 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=30 ttl=64 time=0.065 ms
64 bytes from 10.0.0.21: icmp_seq=31 ttl=64 time=0.071 ms
64 bytes from 10.0.0.21: icmp_seq=32 ttl=64 time=0.066 ms
64 bytes from 10.0.0.21: icmp_seq=33 ttl=64 time=0.064 ms
^C
--- 10.0.0.21 ping statistics ---
33 packets transmitted, 33 received, 0% packet loss, time 32040ms
rtt min/avg/max/mdev = 0.064/114.559/151.225/48.252 ms
root@mininet-vm:~#
```

```
root@mininet-vm:~# ping -c 45 10.0.0.21
PING 10.0.0.21 (10.0.0.21) 56(84) bytes of data.
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=157 ms
64 bytes from 10.0.0.21: icmp_seq=2 ttl=64 time=160 ms
64 bytes from 10.0.0.21: icmp_seq=3 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=4 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=5 ttl=64 time=158 ms
64 bytes from 10.0.0.21: icmp_seq=6 ttl=64 time=157 ms
64 bytes from 10.0.0.21: icmp_seq=7 ttl=64 time=160 ms
64 bytes from 10.0.0.21: icmp_seq=8 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=9 ttl=64 time=161 ms
64 bytes from 10.0.0.21: icmp_seq=10 ttl=64 time=160 ms
64 bytes from 10.0.0.21: icmp_seq=11 ttl=64 time=160 ms
64 bytes from 10.0.0.21: icmp_seq=12 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=13 ttl=64 time=155 ms
64 bytes from 10.0.0.21: icmp_seq=14 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=15 ttl=64 time=161 ms
64 bytes from 10.0.0.21: icmp_seq=16 ttl=64 time=162 ms
64 bytes from 10.0.0.21: icmp_seq=17 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=18 ttl=64 time=157 ms
64 bytes from 10.0.0.21: icmp_seq=19 ttl=64 time=159 ms
64 bytes from 10.0.0.21: icmp_seq=20 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=21 ttl=64 time=159 ms
64 bytes from 10.0.0.21: icmp_seq=22 ttl=64 time=161 ms
64 bytes from 10.0.0.21: icmp_seq=23 ttl=64 time=155 ms
64 bytes from 10.0.0.21: icmp_seq=24 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=25 ttl=64 time=162 ms
64 bytes from 10.0.0.21: icmp_seq=26 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=27 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=28 ttl=64 time=160 ms
64 bytes from 10.0.0.21: icmp_seq=29 ttl=64 time=159 ms
64 bytes from 10.0.0.21: icmp_seq=30 ttl=64 time=0.027 ms
64 bytes from 10.0.0.21: icmp_seq=31 ttl=64 time=0.049 ms
64 bytes from 10.0.0.21: icmp_seq=32 ttl=64 time=0.052 ms
64 bytes from 10.0.0.21: icmp_seq=33 ttl=64 time=0.068 ms
^C
--- 10.0.0.21 ping statistics ---
33 packets transmitted, 33 received, 0% packet loss, time 32038ms
rtt min/avg/max/mdev = 0.027/139.368/162.781/51.784 ms
root@mininet-vm:~#
```

```
root@mininet-vm:~# ping -c 45 10.0.0.21
PING 10.0.0.21 (10.0.0.21) 56(84) bytes of data.
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=181 ms
64 bytes from 10.0.0.21: icmp_seq=2 ttl=64 time=183 ms
64 bytes from 10.0.0.21: icmp_seq=3 ttl=64 time=183 ms
64 bytes from 10.0.0.21: icmp_seq=4 ttl=64 time=182 ms
64 bytes from 10.0.0.21: icmp_seq=5 ttl=64 time=181 ms
64 bytes from 10.0.0.21: icmp_seq=6 ttl=64 time=182 ms
64 bytes from 10.0.0.21: icmp_seq=7 ttl=64 time=183 ms
64 bytes from 10.0.0.21: icmp_seq=8 ttl=64 time=187 ms
64 bytes from 10.0.0.21: icmp_seq=9 ttl=64 time=219 ms
64 bytes from 10.0.0.21: icmp_seq=10 ttl=64 time=190 ms
64 bytes from 10.0.0.21: icmp_seq=11 ttl=64 time=182 ms
64 bytes from 10.0.0.21: icmp_seq=12 ttl=64 time=181 ms
64 bytes from 10.0.0.21: icmp_seq=13 ttl=64 time=182 ms
64 bytes from 10.0.0.21: icmp_seq=14 ttl=64 time=183 ms
64 bytes from 10.0.0.21: icmp_seq=15 ttl=64 time=189 ms
64 bytes from 10.0.0.21: icmp_seq=16 ttl=64 time=183 ms
64 bytes from 10.0.0.21: icmp_seq=17 ttl=64 time=182 ms
64 bytes from 10.0.0.21: icmp_seq=18 ttl=64 time=191 ms
64 bytes from 10.0.0.21: icmp_seq=19 ttl=64 time=182 ms
64 bytes from 10.0.0.21: icmp_seq=20 ttl=64 time=192 ms
64 bytes from 10.0.0.21: icmp_seq=21 ttl=64 time=187 ms
64 bytes from 10.0.0.21: icmp_seq=22 ttl=64 time=181 ms
64 bytes from 10.0.0.21: icmp_seq=23 ttl=64 time=182 ms
64 bytes from 10.0.0.21: icmp_seq=24 ttl=64 time=188 ms
64 bytes from 10.0.0.21: icmp_seq=25 ttl=64 time=182 ms
64 bytes from 10.0.0.21: icmp_seq=26 ttl=64 time=182 ms
64 bytes from 10.0.0.21: icmp_seq=27 ttl=64 time=189 ms
64 bytes from 10.0.0.21: icmp_seq=28 ttl=64 time=182 ms
64 bytes from 10.0.0.21: icmp_seq=29 ttl=64 time=193 ms
64 bytes from 10.0.0.21: icmp_seq=30 ttl=64 time=0.066 ms
64 bytes from 10.0.0.21: icmp_seq=31 ttl=64 time=0.081 ms
64 bytes from 10.0.0.21: icmp_seq=32 ttl=64 time=0.061 ms
64 bytes from 10.0.0.21: icmp_seq=33 ttl=64 time=0.077 ms
^C
--- 10.0.0.21 ping statistics ---
33 packets transmitted, 33 received, 0% packet loss, time 32039ms
rtt min/avg/max/mdev = 0.061/163.641/219.299/61.135 ms
root@mininet-vm:~#
```

Figure 4.14: Priority Case : Connectivity Verification using pings

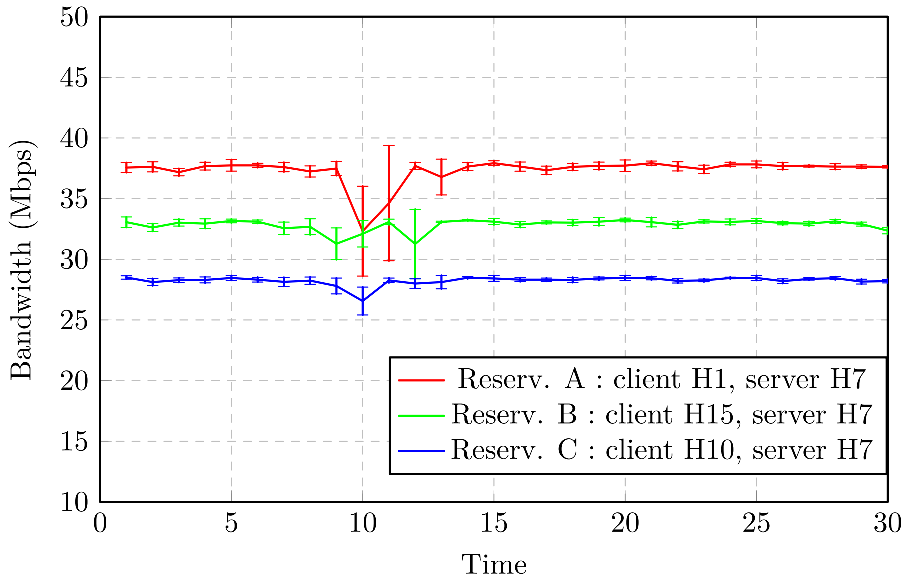


Figure 4.15: Load Balance, Priority Case : Bandwidth

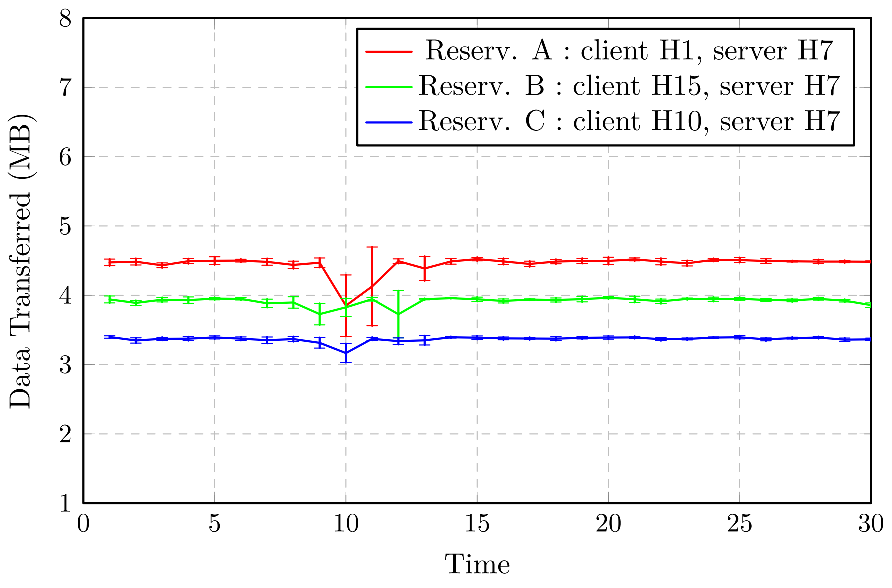
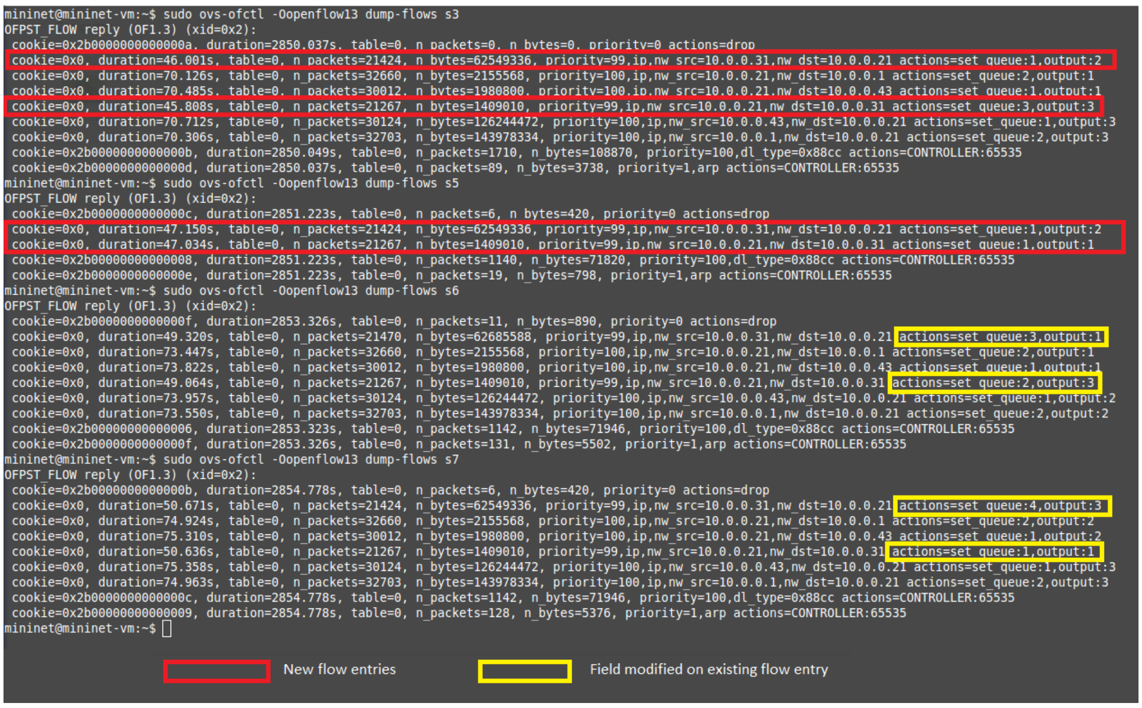


Figure 4.16: Load Balance, Priority Case : Data Transfer

4.4.2.2 maximum rate case



```
mininet@mininet-vm:~$ sudo ovs-ofctl -Oopenflow13 dump-flows s3
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b000000000000a, duration=2850.037s, table=0, n_packets=0, n_bytes=0, priority=0 actions=drop
 cookie=0x0, duration=46.001s, table=0, n_packets=21424, n_bytes=62549336, priority=99,ip,nw_src=10.0.0.31,nw_dst=10.0.0.21 actions=set_queue:1,output:2
 cookie=0x0, duration=70.126s, table=0, n_packets=32660, n_bytes=2155568, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.1 actions=set_queue:2,output:1
 cookie=0x0, duration=70.485s, table=0, n_packets=30012, n_bytes=1980800, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.43 actions=set_queue:1,output:1
 cookie=0x0, duration=45.808s, table=0, n_packets=21267, n_bytes=1409010, priority=99,ip,nw_src=10.0.0.21,nw_dst=10.0.0.31 actions=set_queue:3,output:3
 cookie=0x0, duration=70.712s, table=0, n_packets=30124, n_bytes=126244472, priority=100,ip,nw_src=10.0.0.43,nw_dst=10.0.0.21 actions=set_queue:1,output:3
 cookie=0x0, duration=70.306s, table=0, n_packets=32703, n_bytes=143978334, priority=100,ip,nw_src=10.0.0.1,nw_dst=10.0.0.21 actions=set_queue:2,output:3
 cookie=0x2b000000000000b, duration=2850.049s, table=0, n_packets=1710, n_bytes=108870, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b000000000000d, duration=2850.037s, table=0, n_packets=89, n_bytes=3738, priority=1,arp actions=CONTROLLER:65535
mininet@mininet-vm:~$ sudo ovs-ofctl -Oopenflow13 dump-flows s5
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b000000000000c, duration=2851.223s, table=0, n_packets=6, n_bytes=420, priority=0 actions=drop
 cookie=0x0, duration=47.150s, table=0, n_packets=21424, n_bytes=62549336, priority=99,ip,nw_src=10.0.0.31,nw_dst=10.0.0.21 actions=set_queue:1,output:2
 cookie=0x0, duration=47.034s, table=0, n_packets=21267, n_bytes=1409010, priority=99,ip,nw_src=10.0.0.21,nw_dst=10.0.0.31 actions=set_queue:1,output:1
 cookie=0x2b0000000000008, duration=2851.223s, table=0, n_packets=1140, n_bytes=71820, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b000000000000e, duration=2851.223s, table=0, n_packets=19, n_bytes=798, priority=1,arp actions=CONTROLLER:65535
mininet@mininet-vm:~$ sudo ovs-ofctl -Oopenflow13 dump-flows s6
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b000000000000f, duration=2853.326s, table=0, n_packets=11, n_bytes=890, priority=0 actions=drop
 cookie=0x0, duration=49.320s, table=0, n_packets=21470, n_bytes=62685588, priority=99,ip,nw_src=10.0.0.31,nw_dst=10.0.0.21 actions=set_queue:3,output:1
 cookie=0x0, duration=73.447s, table=0, n_packets=32660, n_bytes=2155568, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.1 actions=set_queue:2,output:1
 cookie=0x0, duration=73.822s, table=0, n_packets=30012, n_bytes=1980800, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.43 actions=set_queue:1,output:1
 cookie=0x0, duration=49.064s, table=0, n_packets=21267, n_bytes=1409010, priority=99,ip,nw_src=10.0.0.21,nw_dst=10.0.0.31 actions=set_queue:2,output:3
 cookie=0x0, duration=73.957s, table=0, n_packets=30124, n_bytes=126244472, priority=100,ip,nw_src=10.0.0.43,nw_dst=10.0.0.21 actions=set_queue:1,output:2
 cookie=0x0, duration=73.550s, table=0, n_packets=32703, n_bytes=143978334, priority=100,ip,nw_src=10.0.0.1,nw_dst=10.0.0.21 actions=set_queue:2,output:2
 cookie=0x2b0000000000006, duration=2853.323s, table=0, n_packets=1142, n_bytes=71946, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b000000000000f, duration=2853.326s, table=0, n_packets=131, n_bytes=5502, priority=1,arp actions=CONTROLLER:65535
mininet@mininet-vm:~$ sudo ovs-ofctl -Oopenflow13 dump-flows s7
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b000000000000b, duration=2854.778s, table=0, n_packets=6, n_bytes=420, priority=0 actions=drop
 cookie=0x0, duration=50.671s, table=0, n_packets=21424, n_bytes=62549336, priority=99,ip,nw_src=10.0.0.31,nw_dst=10.0.0.21 actions=set_queue:4,output:3
 cookie=0x0, duration=74.924s, table=0, n_packets=32660, n_bytes=2155568, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.1 actions=set_queue:2,output:2
 cookie=0x0, duration=75.310s, table=0, n_packets=30012, n_bytes=1980800, priority=100,ip,nw_src=10.0.0.21,nw_dst=10.0.0.43 actions=set_queue:1,output:2
 cookie=0x0, duration=50.636s, table=0, n_packets=21267, n_bytes=1409010, priority=99,ip,nw_src=10.0.0.21,nw_dst=10.0.0.31 actions=set_queue:1,output:1
 cookie=0x0, duration=75.358s, table=0, n_packets=30124, n_bytes=126244472, priority=100,ip,nw_src=10.0.0.43,nw_dst=10.0.0.21 actions=set_queue:1,output:3
 cookie=0x0, duration=74.963s, table=0, n_packets=32703, n_bytes=143978334, priority=100,ip,nw_src=10.0.0.1,nw_dst=10.0.0.21 actions=set_queue:2,output:3
 cookie=0x2b000000000000c, duration=2854.778s, table=0, n_packets=1142, n_bytes=71946, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b0000000000009, duration=2854.778s, table=0, n_packets=128, n_bytes=5376, priority=1,arp actions=CONTROLLER:65535
mininet@mininet-vm:~$ 
```

Figure 4.17: Maximum Rate Case : Flow Tables after Load Balance enforcement

The reservation with lowest maximum rate is the one between hosts 'H10' (10.0.0.31) and 'H7' (10.0.0.21). An alternate path, that would not cross any overloaded link, was searched and found. The new path established was 'openflow:6:1-openflow:3:2-openflow:5:2-openflow:7:3'. By observing Figure 4.17 it is noticeable the switch 'openflow:3' and 'openflow:5' have two more flow entries each one. The switch 'openflow:6' still preserves all flow entries but now the packets between 'H10' and 'H15' are redirected to switch 'openflow:3' through port 1. In turn, 'openflow:3' redirects the packets through port 2 and switch 'openflow:5' through port 2. The new flow entries added on switch 'openflow:3' and 'openflow:5' are inside a red box and the fields changed on two existing flow entries on switch 'openflow:6' are inside a yellow box, Figure 4.12. The new path established is shown in Figure 4.18.

Note that, on the explanation above only the path (with ports) between client ('H10') and server ('H7') is mentioned. The path between server and client works the same way being that the reason why it has not also been mentioned.

Through Figures 4.19 and also Figure 4.20 we verify that no connection was lost during the path replacement process, even though it was a different reservation. In addition, we can also verify that no packet was lost during the process because the number of packets dropped before and after the Iperf test is the same. This is depicted by observing the 'drop' flow rule in Figures 4.10 (before) and 4.17 (after in Rate case). This is due to the fact that the new path was created before tearing down the old one (make-before-break). The correspondent bandwidth and RTT from each reservation remained quite the same during the whole process.

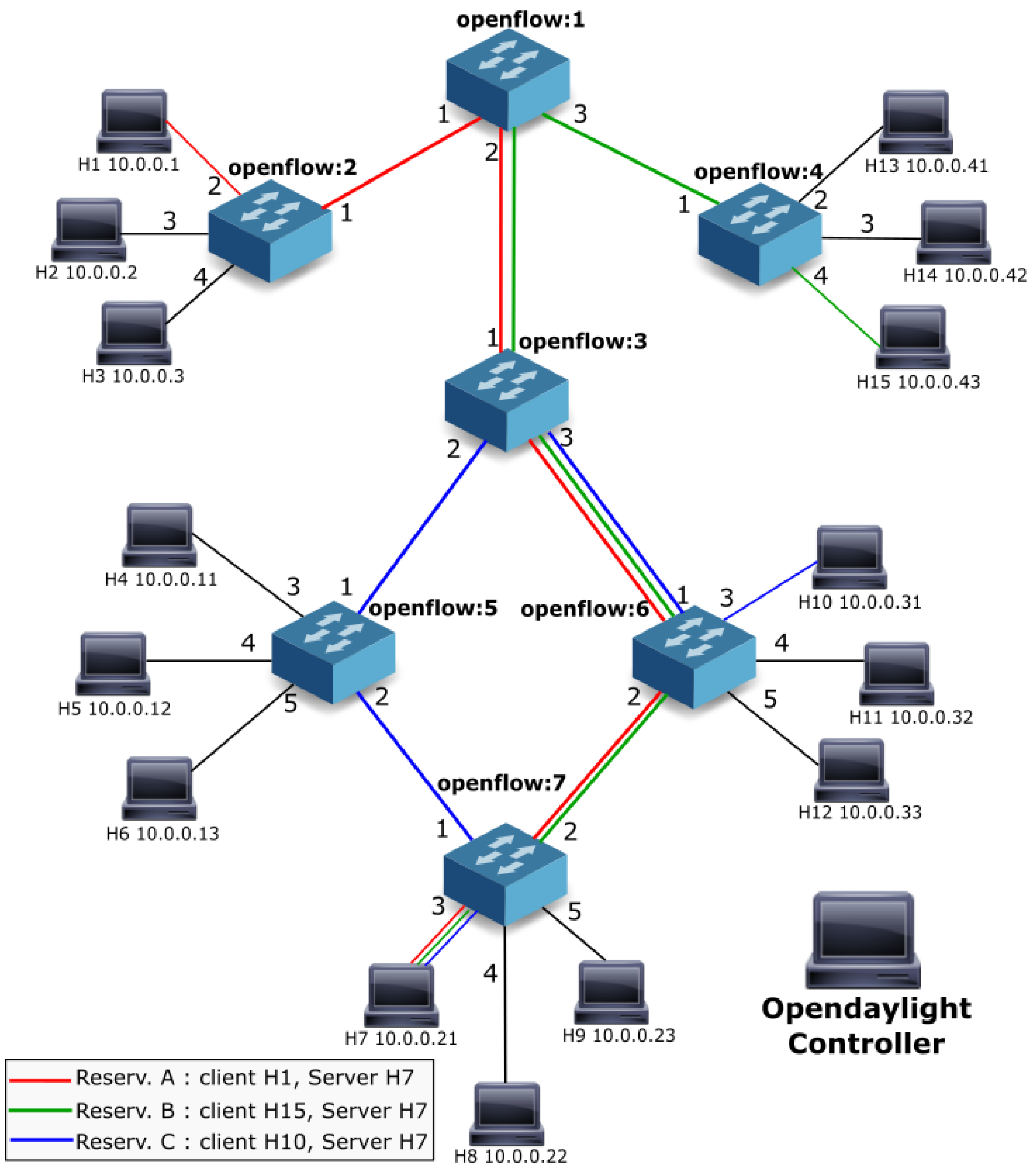
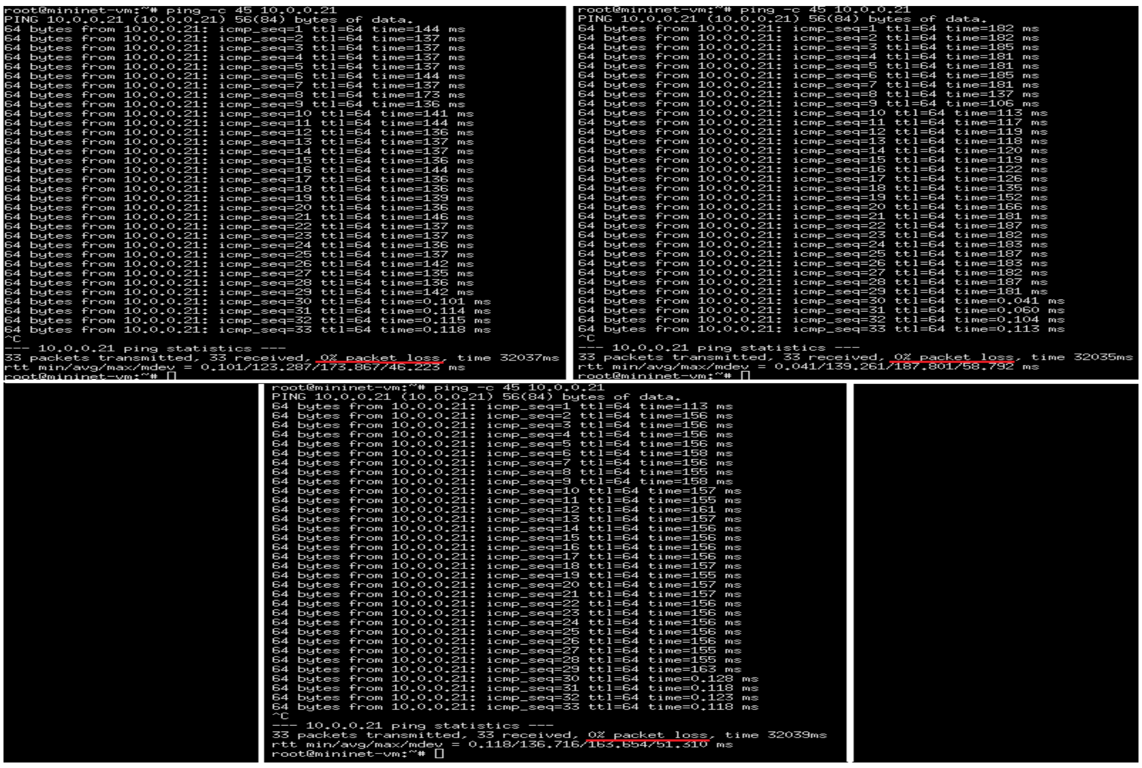


Figure 4.18: Network topology with the paths, after the application of the load balance mechanism with the lowest max-rate parameter



```
root@mininet-vm:~# ping -c 45 10.0.0.21
PING 10.0.0.21 (10.0.0.21) 56(84) bytes of data.
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=144 ms
64 bytes from 10.0.0.21: icmp_seq=2 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=3 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=4 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=5 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=6 ttl=64 time=144 ms
64 bytes from 10.0.0.21: icmp_seq=7 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=8 ttl=64 time=173 ms
64 bytes from 10.0.0.21: icmp_seq=9 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=10 ttl=64 time=141 ms
64 bytes from 10.0.0.21: icmp_seq=11 ttl=64 time=144 ms
64 bytes from 10.0.0.21: icmp_seq=12 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=13 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=14 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=15 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=16 ttl=64 time=144 ms
64 bytes from 10.0.0.21: icmp_seq=17 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=18 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=19 ttl=64 time=139 ms
64 bytes from 10.0.0.21: icmp_seq=20 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=21 ttl=64 time=146 ms
64 bytes from 10.0.0.21: icmp_seq=22 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=23 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=24 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=25 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=26 ttl=64 time=142 ms
64 bytes from 10.0.0.21: icmp_seq=27 ttl=64 time=135 ms
64 bytes from 10.0.0.21: icmp_seq=28 ttl=64 time=136 ms
64 bytes from 10.0.0.21: icmp_seq=29 ttl=64 time=142 ms
64 bytes from 10.0.0.21: icmp_seq=30 ttl=64 time=0.101 ms
64 bytes from 10.0.0.21: icmp_seq=31 ttl=64 time=0.114 ms
64 bytes from 10.0.0.21: icmp_seq=32 ttl=64 time=0.115 ms
64 bytes from 10.0.0.21: icmp_seq=33 ttl=64 time=0.118 ms
^C
--- 10.0.0.21 ping statistics ---
33 packets transmitted, 33 received, 0% packet loss, time 32037ms
rtt min/avg/max/mdev = 0.101/123.287/173.867/46.223 ms
root@mininet-vm:~# 
```

```
root@mininet-vm:~# ping -c 45 10.0.0.21
PING 10.0.0.21 (10.0.0.21) 56(84) bytes of data.
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=182 ms
64 bytes from 10.0.0.21: icmp_seq=2 ttl=64 time=182 ms
64 bytes from 10.0.0.21: icmp_seq=3 ttl=64 time=185 ms
64 bytes from 10.0.0.21: icmp_seq=4 ttl=64 time=181 ms
64 bytes from 10.0.0.21: icmp_seq=5 ttl=64 time=181 ms
64 bytes from 10.0.0.21: icmp_seq=6 ttl=64 time=185 ms
64 bytes from 10.0.0.21: icmp_seq=7 ttl=64 time=181 ms
64 bytes from 10.0.0.21: icmp_seq=8 ttl=64 time=137 ms
64 bytes from 10.0.0.21: icmp_seq=9 ttl=64 time=106 ms
64 bytes from 10.0.0.21: icmp_seq=10 ttl=64 time=113 ms
64 bytes from 10.0.0.21: icmp_seq=11 ttl=64 time=117 ms
64 bytes from 10.0.0.21: icmp_seq=12 ttl=64 time=119 ms
64 bytes from 10.0.0.21: icmp_seq=13 ttl=64 time=118 ms
64 bytes from 10.0.0.21: icmp_seq=14 ttl=64 time=120 ms
64 bytes from 10.0.0.21: icmp_seq=15 ttl=64 time=119 ms
64 bytes from 10.0.0.21: icmp_seq=16 ttl=64 time=122 ms
64 bytes from 10.0.0.21: icmp_seq=17 ttl=64 time=126 ms
64 bytes from 10.0.0.21: icmp_seq=18 ttl=64 time=135 ms
64 bytes from 10.0.0.21: icmp_seq=19 ttl=64 time=152 ms
64 bytes from 10.0.0.21: icmp_seq=20 ttl=64 time=166 ms
64 bytes from 10.0.0.21: icmp_seq=21 ttl=64 time=181 ms
64 bytes from 10.0.0.21: icmp_seq=22 ttl=64 time=187 ms
64 bytes from 10.0.0.21: icmp_seq=23 ttl=64 time=182 ms
64 bytes from 10.0.0.21: icmp_seq=24 ttl=64 time=183 ms
64 bytes from 10.0.0.21: icmp_seq=25 ttl=64 time=187 ms
64 bytes from 10.0.0.21: icmp_seq=26 ttl=64 time=183 ms
64 bytes from 10.0.0.21: icmp_seq=27 ttl=64 time=182 ms
64 bytes from 10.0.0.21: icmp_seq=28 ttl=64 time=187 ms
64 bytes from 10.0.0.21: icmp_seq=29 ttl=64 time=181 ms
64 bytes from 10.0.0.21: icmp_seq=30 ttl=64 time=0.041 ms
64 bytes from 10.0.0.21: icmp_seq=31 ttl=64 time=0.060 ms
64 bytes from 10.0.0.21: icmp_seq=32 ttl=64 time=0.104 ms
64 bytes from 10.0.0.21: icmp_seq=33 ttl=64 time=0.113 ms
^C
--- 10.0.0.21 ping statistics ---
33 packets transmitted, 33 received, 0% packet loss, time 32035ms
rtt min/avg/max/mdev = 0.041/139.261/187.801/58.792 ms
root@mininet-vm:~# 
```

```
root@mininet-vm:~# ping -c 45 10.0.0.21
PING 10.0.0.21 (10.0.0.21) 56(84) bytes of data.
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=113 ms
64 bytes from 10.0.0.21: icmp_seq=2 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=3 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=4 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=5 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=6 ttl=64 time=158 ms
64 bytes from 10.0.0.21: icmp_seq=7 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=8 ttl=64 time=155 ms
64 bytes from 10.0.0.21: icmp_seq=9 ttl=64 time=158 ms
64 bytes from 10.0.0.21: icmp_seq=10 ttl=64 time=157 ms
64 bytes from 10.0.0.21: icmp_seq=11 ttl=64 time=155 ms
64 bytes from 10.0.0.21: icmp_seq=12 ttl=64 time=161 ms
64 bytes from 10.0.0.21: icmp_seq=13 ttl=64 time=157 ms
64 bytes from 10.0.0.21: icmp_seq=14 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=15 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=16 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=17 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=18 ttl=64 time=157 ms
64 bytes from 10.0.0.21: icmp_seq=19 ttl=64 time=155 ms
64 bytes from 10.0.0.21: icmp_seq=20 ttl=64 time=157 ms
64 bytes from 10.0.0.21: icmp_seq=21 ttl=64 time=157 ms
64 bytes from 10.0.0.21: icmp_seq=22 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=23 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=24 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=25 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=26 ttl=64 time=156 ms
64 bytes from 10.0.0.21: icmp_seq=27 ttl=64 time=155 ms
64 bytes from 10.0.0.21: icmp_seq=28 ttl=64 time=155 ms
64 bytes from 10.0.0.21: icmp_seq=29 ttl=64 time=163 ms
64 bytes from 10.0.0.21: icmp_seq=30 ttl=64 time=0.128 ms
64 bytes from 10.0.0.21: icmp_seq=31 ttl=64 time=0.118 ms
64 bytes from 10.0.0.21: icmp_seq=32 ttl=64 time=0.123 ms
64 bytes from 10.0.0.21: icmp_seq=33 ttl=64 time=0.118 ms
^C
--- 10.0.0.21 ping statistics ---
33 packets transmitted, 33 received, 0% packet loss, time 32039ms
rtt min/avg/max/mdev = 0.118/136.716/163.654/51.310 ms
root@mininet-vm:~# 
```

Figure 4.19: Maximum Rate Case : Connectivity Verification using pings

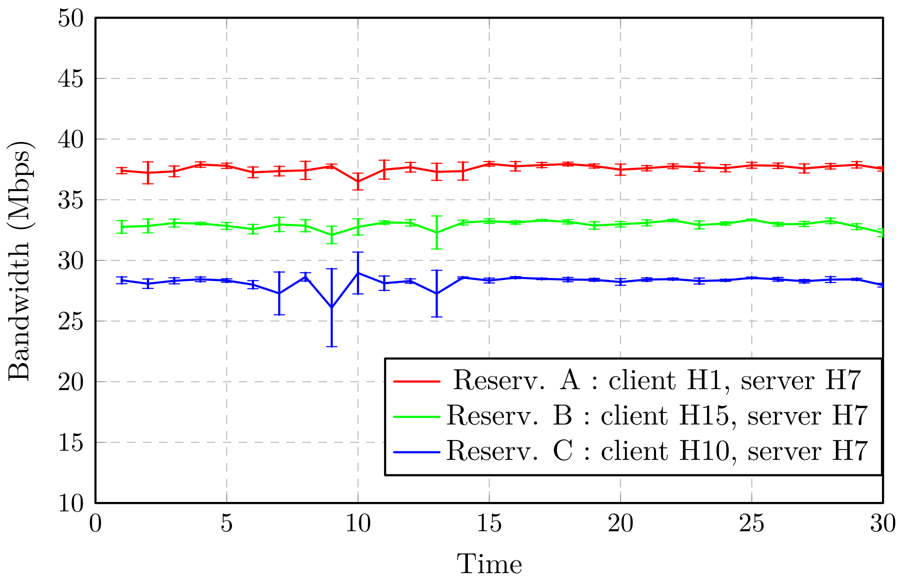


Figure 4.20: Load Balance, Maximum Rate Case : Bandwidth

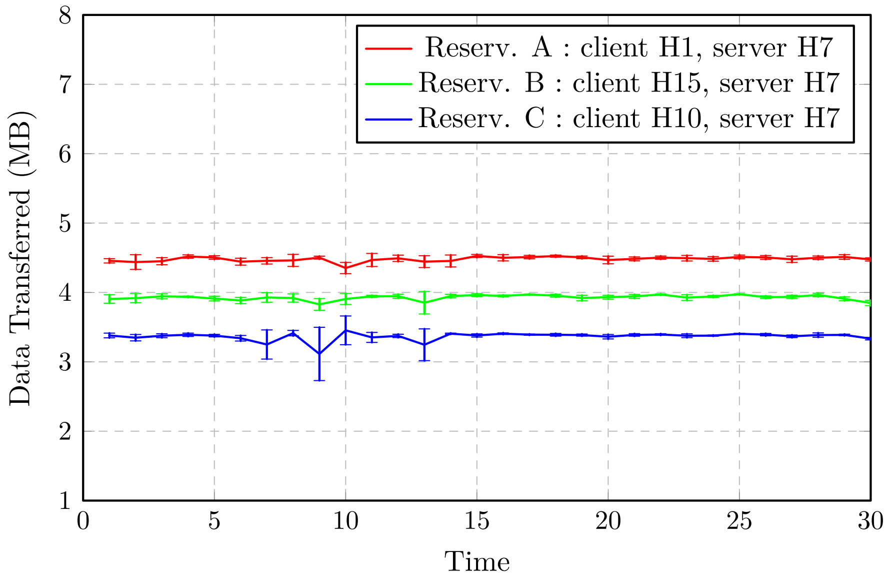


Figure 4.21: Load Balance, Maximum Rate Case : Data Transfer

4.4.3 CONCLUSION

In conclusion there are three important factors to point out. Firstly, the connection between hosts is not lost as well as it is also not affected by the 'Path Replacement' process. Secondly, the selection parameters of a reservation do not impact directly on the delay of the mechanism. Those parameters allow us to choose a given reservation first depending on a given reason or a policy established by the network manager. For example, the maximum rate parameter, mentioned on SubSection 4.4.1, allows reservations with lower bandwidth to be selected first, which could translate into a better resource utilization.

Lastly, on one hand, the delay from the 'Path Discovery' task from the Load Balance mechanism is mainly due to the length of the path and how many reservations were processed to find an alternate path that is valid. On the other hand, the delay from the 'Path Replacement' task from the Load Balance mechanism is mainly due to the length of the path that has to be replaced. The larger the path, more flows and queues have to be replaced and thus the longer it will take for the task is complete.

It is worth to mention that the decrease of throughput that ocurred around $Time = 10s$, in both cases, is due to the increase of processing of the machine. During the execution of these tests we have noticed that when the load on the machine increases, the iperf tests decreases the amount of traffic sent between the nodes during these moments. A similar case happenned when performing iperf tests and, at the same time, capturing the traffic with wireshark [4]. In those tests, the bandwidth results would be relatively lower than the results when wireshark was not running. In short, when the VM is overloaded or the system in general is overloaded or the lack of CPU time to the hosts and switches may lower the performance in iperf tests. Notice that, the whole system is running in the same machine, the controller, the application and all the hosts and switches of the emulated network.

In this particular case the machine load increased when recalculating and sending the flows and queues to the controller. On those brief particular moments, iperf reduced the amount of traffic sent and thus we observed a small decrease in terms of bandwidth that is noticeable in both Figure 4.15 and Figure 4.20. However, as we mentioned above, no packet was lost during the whole test.

[4]https://www.wireshark.org/

CHAPTER 5

Conclusion

5.1 Work Overview

There are several specific purpose applications for QoS and Load balance that perform well in SDN environments, however there is a lack of such application that is external and can be used by several orchestration domains. Current applications are incorporated in the controller they select to work, which could be a barrier to be used by different controller implementations. QLAMES is an application that bears in mind the fact of the eventual standardization of NB APIs, which presents a huge advantage in comparison to other controller's specific applications. That is because it will be able to work with all controller's implementations, as well as being coordinated by several orchestration domains without the need of changing/adapting the application to work with other internal controller APIs. In addition, it incorporates such mechanisms that provide several functionalities (QoS on-demand, monitoring and load balance).

The main objective of this dissertation was to create such an application that could be used by any orchestration domain without the need of creating a new plugin for each controller.

Taking in account the SDN paradigm, there is a lack of such an application that can be integrated with any orchestration domain. For example Openstack has implemented several plugins in several controllers (Ryu, Floodlight, Opendaylight) to allow, in a more easy way, the control of the network using the internal API of each controller. This proof of concept avoids the necessity of creating such API for every controller they want to integrate with, which brings a huge advantage. This way, there is no need to create a plugin that has to adapt to every controller internal API.

The OpenDaylight controller was the elected one to interact with our application. OpenDaylight is a simple controller, easy to install and use, due to the possibility of using a pre-built version. This pre-built version disables the need to compile the controller, because only the NB REST API of the controller was used. In addition, there are several Postman [1] collections that provide knowledge, to developers, about the format of their NB API. However, the information about how to insert queues, using OpenDaylight's OVSDB subsystem, is not so easy to find and understand. Overall, Opendaylight is a good choice to enter in the SDN environment due to its capability of being easy to understand, develop and integrate network mechanisms. It enables the creation of different networking functionalities, in different ways, and/or extending an already existing functionality. In addition, it

[1]https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop

provides several SB protocols to be used which greatly increases the value of using it.

The Mininet emulator was used to create custom topologies (using their Python API), ideal to simulate the performance of the application as it was in a data center environment. Mininet included, by default, two virtual forwarding devices, *Openvswitch* and *ofsoftswitch*. While the first was kernel-based, the second was user-space software switch implementation and thus less efficient. SDN is not well prepared with mechanisms to provide QoS in an easy manner. However, after some research two solutions were found. The first was the use of linux queues while the second was through the use of meters, a mechanism that was included in openflow version 1.3 specification. Openvswitch was the choice due to its performance and being used by the majority of the community, but it didn't support openflow's meter so the queue approach was chosen.

Mininet brought, by default, an older version of Openvswitch that wasn't able to insert the queues, essential to providing the QoS requirememts specified in each reservation. The version was upgraded into a more recent stable version (2.3.1), which allows the creation of the above mentioned queues. When testing the application, we have noticed that the increase of the load of the virtual machine and other applications running on it would influenciate the results obtained (for example while running Wireshark, the iperf tests showed a decrease of bandwidth that could not be explained and if it wasn't running all it would be normal). The second thing we have noticed was that there is a bug on openvswitch, which would cause scalability problems to the proof of concept. The bug consists on the disconnection of the openvswitch connection manager responsible for handling operations such as inserting/removing data from the openvswitch database. When the amount of data is too high in the database, the connection is closed and can no longer be extablished unless the whole database is erased, which would imply the deletion of the queues essential for the QoS provisioning. Despite of the bug being known by the Openvswitch community, there is currently no solution to the problem. The bug was only detected when doing performance tests of the application.

The bug previously mentioned translates into a scalability problem, because it limits the maximum number of queues that can be currently exist in the network. Despite this particular issue, the application has a reasonable performance. It being capable of installing, at the same time, a couple thousands of flows and queues into the network within 1 minute. The tear down process is slightly faster than the installation one. Although deleting from the database requires more time than to insert, it was not needed to do the amount of processing that installation process had to do because the necessary information was already stored in the database. The testing stage also proved that we were able to ensure that the QoS provisioning is correctly established for each reservation within the periods specified by it. In addition, the load balance mechanism is correctly applied without loosing any packet with the change of path.

Overall the application is quite simple, only requiring to the administrator or other orchestration tool to have perfect knowledge of the underlying network and inserting the reservation into the database with all the necessary parameters. thenceforward the application acts automatically, enabling the communication, provisioning the QoS parameters for each reservation and by monitoring the whole network performs load balancing in order to avoid bottlenecks, which in turn decrease the probability of a congestion scenario caused by bursts of traffic between the hosts.

5.2 FUTURE WORK

Even though the proof of concept was successfully implemented and has a decent performance, improvements and extensions may be added in order to increase its quality and provide extra functionality. Some of the most important functionalities that can be improved and implemented in the future are:

- Change the current sequential implementation into a concurrent solution in order to improve the installation/removal of flows and queues in the network.
- Alter the REST requests according to the standard NB API that will eventually be defined.
- Create a fault tolerance and recovery mechanism using several distributed controllers through clustering.
- Construct a Web user inteface to manage the application.

References

[1] S. Shenker, M Casado, T. Koponen, N McKeown, *et al.*, "The future of networking, and the past of protocols", *Open Networking Summit*, 2011.

[2] T. Benson, A. Akella, and D. A. Maltz, "Unraveling the complexity of network management.", in *NSDI*, 2009, pp. 335–348.

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

[4] J. Pan, S. Paul, and R. Jain, "A survey of the research on future internet architectures", *Communications Magazine, IEEE*, vol. 49, no. 7, pp. 26–36, 2011.

[5] A. Ghodsi, S. Shenker, T. Koponen, A. Singla, B. Raghavan, and J. Wilcox, "Intelligent design enables architectural evolution", in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, ACM, 2011, p. 3.

[6] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking a comprehensive survey", *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[7] M. Jammal, T. Singh, A. Shami, R. Asal, and Y. Li, "Software defined networking: State of the art and research challenges", *Computer Networks*, vol. 72, pp. 74–98, 2014.

[8] "Business paper : Deliver hp virtual application networks", HP, Tech. Rep., 2012. [Online]. Available: `http://h17007.www1.hp.com/docs/interopny/4AA4-3872ENW.pdf`.

[9] "Technical white paper : Network transformation with software-defined networking and ethernet fabrics", HP, Tech. Rep., 2012. [Online]. Available: `http://h17007.www1.hp.com/docs/interopny/4AA4-3871ENW.pdf`.

[10] "Positioning paper : Network transformation with software-defined networking and ethernet fabrics", Brocade Communications Systems, Tech. Rep., 2012. [Online]. Available: `http://community.brocade.com/dtscp75322/attachments/dtscp75322/AttendEvents/19/1/network-transformation-sdn-wp.pdf`.

[11] B. Laliberte, "Esg research brief - 'are legacy networks holding back application deployments?'", ESG, Tech. Rep., 2012. [Online]. Available: `http://www.plexxi.com/wp-content/uploads/2012/08/ESG-Research-Brief-Aug-2012.pdf`.

[12] (2015). Interoute - 'what is a private cloud', [Online]. Available: `http://www.interoute.com/cloud-article/what-private-cloud`.

[13] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn", *Queue*, vol. 11, no. 12, p. 20, 2013.

[14] "Specification area", ONF, Tech. Rep., 2013. [Online]. Available: `https://www.opennetworking.org/technical-communities/areas/specification`.

[15] "Charter: Extensibility working group", ONF, Tech. Rep., 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/working-groups/charter-forwarding-abstractions.pdf.

[16] "Charter: Forwarding abstractions working group", ONF, Tech. Rep., 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/working-groups/charter-forwarding-abstractions.pdf.

[17] "Charter: Test and certification working group", ONF, Tech. Rep., 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/working-groups/charter-forwarding-abstractions.pdf.

[18] S. R. (HP) and D. L. (Plexxi), "Northbound interfaces", ONF, Tech. Rep., 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/working-groups/charter-nbi.pdf.

[19] (2015). Open networking foundation introduces northbound interface working group, [Online]. Available: https://www.opennetworking.org/news-and-events/press-releases/1182-open-networking-foundation-introduces-northbound-interface-working-group.

[20] B Nunes, M. Mendonca, X Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking past, present, and future of programmable networks", 2014.

[21] O. N. Fundation, "Software-defined networking: The new norm for networks", *ONF White Paper*, 2012.

[22] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, *et al.*, "Onos: Towards an open, distributed sdn os", in *Proceedings of the third workshop on Hot topics in software defined networking*, ACM, 2014, pp. 1–6.

[23] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, *et al.*, "Onix: A distributed control platform for large-scale production networks.", in *OSDI*, vol. 10, 2010, pp. 1–6.

[24] (2015). Opendaylight controller - md-sal architecture : Clustering, [Online]. Available: https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Architecture:Clustering.

[25] V. Yazici, M. O. Sunay, and A. O. Ercan, "Controlling a software-defined network via distributed controllers", *ArXiv preprint arXiv:1401.7651*, 2014.

[26] B. Pfaff and B. Davie, "The open vswitch database management protocol", RFC Editor, RFC 7047, 2013, http://www.rfc-editor.org/rfc/rfc7047.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7047.txt.

[27] J. D. Case, M. Fedor, M. L. Schoffstall, and J. R. Davin, "Simple network management protocol (snmp)", RFC Editor, STD 15, 1990, http://www.rfc-editor.org/rfc/rfc1157.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1157.txt.

[28] (2015). A primer on northbound apis: Their role in a software-defined network, [Online]. Available: http://searchsdn.techtarget.com/feature/A-primer-on-northbound-APIs-Their-role-in-a-software-defined-network.

[29] (2015). Clarifying the role of software-defined networking northbound apis, [Online]. Available: http://www.networkworld.com/article/2165901/lan-wan/clarifying-the-role-of-software-defined-networking-northbound-apis.html.

[30] (2015). Do sdn northbound apis need standards?, [Online]. Available: http://searchnetworking.techtarget.com/feature/Do-SDN-northbound-APIs-need-standards.

[31] (2015). Where do we stand with sdn's northbound interface?, [Online]. Available: `http://www.webtorials.com/content/2014/04/where-do-we-stand-with-sdns-northbound-interface.html`.

[32] (2015). 7 advantages of software defined networking, [Online]. Available: `http://www.ingrammicroadvisor.com/big-data/7-advantages-of-software-defined-networking`.

[33] (2015). Eight big benefits of software-defined networking, [Online]. Available: `http://www.serverwatch.com/server-tutorials/eight-big-benefits-of-software-defined-networking.html`.

[34] (2015). Enterprise networking: How cisco, hp, juniper, others are tackling sdn, openflow, [Online]. Available: `http://www.eweek.com/c/a/Enterprise-Networking/How-Cisco-HP-Juniper-Others-Are-Tackling-SDN-OpenFlow-183460`.

[35] (2015). Hp broadens openflow support with 16 switch models, [Online]. Available: `http://www.crn.com/news/networking/232600146/hp-broadens-openflow-support-with-16-switch-models.htm`.

[36] (2015). Nicira networks : Disruptive network virtualization, [Online]. Available: `http://web.stanford.edu/class/ee204/2012/Nicira%20Networks.pdf`.

[37] (2015). What's software-defined networking?, [Online]. Available: `https://www.sdxcentral.com/resources/sdn/what-the-definition-of-software-defined-networking-sdn/`.

[38] M. P. Papazoglou and W.-J. Van Den Heuvel, "Service oriented architectures: Approaches, technologies and research issues", *The VLDB journal*, vol. 16, no. 3, pp. 389–415, 2007.

[39] L. Youseff, M. Butrico, and D. Da Silva, "Toward a unified ontology of cloud computing", in *Grid Computing Environments Workshop, 2008. GCE'08*, IEEE, 2008, pp. 1–10.

[40] D. Duncan, X. Chu, C. Vecchiola, and R. Buyya, "The structure of the new it frontier: Cloud computing–part i", *Strategic Faciities Magazine-Pacific and Strategic Holdings Pte Ltd: Singapore*, pp. 67–72, 2009.

[41] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network configuration protocol (netconf)", RFC Editor, RFC 6241, 2011, urlhttpwww.rfc-editor.orgrfcrfc6241.txt. [Online]. Available: `httpwww.rfc-editor.orgrfcrfc6241.txt`.

[42] S. Sezer, S. Scott-Hayward, P.-K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for sdn? implementation challenges for software-defined networks", *Communications Magazine, IEEE*, vol. 51, no. 7, pp. 36–43, 2013.

[43] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern, "Forwarding and control element separation (forces) protocol specification", RFC Editor, RFC 5810, 2010, `http://www.rfc-editor.org/rfc/rfc5810.txt`. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc5810.txt`.

[44] "Charter: Migration working group", ONF, Tech. Rep., 2013. [Online]. Available: `https://www.opennetworking.org/images/stories/downloads/working-groups/charter-migration.pdf`.

[45] E. R. et al., "Migration use cases and methods", ONF, Tech. Rep., 2013. [Online]. Available: `https://www.opennetworking.org/images/stories/downloads/sdn-resources/use-cases/Migration-WG-Use-Cases.pdf`.

[46] OpenDaylight. (2014). Opendaylight a linux foundation collaborative project, [Online]. Available: `httpwww.opendaylight.org`.

[47] K. Lougheed and J. Rekhter, “Border gateway protocol (bgp)”, RFC Editor, RFC 1105, 1989, http://www.rfc-editor.org/rfc/rfc1105.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1105.txt.

[48] J. Vasseur and J. L. Roux, “Path computation element (pce) communication protocol (pcep)”, RFC Editor, RFC 5440, 2009, http://www.rfc-editor.org/rfc/rfc5440.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5440.txt.

[49] (2015). Ten things to look for in an sdn controller, [Online]. Available: https://www.necam.com/docs/?id=23865bd4-f10a-49f7-b6be-a17c61ad6fff.

[50] C. A. Macapuna, C. E. Rothenberg, and M. F. Magalhaes, “In-packet bloom filter based data center networking with distributed openflow controllers”, in *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, IEEE, 2010, pp. 584–588.

[51] A. Voellmy and J. Wang, “Scalable software defined network controllers”, in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, ACM, 2012, pp. 289–290.

[52] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, “Applying nox to the datacenter.”, in *HotNets*, 2009.

[53] (2015). Crossproject:integration group:performance test:results, [Online]. Available: https://wiki.opendaylight.org/view/CrossProject:Integration_Group:Performance_Test:Results.

[54] J. Rexford. (2015). Software defined networking - lectures, [Online]. Available: http://www.cs.princeton.edu/courses/archive/spring12/cos461/docs/lec24-sdn.pdf.

[55] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, “On controller performance in software-defined networks”, in *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, vol. 54, 2012.

[56] S. A. Shah, J. Faiz, M. Farooq, A. Shafi, and S. A. Mehdi, “An architectural evaluation of sdn controllers”, in *Communications (ICC), 2013 IEEE International Conference on*, IEEE, 2013, pp. 3504–3508.

[57] B. Heller, R. Sherwood, and N. McKeown, “The controller placement problem”, in *Proceedings of the first workshop on Hot topics in software defined networks*, ACM, 2012, pp. 7–12.

[58] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, “Logically centralized?: State distribution trade-offs in software defined networks”, in *Proceedings of the first workshop on Hot topics in software defined networks*, ACM, 2012, pp. 1–6.

[59] (2015). Securing sdn deployments right from the start., [Online]. Available: http://www.networkworld.com/article/2840273/sdn/sdn-security-attack-vectors-and-sdn-hardening.html.

[60] J. Metzler, “Understanding software-defined networks”, *InformationWeek Reports*, pp. 1–25, 2012.

[61] (May 2015). Solution brief: Sdn security considerations in the data center, [Online]. Available: https://www.opennetworking.org/solution-brief-sdn-security-considerations-in-the-data-center.

[62] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow enabling innovation in campus networks”, *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[63] (2014). Onf - openflow switch specification version 1.3.1, [Online]. Available: `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf`.

[64] O. N. F. Org. (2014). Open networking foundation, [Online]. Available: `httpswww . opennetworking.org`.

[65] D. Erickson, "The beacon openflow controller", in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, ACM, 2013, pp. 13–18.

[66] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky, "Advanced study of sdn/openflow controllers", in *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, ACM, 2013, p. 1.

[67] (2015). About pox, [Online]. Available: `http://www.noxrepo.org/pox/about-pox/`.

[68] (2015). Ryuinfo, [Online]. Available: `http://thenewstack.io/sdn-series-part-iv-ryu-a-rich-featured-open-source-sdn-controller-supported-by-ntt-labs/`.

[69] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: Towards an operating system for networks", *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[70] (2014). Maestro a system for scalable openflow control, [Online]. Available: `httpcode.google.compmaestro-platform`.

[71] (2014). Floodlight, [Online]. Available: `httpwww.projectfloodlight.orgfloodlight`.

[72] (2015). Ryu, [Online]. Available: `http://osrg.github.io/ryu/`.

[73] (2014). Equinox osgi, [Online]. Available: `http://eclipse.org/equinox/`.

[74] (2014). Osgi alliance, [Online]. Available: `http://www.osgi.org/Technology/WhatIsOSGi`.

[75] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture", in *2014 IEEE 15th International Symposium on*, IEEE, 2014, pp. 1–6.

[76] (2015). Sdn controller wars 2.0, [Online]. Available: `https://www.sdxcentral.com/articles/editorial/controller-onlab-juniper-open-source-sdn-battleground-part1/2013/12/`.

[77] R. Sherwood and Y. KOK-KIONG, *Cbench: An open-flow controller benchmarker*, 2010.

[78] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, "Data center network virtualization: A survey", *Communications Surveys & Tutorials, IEEE*, vol. 15, no. 2, pp. 909–928, 2013.

[79] (2015). Vmware's network virtualization poses huge threat to data center switch fabric vendors, [Online]. Available: `http://viodi.com/2013/05/06/vmwares-network-virtualization-poses-huge-threat-to-data-center-switch-fabric-vendors/`.

[80] (2015). 10 benefits of virtualization in the data center, [Online]. Available: `http://www.techrepublic.com/blog/10-things/10-benefits-of-virtualization-in-the-data-center/`.

[81] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending networking into the virtualization layer.", in *Hotnets*, 2009.

[82] (2015). Openflow 1.3 soft switch - the project, [Online]. Available: `http://cpqd.github.io/ofsoftswitch13/`.

[83] B. Claise, “Cisco systems netflow services export version 9”, RFC Editor, RFC 3954, 2004, `http://www.rfc-editor.org/rfc/rfc3954.txt`. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc3954.txt`.

[84] P. Phaal, S. Panchen, and N. McKee, “Inmon corporation’s sflow: A method for monitoring traffic in switched and routed networks”, RFC Editor, RFC 3176, 2001, `http://www.rfc-editor.org/rfc/rfc3176.txt`. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc3176.txt`.

[85] (2015). Openvswitch org. faq, [Online]. Available: `https://github.com/openvswitch/ovs/blob/master/FAQ.md`.

[86] (2015). Open network install environment (onie), [Online]. Available: `http://onie.org/about/`.

[87] G. Bell, “Failure to thrive: Qos and the culture of operational networking”, in *Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS: What have we learned, why do we care?*, ACM, 2003, pp. 115–120.

[88] W. Kim, P. Sharma, J. Lee, S. Banerjee, J. Tourrilhes, S.-J. Lee, and P. Yalagandula, “Automated and scalable qos control for network convergence”, *Proc. INM/WREN*, vol. 10, pp. 1–1, 2010.

[89] M. S. Seddiki, M. Shahbaz, S. Donovan, S. Grover, M. Park, N. Feamster, and Y.-Q. Song, “Flowqos: Qos for the rest of us”, in *Proceedings of the third workshop on Hot topics in software defined networking*, ACM, 2014, pp. 207–208.

[90] T. Benson, A. Anand, A. Akella, and M. Zhang, “Microte: Fine grained traffic engineering for data centers”, in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, ACM, 2011, p. 8.

[91] H. E. Egilmez, S. T. Dane, B. Gorkemli, and A. M. Tekalp, “Openqos: Openflow controller design and test network for multimedia delivery with quality of service”, *Proc. NEM Summit, Implementing Future Media Internet Towards New Horizons*, pp. 22–27, 2012.

[92] S. Sharma, D. Staessens, D. Colle, D. Palma, J. Goncalves, R. Figueiredo, D. Morris, M. Pickavet, and P. Demeester, “Implementing quality of service for the software defined networking enabled future internet”, in *The European Workshop on Software Defined Networking (EWSDN 2014)*, IEEE, 2014, pp. 49–54.

[93] A. Ishimori, F. Farias, E. Cerqueira, and A. Abelém, “Control of multiple packet schedulers for improving qos on openflow/sdn networking”, in *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, IEEE, 2013, pp. 81–86.

[94] D. Palma, J. Goncalves, B. Sousa, L. Cordeiro, P. Simoes, S. Sharma, and D. Staessens, “The queuepusher: Enabling queue management in openflow”, in *The European Workshop on Software Defined Networking (EWSDN 2014)*, IEEE, 2014, pp. 125–126.

[95] P. M. Mohan, D. M. Divakaran, and M. Gurusamy, “Performance study of tcp flows with qos-supported openflow in data center networks”, in *Networks (ICON), 2013 19th IEEE International Conference on*, IEEE, 2013, pp. 1–6.

[96] A. Juttner, B. Szviatovski, I. Mécs, and Z. Rajkó, “Lagrange relaxation based method for the qos routing problem”, in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, IEEE, vol. 2, 2001, pp. 859–868.

[97] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, “Practical declarative network management”, in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, ACM, 2009, pp. 1–10.

[98] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language", in *ACM SIGPLAN Notices*, ACM, vol. 46, 2011, pp. 279–291.

[99] A. Voellmy, H. Kim, and N. Feamster, "Procera: A language for high-level reactive network control", in *Proceedings of the first workshop on Hot topics in software defined networks*, ACM, 2012, pp. 43–48.

[100] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina, "Generic routing encapsulation (gre)", RFC Editor, RFC 2784, 2000, `http://www.rfc-editor.org/rfc/rfc2784.txt`. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc2784.txt`.

[101] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks", RFC Editor, RFC 7348, 2014, `http://www.rfc-editor.org/rfc/rfc7348.txt`. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc7348.txt`.

[102] T. Benson, A. Akella, A. Shaikh, and S. Sahu, "Cloudnaas: A cloud networking platform for enterprise applications", in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ACM, 2011, p. 8.

[103] (2015). Open networking foundation - north bound interface working group (nbi-wg) charter, [Online]. Available: `https://www.opennetworking.org/images/stories/downloads/working-groups/charter-nbi.pdf`.

[104] (2014). Karaf org., [Online]. Available: `http://karaf.apache.org/`.

[105] T. Ylonen and C. Lonvick, "The secure shell (ssh) transport layer protocol", RFC Editor, RFC 4253, 2006, `http://www.rfc-editor.org/rfc/rfc4253.txt`. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc4253.txt`.

[106] (2015). Opendaylight controller:architectural framework, [Online]. Available: `https://wiki.opendaylight.org/view/OpenDaylight_Controller:Architectural_Framework`.

[107] (2015). Sdn series part six: Opendaylight, the most documented controller, [Online]. Available: `http://thenewstack.io/sdn-series-part-vi-opendaylight/`.

[108] (2015). Difference between ad-sal and md-sal, [Online]. Available: `http://sdntutorials.com/difference-between-ad-sal-and-md-sal/`.

[109] M. Bjorklund, "Yang - a data modeling language for the network configuration protocol (netconf)", RFC Editor, RFC 6020, 2010, urlhttpwww.rfc-editor.orgrfcrfc6020.txt. [Online]. Available: `httpwww.rfc-editor.orgrfcrfc6020.txt`.

[110] (2015). Openvswitch org., [Online]. Available: `http://openvswitch.org/`.

[111] (2015). Openvswitch org. faq : Why ovs, [Online]. Available: `https://github.com/openvswitch/ovs/blob/master/WHY-OVS.md`.

[112] B. Hubert *et al.*, "Linux advanced routing & traffic control howto", *Setembro de*, 2002.

[113] D. G. BALAN and D. A. POTORAC, "Extended linux htb queuing discipline implementations", *International Journal of Information Studies*, vol. 2, no. 2, 2010.

[114] Mininet. (2014). Mininet a linux foundation collaborative project, [Online]. Available: `https://mininet.org/`.

[115] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks", in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ACM, 2010, p. 19.

[116] (2014). Mininet's python api, [Online]. Available: http://mininet.org/api/annotated.html.

[117] (2014). Opendaylight installation guide, [Online]. Available: https://www.opendaylight.org/sites/opendaylight/files/bk-install-guide-20141002.pdf.

[118] (2015). Dijkstra's algorithm in java, [Online]. Available: http://www.algolist.com/code/java/Dijkstra's_algorithm.

APPENDIX A

Appendix

A.1 Configuration details

This section includes all the configuration details that have to be performed in order to setup the architecture scenario. More precisely it indicates which commands are performed, when they are performed and what they do.

a.1.1 Sequence of installation procedures

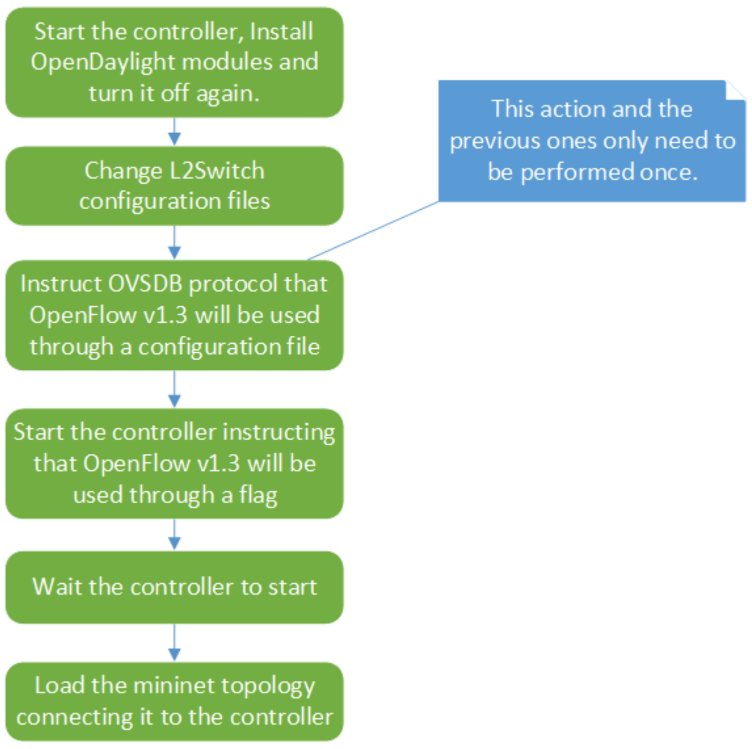

Start the controller, Install OpenDaylight modules and turn it off again.
Change L2Switch configuration files
Instruct OVSDB protocol that OpenFlow v1.3 will be used through a configuration file
Start the controller instructing that OpenFlow v1.3 will be used through a flag
Wait the controller to start
Load the mininet topology connecting it to the controller
This action and the previous ones only need to be performed once.

Figure A.1: Sequence of installation procedures to setup the architecture scenario

a.1.2 ODL'S MODULE INSTALLATION

This configuration is used the first time the controller is executed. The objective is to install all the specified modules into the controller.

```
feature:install odl-openflowplugin-flow-services-ui odl-restconf odl-l2switch-switch
odl-dlux-core odl-ovsdb-all odl-mdsal-apidocs
```

a.1.3 OVSDB'S CONFIGURATION TO OPENFLOWV1.3

This configuration is used to indicate to the OpenDaylight OVSDB subsystem that the version 1.3 of the OpenFlow protocol will be used. It is performed in a file named 'custom.properties', located in the distribution's 'etc' folder and the command added is the following:

```
ovsdb.ofversion = of1.3;
```

a.1.4 CONTROLLER'S CONFIGURATION TO OPENFLOWV1.3

This configuration is executed to run the OpenDaylight controller instructing it that version 1.3 of the openFlow protocol will be used.

```
bin/karaf -of13
```

a.1.5 topology's launch command

This configuration is performed in the Mininet VM after the controller is iniciated. It creates the custom topology defined in the Python file and connects it to the controller using OpenFlow protocol version 1.3.

```
    sudo mn --custom testScenariosTopo.py --topo=mytopo --controller=remote,
ip=192.168.57.1,port=6653 --switch=ovsk,protocol=OpenFlow13
```

A.2 qlames - system structure

This section contains the implementation details, such as the data structures used by the application. It also contains workflows of some important processes performed by the modules of the application.

a.2.1 topology handler

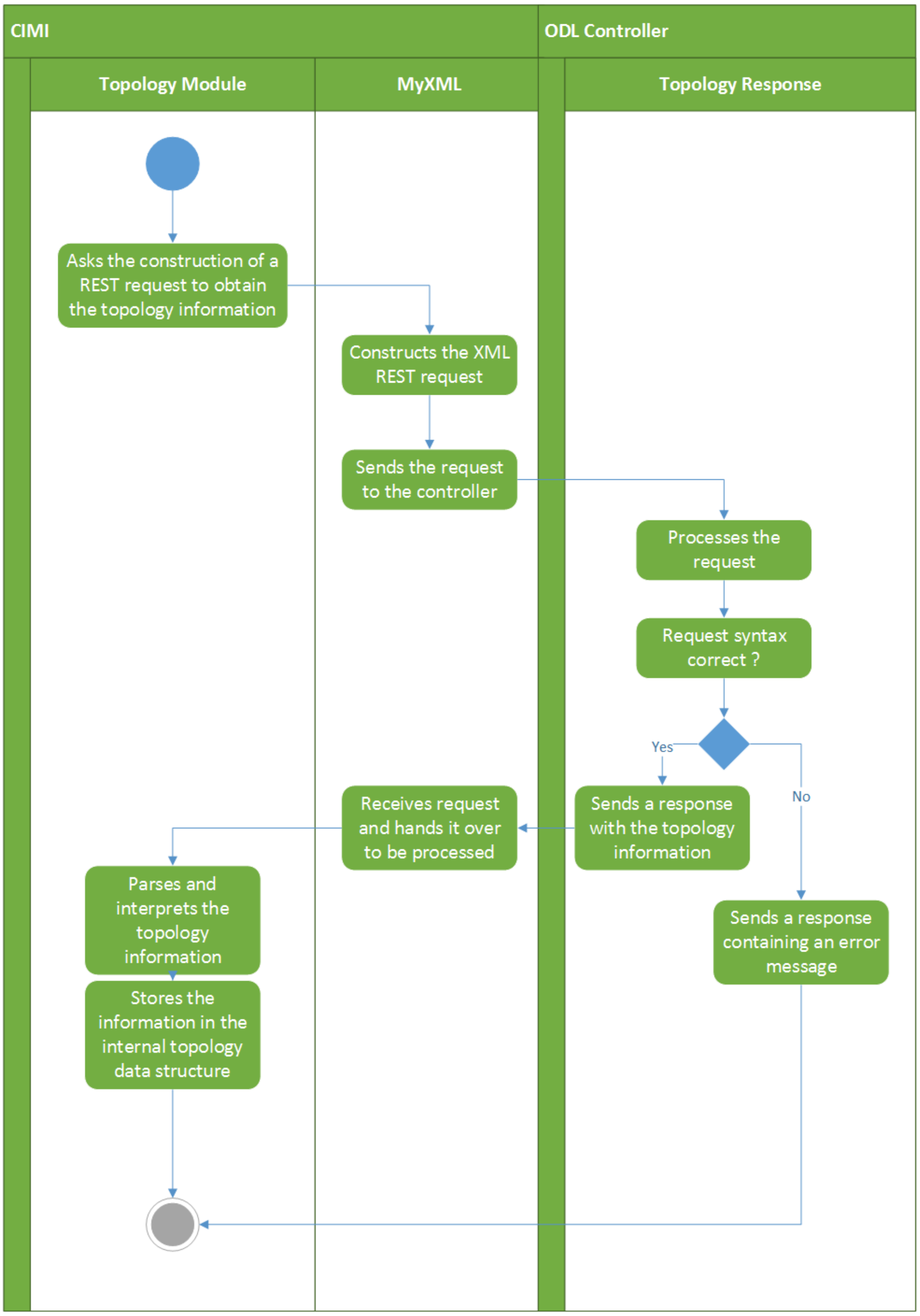
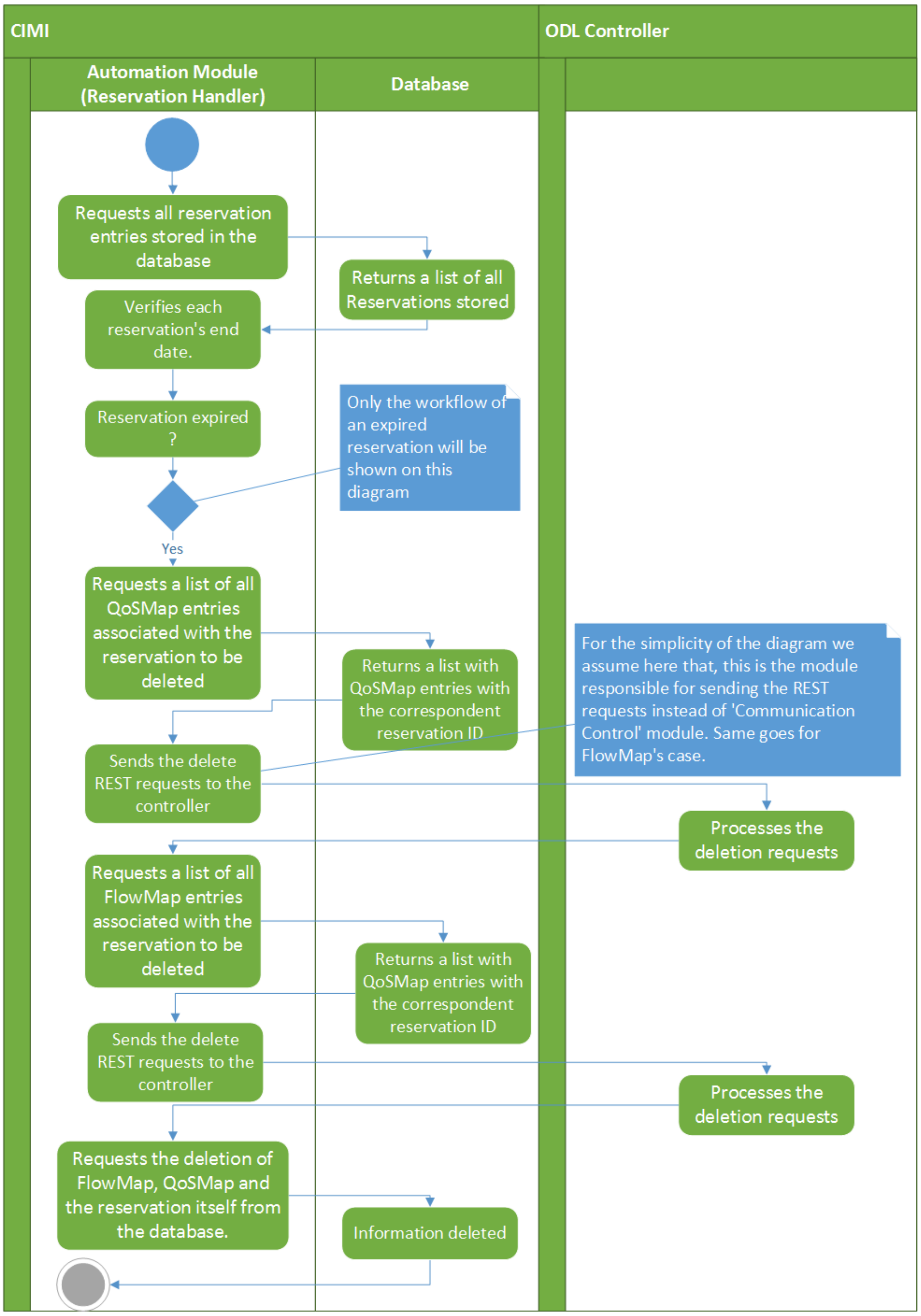

CIMI
ODL Controller
Topology Module
MyXML
Topology Response
Asks the construction of a REST request to obtain the topology information
Constructs the XML REST request
Sends the request to the controller
Processes the request
Request syntax correct ?
Yes
No
Sends a response with the topology information
Sends a response containing an error message
Receives request and hands it over to be processed
Parses and interprets the topology information
Stores the information in the internal topology data structure

Figure A.2: Topology module activity diagram

a.2.2 DATABASE MANAGEMENT



CIMI
ODL Controller
Automation Module (Reservation Handler)
Database
Requests all reservation entries stored in the database
Returns a list of all Reservations stored
Verifies each reservation's end date.
Reservation expired ?
Only the workflow of an expired reservation will be shown on this diagram
Yes
Requests a list of all QoSMap entries associated with the reservation to be deleted
Returns a list with QoSMap entries with the correspondent reservation ID
For the simplicity of the diagram we assume here that, this is the module responsible for sending the REST requests instead of 'Communication Control' module. Same goes for FlowMap's case.
Sends the delete REST requests to the controller
Processes the deletion requests
Requests a list of all FlowMap entries associated with the reservation to be deleted
Returns a list with QoSMap entries with the correspondent reservation ID
Sends the delete REST requests to the controller
Processes the deletion requests
Requests the deletion of FlowMap, QoSMap and the reservation itself from the database.
Information deleted

Figure A.3: Database Management - Reservation's deletion activity diagram

a.2.3 COMMUNICATION CONTROL

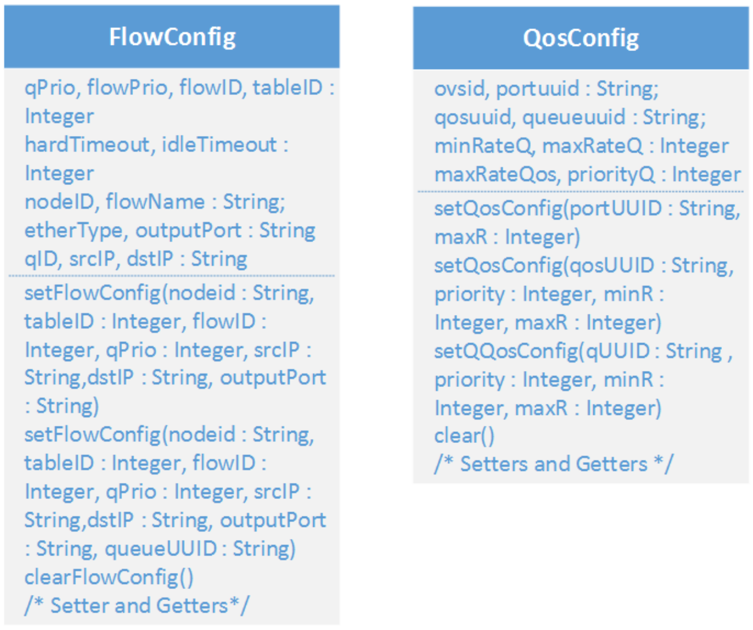
```java
package REST_Requests;

public class Constants {

    public static final double LINK_THRESHOLD = 0.9;    //90%
    public static final boolean applyLoadBalance = true;

    /* HTTP ERROR CODES */
    public static final int OK = 200;
    public static final int CREATED = 201;
    public static final int NO_CONTENT = 204;
    public static final int BAD_REQUEST = 400;
    public static final int UNAUTHORIZED = 401;
    public static final int FORBIDDEN = 403;
    public static final int INTERNAL_SERVER_ERROR = 500;
    public static final int BAD_GATEWAY = 502;
    public static final int SERVICE_UNAVAILABLE = 503;

    public static String ovsID;

    /* Port State */
    public static final boolean UP = true;
    public static final boolean DOWN = false;

    /* GET TYPES XML */
    public static final int topo = 1;
    public static final int table = 2;
    public static final int flow = 3;

    /* GET TYPES JSON */
    public static final int queue = 1;
    public static final int qos = 2;
    public static final int port = 3;
    public static final int bridge = 4;
    public static final int node = 5;
    public static final int iface = 6;

    /* Ethernet types */
    public static final String ipv4 = "2048";
    public static final String arp = "2054";
    public static final String ipv6 = "34525";
    public static final String lldp = "35020";

    /* OF13 Logical Ports */
    public static enum OFLogicalPorts {
        ALL, CONTROLLER, TABLE, IN_PORT, ANY, LOCAL, NORMAL, FLOOD
    }
}
```

Listing 1: class 'Constants'

```
package REST_Requests;

public class BaseURLs {
    /* ODL REST API */
    /* GETS */
    public static final String getTopo =
       "http://192.168.57.1:8181/restconf/operational/
        network-topology:network-topology/";
    public static final String getTable =
       "http://192.168.57.1:8181/restconf/config/
        opendaylight-inventory:nodes/node/<nodeid>/table/<tableid>/";
    public static final String getFlow =
       "http://192.168.57.1:8181/restconf/config/
        opendaylight-inventory:nodes/node/<nodeid>/table/<tableid>/flow/<flowid>";;
    /* PUTS */
    public static final String putFlow =
       "http://192.168.57.1:8181/restconf/config/
        opendaylight-inventory:nodes/node/<nodeid>/table/<tableid>/flow/<flowid>";
    /* DELETES */
    public static final String delTable =
       "http://192.168.57.1:8181/restconf/config/
        opendaylight-inventory:nodes/node/<nodeid>/table/<tableid>";
    public static final String delFlow =
       "http://192.168.57.1:8181/restconf/config/
        opendaylight-inventory:nodes/node/<nodeid>/table/<tableid>/flow/<flowid>";


    /* OVSDB REST API*/
    /* GETS */
    public static final String getNodes =
       "http://192.168.57.1:8080/controller/nb/
        v2/connectionmanager/nodes";
    public static final String getBridges =
       "http://192.168.57.1:8080/ovsdb/nb/
        v2/node/OVS/<ovsid>/tables/bridge/rows";
    public static final String getPorts =
        "http://192.168.57.1:8080/ovsdb/nb/
         v2/node/OVS/<ovsid>/tables/port/rows";
    public static final String getInterfaces =
        "http://192.168.57.1:8080/ovsdb/nb/
         v2/node/OVS/<ovsid>/tables/interface/rows";
    public static final String getQos =
        "http://192.168.57.1:8080/ovsdb/nb/
         v2/node/<ovsid>/tables/qos/rows";
    public static final String getQueue =
        "http://192.168.57.1:8080/ovsdb/nb/
         v2/node/OVS/<ovsid>/tables/queue/rows";

    /* PUT */
    public static final String putQos =
        "http://localhost:8080/ovsdb/nb/v2/node/
         OVS/<ovsid>/tables/QoS/rows/<qosuuid>";
    /* POSTS */
    public static final String postQos =
        "http://localhost:8080/ovsdb/nb/v2/node/
         OVS/<ovsid>/tables/QoS/rows";
    public static final String postQueue =
        "http://192.168.57.1:8080/ovsdb/nb/
         v2/node/OVS/<ovsid>/tables/queue/rows";
    /* DELETES */
    public static final String delQos =
        "http://localhost:8080/ovsdb/nb/v2/node/
         OVS/<ovsid>/tables/qos/rows/<qosuuid>";
    public static final String delQueue =
        "http://localhost:8080/ovsdb/nb/v2/node/
         OVS/<ovsid>/tables/queue/rows/<queueuuid>";

    /* Flows replacers */
    public static String urlFlowReplacer(String baseUrl, String nodeid,
        String tableid){
       String ret = baseUrl;
       /* Replacing fields */
       ret = ret.replace("<nodeid>", nodeid);
       ret = ret.replace("<tableid>", tableid);
       return ret;
    }
    public static String urlFlowReplacer(String baseUrl, String nodeid,
        String tableid, String flowid){
       String ret = baseUrl;
       /* Replacing fields */
```

```
        ret = ret.replace("<nodeid>", nodeid);
        ret = ret.replace("<tableid>", tableid);
        ret = ret.replace("<flowid>", flowid);
        return ret;
    }

    /* Ovs replacers */
    public synchronized static String urlOvsReplacer(String baseUrl,
        String ovsid){
        String ret = baseUrl;
        ret = ret.replace("<ovsid>", ovsid);    // Replacing fields
        return ret;
    }
    public static String urlQosReplacer(String baseUrl, String ovsid,
        String qosuuid){
        String ret = baseUrl;
        ret = ret.replace("<ovsid>", ovsid);    // Replacing fields
        ret = ret.replace("<qosuuid>", qosuuid);    // Replacing fields
        return ret;
    }
    public static String urlQReplacer(String baseUrl, String ovsid,
        String queueuuid){
        String ret = baseUrl;
        ret = ret.replace("<ovsid>", ovsid);    // Replacing fields
        ret = ret.replace("<queueuuid>", queueuuid);    // Replacing fields
        return ret;
    }
}
```

Listing 2: class 'REST URLs'


FlowConfig

qPrio, flowPrio, flowID, tableID : Integer
hardTimeout, idleTimeout : Integer
nodeID, flowName : String;
etherType, outputPort : String
qID, srcIP, dstIP : String

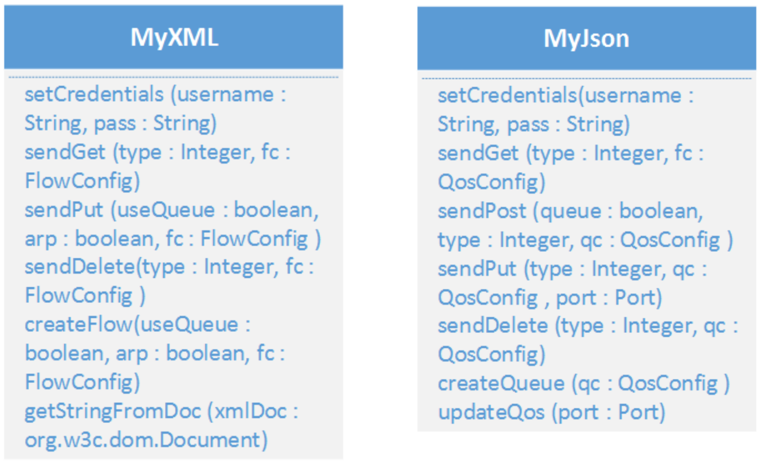
setFlowConfig(nodeid : String, tableID : Integer, flowID : Integer, qPrio : Integer, srcIP : String,dstIP : String, outputPort : String)
setFlowConfig(nodeid : String, tableID : Integer, flowID : Integer, qPrio : Integer, srcIP : String,dstIP : String, outputPort : String, queueUUID : String)
clearFlowConfig()
/* Setter and Getters*/

QosConfig

ovsid, portuuid : String;
qosuuid, queueuuid : String;
minRateQ, maxRateQ : Integer
maxRateQos, priorityQ : Integer

setQosConfig(portUUID : String, maxR : Integer)
setQosConfig(qosUUID : String, priority : Integer, minR : Integer, maxR : Integer)
setQQosConfig(qUUID : String , priority : Integer, minR : Integer, maxR : Integer)
clear()
/* Setters and Getters */

Figure A.4: Communication Control - *FlowConfig* and *QosConfig* data structures


MyXML

setCredentials (username : String, pass : String)
sendGet (type : Integer, fc : FlowConfig)
sendPut (useQueue : boolean, arp : boolean, fc : FlowConfig)
sendDelete(type : Integer, fc : FlowConfig)
createFlow(useQueue : boolean, arp : boolean, fc : FlowConfig)
getStringFromDoc (xmlDoc : org.w3c.dom.Document)

MyJson

setCredentials(username : String, pass : String)
sendGet (type : Integer, fc : QosConfig)
sendPost (queue : boolean, type : Integer, qc : QosConfig)
sendPut (type : Integer, qc : QosConfig , port : Port)
sendDelete (type : Integer, qc : QosConfig)
createQueue (qc : QosConfig)
updateQos (port : Port)

Figure A.5: Communication Control - *MyXML* and *MyJson* methods

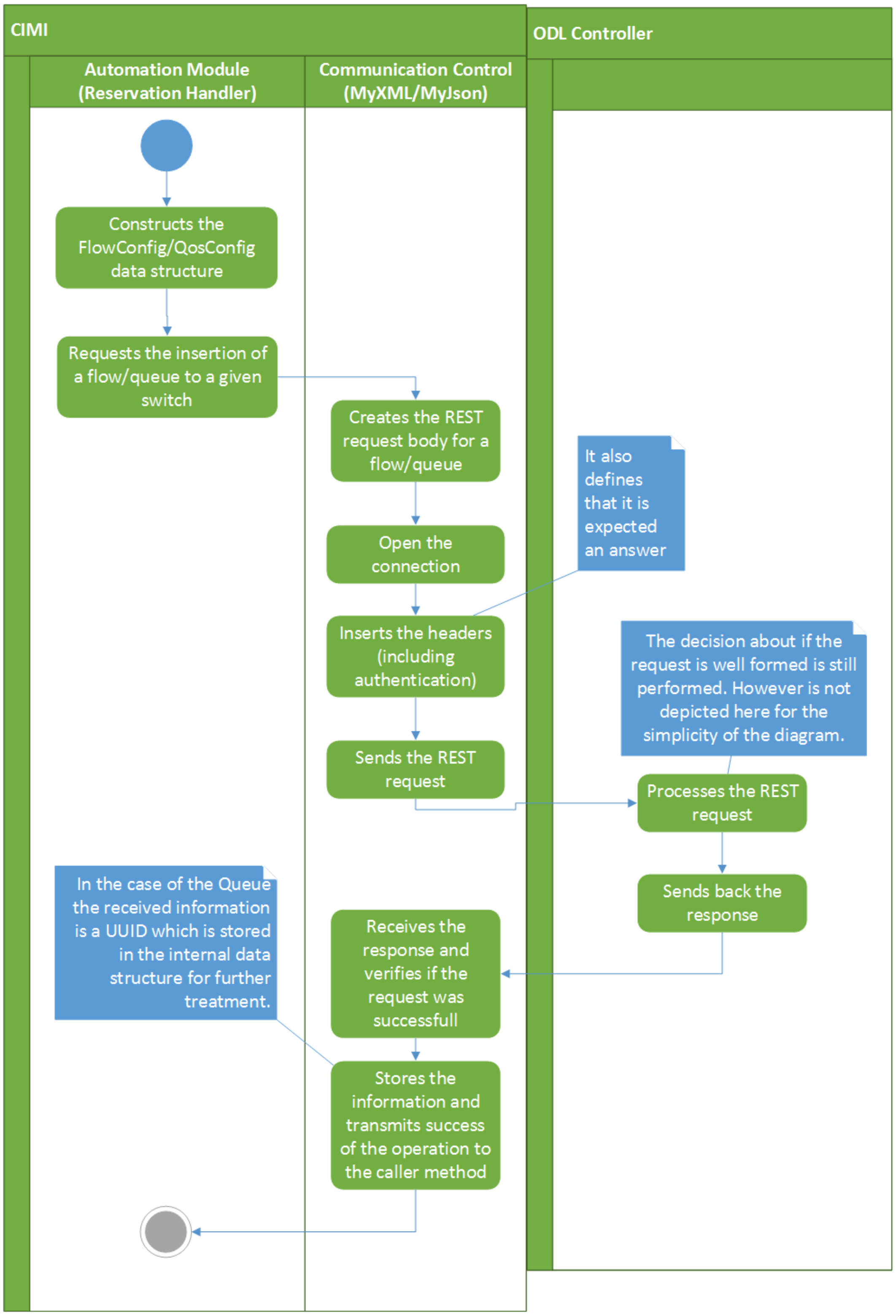

CIMI
ODL Controller
Automation Module (Reservation Handler)
Communication Control (MyXML/MyJson)
Constructs the FlowConfig/QosConfig data structure
Requests the insertion of a flow/queue to a given switch
Creates the REST request body for a flow/queue
Open the connection
Inserts the headers (including authentication)
Sends the REST request
It also defines that it is expected an answer
The decision about if the request is well formed is still performed. However is not depicted here for the simplicity of the diagram.
Processes the REST request
Sends back the response
Receives the response and verifies if the request was successfull
In the case of the Queue the received information is a UUID which is stored in the internal data structure for further treatment.
Stores the information and transmits success of the operation to the caller method

Figure A.6: Communication Control - General workflow activity diagram (Example : adding/removing a queue)

a.2.4 PATH DISCOVERY (DIKJSTRA)

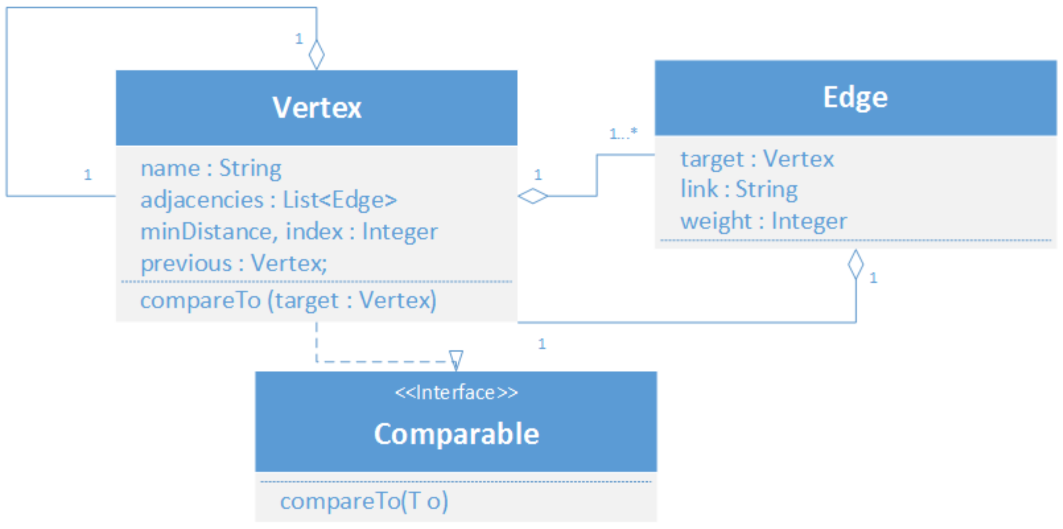
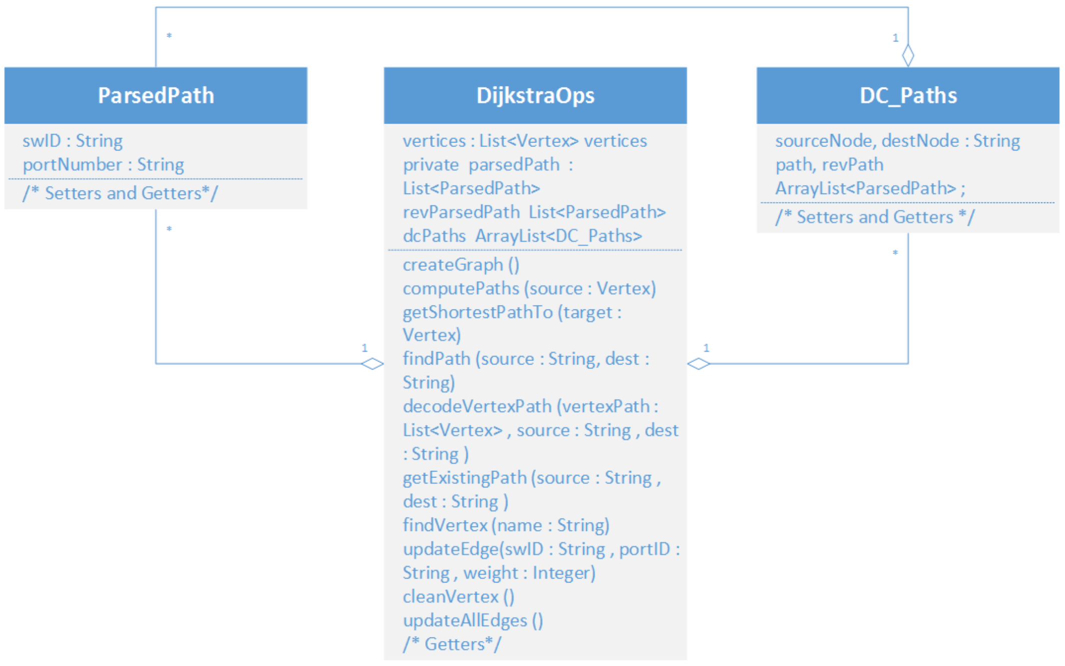


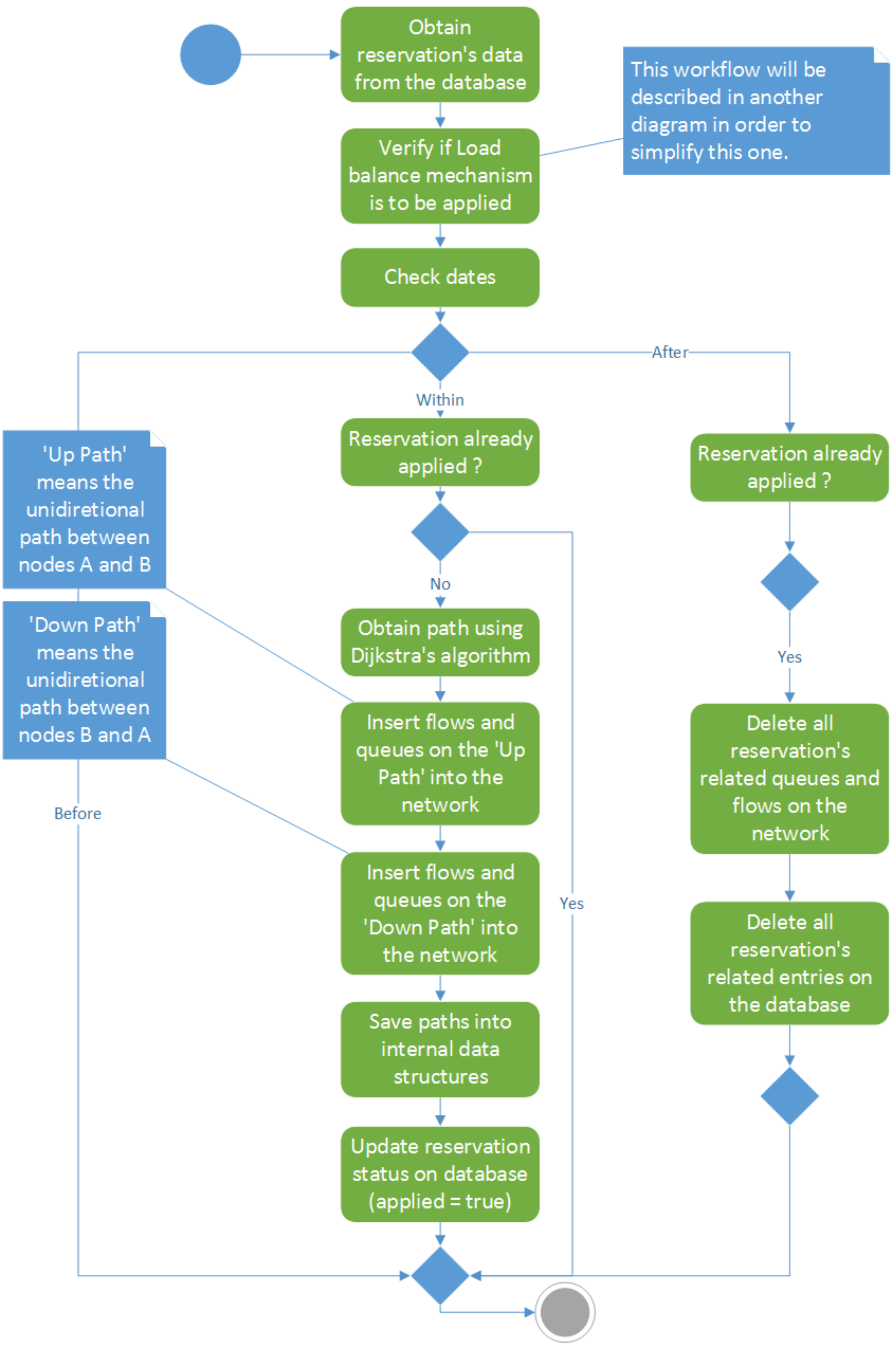
Figure A.7: Path Discovery (Dijkstra) - *Vertex* and *Edge* class diagram



ParsedPath
swID : String
portNumber : String
/* Setters and Getters*/
DijkstraOps
vertices : List<Vertex> vertices
private parsedPath : List<ParsedPath>
revParsedPath List<ParsedPath>
dcPaths ArrayList<DC_Paths>
createGraph ()
computePaths (source : Vertex)
getShortestPathTo (target : Vertex)
findPath (source : String, dest : String)
decodeVertexPath (vertexPath : List<Vertex> , source : String , dest : String)
getExistingPath (source : String , dest : String)
findVertex (name : String)
updateEdge(swID : String , portID : String , weight : Integer)
cleanVertex ()
updateAllEdges ()
/* Getters*/
DC_Paths
sourceNode, destNode : String
path, revPath ArrayList<ParsedPath> ;
/* Setters and Getters */
*
1
*
1
*
1

Figure A.8: Path Discovery (Dijkstra) - Dijkstra operations, *DC_Paths* and *ParsedPath* data structures

a.2.5 AUTOMATION PROCESS



Obtain reservation's data from the database
Verify if Load balance mechanism is to be applied
This workflow will be described in another diagram in order to simplify this one.
Check dates
After
Within
Reservation already applied ?
Reservation already applied ?
'Up Path' means the unidiretional path between nodes A and B
No
Yes
Obtain path using Dijkstra's algorithm
'Down Path' means the unidiretional path between nodes B and A
Insert flows and queues on the 'Up Path' into the network
Delete all reservation's related queues and flows on the network
Before
Insert flows and queues on the 'Down Path' into the network
Yes
Delete all reservation's related entries on the database
Save paths into internal data structures
Update reservation status on database (applied = true)

Figure A.9: Automation Process - Reservation's processing workflow

a.2.6 LOAD BALANCE MECHANISM

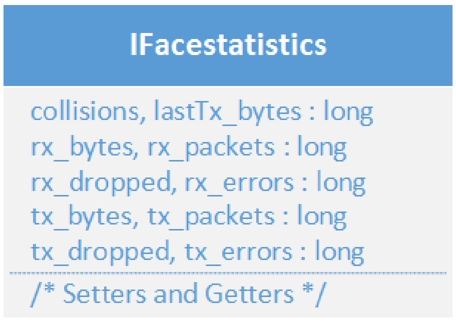

IFacestatistics
collisions, lastTx_bytes : long
rx_bytes, rx_packets : long
rx_dropped, rx_errors : long
tx_bytes, tx_packets : long
tx_dropped, tx_errors : long
/* Setters and Getters */

Figure A.10: Load Balance Mechanism - *IFacestatistics* data structure

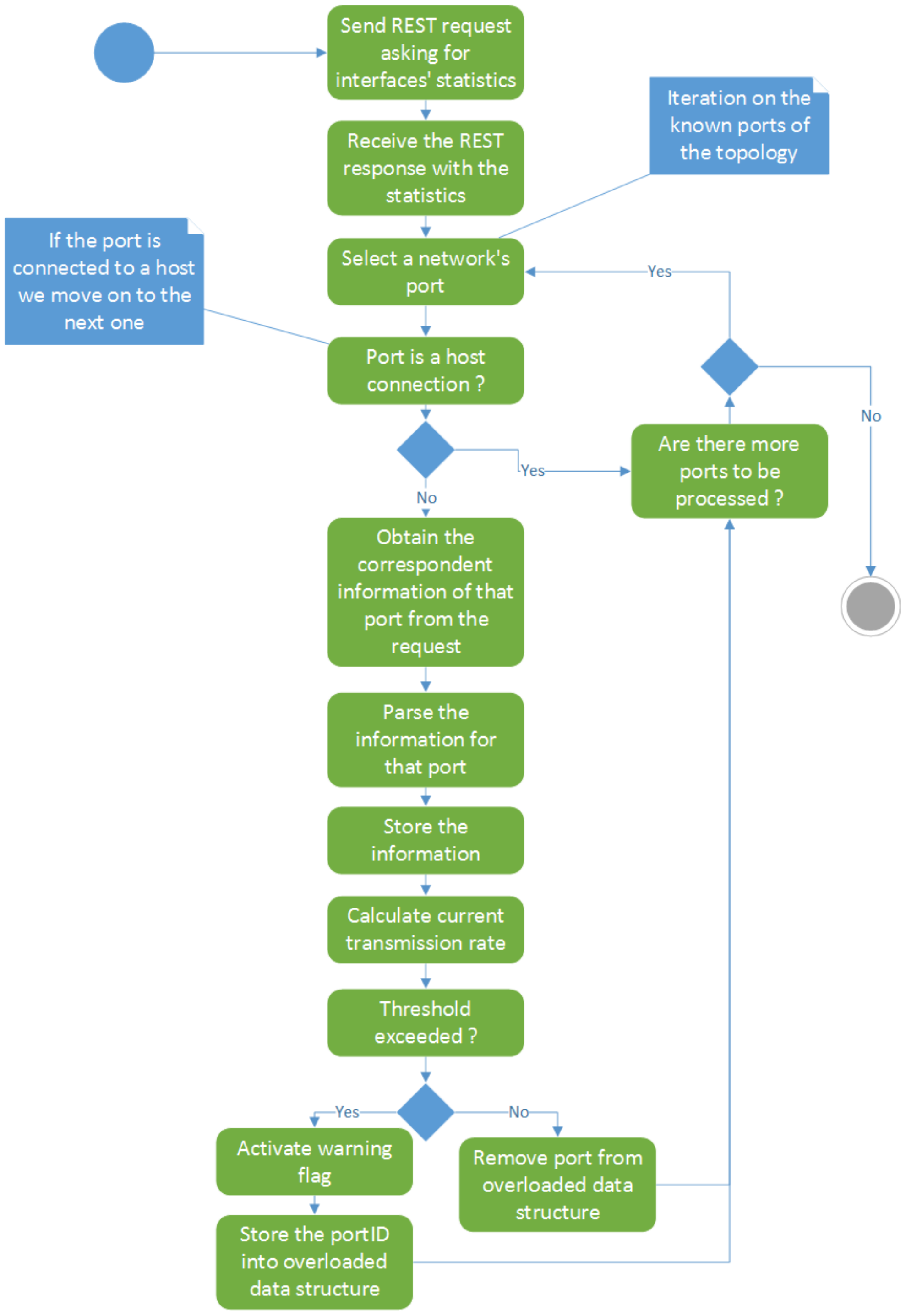
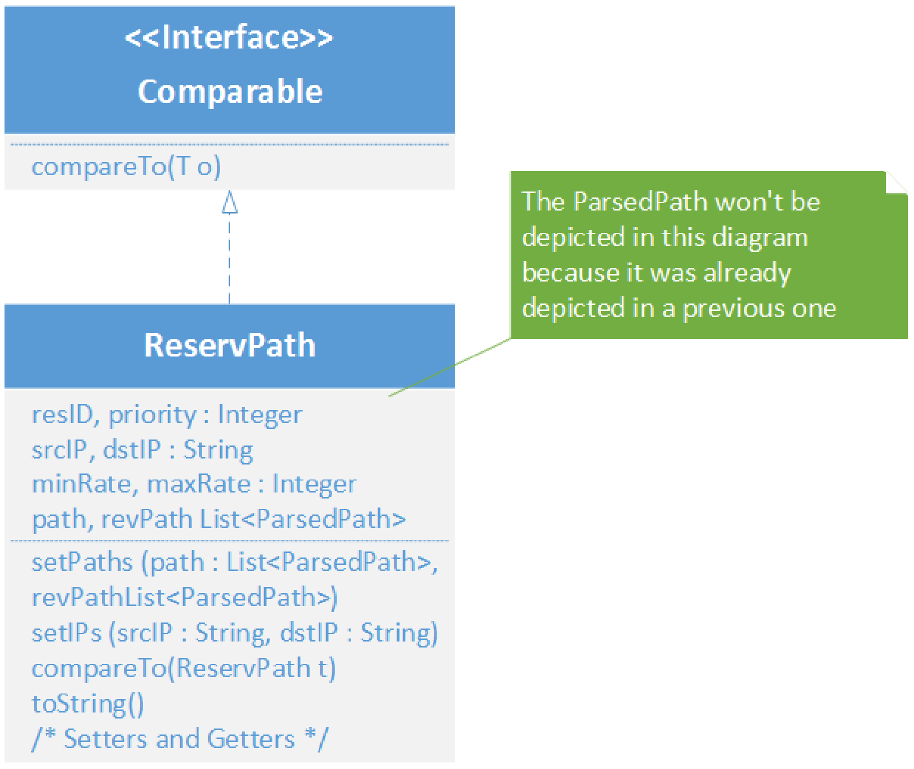

Send REST request asking for interfaces' statistics
Receive the REST response with the statistics
Iteration on the known ports of the topology
If the port is connected to a host we move on to the next one
Select a network's port
Yes
Port is a host connection ?
No
Yes
Are there more ports to be processed ?
No
Obtain the correspondent information of that port from the request
Parse the information for that port
Store the information
Calculate current transmission rate
Threshold exceeded ?
Yes
No
Activate warning flag
Remove port from overloaded data structure
Store the portID into overloaded data structure

Figure A.11: Load Balance Mechanism - Monitoring workflow



<<Interface>>
Comparable
compareTo(T o)
The ParsedPath won't be depicted in this diagram because it was already depicted in a previous one
ReservPath
resID, priority : Integer
srcIP, dstIP : String
minRate, maxRate : Integer
path, revPath List<ParsedPath>
setPaths (path : List<ParsedPath>, revPathList<ParsedPath>)
setIPs (srcIP : String, dstIP : String)
compareTo(ReservPath t)
toString()
/* Setters and Getters */

Figure A.12: Load Balance Mechanism - *ReservPath* data structure

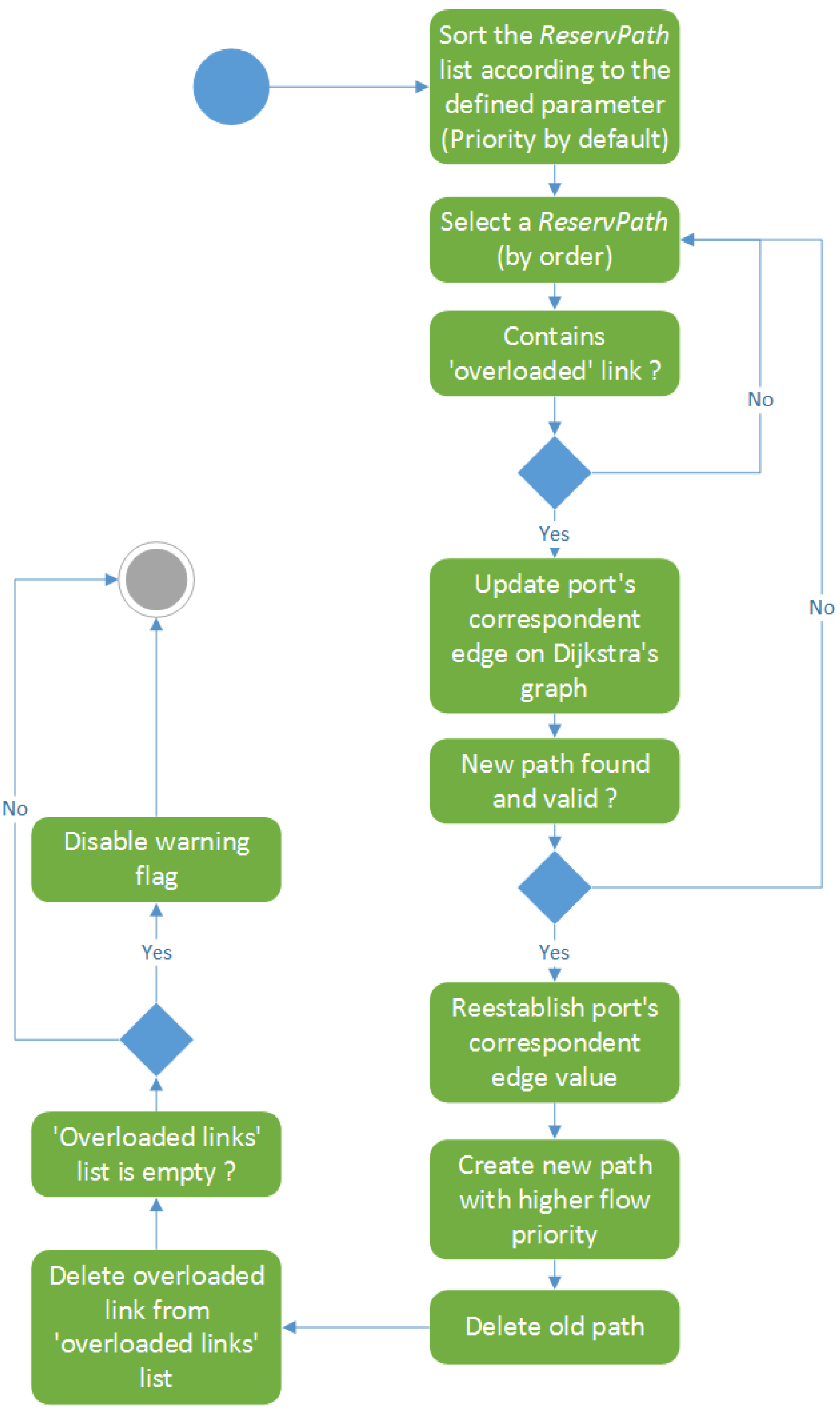

Sort the *ReservPath* list according to the defined parameter (Priority by default)
Select a *ReservPath* (by order)
Contains 'overloaded' link ?
No
Yes
Update port's correspondent edge on Dijkstra's graph
New path found and valid ?
No
Yes
Reestablish port's correspondent edge value
Create new path with higher flow priority
Delete old path
Delete overloaded link from 'overloaded links' list
'Overloaded links' list is empty ?
Yes
No
Disable warning flag

Figure A.13: Load Balance Mechanism - Enforcement workflow

A.3 MININET'S TOPOLOGY SCRIPT

This section contains the Python script that creates the topology to be used in the testing scenarios. This is done by using Mininet's Python API.

```python
from mininet.topo import Topo

class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self ):
        "Create custom topo."

        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        hostA1 = self.addHost( 'h1', ip='10.0.0.1', mask='255.255.255.0'
            ) #A
        hostA2 = self.addHost( 'h2', ip='10.0.0.2', mask='255.255.255.0'
            ) #A
        hostA3 = self.addHost( 'h3', ip='10.0.0.3', mask='255.255.255.0'
            ) #A
        hostB1 = self.addHost( 'h4', ip='10.0.0.11',
            mask='255.255.255.0' ) #B
        hostB2 = self.addHost( 'h5', ip='10.0.0.12',
            mask='255.255.255.0' ) #B
        hostB3 = self.addHost( 'h6', ip='10.0.0.13',
            mask='255.255.255.0' ) #B
        hostC1 = self.addHost( 'h7', ip='10.0.0.21',
            mask='255.255.255.0' ) #C
        hostC2 = self.addHost( 'h8', ip='10.0.0.22',
            mask='255.255.255.0' ) #C
        hostC3 = self.addHost( 'h9', ip='10.0.0.23',
            mask='255.255.255.0' ) #C
        hostD1 = self.addHost( 'h10', ip='10.0.0.31',
            mask='255.255.255.0' ) #D
        hostD2 = self.addHost( 'h11', ip='10.0.0.32',
            mask='255.255.255.0' ) #D
        hostD3 = self.addHost( 'h12', ip='10.0.0.33',
            mask='255.255.255.0' ) #D
        hostE1 = self.addHost( 'h13', ip='10.0.0.41',
            mask='255.255.255.0' ) #E
        hostE2 = self.addHost( 'h14', ip='10.0.0.42',
            mask='255.255.255.0' ) #E
        hostE3 = self.addHost( 'h15', ip='10.0.0.43',
           mask='255.255.255.0' ) #E


        # Switch Level 1
        switch1 = self.addSwitch( 's1' )
        switch10 = self.addSwitch( 's10' )
        # Switch Level 2
        switch2 = self.addSwitch( 's2' )
        switch3 = self.addSwitch( 's3' )
        switch4 = self.addSwitch( 's4' )
        # Switch Level 3
        switch5 = self.addSwitch( 's5' )
        switch6 = self.addSwitch( 's6' )
        # Switch Level 4
        switch7 = self.addSwitch( 's7' )

        # Add links
        #Level 1
        self.addLink( switch10, switch2 )
        self.addLink( switch10, switch3 )
        self.addLink( switch10, switch4 )
        #Level 2
        self.addLink( switch3, switch5 )
        self.addLink( switch3, switch6 )
        #Level 3
        self.addLink( switch5, switch7 )
        self.addLink( switch6, switch7 )
        #Level 4
        self.addLink( switch2, hostA1 )
        self.addLink( switch2, hostA2 )
        self.addLink( switch2, hostA3 )
        self.addLink( switch5, hostB1 )
        self.addLink( switch5, hostB2 )
        self.addLink( switch5, hostB3 )
        self.addLink( switch7, hostC1 )
        self.addLink( switch7, hostC2 )
        self.addLink( switch7, hostC3 )
        self.addLink( switch6, hostD1 )
        self.addLink( switch6, hostD2 )
        self.addLink( switch6, hostD3 )
        self.addLink( switch4, hostE1 )
        self.addLink( switch4, hostE2 )
        self.addLink( switch4, hostE3 )

```

```
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

Listing 3: Mininet's Custom Topology

A.4 MATLAB'S FUNCTIONS

This section contains the Matlab scripts used to calculate the confidence intervals of the results obtained in Chapter 4 such as the bandwidth, data transfer and installation, load balance and tear down delays.

a.4.1 SCENARIO 'BANDWIDTH ON DEMAND'

```
function [ resMean, resVar, uncertainty, lol ] = Delays( time, bw )

    uncertainty = [];
    resMean = [];   resVar = [];

    'Simulation started...'
    resMean = mean( bw(1,:) );
    resVar = var( bw(1,:) );
    uncertainty = norminv(0.95) * sqrt(resVar/time);

    %Write output to file
    fileID = fopen('Results.txt','w');
    init = 'Number of Measures = %d\n';
    fprintf(fileID, init, 10);
    pm = '$\pm$';
    output = '%5.3f % s %5.3f';
    fprintf(fileID, output, resMean, pm, uncertainty);
    fclose(fileID);
end
```

Listing 4: Script to calculate confidence intervals from Bandwidth on Demand scenario

a.4.2 SCENARIO 'INSTALLATION DELAY'

```matlab
function [ resMean, resVar, uncertainty, sum ] = BWTransfer( nMeasures, time, bw )

    uncertainty = [];
    resMean = [];   resVar = [];

    'Simulation started...'
    for i=1:time
        resMean(i) = mean( bw(i,:) );
        resVar(i) = var( bw(i,:) );
        uncertainty(i) = norminv(0.95) * sqrt(resVar(i)/nMeasures);
    end

    %Write output to file
    fileID = fopen('Results.txt','w');
    init = 'Number of Measures = %d\nTime per measure = %d sec\n';
    fprintf(fileID, init, nMeasures, time);
    for i=1:time
        output = '(%d,%5.3f) +- (%5.3f,%5.3f) ';
        fprintf(fileID, output, i, resMean(i), (uncertainty(i)), (uncertainty(i)));
    end

    mean(resMean)
    mean(uncertainty)
    fclose(fileID);

end
```

Listing 5: Script to calculate confidence intervals from Installation delays scenario