ALMA MATER STUDIORUM - UNIVERSITY OF BOLOGNA

Department of Computer Science and Engineering

Master Degree in Computer Engineering

ENHANCING QUALITY OF SERVICE IN SOFTWARE-DEFINED NETWORKS

Supervisor: Professor Antonio CORRADI

Correlator: Professor Mario GERLA

Correlator: Professor Eduardo CERQUEIRA

Correlator: Doctor Luca FOSCHINI

Master's Thesis of:

Francesco ONGARO

Academic Year 2013–2014

Session I

A mia mamma Rita, mio papà Angelo e mio fratello Luca

« Senza entusiasmo,
non si è mai compiuto niente di grande. »
RALPH WALDO EMERSON

Acknowledgements

« What lies behind us and what lies ahead of us,
are tiny matters compared to what lies within us. »
RALPH WALDO EMERSON

« Stay hungry, stay foolish. »
STEVE JOBS

Anche se apparentemente i ringraziamenti sembrano casa di poco conto se comparati con l'intera tesi, alle volte richiedono uno sforzo di pensiero e meditazione notevole. Non si vorrebbe far dispetto a nessuno, ma si è perfettamente consapevoli di quali sono state le persone importanti e i momenti salienti che hanno contraddistinto questi anni di sacrificio. Seguendo questo percorso impegnativo, ora si è arrivati ad un momento conclusivo ed emozionante ma al contempo misterioso e fondamentale per ciò che verrà dopo. Un po' come scollinare a poco a poco una vetta dopo lunghe ore di intensa camminata, con vari momenti di apparente cedimento, ma con la forza e la voglia di arrivare in cima per godersi lo spettacolo, rafforzato da quella sensazione di stanchezza e spossatezza tipiche dopo una lunga scarpinata.

"Time flies" i miei compagni alla UCLA mi dicevano diverse volte, e io dicevo a loro con un velo di amarezza per l'avvicinarsi del momento del ritorno. Sì, partirei proprio dall'ultima esperienza, passata all'University of California, Los Angeles (UCLA) perché è il ricordo più fresco che ho in mente dato che sono

passate appena una manciata di settimane dal mio rientro. Questa affascinante esperienza, inizialmente misteriosa ma rivelatasi poi estremamente positiva e costruttiva, è stata resa possibile grazie al supporto della mia famiglia e delle persone che mi sono state vicine prima e durante i mesi di permanenza a Los Angeles. Il Professor Antonio Corradi, a cui va un sentito ringraziamento per il supporto fornito prima e durante la tesi, ha inoltre reso possibile questa esperienza grazie ai numerosi contatti personali e al prestigio che l'Università di Bologna, seppur piccola confrontata con altri campus universitari, può vantare ed essere orgogliosa di avere (oltre alla invidiata millenaria fondazione). Grazie inoltre al Professor Paolo Bellavista per i preziosi consigli e il supporto che immancabilmente mi ha fornito anche a 10.000 km di distanza. Grazie all'Ingegner Luca Foschini, che ha contribuito alla tesi con interessanti spunti e consigli. All'interno della UCLA, un doveroso e sentito ringraziamento va al Professor Mario Gerla che mi ha ospitato come *Visiting Researcher Scholar* all'interno del proprio laboratorio, il *Network Research Laboratory* (NRL), rendendo la mia permanenza veramente piacevole e stimolante sotto il profilo accademico. Da Lui, così come dal Professor Leonard Kleinrock che ho avuto la fortuna di incontrare, ho imparato che si può essere "grandi" senza bisogno di dover dimostrare di esserlo. Durante la mia permanenza alla UCLA, ho avuto il piacere di essere inoltre seguito dal Professor Eduardo Cerqueira, che non solo si è dimostrato di notevole aiuto e supporto per la ricerca inerente alla tesi, ma ha contribuito allo svolgimento di numerosi meeting utili alla ricerca, nonché ha reso possibile l'instaurarsi di piacevoli momenti di aggregazione, sfociati in un vero e sentito rapporto personale di amicizia. Grazie anche al Professor Giovanni Pau che mi ha sempre fornito utili consigli e suggerimenti con una franchezza che poche persone hanno.

Ma la fine di questo percorso, contraddistinto dall'esperienza passata alla UCLA, non può scindersi da ciò che è venuto prima. E' infatti vero e indiscutibile che, se si è arrivati sin qui, doverosamente si è dovuti passare per una serie di momenti e situazioni, piacevoli e dure allo stesso tempo, che tuttavia hanno reso possibile il raggiungimento di questo importante e gratificante traguardo. Dice la guida alpina e scalatore M. Confortola: « se per cacciare un sogno e raggiungerlo servono forza, determinazione, costanza e un istinto infallibi-

le, è altrettanto vero che in montagna, come nella vita e nel lavoro, in vetta si arriva poco alla volta, campo base dopo campo base, rispettando i tempi e le fasi di acclimatamento ». Ecco quindi che i ringraziamenti li voglio dedicare nuovamente alla mia famiglia, che proprio in quegli anni mi ha continuamente supportato, spronato e aiutato, affinché potessi avvicinarmi ed arrivare in "vetta". Lunghe e fondamentali passeggiate e chiacchierate con mio fratello Luca, che ben sa cosa vuol dire affrontare questo percorso universitario, mi hanno inoltre sempre aiutato, come un faro per i natanti, a mantenere costantemente la prua puntata nella giusta direzione, anche in difficili e burrascose situazioni. Sento inoltre dal profondo di ringraziare Giulia che in quegli anni mi è sempre stata vicina e immancabilmente ha creduto in me. Giulia è stata di grande aiuto affinché non gettassi mai la spugna ma, anzi, cercassi di arrivare fino alla fine del match.

Gli amici, anche se in tanti momenti ho purtroppo dovuto centellinare il tempo con loro per poter arrivare fin qui, è d'obbligo ringraziarli. Grazie per i momenti passati assieme e per gli aperitivi che hanno contraddistinto e scandito alcune nostre serate. Un abbraccio in particolare a Mattia, la cui amicizia e stima si perpetua da molti anni senza mai scolorire ma, anzi, impreziosendosi come un buon vino fa' nella botte. Grazie inoltre a Gabri, France, Corne e Mone per l'amicizia che ci lega da anni. Nonostante le diverse scelte di vita e la distanza che alle volte per mesi ci tiene distanti, riusciamo a trovare il tempo di incontrarci e condividere le nostre diverse esperienze. Credo che questa diversità sia un valore aggiunto per tutti noi. Inoltre, e ne sono orgoglioso, altre amicizie con ragazzi di diverse parti del Mondo sono nate all'interno dell'NRL alla UCLA. Dunque un particolare grazie va all'amico Paul-Louis, al mitico "big" Vince, a Ronedo, a Jerrid e agli altri ragazzi, per aver trascorso preziosi minuti in compagnia davanti a un buon caffè (italiano) o seduti ad un tavolo a gustare piatti messicani o burger americani. "Last but not least", sento di ringraziare l'Ingegner Giulia Mauri, che negli ultimi mesi alla UCLA mi ha aiutato nella stesura della tesi con preziosi consigli e correzioni. Ritengo, inoltre, che Giulia abbia contribuito a rendere unica e irripetibile questa esperienza oltreoceano.

Giusto poche parole sulle due frasi citate in alto. La frase di Ralph Waldo Emerson, filosofo e scrittore statunitense dell'ottocento, penso sia emblematica

e, personalmente, la condivido appieno: « ciò che abbiamo alle spalle e quello che ci sta di fronte, sono cose di poco conto rispetto a ciò che sta dentro di noi ». Essa evidenzia che la persona che siamo diventati e quindi ciò che si ha dentro, la si deve alle innumerevoli scelte, azioni ed esperienze che si sono fatte durante la propria vita. Questo "patrimonio" dunque, ci suggerisce Ralph Waldo Emerson, ha un valore inestimabile ed è unico in ognuno di noi. Esso ci dà la forza di guardare al futuro, anche se pieno di difficoltà e incertezze, e al contempo di sorridere al passato, che sicuramente sarà stato caratterizzato da sfide e ostacoli, con fierezza e fermezza. Dunque ciò che abbiamo dovuto affrontare in passato e quello che il futuro ci riserverà, sono cose di poco conto ("tiny") rispetto alla ricchezza che abbiamo maturato dentro di noi.

Riprendo inoltre la celebre frase di Steve Jobs pronunciata ai neolaureati dell'Università di Stanford durante la cerimonia di laurea : « siate affamati, siate folli ». Essa si riferisce all'approccio che si dovrebbe avere alla propria vita, cercando di viverla e spremerla affinché si riesca a trovare la propria strada senza che nessuno possa imporci le proprie ideologie o limitare la nostra creatività e il nostro essere. Personalmente, ritengo questo aspetto di fondamentale importanza nella propria vita.

Francesco

Sommario

La gestione delle risorse è problema di primaria importanza nelle reti di calcolatori e rimane tuttora un aspetto non risolto e da tenere in considerazione. Sfortunatamente, mentre la tecnologia e l'innovazione evolvono, la nostra infrastruttura di rete è rimasta praticamente nella stessa condizione per decenni, dando origine a quello che comunemente viene definito fenomeno di "ossificazione di Internet".

Il Software-Defined Networking (SDN) è un paradigma emergente nel campo delle reti di calcolatori e consente di controllare, tramite un software centralizzato a livello logico, il comportamento dell'intera rete.

Tale gestione è resa possibile attraverso il disaccoppiamento tra la logica di controllo che governa la rete e l'infrastruttura fisica sottostante di switch e router adibiti all'instradamento del traffico. Il meccanismo che permette al piano di controllo di poter comunicare con il piano dei dati è OpenFlow. Gli operatori che si occupano delle reti sono dunque in grado di scrivere programmi di alto livello per il controllo del comportamento dell'intera rete. Inoltre, la centralizzazione del piano di controllo permette di definire complesse operazioni da eseguire sulle reti, relative ad esempio alla sicurezza o alla gestione e controllo delle risorse, attraverso un unico strumento.

Oggigiorno, l'esplosione delle applicazioni usufruite in tempo reale che hanno delle caratteristiche limitanti di Qualità di Servizio (QoS), porta i programmatori delle reti a dover progettare protocolli che garantiscano adeguate prestazioni. La tesi sfrutta le SDNs e l'utilizzo di OpenFlow per gestire nelle reti

servizi differenziati con una elevata QoS. Inizialmente abbiamo definito un'architettura per la gestione e l'orchestrazione della QoS che permetta di gestire la rete modularmente. Inoltre, viene fornita una integrazione tra l'architettura presentata e il paradigma definito dalle SDN, mantenendo la separazione tra il piano di controllo e quello dei dati.

Il nostro lavoro rappresenta una prima fase di configurazione della rete presso la UCLA (University of California, Los Angeles) in grado di offrire servizi differenziati e stringenti requisiti di QoS. Abbiamo inoltre pianificato di sfruttare la nostra soluzione per gestire l'handoff tra differenti tecnologie di rete, i.e., Wi-Fi e WiMAX. Infatti, il modello può essere eseguito utilizzando diversi parametri, dipendenti dal protocollo di comunicazione usato, ed è in grado di fornire risultati ottimali che possono essere direttamente implementati in una rete di campus universitario.

Abstract

Resource management is of paramount importance in network scenarios and it is a long-standing and still open issue. Unfortunately, while technology and innovation continue to evolve, our network infrastructure system has been maintained almost in the same shape for decades and this phenomenon is known as "Internet ossification".

Software-Defined Networking (SDN) is an emerging paradigm in computer networking that allows a logically centralized software program to control the behavior of an entire network. This is done by decoupling the network control logic from the underlying physical routers and switches that forward traffic to the selected destination. One mechanism that allows the control plane to communicate with the data plane is OpenFlow. The network operators could write high-level control programs that specify the behavior of an entire network. Moreover, the centralized control makes it possible to define more specific and complex tasks that could involve many network functionalities, e.g., security, resource management and control, into a single framework.

Nowadays, the explosive growth of real time applications that require stringent Quality of Service (QoS) guarantees, brings the network programmers to design network protocols that deliver certain performance guarantees. This thesis exploits the use of SDN in conjunction with OpenFlow to manage differentiating network services with an high QoS. Initially, we define a QoS Management and Orchestration architecture that allows us to manage the network in a modular way. Then, we provide a seamless integration between the archi-

tecture and the standard SDN paradigm following the separation between the control and data planes.

This work is a first step towards the deployment of our proposal in the University of California, Los Angeles (UCLA) campus network with differentiating services and stringent QoS requirements. We also plan to exploit our solution to manage the handoff between different network technologies, e.g., Wi-Fi and WiMAX. Indeed, the model can be run with different parameters, depending on the communication protocol and can provide optimal results to be implemented on the campus network.

Contents

1 Introduction **2**

2 Background **6**

2.1 Software-Defined Network paradigm 6

2.1.1 Motivation 6

2.1.1.1 “Classical” switch 7

2.1.2 Early Programmable Networks 8

2.1.2.1 Intelligent Network 8

2.1.2.2 OPENSIG 10

2.1.2.3 GSMP 11

2.1.2.4 Active Network 11

2.1.2.5 4D architecture 12

2.1.2.6 NETCONF 13

2.1.2.7 ForCES 13

2.1.2.8 Ethane 13

2.1.3 SDN Architecture 14

2.1.3.1 Logical layers 15

2.1.3.2 SDN switch 16

2.2 OpenFlow protocol 17

2.2.1 Operating principles 18

2.2.1.1 “Instruction Set” 19

2.2.2 Flows based operation 20

2.2.3 Evolution summary 20
2.2.4 Switch components 22
2.2.4.1 Pipeline processing 24
2.2.4.2 Packet matching 24
2.2.4.3 Table miss 25
2.2.4.4 Flow Entry 26
2.3 Control models 27
2.3.1 Flows insertion 27
2.3.2 Control plane distribution 28
2.4 SDN weaknesses and challenges 28
2.5 Floodlight SDN Controller 30
2.5.1 Architecture 30

3 Dealing with SDN **34**
3.1 Related work 34
3.1.1 How to Monitor Network Parameters with OpenFlow . 35
3.1.2 SDN to improve Quality of Service and Quality of Experience 37
3.2 Mininet Network Emulator 41

4 The QoS-aware Mathematical Model **44**
4.1 Enhanced QoS Architecture 48
4.2 Multi-Criteria Approach 51
4.2.1 Multi-Commodity Flow Problem 51
4.2.2 Constrained Shortest Path Problem 53
4.2.3 Our Multi-Commodity Flow and Constrained Shortest Path Model 55

5 Implementation and Experimental Results **60**
5.1 Hardware Configuration 62
5.2 Network Topology 63
5.3 Mininet Configuration 64
5.4 Mininet Hybrid Configuration 65
5.4.1 Network Services and Tools 67
5.4.2 Stress the Network 68

5.5 Mapping the Network 69
5.6 Inserting the Path 69
5.7 Network Metric Measurement 70
5.7.1 Available Bandwidth 71
5.8 Results 72

6 Conclusion **88**
6.1 Future Work 90

A Python Code **94**
A.1 Network Topology Configuration 94

List of Figures

2.1 "Classical" switch components 8
2.2 Intelligent Network Conceptual Model [1] 9
2.3 4D Project Architecture 12
2.4 ForCES Architecture 14
2.5 Ethane Architecture 14
2.6 SDN Architecture 15
2.7 SDN switch components 16
2.8 OpenFlow switch specification [3] 17
2.9 Packets treatment by the switches OpenFlow-enabled 19
2.10 OpenFlow Instruction Set [2] 20
2.11 OpenFlow evolution summary 21
2.12 OpenFlow switch components [4] 23
2.13 Pipeline packets matching [4] 24
2.14 Packet flow matching [4] 25
2.15 Floodlight architecture [5] 30

3.1 The model of an OpenFlow switch 36
3.2 The functional architecture of the latency monitoring application 37
3.3 OpenFlow-assisted QoE Fairness Framework 39

4.1 Streaming quality and packet loss rate (in percentage) for different content types [6] 47
4.2 QoS Management & Orchestration architecture (red dashed line) 49

5.1 Delay in a real packet-switched network [7] 61
5.2 Network topology model 63
5.3 Mininet hybrid topology network 66
5.4 *Floodlight QoS Advisor v.0.2a* 68
5.5 Throughput measurement using *Iperf* tool in the different scenarios (100Mbps network). 73
5.6 Throughput measurement using both the *Iperf* tool and our module with no packet loss 74
5.7 Throughput measurement using both *Iperf* tool (server side) and our module with 4% of packet loss 74
5.8 Bandwidth usage during the video streaming service. 75
5.9 Temporary network congestion during the video streaming service. 75
5.10 Video streaming and file transfer throughputs during a temporary link congestion. 76
5.11 Video streaming and file transfer throughputs during a permanent link congestion. 76
5.12 Video steaming and file transfer paths in a network without QoS. 77
5.13 Warnings due to a under-threshold throughput. 79
5.14 The "Watch Dog" Flow Chart 80
5.15 Path changing by the QoS architecture during a permanent link congestion. 81
5.16 Video streaming throughput during a permanent link congestion in a network managed by the QoS architecture. 81
5.17 Multiple path changing by the QoS architecture during a multiple link congestion. 82
5.18 Video streaming throughput during a multiple permanent link congestion in a network managed by the QoS architecture. . . . 82
5.19 Path changing by the QoS architecture during a permanent link congestion. 83
5.20 Multi-Commodity Flow throughput during a permanent link congestion in a network managed by the QoS architecture. . . . 83
5.21 Different types of video quality. 86

List of Tables

2.1 Flow entry fields [4] 26

4.1 Application quality requirements 44
4.2 Mean Opinion Score levels [8] 45
4.3 Voice connectivity total delay (one-way) [9] 46
4.4 End-To-End delay in the network gaming [10] 46
4.5 QoS requirements of VoIP, Interactive-Video, and Streaming-Video [11] 47

5.1 Conversion between our mathematical model cost and the MOS levels related to the video streaming 84

Chapter 1

Introduction

The currently used network infrastructure system has been maintained almost in the same form for decades, while technology continues to evolve. Resource management is of paramount importance in network scenarios and it is a long-standing and still open issue. Moreover, in this scenario, the main issue to deal with is the decoupling of the network control logic from the data plane of the network, i.e., the physical routers and switches that forward traffic from sources to destinations. Since there are a lot of emerging network paradigms that are trying to find an efficient alternative to the classical Internet architecture, only a few of them are widespread and successful. In this contest, the Software-Defined Networking (SDN) [2] paradigm is one of the best and most attractive solutions for improving the Internet with more flexibility and adaptability issues. This emerging networking paradigm allows a centralized software program to control the behavior of the whole network by separating the routing decision plane from the forwarding layer. The SDN paradigm needs a mechanism to make the communications between the control and data plane possible. This functionality is obtained by means of an emergent protocol, OpenFlow [3]. SDN, in conjunction with OpenFlow, allows us to write high-level control programs that specify the behavior of the network components. These programs can take care of various network tasks, e.g., security, routing, and resource management.

These tasks are among the most important aspects in all network scenarios,

since the main Internet services are generally still based on the classic Best-Effort paradigm. On the one hand, the Best-Effort service is simple and its simplicity has been the most important factor which has determined its world-wide success. On the other hand, unfortunately, the Best-Effort service does not provide any guarantee on bandwidth, end-to-end delay, and packet loss. Furthermore, nowadays the demand of quality associated to the transport of data and media is growing both in academia and industry. This need represents a very hard challenge and it requires a significant effort towards a QoS-enabled network.

Traditionally, the QoS is defined in terms of availability, i.e., the percentage of time in which the reference system is available and working. In this way, the Service Level Agreements (SLAs) have been defined as a function of percentage availability, e.g., 99.999%, known as “five nines”, that implies a downtime of 5.26 minutes per year. The downtime is the amount of time required to identify and repair a fault in the connection or in the equipment. Furthermore, it is very important to understand that each type of service has different SLA requirements, not only based on availability. Hence, the advanced SLA has to take into account the packet delay and the packet loss parameters in addition to the availability. It follows that the quality of some types of applications depends on delay and/or packet loss (e.g., real-time applications or multimedia transmissions). On the one hand, the telephony services (e.g., VoIP) have strict delay requirements for the packet and codec. In these applications, if a packet reaches its destination after a given delay threshold, the service becomes useless. For the real-time applications, the retransmission of lost packets is also worthless. On the other hand, we have other types of applications that are called elastic. In general, these services are more robust than real-time applications related to packet loss. In fact, they typically allow retransmission, usually accomplished in an end-to-end fashion through the TCP mechanisms. For instance, the File Transfer Protocol (FTP) services and more generally (even if not always) the data transfer applications are referred to as elastic.

It is not easy to guarantee QoS requirements in the traditional Best-Effort networks. Regarding this, the Internet Engineering Task Force (IETF) [12] has proposed different QoS architectures, such as IntServ [13] and DiffServ [14], in

the last decades. However, these proposals have not been very successful or implemented in a wide scale, because they require some fundamental changes on Internet design. In the current Internet architecture, there is also a severe lack of information about the available network resources from the end-to-end point of view. A partial solution came from the Multiprotocol Label Switching (MPLS) and the Border Gateway Protocol (BGP) techniques [15] that are defined to solve these problems. Unfortunately, these solutions lack the real-time reconfigurability and adaptivity. In this scenario, the SDN paradigm can be a fundamental key to overcome the current Best-Effort limitations explained above.

This thesis exploits the use of OpenFlow in SDNs to manage differentiating network services with a high QoS. In particular, we consider a Video Streaming service and a Data Transfer service. Firstly, we define a QoS Management and Orchestration architecture that allows us to manage the network in a modular way. Secondly, we provide a seamless integration between the architecture and the standard SDN paradigm following the separation between the control and data planes.

Then, we give an Integer Linear Programming (ILP) formulation of the problem of guaranteeing a good QoS in terms of packet loss and delay, taking into account the constraints of the network, i.e., maximum acceptable packet loss and delay for each type of service and available bandwidth on the links. Specifically, our model defines a Multi-Commodity Flow Constrained Shortest Path (MCFCSP) problem and takes advantage of both the well known problems derived from Operation Research: the Multi-Commodity Flow Problem (MFP) and the Constrained Shortest Path (CSP). Given the optimal solution of the problem, we integrate the results with an emulated network by means of Mininet [16]. Thus, it is possible to map the different network flows on a real network based on the optimal solution from the model. Moreover, we define various levels of QoS, according to the MOS system [8] for the services that we are considering. Finally, we found a connection between the optimal solution provided by the model and the MOS levels. We used these results to provide evidence of the effectiveness of our model compared to the traditional solution given by the emulator. Then, changing the network conditions, it is easy

to find the new optimal routes between source and destination by means of the model. It is also possible to dynamically map the routes into the network and, most important, to guarantee the QoS necessities.

This work is a first step towards the deployment of our proposal in the University of California, Los Angeles (UCLA) campus network with differentiating services and stringent QoS requirements.

Thesis Outline

This thesis describes a new architecture that allows enhanced QoS in SDN networks. The architecture is composed of several modules that give us the possibility to retrieve information about the network status and manage the switches according to our mathematical model. The remainder of the thesis is structured as follows:

- **Chapter 2** provides an overall view about the SDN paradigm, starting from the early solutions that laid the foundation for SDN and then focusing on the OpenFlow protocol characteristics. The SDN weaknesses and the Floodlight controller are also presented.

- **Chapter 3** gives an overview of the state of the art and presents Mininet, an emulated environment suitable for dealing with SDN.

- **Chapter 4** describes our novel architecture that makes an enhanced QoS model in SDN networks possible. This chapter also details the mathematical model in depth, based on the Multi-criteria approach, which is the core of our architecture.

- **Chapter 5** provides the implementation of the proposed architecture and also the performance assessment and the numerical evaluation.

- **Chapter 6** lists open research problems and provides the conclusions.

Chapter 2

Background

2.1 Software-Defined Network paradigm

Sofware-Defined Networking (SDN) was conceived at the UC Berkeley and Stanford University in 2008. The Open Networking Foundation (ONF) [17], a non-profit industry consortium founded in 2011, is dedicated to the promotion and adoption of SDN through open standards development like as OpenFlow™ protocol. The purpose of this chapter is to give a brief overview of the SDN architecture, paradigm, and protocol.

2.1.1 Motivation

The term **Internet ossification** [18] expresses the difficulty of the Internet to evolve in terms of both its physical infrastructure as well as its protocols and performance. The Internet is considered part of our society critical infrastructure and it has a huge deployment base (like as in transportation, power grids, water supply, etc.) that makes its evolution not simple. Furthermore, it is imperative that the Internet would be able to evolve to address new challenges as represented by new applications and services that are becoming increasingly more complex and demanding.

With a more detailed view, in current networks, it is very hard to deploy new protocols, services, resources optimization, and traffic differentiation, because the routers and the switches are usually "closed" systems, often with

limited functionalities and vendor-specific control interfaces. Furthermore, the lack of a common control interface to the various network devices may require high efforts for the configuration or the policy enforcement of them. Thus, it is more difficult for the network infrastructure to evolve, once it has been deployed and in production. Nevertheless, “middleboxes” like as firewalls, Intrusion Detection Systems, and Network Address Translators, were used to overcome, in a “workaround way”, these limitations and circumvent the network ossification effect.

The network infrastructure ossification issues are largely attributed to the tight coupling between the **control logic** and the **forwarding hardware** which means that the decisions about the data flowing through the network are taken directly on-board from each “classical” network device.

2.1.1.1 “Classical” switch

In a classical router or switch, the fast **packet forwarding** (data plane) and the high level **routing decisions** (control plane) occur on the same device, as depicted in Figure 2.1. In this figure, the main components respectively are:

1. The **forwarding elements** are generally **Application-Specific Integrated Circuits** (ASIC), network-processors, or general-purpose processor-based devices that handle data path operations for each packet. They are designed to perform very quickly one particular function: to forward frames and packets at wire speed (line-rate). Furthermore, they are able to increase the lookup functions using very specialized memory resources like as **Content Addressable Memory** (CAM) or **Ternary Content Addressable Memory** (TCAM) to hold the forwarding information.

2. The **control elements** in general are based on general-purpose processors that provide control functionalities, like routing and signaling protocols.

The main difference between a “classical” switch and a switch able to work on an SDN network, is the architecture, as explained further below. However, before explaining the details of the SDN architecture, we focus on pioneer works that had provided inspiration for the SDN. In these works, the fundamental keys had been the decoupling of control plane from data plane and the

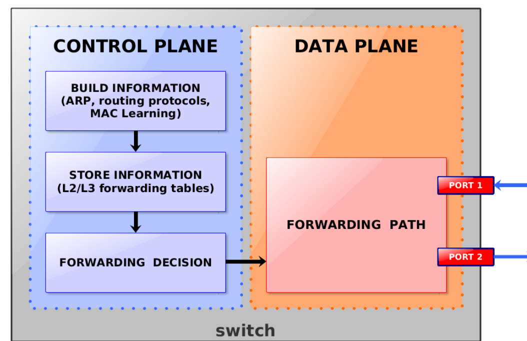


Figure 2.1: "Classical" switch components

network programmability, as explained in the next section.

2.1.2 Early Programmable Networks

The following chapter discusses the "early programmable networks", starting from the Eighties to nowadays. In the 1980s, the idea of centralized and decoupled control network started in the telecommunication field with the Intelligent Network architecture. Then, in the mid 1990s, the programmable network principles were established with some projects, precursors of the current SDN paradigm.

2.1.2.1 Intelligent Network

The **Intelligent Network** (IN) was an architectural concept, introduced by the Bell Communications Research group, that was applied to the development of new services in **wireline telephone**. IN enables the real-time execution of network services and customer applications in a distributed environment. Through separation between the software that controls the basic switch functionalities and the software that controls the call progression, the main goal was the rapid development of differentiating services. Furthermore, the main components of the IN architecture are depicted in Figure 2.2 and they represents a framework called **Intelligent Network Conceptual Model** (INCM). Specifically, the INCM model is composed of four planes, as described in [19]:

1. The **Service Plane** (SP) describes services from the user prospective. It consists of one or more **Service Features** (SF) that represent a service

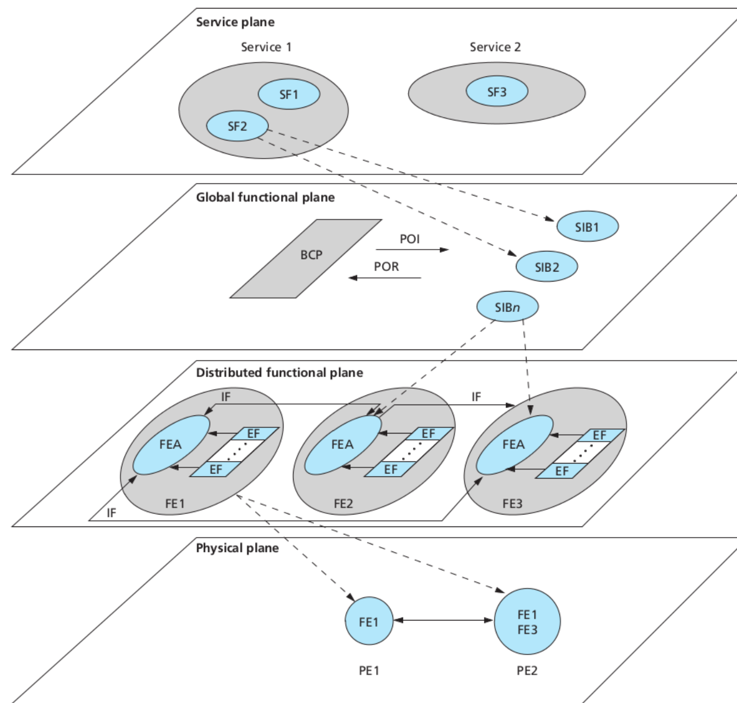


Figure 2.2: Intelligent Network Conceptual Model [1]

component. Furthermore, a service component can be a complete service or part of a service. This composition principle can make the services customization by the subscribers instead of the telco operators possible.

2. The **Global Functional Plane** (GFP) deals with service creation and models the network as a unique and global virtual machine. This plane contains the **Service Independent Building Blocks** (SIB) that are a set of standard and reusable capabilities used to build features and services. Furthermore, the **Basic Call Process** (BCP) is a SIB from which a service is launched. In this plane, a service consists of a chain of SIBs which can be viewed as a script.

3. The **Distributed Functional Plane** (DFP) defines the functional architecture. It is composed of a set of **Functional Entities** (FE) that realize the network functionalities. The main functions of this plane are:

   a. The **Service Control Function** (SCF) that contains the service logic and controls the execution of the service.

b. The **Service Switching Function** (SFF) that provides a standardized interface between the SCF and the switch, allowing the control of them.

c. The **Specialized Resource Function** (SRF) that performs user interaction functions through established connections.

d. The **Service Data Function** (SDF) which performs related data processing function used to update and retrieve user information.

e. The **Service Management Function** (SMF) that handles the activities related to the service deployment, service control, service monitoring, service provisioning, and service billing.

f. The **Service Creation Environment Function** (SCEF) which allows the service definition, development, and testing on the network.

4. The **Physical Plane** (PP) corresponds to the physical architecture of the IN and it is composed of the **Physical Entities** (PE) and the interfaces among them. Furthermore, the FE entities of the DFP plane are directly implemented into the PE in the PP plane and the interactions among different PEs are possible through the **Intelligent Network Application Protocol** (INAP).

Thus, the IN architecture is based on a centralized control and the service control is completely separated from call control as happens in the SDN networks. The Open Signaling Working Group also presented an architecture to make the network programmability possible, as explained in the next section.

2.1.2.2 OPENSIG

At the beginning, the **Open Signaling Working Group** (OPENSIG), proposed a solution based on an open and programmable network interface to access the network hardware. The motivation originally came from the observation that monolithic and complex control architectures could be restructured as a minimal set of layers. By means of this partition and by using an open interface, it became possible to easily access the services located in each layer. This approach was used to introduce programmability in the control plane of the telecommunication networks based on ATM [20]. Furthermore, an IETF

group understood the need to define a common protocol, called GSMP, suitable for the network device management.

2.1.2.3 GSMP

Starting from the programmable network interface idea, an IETF working group developed the **General Switch Management Protocol** (GSMP), a protocol specifically designed for the management of the switches by external components. Furthermore, the protocol allows the external controller to get network statistic information, manage connections, ports, and resources of the switches [21]. GSMP v. 3 is the last version of the protocol, published in June 2002. Another important networking group, Active Network, also defined a novel architecture to improve the network programmability, as detailed above.

2.1.2.4 Active Network

Around 1997, the **Active Networking Group** proposed an innovative programmable network architecture approach, named **Active Network**, in which the switches perform customized operations on the data messages flowing through them [22]. The main objective was to decouple network services from the hardware allowing the loading of new services into the infrastructure by need. The network is considered “active” in the sense that the switches can perform computation on the packet contents. Moreover, there are two different approaches to build active networks that are respectively considered “discrete” (out-of-band) and “integrated” (in-band) [23]:

1. The **“programmable switch”** approach is based on the injection of customized programs into the active nodes (switches or routers). Through the examination of the message header, the specific programs can process the packets and perform computations like as: modify, store or redirect the data. In the Internet based on the Active Network architecture, it is possible to dynamically load code into the nodes through a “back door”, achieving a router extensibility purpose. It follows that the main goal of this approach is focused on the separation between the injection programs and the processing of the messages.

2. The **"capsules"** approach is based on the possibility to embed a program fragment and data into every packet (called capsule). Therefore, when a capsule reaches an active node, the node is able to evaluate and process the embedded code.

These approaches aim to make the basic network services selectable on a per packet basis, reducing the deployment time and allowing the network programmability. Moreover, about ten years ago, another architecture was developed to reach a flexible network programmability by decoupling the data plane and control plane, as described below.

2.1.2.5 4D architecture

In the mid of 2000, the **4D architecture** [24] was developed with a clear separation between the routing decision logic and the protocols governing the interaction with the network elements. As depicted in Figure 2.3, the network control functions are divided into 4 planes: decision, dissemination, discovery, and data.

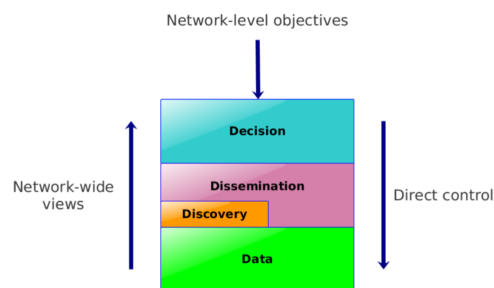


Figure 2.3: 4D Project Architecture

In the 4D architecture, each plane has a clear task:

1. The **decision plane** is responsible for the network configuration through the management of the router Forwarding Information Base.
2. The **dissemination plane** is accountable for the network state information gathering such as link up/down information.
3. The **discovery plane** enables devices to discover their directly connected neighbors.

4. The **data plane** is responsible for the network traffic forwarding.

This architecture provides direct inspiration for later works such as **NOX** (Network Operating System), which proposed an "operating system for networks" that provides an uniform and centralized programmatic interface to the entire network [25]. Moreover, the IETF Network Configuration Working Group presented a protocol to make the network configuration possible.

2.1.2.6 NETCONF

In 2006, the IETF Network Configuration Working Group proposed the **NETCONF Configuration Protocol** [26]. NETCONF protocol was not designed for enabling direct control of the switches, but as a management protocol for modifying the configuration of the network devices by means of API. Unfortunately, in this solution there is no separation between the data and the control plane. However, the ForCES IETF group developed an architecture and a protocol to separate the forwarding plane from the routing plane.

2.1.2.7 ForCES

The **IETF Forwarding and Control Element Separation** (ForCES) Working Group defined the architectural framework and the associated protocols to standardize the exchange of information between the control and forwarding plane [27]. This approach, drawn in Figure 2.4, aims to improve the interoperability and flexibility enabling a rapid innovation in the control and forwarding planes. Since 2003, the ForCES protocol has been undergoing standardization.

The next section also present the Ethane architecture that is considered as the predecessor of the OpenFlow protocol, laid the foundation for what SDN would become.

2.1.2.8 Ethane

The **Ethane** project [28] defines a new network architecture for enterprise networks, started around 2006. The network is composed of the Ethane switches, that include the flow tables, and the controller. By means of a secure channel, the controller can communicate with the switches and can decide whether

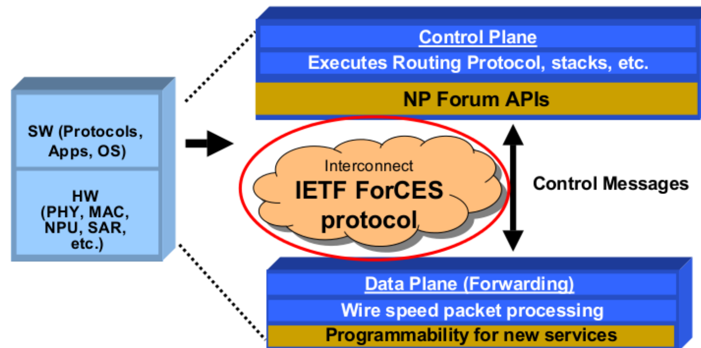


Figure 2.4: ForCES Architecture

a packet should be forwarded or not. As depicted in Figure 2.5, the Ethane architecture is composed of an external and centralized controller that can interact with the Ethane switches for managing policy and security.

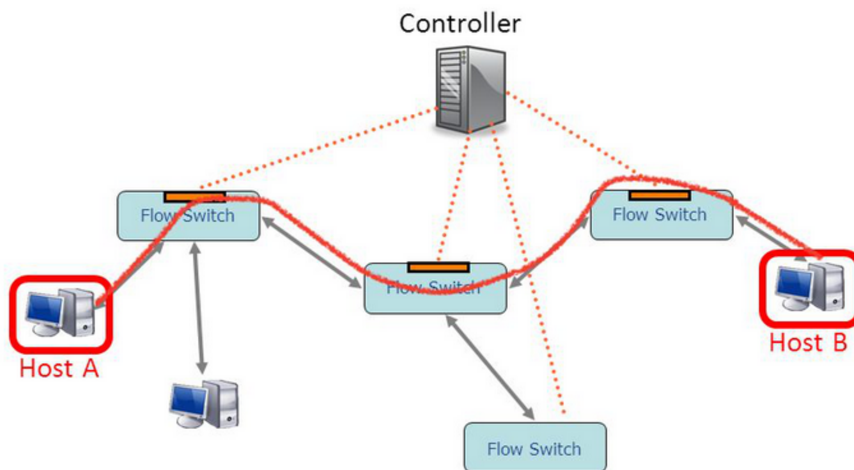


Figure 2.5: Ethane Architecture

As we can clearly see in Figure 2.5, the Ethane architecture makes the separation between the control plane and the data plane possible. This is an important aspect that is the core of the SDN architecture, as detailed above.

2.1.3 SDN Architecture

In the following section, we are interested in explaining the main components of the SDN architecture. The SDN architecture represents a new network-

ing paradigm that decouples the **control plane** from the **data plane**, facilitating the network evolution, interoperability, and scalability. This decoupling, that is the SDN "core", is possible through the switch components separation, as explained further in Section 2.1.3.2. Furthermore, the main differences between the "classical" network and the new networking paradigm are described through three different logical layers as detailed below.

2.1.3.1 Logical layers

The SDN architecture can be represented by three different logical layers [2], as shown in Figure 2.6.

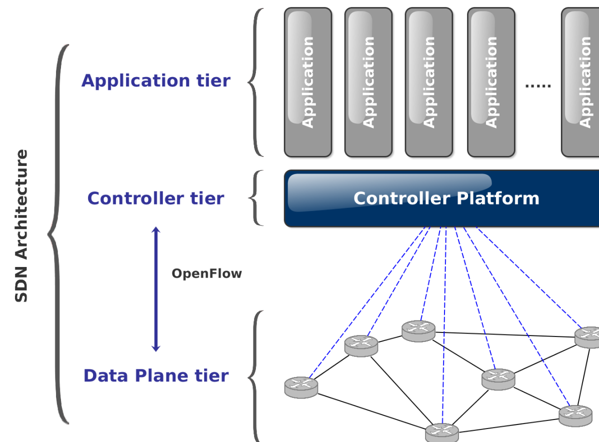


Figure 2.6: SDN Architecture

Specifically, each tier has different functionalities:

1. The **data plan** tier portrays the network infrastructure composed of physical devices (e.g., switches and routers).

2. The **controller** tier represents the "network intelligence" and it is logically centralized in the SDN controller, virtually located in this layer. This solution allows the controller to maintain a global view of the network, placed in the **infrastructure layer**.

3. The **application** tier represents the layer where network operators and administrators can operate. By centralizing the network state in the controller tier, at the application layer it is possible to configure, manage,

secure, and optimize network resources via dynamic, automated SDN programs. Moreover, the network operators can directly write and deploy customized programs themselves without waiting for the vendors releases that could take long time.

Hence, the layers abstraction described above facilitates the programmers to operate on a network abstraction layer instead of thousands of different physical devices, through Application Programming Interfaces (APIs). However, this abstraction is possible only if the under layer infrastructure makes the interaction with itself possible. To reach this goal, the SDN architecture requires physical devices with different characteristics compared with the "classical" switch, as explained in the next section.

2.1.3.2 SDN switch

On the one hand, by means of the SDN architecture, the network becomes a "simple" packets forwarding element. On the other hand, the high-level routing decisions and state information are centralized in an external and separate server controller, instead of enforcing policies and running protocols on a convolution of scattered devices, as shown in Figure 2.7.

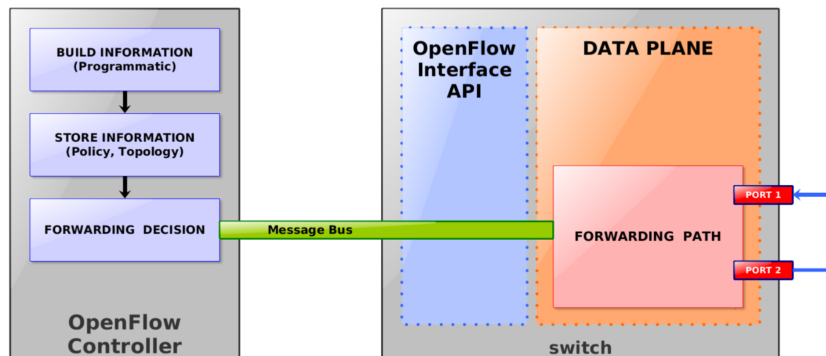


Figure 2.7: SDN switch components

SDN, by separating the control plane from the data plane, can offer a flexible network automation and management framework. This framework makes the development of tools for automating tasks (that are done manually today) possible. These automation tools can reduce operational overhead decreasing network instability introduced by operator error. Unfortunately, the vendors

software environments is typically proprietary and closed and they do not make the management and tweaking of the network easy. However, the SDN architecture can facilitate innovation and enable simple programmatic control of the network data-path giving rise to the idea of programmable networks. This is an important aspect that allows to lower the barrier to the entrance for new ideas.

Clarified the architecture, described further in Section 2.2.4, the next section explains how it is possible to communicate among the different layers.

2.2 OpenFlow protocol

OpenFlow™ is an open standard protocol, specifically designed for the SDN networks, that allows the communication between the control and data planes and permits the manipulation of the latter. As illustrated in Figure 2.8, the OpenFlow switches and the controller can communicate via the OpenFlow protocol over a secure channel. The protocol defines different messages such as packet-received, send-packet-out, modify-forwarding-table, and get-stats, that can be exchanged between the switch and the controller.

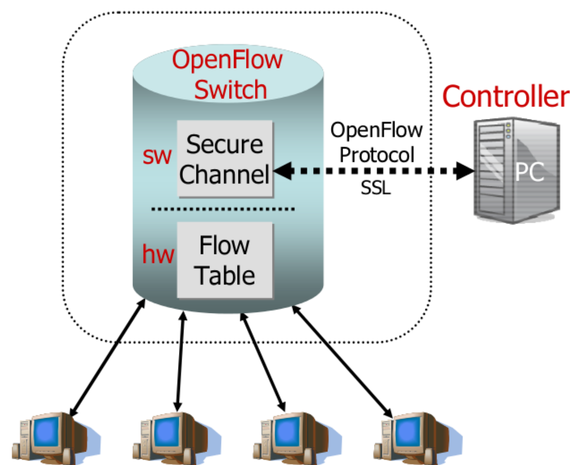


Figure 2.8: OpenFlow switch specification [3]

At the beginning, the OpenFlow protocol was developed at the Stanford University around 2008 for enabling researchers to run experimental proto-

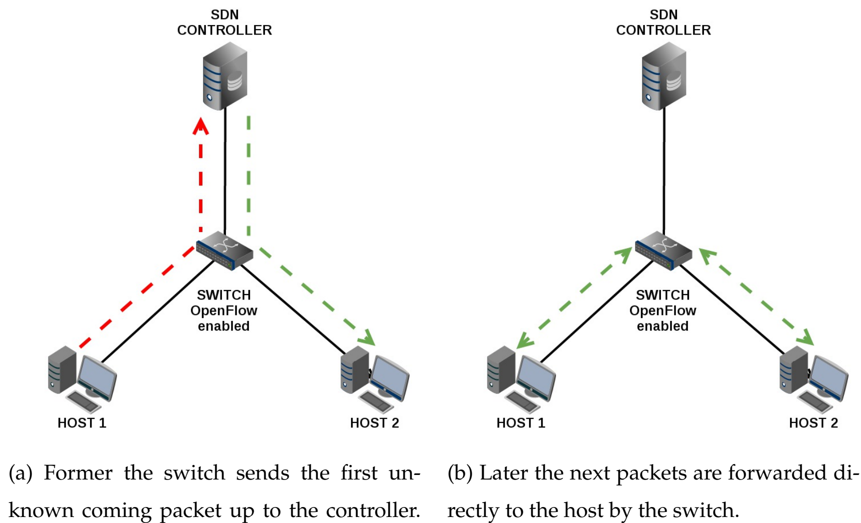
cols in the campus networks [3]. Presently, OpenFlow is added as a feature to commercial network devices and it provides a standardized vendor-agnostic interface to access the Ethernet switches, routers and wireless access points. Moreover, these devices, called "OpenFlow-enabled", allow the access without requiring vendors to expose the internal workings of their products. Thus, the OpenFlow protocol makes the deployment of innovative routing and switching protocols easy. Furthermore, it can be used for applications such as virtual machine mobility, high-security networks and next generation IP-based mobile networks.

The next sections describe the OpenFlow operativeness in depth, starting from a general overview, proceeding with the protocol evolution summary and reaching the protocol details according to the newest specification [4].

2.2.1 Operating principles

The following section aims to provide a first general overview of the interactions among the switches and the controller. When an OpenFlow Switch receives a packet that it has never seen before and for which it has no matching flow entries, it sends this packet, called *packet-in* to the controller as in Figure 2.9a. Then, the controller takes a decision on how to handle this packet. It can drop the packet, or add a flow entry directing into the switch. In case of flow entry insertion, the switch learns how to forward similar packets in the future, as shown in Figure 2.9b.

While additional details of these interactions are further described below, by now it is interesting to figure out a parallelism between the steps described above and the cache interaction of a Central Processing Unit (CPU). In particular, when a *cache miss* occurs, the actions typically taken by a CPU are comparable to the OpenFlow protocol interactions in case of no match against the Flow entries. In fact, when a CPU needs a specific data, the first step is to search it into the cache (starting from the nearest one, e.g., L1 layer cache), like the lookup phase into the switch. If the data is in the cache, a *cache hit* occurs and the CPU is able to continue with the next instruction. Otherwise, the CPU has to manage a *cache miss* retrieving the data somewhere else (e.g., from the L2 layer cache or directly from the RAM or, if necessary, from the disk). This ap-



(a) Former the switch sends the first unknown coming packet up to the controller.

(b) Later the next packets are forwarded directly to the host by the switch.

Figure 2.9: Packets treatment by the switches OpenFlow-enabled

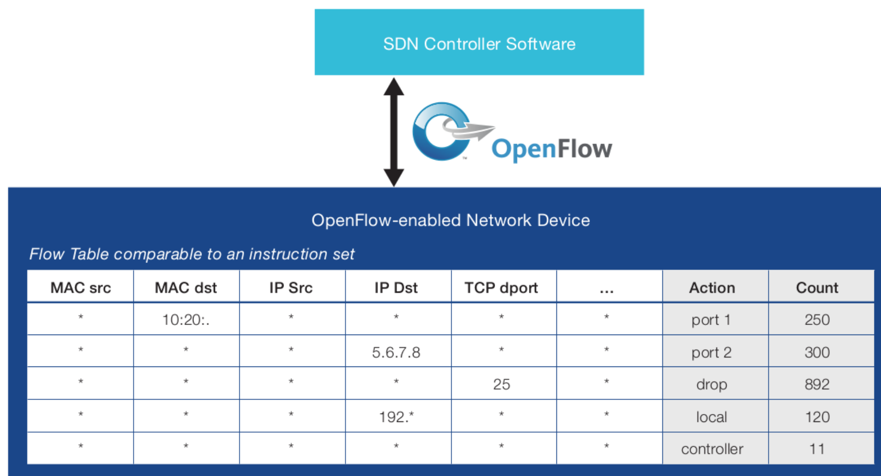
proach is similar to the steps done into the switch in case of no matching flow entries.

A further interesting comparison can be made between the OpenFlow protocol and the functionalities (i.e., *Instruction Set*) of a CPU.

2.2.1.1 "Instruction Set"

The features offered by the OpenFlow protocol can be assimilated to the Instruction Set Architecture (ISA) of a CPU, as drawn in Figure 2.10. Since the instruction set allows to access the internal architecture of a CPU (memory, registry, etc.), the OpenFlow protocol provides to the external software application the primitives that can be used to program the forwarding plane of the network devices.

The main actions that the protocol can take are based on the data flows setting inside the switches, as explained below.



MAC src	MAC dst	IP Src	IP Dst	TCP dport	...	Action	Count
*	10:20:.	*	*	*	*	port 1	250
*	*	*	5.6.7.8	*	*	port 2	300
*	*	*	*	25	*	drop	892
*	*	*	192.*	*	*	local	120
*	*	*	*	*	*	controller	11

Figure 2.10: OpenFlow Instruction Set [2]

2.2.2 Flows based operation

The OpenFlow protocol widely uses the concept of “flows” to identify network traffic based on pre-defined matching rules that can be statically or dynamically programmed by the SDN control software. Not only OpenFlow permits the network programming on a per-flow basis, but also provides a granular control of the data flows, enabling the network to dynamically adapt the resources by need. However, this per-flow control is generally not possible in the current IP-based routing schemes. In fact, in that case, all flows between two end points must follow the same path through the network, regardless of their different requirements.

Before going in depth into the last OpenFlow specifications, a look at the protocol evolution summary can give us the idea of the improvements that have been done, as detailed in the next section.

2.2.3 Evolution summary

The first OpenFlow protocol release, the 1.0 version, was conceived on December 2009. Then, passing through intermediate protocol evolution, the OpenFlow protocol has reached the recent and stable 1.4 version, as summarized in Figure 2.11.

Specifically, the main functionalities of the first OpenFlow 1.0 version [29]

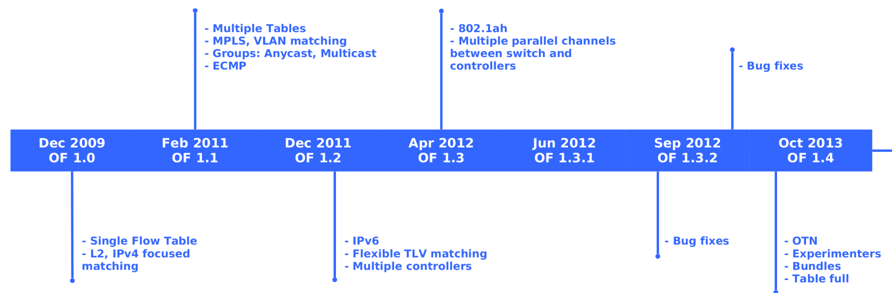


Figure 2.11: OpenFlow evolution summary

are:

1. The **single logical table** for the implementation of the flow rules. Moreover, this aspect limits the full utilization of hardware ASIC capabilities.

2. The **groups** for the creation of group ports, similar to link aggregation in legacy networks. This is suitable for multipathing or redundancy.

3. The **virtual LAN** (VLAN) suitable for a coarse tag supporting.

4. The **virtual ports** that extend OpenFlow beyond physical ports enabling OpenFlow to be used to implement network virtualization for multi-tenancy at scale.

5. The **connection interruption management** useful in case of connection loss. Generally, when the controller connection fails or is terminated and cannot connect to a backup controller, the switch goes into "emergency mode" and immediately resets the current TCP connection. In this state, the matching process is dictated by the emergency flow table entries (marked with the emergency bit), whereas any other entries are deleted. The switch continues to operate in OpenFlow mode until it reconnects to a controller.

However, several new features have been built in the last OpenFlow 1.4.0 specification [4]. The most important protocol improvements are the following:

1. The **Type-Length-Value (TLV) format improvement** to enhance protocol extensibility. The TLV structure is suitable for supporting additional future experimentation.

2. The **multi-controller support** makes an enforced interaction and synchronization among controllers possible. Specifically, the flow monitoring allows a controller to identify in a switch the changes made by other controllers. In addition, when a group table or meter table switch are updated, the controllers associated with that device are notified.

3. The **atomic execution** of a bundle of instruction to avoid intermediate states. In particular, a controller should not receive any notification resulting from the partial application of the bundle (bundle fails).

4. The **fine-grained rule capacity** for helping the controller to manage its capacity limitations for storing rules. Specifically, in case of table full, the switches can proactively evict rules according to the importance of the entries. If the rule tables are filling up, some "vacancy events" can also be used as early detection system to warn the controller.

5. The **optical port support** allows us to deal with fiber-optic networking, by managing frequency and power involved in optical communication.

The next section explains in depth the main characteristics of the protocol, according to the OpenFlow 1.4.0 specification.

2.2.4 Switch components

As discussed in Section 2.1.3.2, the OpenFlow switch is a fundamental part of SDN. Each switch, which is represented as basic forwarding hardware accessible via an open interface, comes in two varieties:

1. The **"pure" OpenFlow switches** have no legacy features or on-board control, and completely rely on a controller for forwarding decisions.

2. The **"hybrid" switches** support OpenFlow in addition to the traditional operation and protocols, making backwards compatibility possible (most commercial switches available today are hybrids).

Furthermore, each switch is composed internally of three different components, as sketched in Figure 2.12.

As illustrated in Figure 2.12, the main parts of a switch are the following:

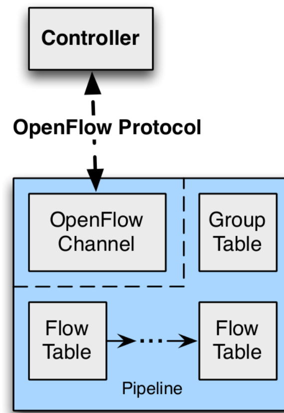


Figure 2.12: OpenFlow switch components [4]

1. An **OpenFlow channel** that allows the communication and management between the external controller and the switch via the OpenFlow protocol.

2. One or more **flow tables** that store the **flow entries** for performing packet lookup and forwarding.

3. The **group table**, a special kind of table designed to perform operations that are common across a set of flows. This approach enables complex forwarding actions such as multipath and link aggregation.

The OpenFlow channel is the interface that connects each OpenFlow switch to a controller. By means of this interface, the controller can manage several switches sending and receiving messages from them, according to the OpenFlow protocol. The OpenFlow channel also allows a secure communication among the switches and the controller using the Transport Layer Security (TLS) cryptographic protocol (by the way, the communication may be run directly over TCP).

Since the OpenFlow tables are considered the core of OpenFlow, a detailed analysis is required. Specifically, the OpenFlow tables are composed of some fundamentals mechanisms. As described in the next section, the most important are: the packet matching, that extracts the packet header and executes actions associated to its, and the pipeline processing, that allows the switches to forward and process a packet through the table chain.

2.2.4.1 Pipeline processing

Every OpenFlow switch can contains multiple flow tables, that are generally composed of several flow entries, as explained in Section 2.2.4. The pipeline processing mechanism specifies how the packets have to interact with each flow table, as depicted in Figure 2.13.

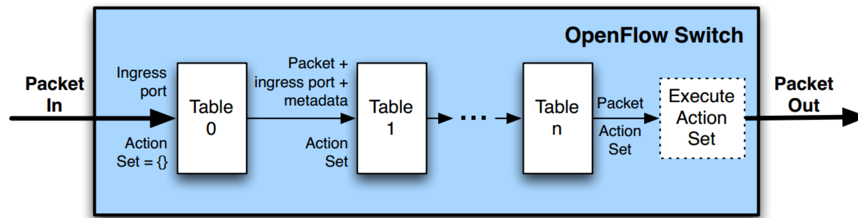


Figure 2.13: Pipeline packets matching [4]

As illustrated in Figure 2.13, each packet is matched against the flow entries starting at the first flow table, called *flow table 0*. Then, depending on the outcome of the previous match, the pipeline processing can continue, going forward to the next subsequent flow table for further processing.

For each table, a packet matching can occur and, consequently, some specific actions can be taken by the switch. Since the packet matching is the first entry point consulted for the lookup procedure, more details concerning this mechanism are necessary.

2.2.4.2 Packet matching

When a packet arrives at the Flow Table, the packet match fields, that can be different according to the packet type, are extracted from the packet header and they are used for the table lookup. Moreover, the matches can be performed against the information related to the ingress port and, in case, the metadata fields, that can be also used to pass information between tables. Thus, if a matching entry is found, the switch executes the instruction set associated with the matched flow entry. These instructions typically can contain actions (like as packet forwarding, packet modification, and group table) or they can modify the pipeline processing. Furthermore, if some actions are applied during the pipeline processing, the modifications are reflected in the packet match

fields, which represent the current packet state. In any case, when the instruction set associated with a matching flow entry does not specify a next table, the pipeline processing stops. Only at that time, the packet is processed with its associated actions set and usually forwarded, as shown in Figure 2.14

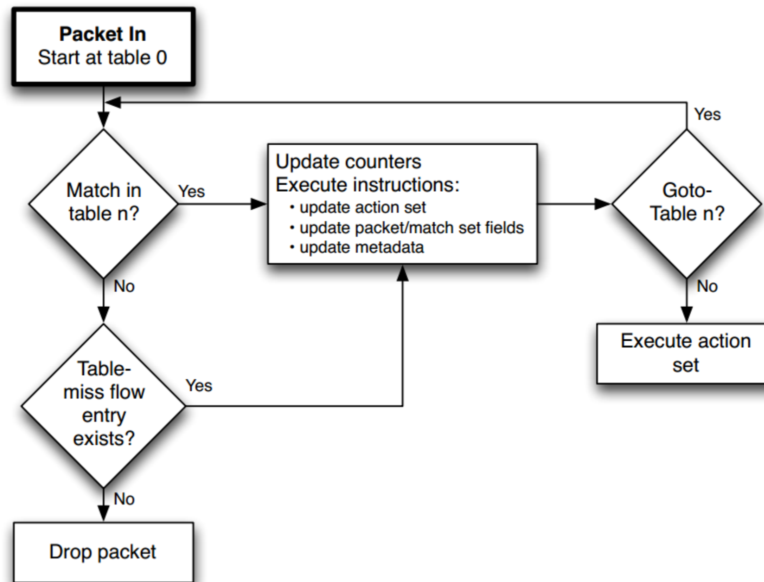


Figure 2.14: Packet flow matching [4]

However, if the lookup phase does not match any entry, a so-called *table-miss* event occurs, as explained in the section below.

2.2.4.3 Table miss

When no packet match is found during a lookup phase, a *table-miss* event occurs. Each flow table must support a *table-miss* flow entry to process table misses, according to the specification. In case of table miss, the switch can take some actions according to the instruction set defined at the *table-miss* flow entry. These instructions can specify to forward the packet to the controller, or to drop it, or to simply continue to the next subsequent flow table. However, if the *table-miss* flow entry does not exist, the packets unmatched by flow entries are discarded by default.

The next section describes in detail the flow entry element which is the fundamental part of the flow table.

2.2.4.4 Flow Entry

As asserted in Section 2.2.4, the flow table is one of the main components that each OpenFlow switch must have (at least one), according to the specification. Each flow table consists of a set of flow entries that have some specific fields, as illustrated in Table 2.1.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

Table 2.1: Flow entry fields [4]

Specifically, the flow entry fields are the following:

1. **Match fields**, consist of the ingress port, packet headers, and optionally metadata specified by a previous table. The **packet header** information of the incoming packets is compared with the match field of each flow entry and if there is a match, the packet is processed according to the action contained by that entry.

2. **Priority**, useful to specify a matching precedence of the flow entry.

3. **Counters**, used to collect statistics for a particular flow or port or queue, such as number of received packets or bytes and duration of the flow.

4. **Instructions**, used to modify the action set or the pipeline processing.

5. **Timeouts**, suitable for specifying the maximum amount of time or idle time before the flow is expired by the switch.

6. **Cookie**, can be used by the controller to filter flow statistics. Furthermore, they are useful to recognize a specific flow.

Each flow table entry is uniquely identified by its match fields and priority. The entry which has all field omitted and priority equal 0, is called *table-miss* flow entry, as previously detailed in Section 2.2.4.3.

The next section discusses the different control models that characterize OpenFlow and consequently the network behavior.

2.3 Control models

In the SDN, the behavior of the network is partly regulated by the controller that represents the main SDN component, and also it allows us to maintain a general view of the network and to manage the conduct of the switches. Furthermore, the decoupling of the control and data layer allows the controller to provide a programmatic interface to the network, where applications can be written to perform management tasks and various functionalities. Moreover, two important aspects in a SDN network are the **scalability** and the **performance** of the network controller. They depend, in part, on the control models related to the OpenFlow controllers. Thus, the target of the following sections is to give us a general overview of the control models that can be used for managing a SDN network.

2.3.1 Flows insertion

In general, there are three different approaches for the insertion of the flows into the switches OpenFlow-enabled:

1. In the **reactive** approach, after the first packet arrival to the switch, if there are not matching flow entries in the flow table, the packet is forwarded to the controller. The controller can make some decisions (e.g., drop or forward the packet) and, in case, insert the flow entry into the switch. Then, the next packets related to that specific flow, will be managed directly by the switch according to the flow entries.

2. In the **proactive** mode, before the packets arrival, the controller can proactively insert the flow entries into the switches. In that case, when the packets arrive, the switches know how to manage those flows without interactions with the controller.

3. The **predictive** behavior uses the historical data regarding the network performance to make the adjustment of the routes and flows possible.

It follows that the amount of the packets exchanged between the switches and the controller is reduced in the proactive approach. Furthermore, the proactive mode makes the “make-before-break” approach possible. In other words,

retrieving information related to the **network status** is useful to understand what is going on and to prevent a down link by finding a new path.

One other important aspect regards the control plane distribution, which is directly correlated to both the scalability and the availability of the network controller.

2.3.2 Control plane distribution

The distribution of the control plane is not specified by the OpenFlow protocol, as well as the controller-to-controller communication. The controller can be implemented as a single **centralized** or a **distributed** server. In case of a distributed solution, different schemes can be implemented (e.g., one main controller and some back-up controllers updated by a hot/cold copy model). However, any type of distribution or redundancy in the control plane can be useful to enforce the **availability** and **scalability** of the controller.

The main weaknesses and challenges related to SDNs are explained in Section 2.4, according to the control models above mentioned.

2.4 SDN weaknesses and challenges

The focus of this section concerns the weaknesses and challenges of dealing with the SDN networks and the OpenFlow protocol. SDN and OpenFlow offer a way to make simple the prototyping, deployment, and management of the network elements. However, we must also take into consideration some interesting aspects that can lead the network in an unsafe or unavailable condition [30] [31], as follows:

1. The **availability** of the controller is the main aspect that is necessary to consider. The tight dependence between the switches and the controller whenever a modification of the rules is necessary, could become a problem. Moreover, if the network design takes into account only one **centralized** controller, it could become a "single point of failure". A **distributed** approach could be implemented to guarantee the availability and avoid a potential undesirable failure. In addition, some redundancy or backup solution could be used for enforcing the robustness.

2. The **security** is also important. In SDN, the controller is a component with a critical knowledge of the network and this aspect exposes the controller to possible attacks and threats. Additionally, the channels among the controller and the switches could be vulnerable. According to the OpenFlow specification, it is possible to use a secure communication by means of the TLS protocol, but its usage depends on the design of the network since it is not required.

3. The **consistency** of the flow tables is also a potential issue. Since several controllers can manage the same flow tables, for instance, a production hardware controller and some other experimental controllers, it follows that the latter will be the "weakest link in the chain". Consequently, they could be subject to lower security controls, leading the flow tables in an inconsistent state. An implementation of the flow visor can be suitable for avoiding these potential threats.

4. The **scalability** of the network also depends on the controller, that potentially can become a "bottleneck". In case too many packets reach the controller, performance issues can occur in the network. It follows that it is important to take into account the distribution of the control plane, presented in Section 2.3.2, for avoiding these undesirable problems.

5. The **performance** of the network can also be related to the control model adopted. Since the flow table size is limited, the management of a very large number of flows is still a strong challenge. However, a well designed network could reduce performance issues through a proactive approach. In fact, as previously analyzed in Section 2.3.1, the proactive approach reaches better performance than the reactive mode because it limits the amount of messages exchanged among the controller and the switches.

The aspects described above have been taken into consideration by the community researchers and represent the future challenges for the SDNs.

The last section of this chapter presents one of the most widespread controller, Floodlight, suitable for dealing with SDNs.

2.5 Floodlight SDN Controller

Many SDN controllers have been developed since the introduction of SDN [18]. However, one of the most widespread OpenFlow controller is Floodlight [32]. Floodlight is a Java-based open source software based on the Beacon controller implementation developed at the Stanford University [33], that works with physical and virtual OpenFlow switches. The last release of Floodlight is the version 0.90 and, the following section explains the architecture and the main characteristics of the Floodlight.

2.5.1 Architecture

The Floodlight controller realizes a set of common functionalities to control and inquire an OpenFlow network. As shown in Figure 2.15, the Floodlight architecture is modular and it is clear the relationship among the controller, the applications built as Java modules compiled with Floodlight, and the applications built over the Floodlight REST API.

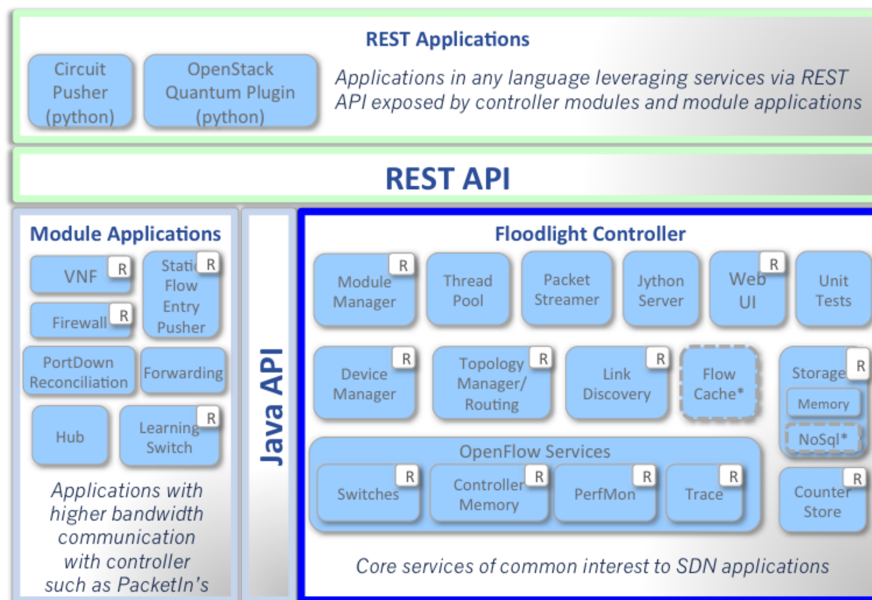


Figure 2.15: Floodlight architecture [5]

As depicted in Figure 2.15, the main modules related to the controller core are the following:

1. The **Link Discovery** is responsible for discovering and maintaining the status of links in the OpenFlow network by means of the Link Layer Discovery Protocol (LLDP) and the Broadcast Domain Discovery Protocol (BDDP). Furthermore, the controller periodically commands through the Link Discovery module the switches to flood LLDP and BDDP packets to all their ports. A discovery protocol packet typically contains the Data Path Identifier (DPID) of the sender switch together with the port of the switch which the message originates from. Thus, the controller discovers the direct and indirect connections between the switches. Specifically, a direct link will be established if a LLDP is sent out to one port and the same LLDP is received on another port. It follows that the ports are directly connected. On the opposite side, thanks to the BDDP protocol, it is possible to find indirect connections by means of a flooding approach. In this case, if a BDDP packet is sent out to a port and received on another, a broadcast link is created. It means that there is another layer 2 switch that is not under the control of the controller between these two ports. Further, the controller is also able to verify the connection liveliness with periodical checks.

2. The **Topology Manager** is designated to maintain the topology information updated for the controller, as well as to find routes in the network. The topologies are computed by means of the link information retrieved from the Link Discovery service. All the information about the current topology is stored in an immutable data structure called the topology instance. Thus, the controller can determine the shortest path (using the Dijkstra algorithm) from a source switch port to a destination switch port, equipped with the knowledge of the network topology.

3. The **Device Manager** tracks devices, through the *packet-in* requests, connected to the network and defines the destination device for a new flow. By default, the module uses the MAC address and the VLAN to identify uniquely a device. Furthermore, the Device Manager learns about other important pieces of information such as IP addresses as well permits to know the device attachment points. In that case, if a *packet-in* is received on a switch, an attachment point will be created for that device. Finally,

this component also ages out attachment points, IPs, and devices themselves. It exploits the last seen timestamps to keep control of the aging process.

4. The **Storage** service realizes a storage based on a NoSQL style. It also supports a notification mechanism for changes in the database.

Furthermore, as also depicted in Figure 2.15, some other important application modules are:

1. The **Forwarding** component, loaded by default, makes the forwarding of packets between two devices possible, realizing a reactive forwarding approach. It is designed to work in networks that contain both switches OpenFlow-enabled and "regular" switches. However, it works correctly only if loops among groups of switches OpenFlow-enabled and groups of the "regular" switches do not exist. Moreover, the Forwarding module individually handles each packet, and severely limits the performance.

2. The **Learning Switch** implements a behavior similar to the "regular" L2 learning switch. Thus, the module detects and learns about new devices based on their MAC addresses. When the controller detects a new flow, the Learning Switch identifies the input and the output switches, as well as all other switches on the shortest path between the start and the end point. Once a path has been found, the module installs the appropriate OpenFlow rules for handling the new flows on all participating switches.

3. The **Static Flow Entry Pusher** allows a user to manually insert flows into an OpenFlow-enabled switch through REST API. Thus, in practice, this module realizes a proactive forwarding approach.

4. The **PortDown Reconciliation**, when a port or a link goes down, reconciles flows across a network. Following a link discovery update, the module discovers and deletes flows directing traffic towards the downed port. Then, it re-evaluates the path that the traffic has to take according to the updated topology. It follows that if this module is not enabled, the persistent traffic continues to be routed to a downed port.

5. The **Firewall** module implements a generic reactive firewall software that enforces the Access Control List (ACL) rules on the OpenFlow-enabled switches. Specifically, the ACL rules are sets of conditions that allow or deny a traffic flow at its ingress switch. Furthermore, the first packet(s) of a traffic flow is matched against the set of the existing firewall rules through the monitoring of each *packet-in* triggered by. The rules are also sorted based on assigned priorities and are matched against the *packet-in* header fields.

On top of both the controller core modules and the application modules is placed the **REST API** layer. This tier allows the modules to expose their REST APIs over HTTP realizing a flexible architecture.

Floodlight supports both the forwarding approaches (argued in Section 2.3.1) by means of the Forwarding and the Static Flow Entry Pusher modules described above. The controller has also a separate module system to load the modules. It is possible to enable/disable the loading of some modules, and consequently the behavior of the controller, simply by means of an editable *properties* file. This approach permits to swap out implementations of modules without modifying modules that depend upon them. Hence, the modular architecture and the REST API service supply an adaptable and extendible framework for dealing with SDNs, enforcing the code modularity.

However, some limitations are still present. Firstly, at the moment, Floodlight supports only the OpenFlow 1.0 specification and the timeline for support of 1.2/1.3 is currently unknown. Secondly, since Floodlight stores the data in the volatile memory, all states will be lost when the controller is turned off. Finally, there is no isolation of data enforced. It means that if a module creates a table, another module could potentially overwrite this data creating inconsistency.

The next chapter discusses some important related works in the SDN field and an interesting network emulator, Mininet, suitable for dealing with SDNs.

Chapter 3

Dealing with SDN

The following chapter provides an overview of the main related works about SDN and OpenFlow. Specifically, we present several important papers that deal with the Quality of Service (QoS) and the Quality of Experience (QoE) in SDN. A few interesting works related to the network monitoring and resource management are also described in the next section. Finally, we discuss a widespread network emulator, called Mininet, often used for dealing with SDN.

3.1 Related work

Recently, SDN in conjunction with OpenFlow have attracted the attention of both academia and industry. They allows the software-based controller to manage the forwarding information in the switches. Furthermore, the switches OpenFlow-enabled become "simple" forwarders that route the network traffic according to the rules set by the controller. Hence, SDN enables the researchers to test new ideas (e.g., novel algorithms, different protocols, or customized architecture) in a production environment by decoupling the control and data planes. In addition, the network status monitoring is of utmost importance for managing the resources and making an enhanced QoS possible. The next section details some interesting works related to the network monitoring in SDN.

3.1.1 How to Monitor Network Parameters with OpenFlow

Even if many papers focus on the problem of network management, monitoring and control to improve the QoS perceived by the customers, only some provide solutions for measuring network performance, e.g., latency, throughput, and packet loss.

The authors in [34] implement *OpenNetMon* to monitor latency, throughput and packet loss in OpenFlow networks. *OpenNetMon* is a POX OpenFlow controller module that monitors per-flow QoS metric. This application allows to determine on-line whether the end-to-end QoS parameters are satisfactory. Then, the application sends the data relative to throughput, delay and packet loss to the controllers for Traffic Engineering (TE) purposes. The throughput and packet loss are obtained from polling flow source and destination switches. The *OpenNetMon* regularly sends polling messages to the switches to retrieve Flow Statistics. Then, it receives the amount of bytes sent and the duration of each connection from which is possible to measure the throughput. Notice that the polling is done for every path between every node pair to be monitored. Moreover, the polling is adaptively changed based on new flows arrivals and changes. Per-flow packet loss is calculated by subtracting the increase of the packet counter of the source switch with that of the destination switch. While, the latency is harder to measure. It is derived by injecting probe packets into the switch data planes on the same path of each flow. Then, the controller can measure the delay by computing the difference between the packets departure and arrival times, subtracting with the estimated switch-to-controller (Round Trip Time) RTT, using the following formula: $t_{delay} = (t_{arrival} - t_{sent} - 1/2(RTT_{s1} + RTT_{s2}))$. This paper presents also some implementation specific details that are useful for the research community [34]. The main drawback is that the authors do not suggest how the controller can find the new paths based on the real-time data.

Another way to measure network parameters is to use an analytical model. Indeed, [35] presents such a model to evaluate the forwarding speed and blocking probability of an OpenFlow network. The model is based on the queuing theory and then tested on an OpenFlow switch and controller by means of a simulation: authors assume that the OpenFlow architecture can be viewed as

a feedback oriented queuing system model divided into two systems. The first is a forwarding queuing system of a type M/GI/1 with a M/M/1 Markovian server. While the second is a feedback queuing system of the delay-loss type $M/GI/1 - S$ with a $M/M/1 - S$ server. Then, the controller is modeled as the feedback $M/M/1-S$ queuing system. The following Figure 3.1 shows the complete model of the forward and feedback queuing systems. The performance

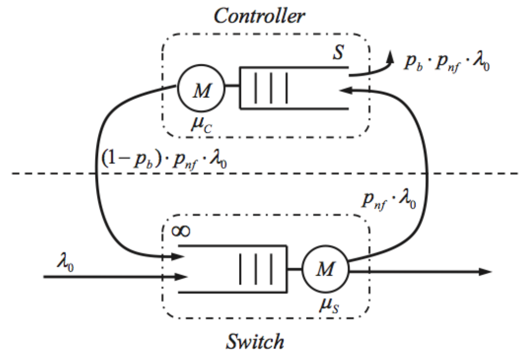


Figure 3.1: The model of an OpenFlow switch

parameters are the total sojourn time of a packet through the system and the probability of dropping a packet. The results show that the sojourn time depends on the processing speed of the controller. Thus, it can be deduced that the controller performance limits the installation of new flows on the network. The main advantage of using an analytical model is that it can provide results in less time than using a simulation. However, the paper does not exploit the fact that the measurements can be used to improve the QoS.

One of the most important metric in network evaluation is network latency. The paper [36] suggests a way to measure the link latency from an OpenFlow controller. The idea is to send a special packet on the link from the controller and back and measuring the amount of time it requires to do so. The Figure 3.2 depicts the functional architecture for the latency monitoring. To do so, the controller c sends a request to a switch s_1 to being forwarded through a particular port. The second switch s_2 sends back the packet to the controller. Then, it can be deduced from the received time and the Timestamp how long the packet took to complete its trip. Finally, it is necessary to subtract the time spent in the links and in the switches. Thus, the link latency is computed as follows:

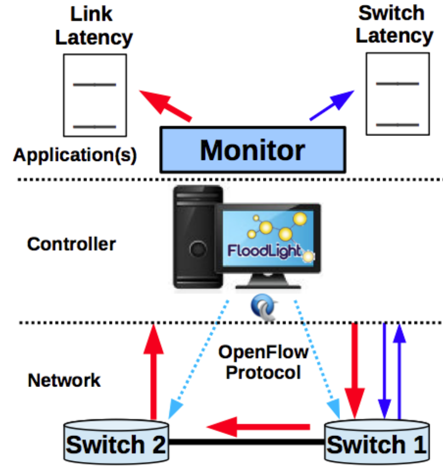


Figure 3.2: The functional architecture of the latency monitoring application

$Latency(s_1, s_2) = T_{total} - \frac{RTT_{c \to s_1}}{2} - \frac{RTT_{c \to s_2}}{2} - \mathbf{C}$. Where **C** is the calibration value of the controller. This solution is implemented on an OpenFlow testbed showing the effectiveness. The paper only miss to use latency measurements for providing new network capabilities.

The next section presents some interesting works focused on improving the QoS and the QoE in SDN.

3.1.2 SDN to improve Quality of Service and Quality of Experience

SDN decouples control and forwarding layers of routing, as an efficient way to provide new QoS architectures over OpenFlow networks.

A solution for scalable video streaming over Open Flow network is presented in [37]. The authors suggest and solve two optimization problems for controller design. The solutions provide new routing path for lossless and lossy QoS flows. The results show that the average quality of video stream is improved by 14% by rerouting the base layer, and it can also be improved by another 6.5% by rerouting the enhancement layer.

OpenQoS [38] is a proposal for multimedia delivery with end-to-end QoS support. The routes of the multimedia traffic are optimized dynamically to respect the QoS requirements, such as packet loss and latency. The paper sug-

gests a dynamic QoS routing for QoS flows while other flows remain on their shortest path. The results show that the solution can guarantee seamless video delivery with little or no video artifacts experienced by the end users. Moreover, they show that the service does not have adverse effect on other types of traffic.

Moreover, the paper [39] also design a QoS architecture based on OpenFlow, suggesting an optimization framework. The scheme is exploited for enhancing QoS-enabled streaming of scalable encoded videos with two level of QoS: the base layer and the enhancement layer. The problem is posed as a Constrained Shortest Path (CSP) problem and then solved. The results show that there is a significant improvement in the quality of scalable video streaming. The rerouting of base layer video only is sufficient to get important improvement over streaming scalable or non-scalable video with best effort quality. The dynamic rerouting of the enhancement layer is useful when the network congestion is high and the base layer bit rate is low.

Another optimization problem for QoS flow routing is suggested in [40]. The paper describes an architecture to support QoS flows in an OpenFlow environment with a centralized controller. The problem formulation provides routes for QoS flows that are translated into flow tables for QoS traffic. Moreover, the authors setup an environment of the controller, to receive a QoS contract, to configure forwarders for QoS flows, to monitor the network and to switch-over to an alternate route under congestion or failure.

The topic of QoE is considered in many works, [41] proposes an OpenFlow-assisted QoE Fairness Framework (QFF) to improve the QoE of multiple clients. By exploiting the SDNs and OpenFlow, the paper suggests a way to optimize the QoE for all video streaming devices in a network, considering also the device and network requirements. The main characteristic of QFF is to dynamically adapt the video flow in order to guarantee network-wide QoE fairness. The QFF has been implemented and evaluated by means of MPEG-DASH and OpenFlow. Exploiting OpenFlow, QFF monitors the status of all DASH video application in the network. Then, QFF can accordingly take some decisions about how to allocate the network resources. The following Figure 3.3 depicts an high-level view of the QFF.

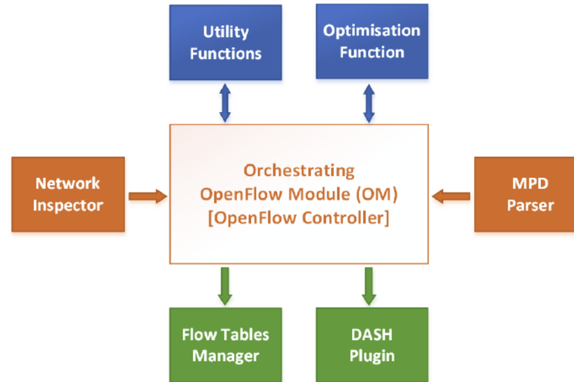


Figure 3.3: OpenFlow-assisted QoE Fairness Framework

The architecture is composed by an OpenFlow Module (OM), that runs on the OpenFlow controller of the network and is responsible of managing the main QFF functionalities. The inputs to the OM are the Network Inspector and the MPD Parser, that provide the network and clients status. While, the outputs are the Flow Tables Manger and the DASH Plug-in, that ensure that the decisions are propagated to the network. Finally, there are the Utility Functions and the Optimization Function that dynamically optimize the QoE fairness interacting with the OM. The QFF is evaluated in a home networking scenario. The results show that QFF provides network stability and optimizes video streaming QoE among different devices in a network. The main drawback of this approach, as stated by the authors, is that the Utility Function depends on the characteristic of the test video source. The Utility Function should be more flexible and adaptive to various video contents, for instance to add the possibility to measure some metrics related to the improvement of QoE.

Then, the paper [42] suggest a scheme for HTTP video quality optimization exploiting SDN. It proposes a HTTP video content delivery scheme in the SDN scenario. The SDN paradigm increases the network intelligence and scalability and, also allows user to obtain video resource from the nearest storage router providing better video quality and user QoE. Then, the authors test by means of five experiments the video quality in presence of different round trip time delay depending on round trip delay results of HTTP video access in SDN and actual network. Finally, the user QoE of the video is computed and compared

for all the experiments. The results show that the video quality is substantially improved with SDN scheme and the QoE achieve more than a good level.

The paper [43] presents QoSFlow to improve the flexibility of QoS control. The aim of QoSFlow is to allow the control of multiple packet schedulers. Thus, it brings the Linux traffic control into the Open Flow networks. The authors analyze the performance in terms of response time of the packet scheduler operations running on data path level, maximum bandwidth capacity, hardware resource utilization rate, bandwidth isolation and QoE.

We also consider the proposal of [44], where the QoE measurement is investigated. The authors suggest three approaches: subjective, objective and hybrid approach. The first one depends on the quality score given by humans according to their point of view and perception but it is very expensive in terms of manpower. The objective approach is based on algorithms and metrics already defined in the literature, the most common metric is the Peak Signal-to-Noise Ratio (PSNR). However, the PSNR is not a real-time mechanism because it requires to reconstruct the image at the receiver before computing the PSNR. Finally, the hybrid approach works in real-time and it is based on statistic learning. The latter gains advantages from the other approaches because it is more accurate. It is very important to define a method to measure the QoE for managing network resources and provide efficient services.

While, the paper [45] suggests SWAN (Software-driven WAN), an efficient system that allows inter-Data Center WANs to carry more traffic. In particular, it globally coordinates the sending rates of different services and, centrally allocates network paths. Thus, SWAN dynamically chooses how much traffic each service can send and manages the network data plane to carry the traffic. The main problem is the congestion that can be caused by the frequent updates to the network data plane. However, the authors suggest to leave a "scratch" capacity at each link to enable a congestion-free plan to update the data plane. Then, they develop SWAN and test it finding that their system can carry 60% more traffic than the current practice with no update overhead and, showing that the changes to the network paths are quick.

Unfortunately, often it is expensive and not simple to configure and test an SDN environment composed of real devices, especially if the network is very

large. However, a few SDN network emulators have been developed to make these tasks simple. One of the most widespread SDN emulators is Mininet and it is presented in the next section.

3.2 Mininet Network Emulator

Mininet is an open source project for rapidly prototyping large networks on the constrained resources of a single device e.g., a laptop [16]. More specifically, it is a network emulator that permits to deal with SDN networks. This emulated environment can be executed in a Virtual Machine (VM) (e.g., VirtualBox or VMware) or directly on a native Linux distribution. Mininet is able to run a collection of virtual end-hosts, switches, routers, and links on a single Linux kernel. Furthermore, it allows us to create many custom topologies and emulate some link parameters like a real Ethernet interface, e.g., link speed, packet loss, and delay. Thus, by means of Mininet, it is possible to create an emulated network that reproduces a hardware network, or a hardware network that resembles a Mininet network, and run the same binary code and applications on either platform. Specifically, it is possible to create a customized network by using Python APIs or directly build some simple network topologies through the Command-Line Interface (CLI). The CLI also provides some useful commands suitable for retrieving topology information, debugging the network, or testing the connectivity. Mininet can also work with several different SDN controllers, e.g., Floodlight, as described in Chapter 2.

However, some restrictions exist. Since Mininet permits to emulate a large network in a single device, it follows that the performance of the emulator directly depends on the available resources supplied by the host. On the one hand, this feature allows the researchers to rapidly test new algorithms and protocols in a *built-in* environment. On the other hand, an experiment conducted in a device with limited computational resources could alter the network behavior. In fact, since Mininet is an emulator which does not have a strong notion of virtual time (unlike a simulator), it does not allows us to correctly emulate high link speed. Currently, Mininet does not perform a Network Address Translation (NAT) out of the box, thus the nodes cannot be di-

rectly connected to the Internet. This aspect is further explained in Section 5.4 of Chapter 5. Moreover, all the Mininet hosts share the host file system and the Process ID (PID) space. It follows that the Mininet processes are not completely isolated, consequently, they could interfere with some other daemon processes that are running in the same space.

The next Chapter describes our architecture to enhance QoS in SDNs in depth and presents the mathematical model based on the Multi-Commodity Flow Problem and the Constrained Shortest Path Problem.

Chapter 4

The QoS-aware Mathematical Model

The QoS is of paramount importance in a SDN scenario, as explained in the previous chapters. Furthermore, it is very important to take into account the relation between the type of applications (e.g., real-time vs. file transfer data) and the network parameters as available bandwidth, packet loss, delay, and jitter. In Table 4.1 are summarized few examples of quality requirements of a set of relevant applications. Specifically, the loss tolerance indication is

APPLICATION	CHARACTERISTICS	LOSS TOL.	DELAY TOL.	JITTER TOL.
Network control	Mostly inelastic	Low	Low	Yes
Telephony	Inelastic, low rate	Very low	Very low	Very low
Signaling	Short packets, delay critical	Low	Low	Yes
Multimedia conferencing	Reacts to loss	Low medium	Very low	Low
Real-time interactive	Inelastic, variable bit rate	Low	Very low	Low
Multimedia streaming	Elastic, variable bit rate	Low medium	Medium	Yes
Broadcast video	Inelastic, non variable bit rate	Very low	Medium	Low
Low latency data	Elastic, variable bit rate	Low	Low medium	Yes
OAM	Both elastic and inelastic	Low	Medium	Yes
High throughput data	Elastic	Low	Medium high	Yes
Low-priority data	Elastic	High	High	Yes

Table 4.1: Application quality requirements

related to the application resilience in case of packet dropping (that typically occurs when the network is congested). Moreover, the delay tolerance refers to the application robustness when the packet delivery is prone to the latency.

Finally, the jitter tolerance indicates the application strength in the event of variation in packet delivery delay. Thus, the application quality requirements give us a general idea of the correlation between the application type and the network parameters.

However, we are interested in finding a map between the network parameters described above and a general reference scale. This mapping is useful to have a connection between the network status and the QoS from the user point of view. To make it possible, we have taken into account the well known Mean Opinion Score (MOS) model, that is a metric (conceived in telephony networks) to measure the users satisfaction level. It has been standardized by the Telecommunication Standardization Sector (ITU-T) [46] group in the ITU-T P.800 standard [8] and it is composed of five different levels (fractional values are admitted), as sketched in Table 4.2. It follows that a MOS score level equal

MOS	QUALITY	IMPAIRMENT
5	Excellent	Imperceptible
4	Good	Perceptible but not annoying
3	Fair	Slightly annoying
2	Poor	Annoying
1	Bad	Very annoying

Table 4.2: Mean Opinion Score levels [8]

to 5 corresponds to completely satisfied users, while a MOS level equal to 1 corresponds to a population of unsatisfied users. At the beginning, the model was exploited to retrieve the average MOS value of classic PCM fixed telephony, that amounts to 4.3 (used as a reference to judge the MOS value of a telephone system).

Moreover, the ITU-T group has also defined, in the recommendation G.114 [9], some delay ranges in the telephone field. The recommendation document suggests that the total delay (called "one-way" or "mouth-to-ear") in a voice connectivity should be less than $400ms$, as indicated in Table 4.3. However, the same recommendation specifies that the total delay in a VoIP call should be less than $100 - 150ms$.

Nowadays, a lot of papers in different fields refer to this classification. Furthermore, the MOS score level is often used as a reference point for defining

DELAY(ms)	QUALITY
< 150	Acceptable
150 - 400	Acceptable but not desirable
> 400	Unacceptable

Table 4.3: Voice connectivity total delay (one-way) [9]

some parameters range. It follows that for each type of application, there exist many different mappings between the MOS classification and the network parameters depending on the amount of packet loss, delay, jitter, or a combination of few of them. For instance, the delay value is very important in network gaming, as depicted in Table 4.4. As detailed in the paper [10], the quality per-

DELAY(ms)	MOS	QUALITY
< 50	5	Excellent
30 - 100	4	Good
100 - 150	3	Fair
150 - 200	2	Poor
> 200	1	Bad

Table 4.4: End-To-End delay in the network gaming [10]

ceived from the user's point of view is intolerable if the delay is greater then $200ms$. However, if the delay is less then $100ms$, the user experience becomes good or excellent.

Even the authors of the paper [6] have taken into account the connection between the packet loss and the quality defined by means of the MOS score, as depicted in Figure 4.1. Specifically, it has been evaluated the packet loss that can occur in different content types, such as news, film trailer, football, and music video. As illustrate in Figure 4.1, the multimedia services have various packet loss range depending on the different content types (with bit rate values from 24 kbps to 256 kbps). There exists a strong correlation between the packet loss and the quality from the user's point of view, as clearly illustrated in Figure 4.1. Specifically, when the amount of packet loss is approximately equal to 5%, the QoS perceived by the users is poor (according to the MOS model).

Even CISCO has brought to light [11] some reference point about the QoS needed for VoIP, Interactive-Video (IP Videoconferencing) and Streaming-Video, as summarized in Table 4.5.

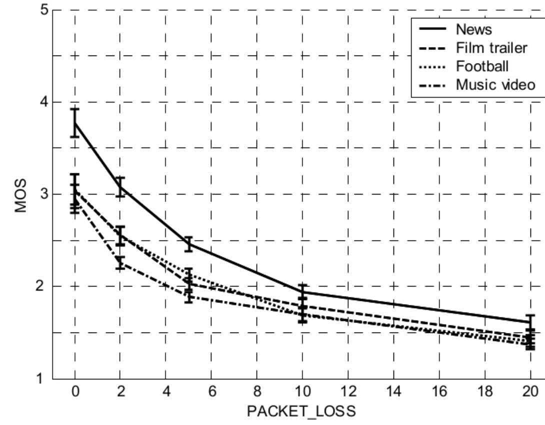


Figure 4.1: Streaming quality and packet loss rate (in percentage) for different content types [6]

APPLICATION	DELAY	PACKET LOSS	JITTER	BANDWIDTH
VoIP	$\leqslant$ 150 ms *	$\leqslant$1 %	< 30 ms *	21-320 kbps
Interactive-Video	$\leqslant$ 150 ms *	$\leqslant$1 %	$\leqslant$30 ms *	n.a.
Streaming-Video	$\leqslant$ 4-5 s *	$\leqslant$ 5 %	n.a.	n.a.

* "one-way" value

Table 4.5: QoS requirements of VoIP, Interactive-Video, and Streaming-Video [11]

We provide a few comments about the values indicated in Table 4.5:

1. VoIP is not tolerant of packet loss (ideally, there should be no packet loss for VoIP) especially if it uses a compressed codec.

2. The maximum video streaming delay depends on video application buffering capabilities, thus it can be less than the value indicated.

3. The acceptable jitter is not a fundamental parameter in video streaming applications, thus there are no significant jitter requirements.

4. The streaming video bandwidth requirement is related to the encoding format and the rate of the video stream, hence it is not a fixed value.

Furthermore, both in multimedia and telephony applications, the QoS can also depend on the codec type used, the error recovery mechanism, and the content bit rates. It follows that it is not simple to have an accurate measure of the quality from the user's point of view because it is necessary to take into account a lot of different variables. In our proposal, we have primarily taken

into account the available bandwidth, the packet loss, and the delay. Nevertheless, our mathematical model can be easily extended to satisfy more constraints related to other network or service parameters.

Moreover, the purpose of this chapter is to describe in depth our proposed enhanced QoS architecture for SDN networks. To reach this goal, it is very important to continuously supervise the network conditions, make the right decisions, and accordingly manage the devices. Then, the subsequent section explains our mathematical model, based on the multi-criteria approach, that implements the "brain" of the network. Finally, we have defined different thresholds of QoS and we have put in relation the values supplied by the mathematical model with the MOS model described above.

4.1 Enhanced QoS Architecture

This section describes our proposed architecture, indicated with a red dashed line in Figure 4.2. Our proposal aims to reach an enhanced QoS in SDNs by means of several functionalities. On the one hand, the architecture continuously retrieves the network parameters to improve the awareness of the network status. On the other hand, the presented solution calculates the best path according to the specific flow constrains. It reaches this goal by exploiting the multi-criteria approach, as explained in Section 4.2. Specifically, the architecture is composed of the following logical modules:

a. The **Network Topology Mapper** incessantly retrieves information about the real network topology (links and nodes) and maps it into a structure, i.e., the **Link Connection**. The mapping of these information is useful for the **Static Path Inserter** and the **Path Finder** module for running algorithms on the structure and finding offline the best path.

b. The **Static Path Inserter** inserts the flow entries into the switches, through the Floodlight REST API, to configure the paths when a manual configuration is required. This static insertion is suitable for configuring the paths according to some external decisions, e.g., overloading some particular links instead of others or forcing specific paths in such a way to

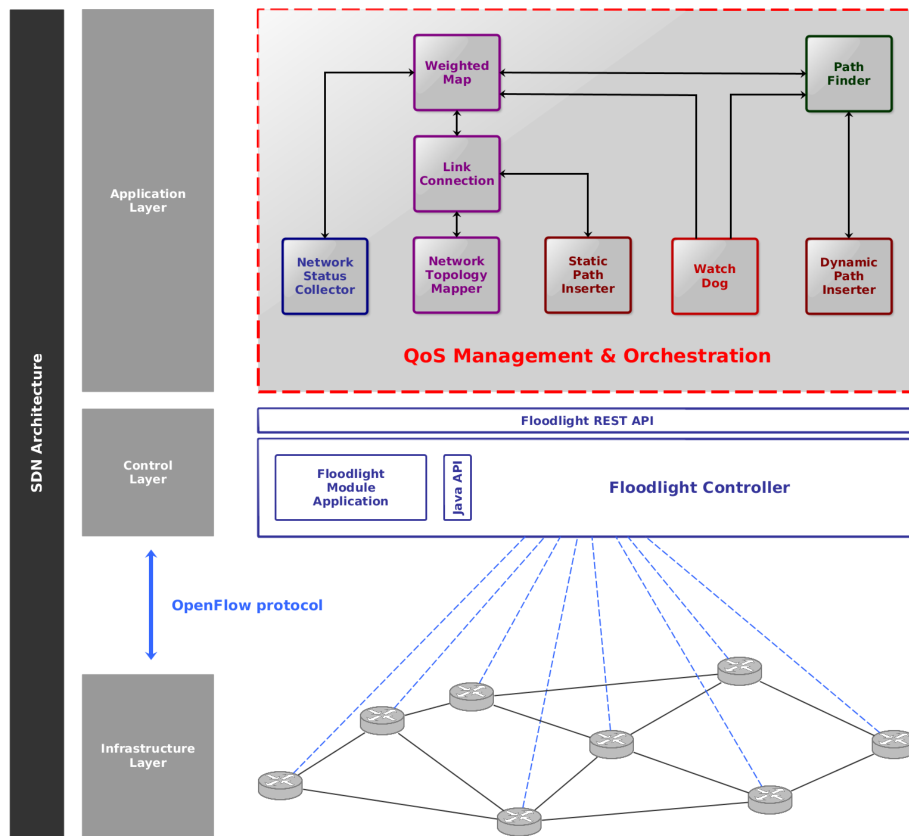


Figure 4.2: QoS Management & Orchestration architecture (red dashed line)

avoid some switches. The module uses also the **Link Connection** to get the correct ports involved in the path.

c. The **Network Status Collector** continuously collects information about the network status (e.g., the available bandwidth, the packet loss, and the latency of the network links). These information are stored in the **Weighted Map** structure to allow the **Path Finder** module to find the best path. The reliability of the measurements depends on the quality of the data stored in the switch OpenFlow-enabled. However, the current Floodlight version (that supports OpenFlow version 1.0) does not offer the possibility to retrieve these informations as a "built-in" feature. Thus, we have implemented these specific functionalities as described in Chapter 5.

d. The **Dynamic Path Inserter** injects the flow entries into the switches to

set the best path found by the **Path Finder** module. On the one hand, the module exploits the **Path Finder** for calculating the best path. On the other hand, the Dynamic Path Inserter uses the Floodlight REST API for deploying the rules into the devices. Thus, this component is the "actuator" of the architecture.

e. The **Path Finder** implements the mathematical model for finding the best path according to both the flow requirements and the network status. The model uses the multi-criteria approach, as detailed in Section 4.2. Specifically, the Path Finder exploits the information stored in the **Weighted Map** and decides which is the best path for each flow. This module supplies the new path to the **Dynamic Path Inserter** when required.

f. The **Watch Dog** module is responsible for the triggering of the path changing. It continuously analyzes the network information collected by the **Network Status Collector** and decides when it is necessary to trigger a path changing. In this case, the **Watch Dog** directly commands the **Path Finder** to retrieve the best path according to the network conditions stored into the **Weighted Map**. Consequently, the module uses the **Dynamic Path Inserter** for putting the flow entries into the switches involved in the new path.

g. The **QoS Management & Orchestration** is the proposed architecture core and uses all the modules for the network management. Specifically, this "wrapper" module has to dynamically analyze the network status, decide when it is necessary to redefine a new path for a specific flow, find which is the best path for that flow, and insert the rule in the switches (in the form of flow entry). Moreover, by using the **Dynamic Path Inserter**, we are able to achieve the "make-before-break" approach, described in the Chapter 2, and offer the best QoS as possible.

The modular design of the proposal in conjunction with the flexibility supplied by the Floodlight REST API interface, provides a flexible architecture that can be easily extended to operate with different controllers or to realize additional functionalities.

The next section describes in depth the main modules of the architecture. Specifically, it is presented the multi-criteria approach and the mathematical model, exploited by the architecture, to enhance the QoS in SDNs.

4.2 Multi-Criteria Approach

The **multi-criteria approach** is the core of our proposal and it allows us to assign a flows to a specific path taking into account both the link parameters (e.g., packet loss, latency, and available bandwidth) involved in the path and the flow requirements. To reach this goal, the proposed architecture has to continuously monitor the network parameters to adapt the resources by need. The architecture is also able to find the best path (or more than one best path in case of multiple flows) that can satisfy the flow necessities by means of our mathematical model. The mathematical model is based on two well known problems that have been addressed by Operations Research: the **Multi-Commodity Flow Problem** and the **Constrained Shortest Path Problem**, as further described below. The target of our architecture is to enhance the QoS in SDNs.

4.2.1 Multi-Commodity Flow Problem

The **Multi-Commodity Flow Problem** (MFP) is a particular network flow problem where multiple "commodities", i.e., different traffic flows (demands/services), should be sent from various sources to distinct destination nodes. In particular, we suppose to have a network of interconnected nodes where each link has a particular capacity, i.e., bandwidth, and a cost associated to it. Moreover, the flowing traffic along the links between each pair of nodes consumes an amount of bandwidth. The goal is to find the optimal set of routes through the network for each of those commodities with the minimum total flow cost. The constraint is that the total flow on a link should not exceed the link capacity.

Thus, we define the mathematical notation for the MFP. The network is represented by an oriented graph $G = (N, A)$, where N is the set of nodes and A is the set of arcs between each pair of nodes. The arcs E are bi-directional and they have associated the available bandwidth b_{ij} and the cost per unit of flow

c_{ij}. The set of different traffic flows to be routed on the graph is represented by K. For each flow, three parameters are given:

i. the source node s_k;

ii. the destination/terminal node t_k;

iii. the amount of flow f_k to be sent from the source node to the destination node.

Notice that we assume that there exists no pair of flows with the same origin and destination.

In the following, we provide the ILP formulation for the MFP problem. The optimization objective is to route all the flows in the network with the minimum cost.

Sets:

- Nodes: $n \in N$
- Arcs: $(i, j) \in A$
- Edges: $(i, j) \in A \cup (j, i) \in A$

Variables:

- $x_{ij}^k \geqslant 0$: amount of the flow corresponding to the service k routed on the link (i, j).

Parameters:

- $b_{ij} \geqslant 0$: available bandwidth on the link (i, j);
- $c_{ij} \geqslant 0$: cost of the link (i, j);
- $s_k \in N$: source of the flow k;
- $t_k \in N$: destination of the flow k;
- $f_k \geqslant 0$: amount of the flow k to be sent from the source to the destination.

Objective Function:

$$\min: \sum_{(i,j)\in A} \sum_{k\in K} c_{ij} x_{ij}^{k} \tag{4.1}$$

Constraints:

$$\sum_{(i,j)\in A} x_{ij}^{k} - \sum_{(j,i)\in A} x_{ji}^{k} = \begin{cases} f_k & \text{if} \quad i = s_k, \\ -f_k & \text{if} \quad i = t_k, \\ 0 & \text{if} \quad i \neq s_k, t_k \end{cases} \qquad \forall i \in N, \forall k \in K \tag{4.2}$$

$$\sum_{k\in K} x_{ij}^{k} \leqslant b_{ij} \quad \forall (i,j) \in A \tag{4.3}$$

$$x_{ij}^{k} \geqslant 0 \quad \forall (i,j) \in A, \forall k \in K \tag{4.4}$$

The objective function 4.1 represents the **cost minimization** that depends on the cost of the used links. The first constraint 4.2 is related to the **flow balancing**. It takes into account the well known **Flow Conservation Law** that explains that the total flow incoming into each vertex is equal to the total flow outgoing from the same vertex, with the exception of the source and the terminal. Equation 4.3 refers to the **arch capacity** constraint and imposes a limit on the **available bandwidth** of each link. Finally, the Equation 4.4 defines the **variables domain** and guarantees that the decision variable is positive. About the problem size, we notice that the number of variables is $|A||K|$ and the number of constraints is $|N||K| + |A|$. This problem is *NP-complete* [47].

4.2.2 Constrained Shortest Path Problem

The **Constrained Shortest Path** problem (CPS) is an extension of the **Shortest Path** (SP) problem and it calculates a shortest path fulfilling a set of constraints. In particular, we have a network of nodes where each link is defined by various parameters, e.g., in our case the delay and the packet loss. Moreover, each link has a cost associated to it. The goal is to find the optimal path between a source and a destination with the minimum cost after pruning those

links that violate a given set of constraints. In our case, the constraints are that the delay and the packet loss do not exceed the maximum acceptable values. However, a lot of constraints can be considered, e.g., the bandwidth per link, the number of links traversed, the number of included or excluded nodes, depending on the network.

Thus, we define the mathematical notation for the CSP. The network is represented by an oriented graph $G = (N, A)$, where N is the set of nodes and A is the set of arcs between each pair of nodes. The arcs E are bi-directional and they have associated the delay d_{ij}, the packet loss $p_{i,j}$, and the cost c_{ij}. The flow is routed on the graph from the source node s to the destination node t.

We define above the ILP formulation for the CSP problem. The optimization objective is to route a flow in the network along the shortest path.

Sets:

- Nodes: $n \in N$
- Arcs: $(i, j) \in A$
- Edges: $(i, j) \in A \cup (j, i) \in A$

Variables:

- $x_{ij} \in \{0, 1\}$: boolean variable that is 1 if the flow is routed on the link (i, j), 0 otherwise.

Parameters:

- $c_{ij} \geqslant 0$: cost of the link (i, j);
- $s \in N$: source of the flow;
- $t \in N$: destination of the flow;
- $P_{max} \geqslant 0$: maximum acceptable value for the packet loss;
- $p_{ij} \geqslant 0$: packet loss on the link (i, j);
- $D_{max} \geqslant 0$: maximum acceptable value for the delay;
- $d_{ij} \geqslant 0$: delay on the link (i, j);

Objective Function:

$$\min: \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{4.5}$$

Constraints:

$$\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = \begin{cases} 1 & \text{if} \quad i = s, \\ -1 & \text{if} \quad i = t, \\ 0 & \text{if} \quad i \neq s, t \end{cases} \qquad \forall i \in N \tag{4.6}$$

$$\sum_{(i,j)\in A} p_{ij} x_{ij} \leqslant P_{max} \tag{4.7}$$

$$\sum_{(i,j)\in A} d_{ij} x_{ij} \leqslant D_{max} \tag{4.8}$$

$$x_{ij} \in \{0, 1\} \quad \forall (i, j) \in A \tag{4.9}$$

The objective function 4.5 represents the **cost minimization** that depends on the cost of the used links. The first constraint 4.6 is related to the **flow balancing** as explained in Section 4.2.1. Equations 4.7 and 4.8 refer to the maximum acceptable value for the **packet loss** and the **delay**, respectively and P_{max}, D_{max} impose the limit. Finally, the Equation 4.9 defines the **variables domain** and guarantees that the decision variable is 0 or 1. We notice that, about the size of the problem, the number of variables is $|A|$ and the number of constraints is $|N|$. This problem is *NP-complete* [47].

4.2.3 Our Multi-Commodity Flow and Constrained Shortest Path Model

Our model takes into account the **MFP** and **CSP** problems, we call it **Multi-Commodity Flow and Constrained Shortest Path** (MCFCSP). The MCFCSP model permits to find for each service the related shortest path according to the given set of constraints. We suppose to have a network of nodes where each link has an associated delay, packet loss, and bandwidth. Moreover, each

link has a cost associated to it that is computed as the weighted sum of the delay and packet loss. Furthermore, we take into account multiple flows that correspond to different services and that should be sent from various sources to distinct destinations. Our goal is to find the optimal set of routes through the network for each of the commodities with the minimum flow cost subject to some constraints. Our constraints are the maximum acceptable delay and packet loss, and the available bandwidth on the links.

Thus, we define the mathematical notation for the MCFCSP. The network is represented by an oriented graph $G = (N, A)$, where N is the set of nodes and A is the set of arcs between each pair of nodes. The arcs E are bi-directional and they have associated the available bandwidth b_{ij}, the delay d_{ij}, the packet loss $p_{i,j}$, and the cost per unit of flow c_{ij}. Specifically, we compute the cost c_{ij} as follows:

$$c_{ij} = \alpha d_{ij} + \beta p_{ij} \quad \forall (i, j) \in A \tag{4.10}$$

Where α and β are the scale factors. This computation allows to weight the cost based on the importance of the delay and the packet loss for a particular flow. Thus, we can manage these parameters according to the requirements of the type of service. For example, the Interactive Multimedia Applications have strict end-to-end delay requirements, so we can put $\alpha = 1$ and $\beta = 0$ in order to take into account only the delay. Otherwise, a Medical Data Applications do not allow high value of packet loss. The set of different traffic flow to be routed on the graph is represented by K. For each flow, five parameters are given:

i. the source node s_k;

ii. the destination/terminal node t_k;

iii. the amount of flow f_k to be sent from the source node to the destination node;

ix. P^k_{max} the acceptable value of the packet loss for each service;

x. D^k_{max} the acceptable value of the delay for each service;

In the following, we also provide the ILP formulation for the MCFCSP problem. The optimization objective is to route all the flows in the network along the shortest path, with the minimum cost.

Sets:

- Nodes: $n \in N$
- Arcs: $(i, j) \in A$
- Edges: $(i, j) \in A \cup (j, i) \in A$

Variables:

- $x_{ij}^k \geqslant 0$: amount of the flow corresponding to the service k routed on the link (i, j).

Parameters:

- $b_{ij} \geqslant 0$: available bandwidth on the link (i, j);
- $c_{ij} \geqslant 0$: cost of the link (i, j), computed as $\alpha d_{ij} + \beta p_{ij}$;
- $\alpha \geqslant 0$: scale factor for the delay;
- $\beta \geqslant 0$: scale factor for the packet loss;
- $s_k \in N$: source of the flow k;
- $t_k \in N$: destination of the flow k;
- $f_k \geqslant 0$: amount of the flow k to be sent from the source to the destination.
- $P_{max}^k \geqslant 0$: maximum acceptable value for the packet loss;
- $p_{ij} \geqslant 0$: packet loss on the link (i, j);
- $D_{max}^k \geqslant 0$: maximum acceptable value for the delay;
- $d_{ij} \geqslant 0$: delay on the link (i, j);
- $B^k \geqslant 0$: bandwidth required by the service k;

Objective Function:

$$\min: \sum_{(i,j) \in A} \sum_{k \in K} c_{ij} x_{ij}^k \tag{4.11}$$

Constraints:

$$\sum_{(i,j)\in A} x_{ij}^k - \sum_{(j,i)\in A} x_{ji}^k = \begin{cases} f_k & \text{if} \quad i = s_k, \\ -f_k & \text{if} \quad i = t_k, \\ 0 & \text{if} \quad i \neq s_k, t_k \end{cases} \qquad \forall i \in N, \forall k \in K \tag{4.12}$$

$$\sum_{(i,j)\in A} p_{ij} x_{ij}^k \leqslant P_{max}^k \quad \forall k \in K \tag{4.13}$$

$$\sum_{(i,j)\in A} d_{ij} x_{ij}^k \leqslant D_{max}^k \quad \forall k \in K \tag{4.14}$$

$$\sum_{k\in K} B^k x_{ij}^k \leqslant b_{ij} \quad \forall (i,j) \in A \tag{4.15}$$

$$x_{ij}^k \in \{0,1\} \quad \forall (i,j) \in A, \forall k \in K \tag{4.16}$$

The objective function 4.11 represents the **cost minimization** that depends on the delay and packet loss of the used links. The first constraint 4.12 is related to the **flow balancing** as explained in Section 4.2.1. Equations 4.13 and 4.14 refer to the maximum acceptable value for the **packet loss** and the **delay**, respectively and P_{max}^k, D_{max}^k impose the limit for each service k. Equation 4.15 refers to the **arch capacity** constraint and imposes a limit on the **available bandwidth** of each link considering all the flows k. Finally, the Equation 4.16 defines the **variables domain** and guarantees that the decision variable is 0 or 1. About the problem size, we notice that the number of variables is $|A||K|$ and the number of constraints is $|N||K| + |A| + |K|$. This problem is *NP-complete* [47].

The mathematical model also allows us to define the maximum acceptable value for the packet loss and delay, respectively indicated by P_{max} and D_{max}, for each k-th commodity flow. The maximum acceptable values are fundamental keys to respect specific flow type requirements, according to Table 4.1, Table 4.5, and Figure 4.1. We can also exploit the α and β scale factors of Equation 4.10 to dynamically tweak the weighted cost in accordance with the importance of the delay and the packet loss for a particular flow. It follows that, on the one hand, we can assign a maximum acceptable value of packet loss and

delay and, consequently, obtain both the path that satisfy all the constraints and the correlated minimum cost. On the other hand, we can modify the maximum acceptable values and, incidentally, verify the cost minimization function for guaranteeing a specific QoS level, according to the Table 5.1.

The next chapter presents the implementation and the experimental results taking into account the presented architecture and the mathematical model defined above. Chapter 5 also discusses the correlation between the QoS level (using the MOS model as a reference for the QoS) and the cost minimization function that we found during the experiments.

Chapter 5

Implementation and Experimental Results

This chapter proposes the implementation of our novel architecture and its core that is embodied in the MCFCSP model, based on the multi-criteria approach described in Section 4.2. The mathematical model is realized as an external module on top of the Floodlight controller, presented in Section 2.5. Furthermore, the module is implemented in Java and it uses *AMPL* (A Mathematical Programming Language) [48] pairs with the CPLEX solver [49] to solve the ILP problem related to the path finding. Specifically, *AMPL* is an algebraic modeling language for describing high-complexity problems, e.g., optimization and scheduling problems, while CPLEX is the optimization engine, developed by IBM, for solving ILP problems. Hence, in practice, through our MCFCSP mathematical model and the interface with the solver, we are able to continuously check if a new best path exists. If a better solution is found, the module can decide to insert the flow entries into the switches involved with the new path. To make the testing of our mathematical model possible, we developed a network topology in the Mininet network emulator, presented in Section 3.2. The testing codes allows us to compare different solutions given by the MCFCSP model and evaluate the QoS.

Furthermore, in a real packet-switched network, a few delay times can occur due to queuing, processing, transmission, and propagation, as depicted in

Figure 5.1.

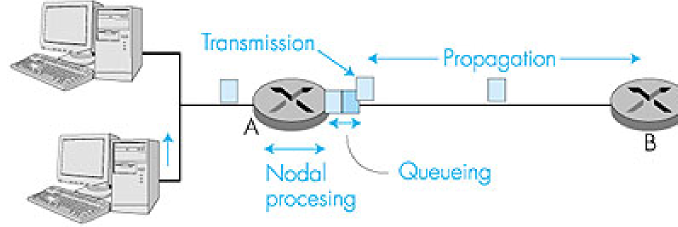


Figure 5.1: Delay in a real packet-switched network [7]

Hence, we can calculate the total delay as follows:

$$d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop} \tag{5.1}$$

The d_{nodal} delay in Equation 5.1 is the latency at a specific router and it is composed of the following different delays [7]:

1. d_{proc} is the delay time to process the packet and, in high-speed routers, is typically on the order of μs (microseconds) or less. However, in a SDN network, we have to take into account the time for the processing by the external Controller (in general only for the first packet flow).

2. d_{queue} is the amount of time that a packet has to wait for the transmission along the link. The queuing delay of a specific packet depends on the number of other packets in the queue, thus, the delay of a given packet can vary significantly from packet to packet. On the one hand, if the queue is empty and no other packet is currently being transmitted, the packet queuing delay is approximately zero. On the other hand, if the traffic is heavy and many other packets are also waiting to be transmitted, the queuing delay will be long. Queuing delays can be on the order of ms (milliseconds) to μs in practice.

3. d_{trans} is the transmission delay, also called the "store-and-forward" delay. It represents the amount of time required to transmit all the packet bits along the link. For instance, if L is the length of the packet (in bits), R is the transmission rate of the link (typically in Mbps), it follows that the transmission delay is L/R. Transmission delays are typically on the order of ms or less in practice.

4. d_{prop} is the delay due to the propagation speed of the link. The propagation speed depends on the physical link (e.g., twisted-pair copper wire or multi-mode fiber) and is in the range of $2 * 10^8$ or $3 * 10^8$ m/s, thus it almost equal to the speed of the light. Hence, if d is the distance between two routers and s is the propagation speed, it follows that the propagation delay is d/s. In general, the propagation delays are on the order of ms in wide-area networks.

Thus, we consider into our emulated environment an average delay of $10ms$ for each link, according to the Equation 5.1. Then, we increment these values from $10ms$ to $100ms$ for each link during the simulation to check the results of our model. We also consider the amount of packet loss, according to the details given in Chapter 4. In general, a correlation between the fraction of packet loss and traffic intensity exists. In fact, if the congestion increases the packet loss becomes more intense. Specifically, if a link has a total packet loss of 1%, it means that for every one thousand packets transmitted from the source to the destination, ten packets are dropped.

We want to demonstrate, in the sections below, that without a proactive approach the network can rapidly become congested reaching, incidentally, a very poor QoS.

5.1 Hardware Configuration

Since Mininet runs into our laptop, the performance of the network emulator depends on the hardware characteristics. Furthermore, we use Mininet v.2.0 in a VM ritualized with the Oracle VirtualBox (version 4.3.2) software. The laptop characteristics used for the tests are:

- Processor: Intel® Core™2 Duo T9300 @ 2.50 GHz x 2
- Memory: Corsair SODIMM DDR2 2 x 2 $GBytes$ @ 667 MHz
- Disc: Solid State Disk 250 $GBytes$
- O.S.: Debian 7.5 (Wheezy) 64-bit with the Kernel Linux 3.2.0-4-amd64

The hardware constraints are very important especially if it is required to emulate a network with a large number of nodes or with a large links capacity (e.g., Gigabit Ethernet). Moreover, for conducting our experiments in the "hybrid" scenario, as explained in Section 5.4, we use a wireless router (a D-Link DSL-G604T, 54Mbps) and real devices such as Android smartphones and laptops with different Operations Systems.

The next section details the network topology configuration used for the testing of our proposal.

5.2 Network Topology

For our tests, we consider a network topology composed of a video streaming server, a medical server, and two different clients, as drawn in Figure 5.2.

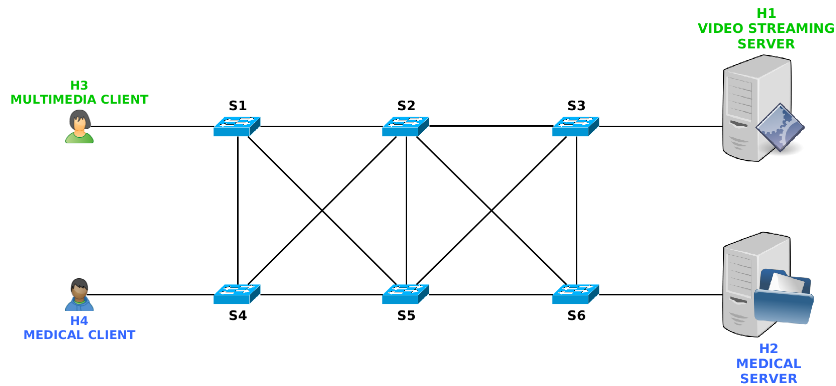


Figure 5.2: Network topology model

Specifically, we want to study in an emulated environment the feasibility of a real network scenario at the UCLA campus in which different flow types (e.g., some general multimedia contents and some medical information related to the hospital patients) can share the same physical infrastructure and incidentally can be treated in different ways. This different treatments will depend on the specific QoS requirements connected to the flow type, i.e., a medical data is more important than a generic multimedia data. As detailed in Section 5.4.1, we define the different flows as follows:

1. The **multimedia flow** is composed of a video streaming data that comes

from the video streaming server.

2. The **medical flow** is modeled as a general file transfer data that comes from an FTP server.

Moreover, the particular topology shape depicted in Figure 5.2 ensures path diversity and gave us the possibility to test our mathematical model with multi-commodity flows, as discussed in Chapter 4.

The next section describes the implementation of the network topology in the Mininet emulator.

5.3 Mininet Configuration

In this section we present the configuration of the Mininet emulator, according to the network topology shown in Figure 5.2. To make the implementation of the emulated network topology possible, we use the Python script file detailed in Appendix A.1. In particular, by means of the functions $addHost$, $addSwitch$, and $addLink$, we can add to the Mininet topology the network elements such as $HOST$, $SWITCH$, and $LINK$ respectively.

```
1  net = Mininet( controller=RemoteController, link=TCLink)
2  net.addController('c0', ip='192.168.2.252', port=6633)
3  HOST = net.addHost('host', mac='aa:aa:aa:aa:aa:01')
4  SWITCH = net.addSwitch('switch', listenPort=6634)
5  net.addLink(HOST, SWITCH, delay='1ms', loss=0, bw=10)
```

Furthermore, as specified in the script above, we set the network parameters such as packet loss, delay, and bandwidth available, according to the considerations made at the beginning of this chapter. In practice this piece of Python code creates a link between $HOST$ and $SWITCH$ with $10ms$ of delay, 0% of packet loss, and $10Mbits$ of available bandwidth.

During our tests, we change the link parameters to check the network behavior using our proposal. Incidentally, it is possible to verify that the packet loss and delay values are consistent with the network parameters defined by means of the Python script, using the Mininet CLI as follows:

```
1  root@mininet-vm:/# ping -f -c 1000 10.0.0.1
2  PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
```

```
3  --- 10.0.0.1 ping statistics ---
4  1000 packets transmitted, 840 received, 16% packet loss, time 13258ms
5  rtt min/avg/max/mdev = 80.906/83.784/228.861/11.655 ms, pipe 18, ipg/ewma
       13.271/82.736 ms
```

The *ping* shell command above gives us the RTT statistics about 1000 ICMP packets from the client H3 to the server H1, as depicted in Figure 5.2. In that case, we define a packet loss of 2% and a delay of $10ms$ in our Python script. The average delay found is approximately $84ms$ because, according to the topology depicted in Figure 5.2, the packets have to pass through 4 links (3 hops distance) to reach the destination, and they have to come back. Thus, if each link has a delay of $10ms$, the RTT is approximately $10 * (4 * 2)ms = 80ms$ (4 links in each direction). The same approach gives us the total packet loss. In fact, if each link has a packet loss of 2%, the total loss is approximately $2 * (4 * 2)\% = 16\%$ (4 links in each direction).

Moreover, we use the Floodlight SDN controller version 0.90, explained in Chapter 2, that runs into our Eclipse IDE. On top of the controller we implement our architecture detailed in Chapter 4. We also test our proposal in a particular scenario, that we call "hybrid", composed of the Mininet emulator and a few real devices, as explained in Section 5.4.

5.4 Mininet Hybrid Configuration

In this section, we suggest a Mininet configuration useful to the communication between the network emulator and a few real devices, such as smartphones/laptops, as depicted in Figure 5.3. In particular, the devices are connected to the workstation (where Mininet is executed) by means of a Wi-Fi router. The nodes inside Mininet have private addresses, hence, to make the interaction between the nodes inside and outside Mininet possible, it is necessary to realize a NAT in S1, as depicted in Figure 5.3. It is also compulsory to define a set of rules into the *IPTable* structure (for the Linux kernel firewall configuration) to forward the traffic from the public to the private network. Thus, setting the NAT and the *IPTable* rules, the system can act as a gateway and provide Internet access to multiple hosts on a local network using a single public IP address. We use a Python configuration script (based on the script $nat.py$,

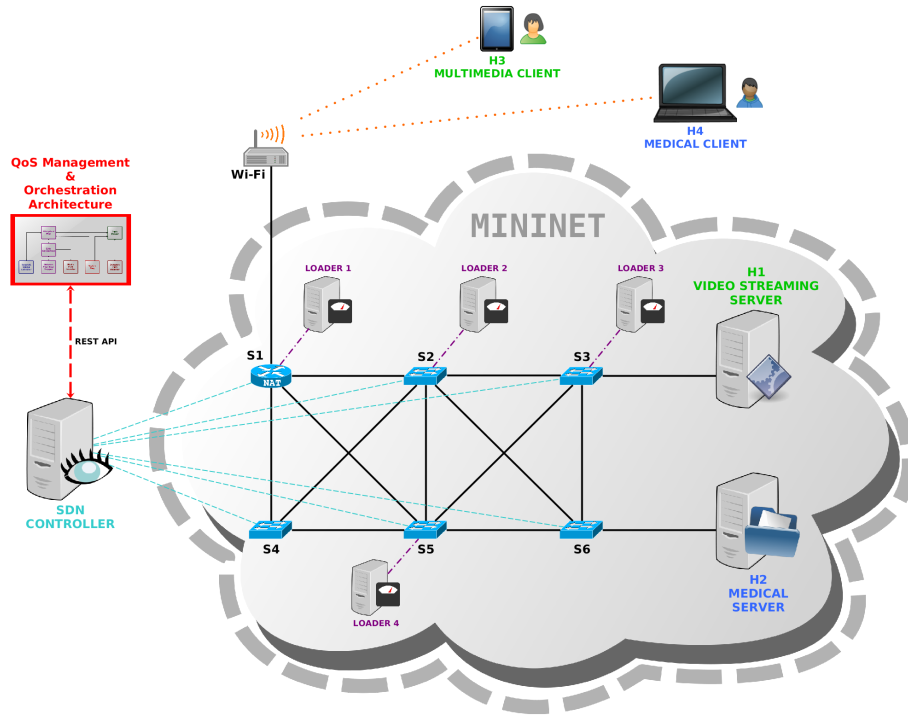


Figure 5.3: Mininet hybrid topology network

supplied with the software Mininet v.2.0 [50]). The *IPTable* configuration defined above allows the NAT to forward the traffic from the network outside Mininet to the servers inside, i.e., video streaming server (bound on port 1234) and the medical server (bound on port 21) respectively.

```
1 iptables -t nat -A PREROUTING -p tcp --dport 1234 -i eth0 -j DNAT
      --to-destination 10.0.0.1:1234
2 iptables -t nat -A PREROUTING -p ICMP -i eth0 -j DNAT --to-destination 10.0.0.2
3 iptables -t nat -A PREROUTING -p tcp --dport 21 -i eth0 -j DNAT --to-destination
      10.0.0.2:21
4 iptables -t nat -A PREROUTING -p tcp --dport 5001 -i eth0 -j DNAT
      --to-destination 10.0.0.2:5001
```

As we can see in the code above, we define an additional rule for ICMP messages forwarding (e.g., used by the *ping* command) to the medical server. This rule is useful to verify the Round Trip Time (RTT) between the clients and the medical server, as explained in Section 5.3. The last *IPTable* rule regards the forwarding of the traffic data generated by the `Iperf` command to overload the network, as explained in Section 5.4.2.

5.4.1 Network Services and Tools

For the video streaming flow, on the server side, we use a VLC media player [51], as a video streaming server, configured as follows:

```
1 cvlc -vvv file_video.ogg input_stream --sout
    '#standard{access=http,mux=ogg,dst=10.0.0.1:1234}'
```

As we can see above, we used the cvlc command (that allows us to use the media player without graphical interface) with the arguments for defining the http protocol and the port 1234 where we want to bind the streaming service. Relative to the video content, we use the "Big Buck Bunny" open movie [52], encoded in ogg format with a resolution of 1280x720 pixel (HD). On the client side, we also use the VLC media player to receive the network video stream.

For the file transfer flow, on the server side, we employ a simple FTP server based on the Apache FtpServer Java project [53] and bound on port 21. On the client side, we use the *wget* Linux command below to download a general file from the FTP server.

```
1 wget --no-passive-ftp
    ftp://USERNAME:PASSWORD@SERVER_IP:SERVER_PORT/FILE_TO_DOWNLOAD
```

We specify the $--no-passive-ftp$ parameter to avoid the use of alternative ports by the *wget* command, since the FTP server is behind a NAT.

Moreover, we collect network informations by means of our architecture module, as detailed in Section 5.7, and also through the *bmon* and *ntop* Linux commands.

```
1 bmon -o ascii -p INTERFACE -r 0.4
```

The *bmon* command give us the bandwidth usage related to a specific interface by means of the $-p$ parameter. With the $-r$ argument we are able to set the reading interval in which the input module will be called. In practice, these tools allow us to get an additional general view of the bandwidth usage for each interface into the emulated network. Another simple tool such as the *ping* command, also gives us an idea about the network status, as mentioned in Section 5.3.

Moreover, we develop a graphical tool, called *Floodlight QoS Advisor* that helps us during the network management, as illustrated in Figure 5.4. The currently *Floodlight QoS Advisor* is an "alpha" version. It allows us to inspect the switches for retrieving statistic information, as illustrated in the *Snoop* tab. The tool also gives us the possibility to modify or delete flow entries by means of the *Modify* tab and it makes the throughput measurement of several ports possible through the *RealTime* tab.

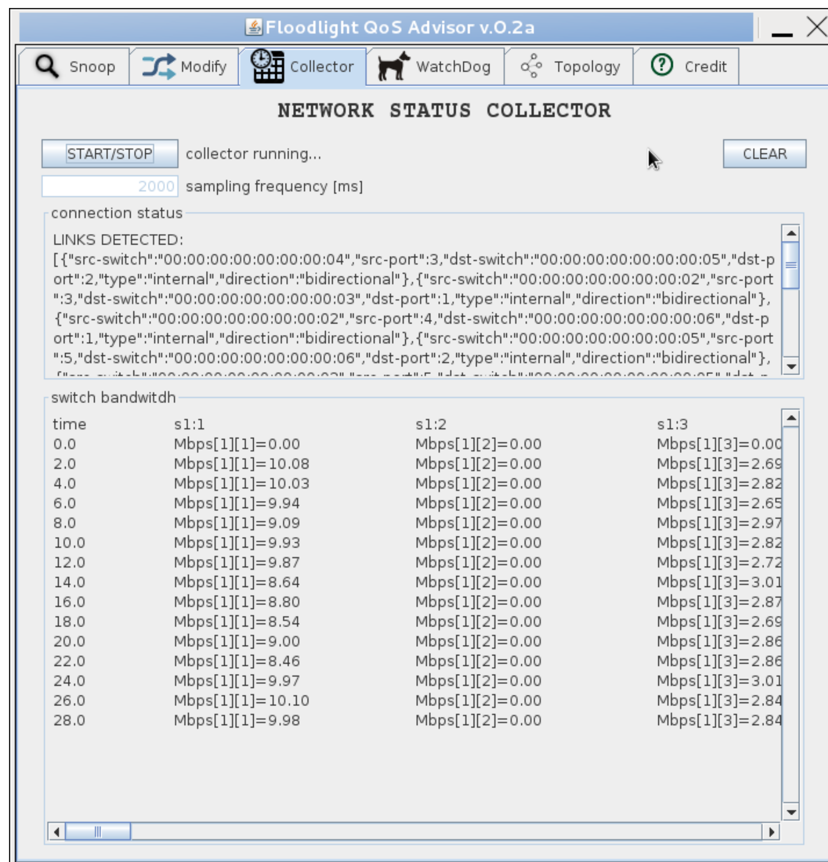


Figure 5.4: *Floodlight QoS Advisor v.0.2a*

The next section explains the configuration that we use to stress the network.

5.4.2 Stress the Network

We use a few Loader servers, connected to the switches, to make the network overloading (congestion) possible. Incidentally, we measure the network

status (e.g., the available bandwidth) and verify the quality of the services (for instance, the file transfer and the multimedia video streaming) to find a relation between them. The Loader server uses the *Iperf* Linux command (in a *server mode*) to inject flow into the network. Moreover, we put another Loader server (in a *client mode*) to make the overloading between two nodes possible. The Loaders are connected to different switches, for instance, S1 and S3, to realize a network congestion on the links between S1 and S2 and between S2 and S3. We try different network congestion configurations simply attaching the Loader servers to various switches.

```
1 iperf -s
2 iperf -c [SERVER IP] -t [OVERLOADING DURATION] -d
```

As we can see above, we have to run the *Iperf* command in a *server mode* ($-$s argument) and in a *client mode* ($-$c parameter) respectively into the different Loader servers. With the $-$d argument we define the bidirectional traffic flow for a specific duration ($-$t).

5.5 Mapping the Network

The topology mapping can be realized in different manners, e.g., exploiting the information stored in the Floodlight or through the REST API. We map the network topology into a simple binary file to make the operations easy. However, the architecture modularity gives us the possibility to develop this part at a later stage. In the structure we store, for each switch, the connected ports and the next hop reachable by following these ports. This file is inspected by the *Dynamic Path Inserter* module to find the specific ports involved in the path given by the *Path Finder*. Then, the *Dynamic Path Inserter* uses the retrieved ports to set the flow entries in the switches, as described in the next section.

5.6 Inserting the Path

The *Dynamic Path Inserter* module is in charge of path insertion into the switches, as detailed in Section 4.1 of Chapter 4. In practice, it is an actuator of the path found by the *Path Finder*. The module has to inquire the topology

map, defined in Section 5.5, for finding the ports involved in the path and thus finalize the insertion path operations, as follows:

```
1  public static void putFlowForward(String controller, String sw, String
       portService, String destPort) {
2   ...
3   putFlow(controller, sw, flowName, cookie, priority, portService, destPort,
        destMac);
4   ...
5  }
6
7  public static void putFlowBackward(String controller, String sw, String
       portService, String destPort) {
8   ...
9   putFlow(controller, sw, flowName, cookie, priority, portService, destPort,
        srcMac);
10  ...
11 }
```

In the code above, we define in the $putFlowForward$ method a rule for the traffic from a client to a server (e.g., video streaming server). The rule above defines, for each packet with a destination MAC address ($destMac$ of the server) and a specific port ($portService$ e.g., 1234 port), to forward the traffic to a specific switch destination port ($destPort$ argument). We also define a method, called $putFlowBackward$, to insert the flows in the opposite direction (from a server to a client). In this case, we have to forward the traffic that comes from the source server ($srcMac$) to the client through a different destination port ($destPort$). The $cookie$ and $priority$ parameters allow us to specify a cookie ID and a flow priority respectively.

The next section presents the measurement of the network parameters, in particular the available bandwidth.

5.7 Network Metric Measurement

As explained in the Chapter 2, the network parameters are fundamental aspects because they allow us to understand the status of the network and consequently take the right decisions. Thus, by means of those values, we are able to build a weighted map and consequently find the best path by means of our multi-criteria approach, as explained in the Section 4.2. The weighted

map is a binary file where the amount of bandwidth, packet loss, and latency between every switch are stored. This structure is used as input data by the mathematical model that is the *Path Finder* core, as described in Section 4.1 of Chapter 4.

The following section explains how it is possible to retrieve the available bandwidth values through the OpenFlow protocol.

5.7.1 Available Bandwidth

We retrieve the available bandwidth related to a specific link by means of the amount of packets passed through the switch port connected to it. Specifically, the amount of bytes transmitted or received from/to a port are related to the low-level data transmission (i.e., throughput, not the goodput). Our architecture allows us to continuously sample, for each switch port, the transmitted or received bytes with a specific frequency. Hence, comparing the retrieved values in two different instants, it is possible to approximately know which is the bandwidth usage of the link connected to that port. The throughput computation is detailed below:

```
1  int lastPacketsCounted = 0;
2  int PacketsCounted = 0;
3  float bandwidth = 0;
4  float bandwidthKBps = 0;
5  float bandwidthMBps = 0;
6  float bandwidthMbps = 0;
7  float freq = 1000; //milliseconds
8  boolean status = true; //to block the sampling
9  String str = null;
10 String type = "port";
11
12 public void run() {
13  try {
14   while(status) {
15    ...
16    // Getting the statistics about the port from a specific switch
17    str = FlowInfo.getCounterStatisticsSwitch(IPs.getControllerIP(), sw, type );
18    // Retrieving the "receiveBytes" value by means of the Java Regex (Regular
         Expressions)
19    packetsCounted  = StringAttributeValueExtractor.retrieveAttributeValue(str,
         port, "receiveBytes");
20    bandwidth = (packetsCounted - lastPacketsCounted) / (freq/1000);
21    bandwidthKBps = bandwidth / 1024;         //KiloByte / sec
```

```
22        bandwidthMBps = bandwidth / (1024*1024);  //MegaByte / sec
23        bandwidthMbps = bandwidthMBps * 8;        //Megabit  / sec
24        // Updating the last value of bytes
25        lastPacketsCounted = packetsCounted;
26        // Collecting the statistics
27        writer.println(  timeLaps + "\t" + String.format( "%1.2f", bandwidthMbps ) +
              "\t" + String.format( "%1.2f", bandwidthMbps2 ) + "\t" + String.format(
              "%1.2f", bandwidthMbps3 ));
28        Thread.sleep((long) freq); //milliseconds
29      }
30    } catch(Exception e) {
31       System.out.println(e.getMessage());
32      }
33  }
```

As we can see in the code above, the Java class involved in bandwidth measurement continuously retrieves statistics information from a specific switch (sw) by means of the $retrieveAttributeValue$ method. This method calls a floodlight REST API to directly get the information from the devices. We update the values of the new and last received bytes to make the bandwidth computation possible. Then, the method stores the found values into a weighted map, as detailed in Section 5.7, to allow the *Path Finder* module to compute the new path with fresh data.

The next section presents the results of the thesis with a focus on the difference in terms of service throughput and QoS between a scenario with and without a QoS management.

5.8 Results

In this section, we provide simulation results about our proposal. The first step is to simply measure the maximum reachable throughput (bandwidth consumption) by means of the *Iperf* tool in the two different scenarios, depicted in Figures 5.2 and 5.3 and called respectively "regular" and "hybrid" in Figure 5.5 below. At the beginning, we set the Mininet network with a $100Mbps$ of link capacity, $1ms$ of delay, and 0% of packet loss for each connection. To make it possible, we used the *Iperf* Linux command to inject a flow into the network. In one case, we measure the throughput between the Loader server connected to S1 and the server H2, in the other case, we retrieved the throughput values

from the same server H2 to a laptop located outside Mininet (H4), as illustrated in Figure 5.3.

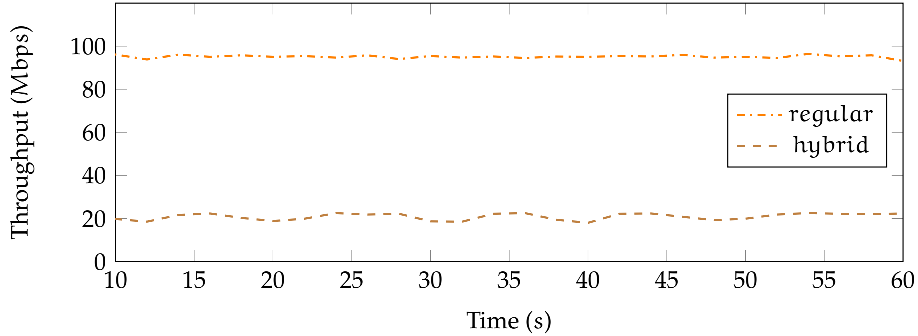


Figure 5.5: Throughput measurement using *Iperf* tool in the different scenarios (100Mbps network).

As we can see in Figure 5.5, although our Wi-Fi has a theoretical 56Mbps of bandwidth, the maximum reachable throughput outside Mininet (i.e., the "hybrid" trend) is about 20Mbps. This value is due to some well known Wi-Fi limitations, e.g., channel interference, signal degradation (the Wi-Fi router is behind a thin wall and about 10 meters far from the testbed), and so on. We decide to use for our tests a 10Mbps of Mininet link capacity to avoid the "bottleneck" limitation related to the Wi-Fi interface. Hereafter, we only consider the "hybrid" network topology configuration, detailed in Section 5.4, because it represents a scenario more close to a real environment.

We are also interested in comparing the values given by the *Iperf* tool with the values supplied by our module (called *Network Status Collector* in Figure 4.2 in Chapter 5) to verify the reliability of our measures. To reach this goal, we try two different network configurations and we compare the values retrieved in both scenarios. The test are done using both the *Iperf* tool (collecting the statistics given by the *Iperf* server connected to H2) and our *Network Status Collector* module (getting the values from the switch S6 that is directly connected to the server H2). The sampling is done every 2ms for 60s. Specifically, in the first experiment the network topology is set with a 0% of packet loss, 1ms of delay for each link, and 10Mbps of maximum available bandwidth. The trends are illustrated in Figure 5.6.

We also repeat the experiment above with a total packet loss of 4%, 1ms of

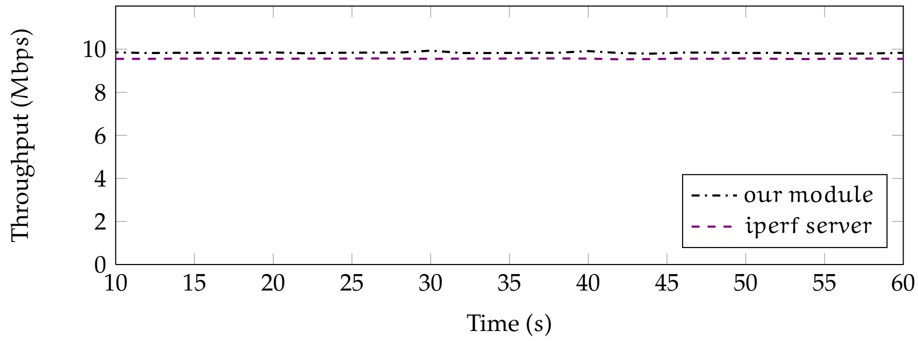


Figure 5.6: Throughput measurement using both the *Iperf* tool and our module with no packet loss

latency for each link, and 10Mbps of link capacity. The trends are depicted in Figure 5.7.

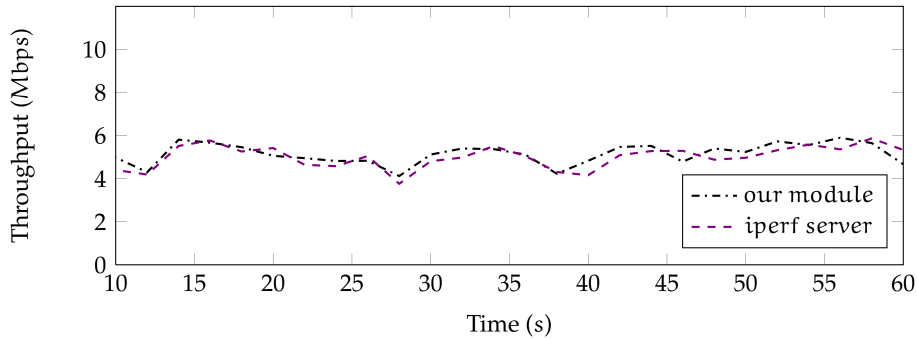


Figure 5.7: Throughput measurement using both *Iperf* tool (server side) and our module with 4% of packet loss

As clearly shown in Figure 5.7, with a total packet loss of 4%, the throughput is around 5Mbps and it has an oscillatory behavior. We notice in both Figure 5.6 and Figure 5.7 that the values retrieved by the *Iperf* tool and through the *Network Status Collector* module are very similar. Thus, we can assert that the values retrieved by our module are reliable. Moreover, from here on, we show our results based on values retrieved through our module.

The SDN paradigm in conjunction with OpenFlow protocol also gives us the possibility to measure the values directly from each switch OpenFlow-enabled, not only between the end points as generally the network tools do (e.g., *Iperf*).

The second step is to analyze the network behavior during a link congestion

to verify the need of a QoS management. Initially, we measure the bandwidth usage during the video streaming without network congestion, as illustrated in Figure 5.8.

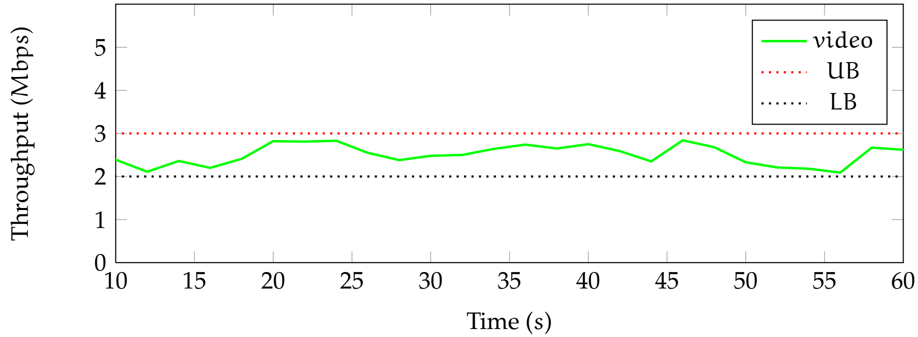


Figure 5.8: Bandwidth usage during the video streaming service.

As we can see in Figure 5.8, the video streaming throughput is approximately between a lower bound (LB) of 2Mbps and an upper bound (UB) of 3Mbps. These values are related to the video characteristics, e.g., resolution, format, and compression. In this case the video content is a ogg file with HD resolution, as specified in Section 5.4.1 above.

Then, starting from the scenario explained above, we temporary overload the network for verifying the throughput trend in case of link congestion. Specifically, to make it possible, we use the Loader servers 1 and 2 that are directly connected to the switches S1 and S2 respectively, as in Figure 5.3. The congestion is temporary done during the video streaming service and the situation is depicted in Figure 5.9.

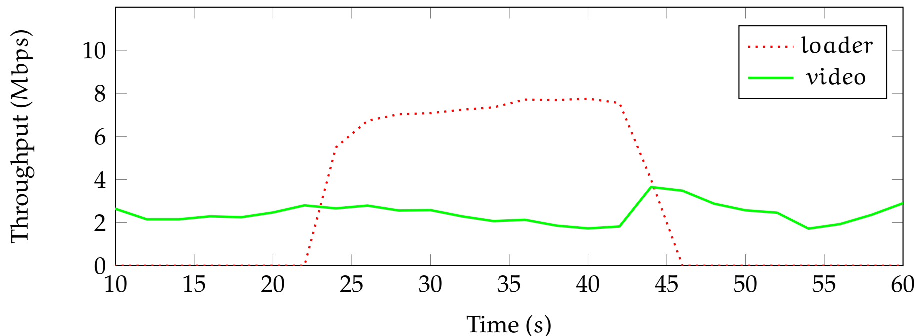


Figure 5.9: Temporary network congestion during the video streaming service.

As shown in Figure 5.9, during the network congestion (forced from approximately the instant 22s to 42s), the video streaming throughput decreases under the critical threshold of 2Mbps. It follows that, in case of a short link congestion, the video is blocked just only for a few seconds by lack of sufficient available bandwidth. This issue is limited in part by the video player buffer.

Unfortunately, this unwanted situation could become even worse if the congestion window is longer than the last scenario, as illustrate in Figure 5.10.

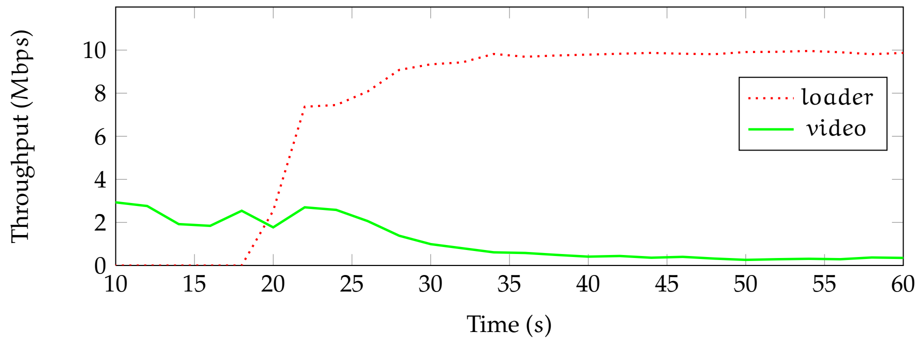


Figure 5.10: Video streaming and file transfer throughputs during a temporary link congestion.

As in Figure 5.10, when the network overloading starts, the video streaming falls below the critical threshold (2Mbps) approximately after 10 seconds. Consequently, in case of long network congestion, the video is completely blocked and the buffering cannot be useful to limit this issue.

We also overload the network during the execution of both the video streaming and file transfer services (we limit the maximum throughput to 4Mbps to emulate a real scenario during the tests), as represented in Figure 5.11.

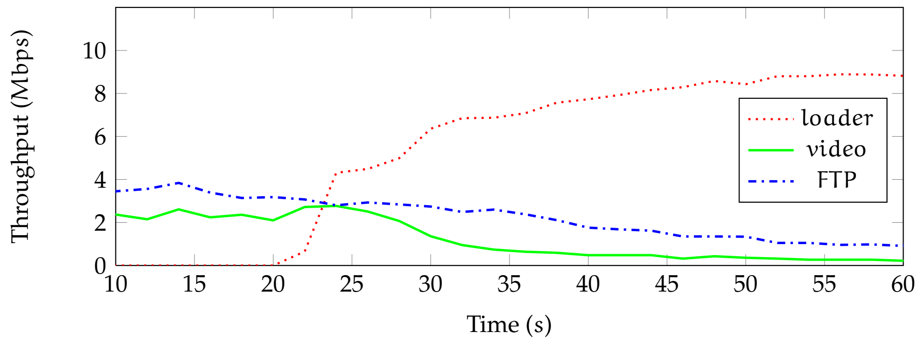


Figure 5.11: Video streaming and file transfer throughputs during a permanent link congestion.

It goes without saying that while a lack of sufficient bandwidth impacts negatively on the video streaming service, the file transfer just only slow down during the network congestion. However, the total amount of time required for a file transfer proportionally increases with the decreasing of the available bandwidth. Thus, it is necessary to find a tradeoff and it depends on the type of service "behind" the file transfer.

The first important consideration is that if it was possible to analyze the network status, it would be feasible to define the network paths to avoid the link congestion and, consequently, provide strict bandwidth guarantees to the video streaming service. Unfortunately, neither the Mininet switches (that are just packet forwarders) nor the Floodlight controller are aware of the network status. Specifically, the natural approach of SDN controllers (without QoS such as Floodlight) is to choose the paths by means of Dijkstra algorithm. However, the shortest path is calculated without taking into account the network status, e.g., packet loss, latency, or available bandwidth. Moreover, the defined paths are generally kept over time. It follows that, in case of multi-commodity flows (as in our scenario), the SDN controller could assign the same entire path (in the worst case) or a part of it (e.g., a few links) to each flow, as sketched in Figure 5.12.

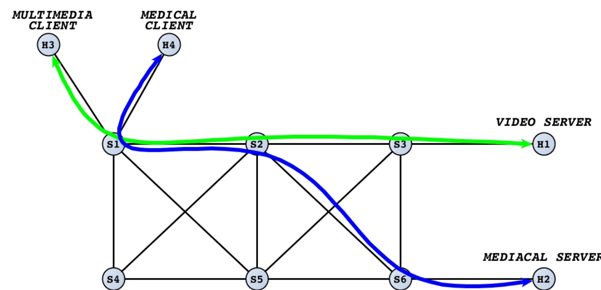


Figure 5.12: Video steaming and file transfer paths in a network without QoS.

Thus, the video steaming and file transfer services have to share the same available bandwidth. This aspect has a strong impact on the network performance proportionally to the increasing of the number of flows on the same path and it could potentially lead to a congestion collapse of the network.

The third step is to evaluate the network behavior during the use of our architecture to enhance the QoS. We want to illustrate the QoS reached by

means of our mathematical model and the network monitoring. We start to consider that without awareness of the network available bandwidth, the controller cannot know which are the overloaded links and, consequently, it is not able to find the best path. Hence, under these considerations, a congestion situation can easily occur just only if a single path link is overloaded (that becomes the network bottleneck). In addition, when it is necessary to deal with multi-commodity flows, it is mainly important to take into account that the amount of flows can be more than one at the same time. It follows that it is crucial to efficiently assign the path to each flow, according to both the link available bandwidth and the single flow throughput requirements. In practice, to make this decision possible, we decide to set our architecture to retrieve and store the amount of available bandwidth for each link, every 2 seconds. This task is realized by the *Network Status Collector* module, as explained in Section 4.1 of Chapter 4. Incidentally, we use the *Path Finder* module for discovering the best path. The *Path Finder* uses the network status data, collected by the *Network Status Collector*, as constraints in the MCFCSP problem formulated in Section 4.2.3 of Chapter 4. Furthermore, the specific bandwidth requirements for each type of service (i.e., video streaming and file transfer) are added as constraints to the ILP problem. The output of the model is the best path as possible for each flow, according to the constraints.

Relative to the topology depicted above in Figure 5.3, the first test is to verify the *Path Finder* output. We set the flow bandwidth requirements to $4Mbps$ and $3Mbps$ respectively for the file transfer service and the video streaming, according to the results depicted in Figure 5.8. At the beginning we consider for each link a delay of $1ms$ and an ideal packet loss equal 0%. Moreover, we define a video streaming throughput threshold suitable to trigger the path changing by means of the *Watch Dog* module, as described in Chapter 4. The threshold value is related to the average throughput of the video streaming used in our test (it depends on the format and video compression). Specifically, we take the minimum throughput value, that is $2Mbps$, as in Figure 5.8 and we subtract the 10% to avoid borderline throughput value. Hence, the *Watch Dog* module will trigger the changing of the path, when the throughput falls below the threshold of $1.8Mbps$. However, during a regular video streaming, it is possible that

the value of the throughput cross the lower bound of 1.8Mbps, as depicted in Figure 5.13.

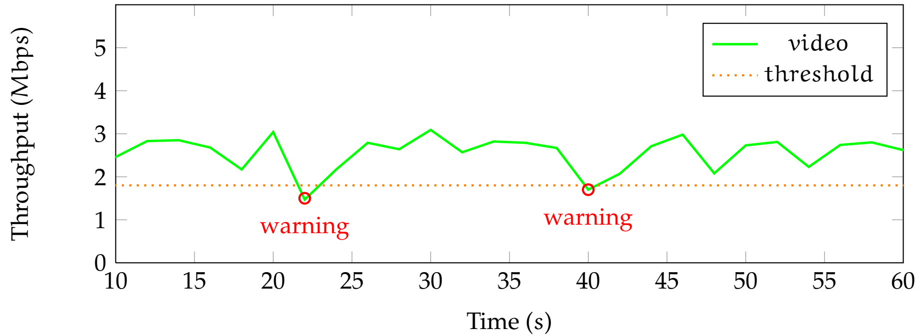


Figure 5.13: Warnings due to a under-threshold throughput.

In this case, in general, the video player is able to manage the temporary lack of bandwidth by means of its buffer. However, without a precaution, this situation could lead to a frequent path changing. We call it a "false-positive" warning, as depicted in Figure 5.13. To limit this unwanted behavior, we add to the *Watch Dog* module the capacity to manage these warnings, as schematized with the flow chart in Figure 5.14.

Specifically, the *Watch Dog* Flow Chart reasons as follows: if a warning occurs for the first time, the *Watch Dog* does nothing and, incidentally, it updates the *counter*. Then, if it does not occur another warning for a specified time, e.g., 5 seconds, the warning is deleted, otherwise, it triggers the changing of the path. This is a simple behavior that avoid some "false-positive" warning. However, we can implement a bit more sophisticated behavior just only by tweaking this module. It is clear, on the one hand, that if we delete the first warning too quickly, the risk is to remove an important warning that indicate an imminent congestion. On the other hand, if we wait for a long time before deleting the first warning, it is possible to change the path when it is not necessary. Thus, this aspect is a trade-off between a reactive behavior to prevent congestion collapse and an useless network overloading due to a path changing when not necessary.

We start the first test with our QoS module to avoid the block of the video due to a network congestion, as depicted in Figure 5.15.

As we can see in Figure 5.15b, through a dynamic changing of the path we

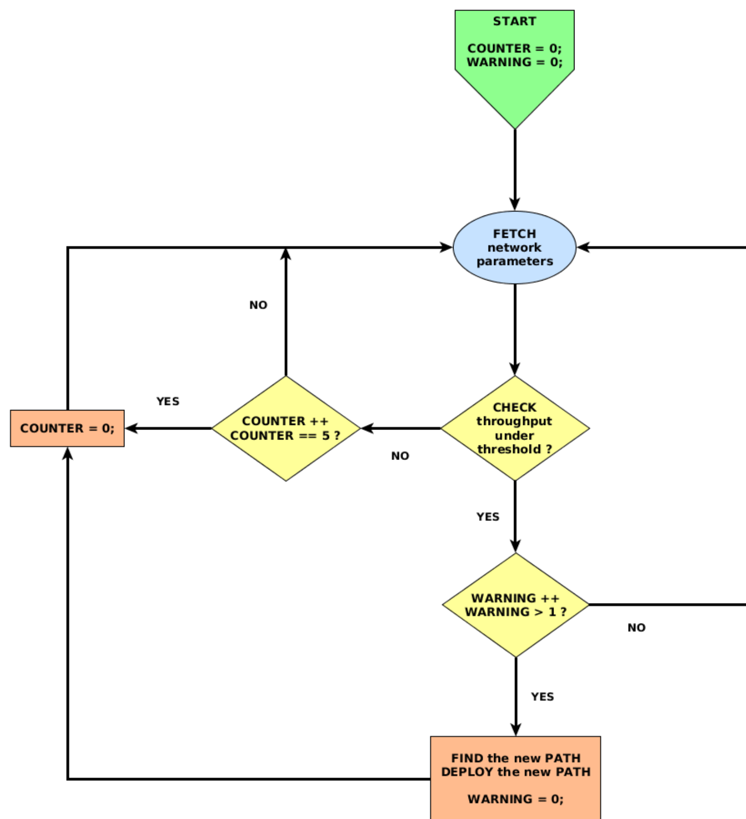


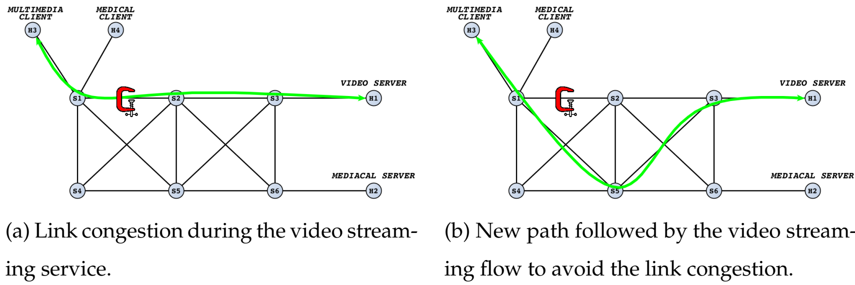
Figure 5.14: The “Watch Dog” Flow Chart

are able to avoid the link congestion between the switches S1 and S2. In this case, the video streaming throughput does not significantly suffer from the network congestion and it falls below the critical threshold just only for a short time, as shown in Figure 5.16.

Fortunately, in general, the video player buffering is able to manage this short lack of bandwidth and the result is a good video quality with a very short block or no block at all.

Comparing the Figures 5.10 and 5.16 respectively with and without the QoS module, it is clear that in the latter scenario the architecture is able to keep the video steaming throughput higher than the former scenario. Since the video streaming service needs a minimum amount of available bandwidth, the network without a QoS mechanism is not able to guarantee the video streaming requirements.

Starting from the last scenario, we also verify the network behavior in case



(a) Link congestion during the video streaming service.

(b) New path followed by the video streaming flow to avoid the link congestion.

Figure 5.15: Path changing by the QoS architecture during a permanent link congestion.

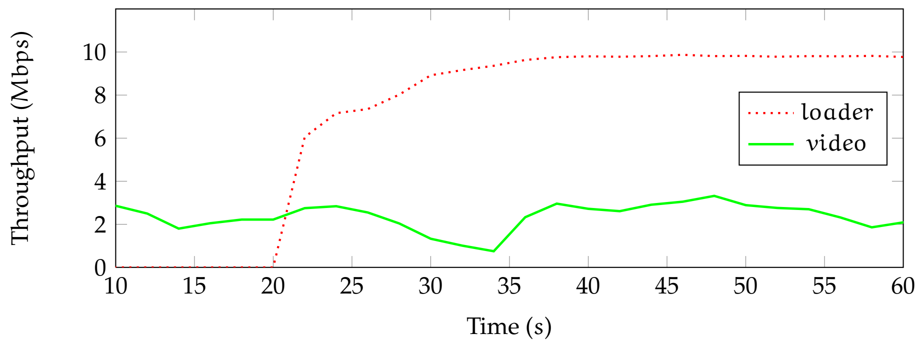


Figure 5.16: Video streaming throughput during a permanent link congestion in a network managed by the QoS architecture.

of multiple congestions, as depicted in Figure 5.17.

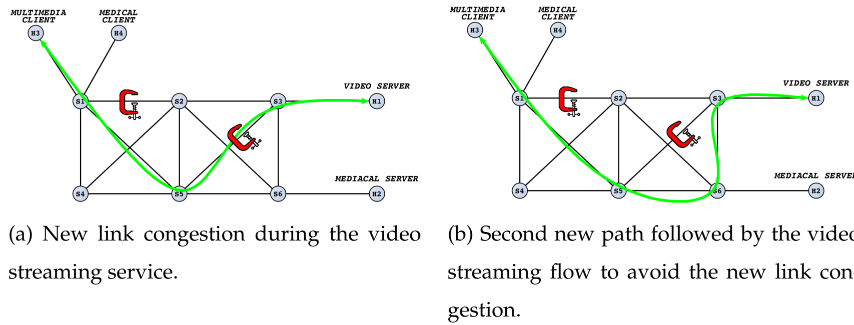
The QoS architecture is also able to identify the second link congestion, between the switches S3 and S5 and, consequently, change the path again, as in Figure 5.17.

Incidentally, the video steaming throughput keeps a fair QoS, falling under the threshold just only for a few seconds, as illustrated in Figure 5.18.

In this case, there are two link congestions (between S1 and S2 and between S3 and S5) at the same time. However, the QoS architecture is able to manage the multiple path changing to keep the video streaming through as high as possible.

Moreover, we test our architecture during a multi-commodity flow scenario, as depicted in Figure 5.19.

As we can see in Figure 5.19b, the QoS architecture redirects the two different flows, related to the video streaming and the file transfer, to different paths (with a common link).



(a) New link congestion during the video streaming service.

(b) Second new path followed by the video streaming flow to avoid the new link congestion.

Figure 5.17: Multiple path changing by the QoS architecture during a multiple link congestion.

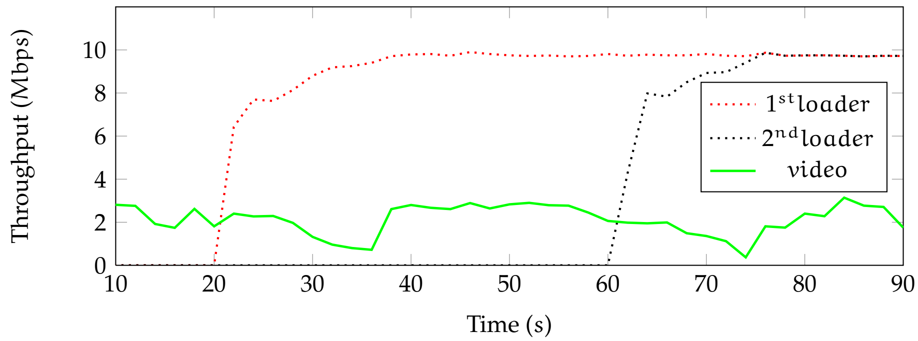
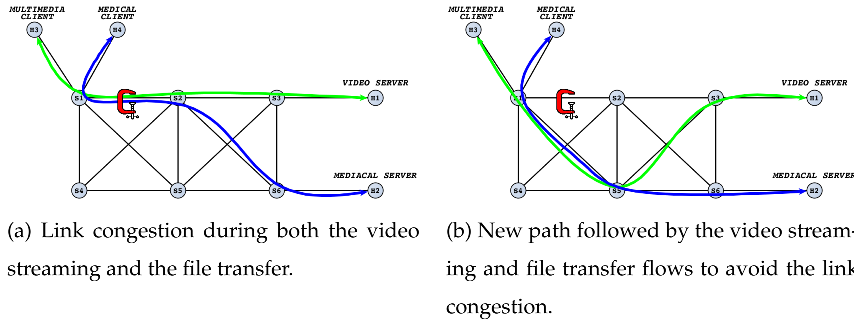


Figure 5.18: Video streaming throughput during a multiple permanent link congestion in a network managed by the QoS architecture.

In figure 5.20 it is also depicted both the video streaming and file transfer throughput.

It is quite clear that in this case the bandwidth management is better than the scenario without the QoS module. Thus, with a QoS architecture, it is possible to keep an high throughput and consequently, a good quality.

At this point, we test our proposal putting into the simulation different values of packet loss evaluating the QoS. Incidentally, for each packet loss percentage, we calculate the value of the mathematical model giving to it the input parameters about the network status. Then, the same procedure is repeated using different values of delay. We also test several combinations of delay and packet loss to observe the QoS behavior. Using this approach, and by exploiting the Equation 4.11 of the MFPCSP model detailed in Chapter 4, we find a correlation between the cost given by the objective function and the MOS score levels, as detailed in Table 5.1.



(a) Link congestion during both the video streaming and the file transfer.

(b) New path followed by the video streaming and file transfer flows to avoid the link congestion.

Figure 5.19: Path changing by the QoS architecture during a permanent link congestion.

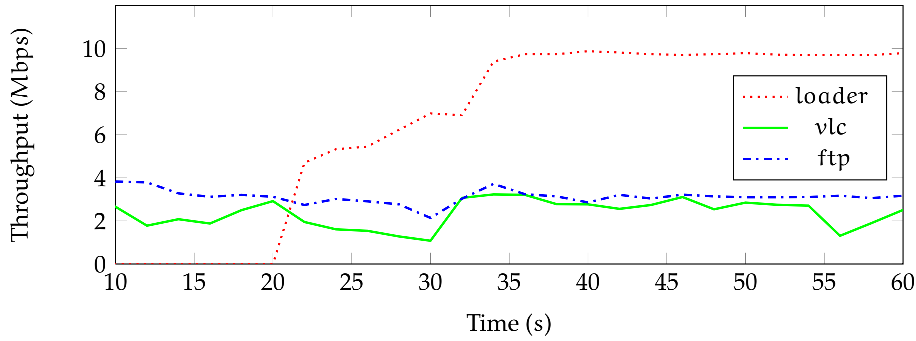


Figure 5.20: Multi-Commodity Flow throughput during a permanent link congestion in a network managed by the QoS architecture.

As we can see in Table 5.1, the connection between the value ranges given by the mathematical model and the MOS levels indicates that, in case of cost values greater than 80, the QoS becomes very low. It is also possible to identify some additional "bad" levels. For instance, we found that with a total cost between 100 and 150 (i.e., 10% and 15% of packet loss respectively), it is possible to start the streaming but the video is completely blocked or it is blocked every $2-3$ seconds. The same situation occurs when the total delay is around $250-300ms$. Finally, with a total packet loss of about 18% (cost value $\approx$ 150) the video streaming does not start at all.

Moreover, we observe some other interesting characteristics about the relation between the cost function and the QoS. Starting from the Equation 4.10 in Chapter 4, we can define the cost equation as follows:

$$\text{cost} = \alpha * \text{delay}_{\text{tot}} + \beta * \text{packet loss}_{\text{tot}} \tag{5.2}$$

MODEL COST	MOS	QUALITY	IMPAIRMENT
< 40	5	Excellent	No block at all
40 - 55	4	Good	No block or a sporadic very short block
56 - 69	3	Fair	A couple of short blocks ($1-2$ s)
70 - 79	2	Poor	Several blocks ($2-4$ seconds long)
> 80	1	Bad	A lot of blocks (> 10) with long duration ($7-10$ s)

Table 5.1: Conversion between our mathematical model cost and the MOS levels related to the video streaming

Thus, exploiting the α and β scale factors of Equation 5.2, it is possible to tweak the weighted cost according to the delay and the packet loss requirements for a particular flow. Specifically, we observe that if the delay is not present or very small, e.g., $1ms$, then we can take into account only the β scale factor and set it to 10. Hence, the Equation 5.2 becomes:

$$cost = 10 * packet\ loss_{tot}$$

In this case, for example, if the total packet loss is equal to 6% (with a very small delay), the video streaming quality is quite fair, according the correlations between the QoS and the cost function detailed in Table 5.1.

We also noticed that if the packet loss is equal to 0, then we can ignore it and set the α scale factor to approximately 1.2. In this case the Equation 5.2 becomes:

$$cost = 1.2 * delay_{tot}$$

For instance, if the total delay is equal to $50ms$ (with 0% of packet loss), it follows that the video streaming quality is fair, according to Table 5.1.

Furthermore, if both the packet loss and the delay occur, we notice that the Equation 5.2 should take into account an additional scale factor equal to 2 as follows:

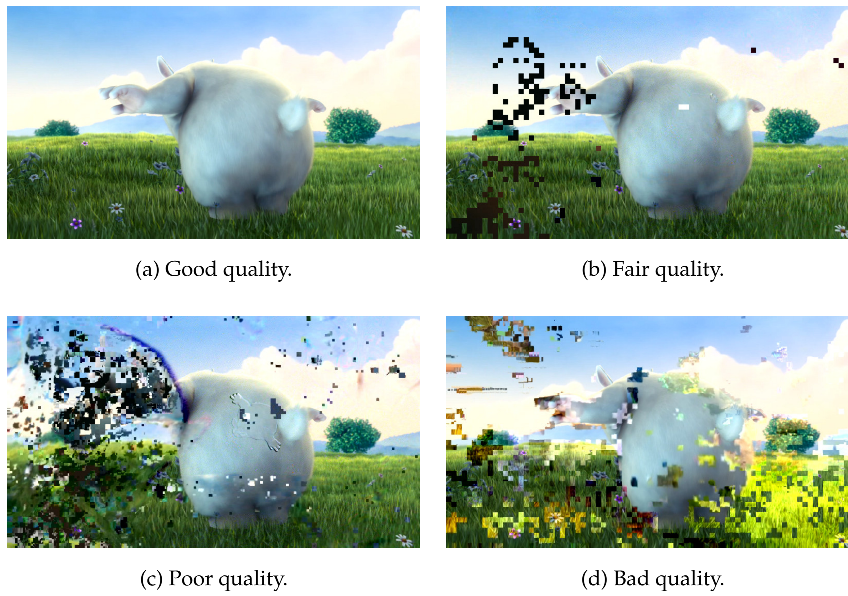
$$cost = (1.2 * delay_{tot} + 10 * packet\ loss_{tot}) * 2$$

For example, if the total packet loss and the delay are respectively about 2% and $12ms$, than the Equation 5.8 gives us a value of 68. In this case, the quality of the video streaming is quite fair, according to Table 5.1. This means that when a video streaming is affected by a combination of delay and packet loss, it is necessary to take into account an additional scale factor (equal to 2) to keep actual the relation between the QoS and the cost function.

Relative to a file transfer, in general, there are not strict constraints about the packet loss, delay, and available bandwidth. However, the increasing of the packet loss and delay can considerably decrease the file transfer throughput. For example, without specific file transfer bandwidth limitations we notice that the maximum throughput is around $9.1Mbps$ (in our $10Mbps$ network). In case of a total delay equal to $80ms$, in general the file transfer starts slowly and then it is able to reach an high throughput value, around $8Mbps$. Unfortunately, increasing the total delay until a significant value of $500ms$, the maximum throughput reachable is approximately $1.8Mbps$ with a very high oscillatory behavior. The same situation occurs with about 8% of total packet loss. However, increasing the packet loss to a value larger than 15%, the connection is lost after a short time. If both the delay and the packet loss occur at the same time, it is possible to verify that the file transfer service quality decreases rapidly. Specifically, with a 6% of packet loss and, incidentally, a total delay of $80ms$, the maximum throughput is approximately $0.25Mbps$ with a very high oscillatory behavior.

The relations above are very important because allow us to define a mapping between the mathematical basics and the QoS in our hybrid environment. It is clear that there are a lot of variables depending on the interferences, the signal noise, and so on.

With our QoS architecture, that continuously inquires about the network status and allocates the paths if necessary, we observed that it is possible to avoid, or strongly reduce, the link congestion effects. As shown in several figures above, the QoS architecture gives us the possibility to keep a throughput as high as possible and, consequently, the quality of the service. This aspect is very important especially if the services require a minimum amount of available bandwidth. In this case, it is necessary to analyze the service throughput for guaranteeing a good quality, in particular if the services are bandwidth-sensitive such as video streaming or VoIP call. It is also fundamental to take into account the amount of packet loss and delay to reach a good QoS, according to the service type requirements. A lack of sufficient bandwidth as well as high packet loss or delay, can impact very negatively on the video streaming service, as we can see in Figure 5.21.



(a) Good quality.

(b) Fair quality.

(c) Poor quality.

(d) Bad quality.

Figure 5.21: Different types of video quality.

The next chapter provides the conclusion and the future works.

Chapter 6

Conclusion

The "Internet Ossification" is a well known phenomenon that brings the research community in exploring new network paradigms and solutions. In a ever-changing world, where dynamic bandwidth needs are no longer negotiable, network adds, moves, and changes are time-consuming and burdensome. What is really needed is the ability to respond in real-time to the network requirements, where an individual can supply the bandwidth by using the power of abstraction. SDN and OpenFlow provide the framework required to make this possible creating a programmable network involving both existing infrastructure and next-generation systems and making them substantially more dynamic. Moreover, SDN can simplify the software needed to deliver services, improving the use of the network and shortening delivery times, leading to increased revenue.

The main problem of the today networks is the ability to manage differentiating services guaranteeing the various QoS requirements. With this thesis, we deeply analyzed the problem and showed that we are able to enhance the QoS in a SDN scenario exploiting a novel QoS architecture. Moreover, we achieve a better utilization of the the network resources by means of the multi-criteria approach and our Multi-Commodity Flow Constrained Shortest Path model. The model takes into account the network requirements, i.e. bandwidth, packet loss and delay, to find the shortest path between each couple of client and server for each service. Our QoS architecture continuously inquires the network sta-

tus and allocates new paths, if it is necessary, considering the constraints relative to packet loss, delay and bandwidth. The results show that it is possible to avoid, or strongly reduce, the link congestion effects. Moreover, the QoS architecture gives us the possibility to keep a good level of throughput and, consequently, the needed quality of each service. This aspect is very important especially if the services require a minimum amount of available bandwidth. A lack of sufficient bandwidth as well as high packet loss or delay can impact very negatively on the video streaming service. Furthermore, we are able to map the results given by our MCFCSP model into a MOS scale to provide an opinion score from the client point of view. The evidence of the MOS is provided through video streaming screenshots.

We can conclude that SDN and OpenFlow will most likely become pervasive technologies in the future networks, since they have an enormous potential contribution in a large number of different application fields, as explained in the next section. The modular design of the proposal in conjunction with the flexibility supplied by the Floodlight REST API interface, provides a flexible architecture that can be easily extended to operate with different controllers or to realize additional functionalities. Our model and evaluations are a first step into the definition of a QoS enhanced architecture that can provide differentiating services with some QoS guarantees. We notice that the bandwidth calculator can sometimes give a little inaccurate values if the sampling is too short. It depends on the possible delays that occur in the network, and, consequently, the bandwidth calculation may be a bit overestimated. Thus, we think to ameliorate this tracer in order to get better results. Furthermore, we can improve the network mapping by using a dynamic structure or directly the data stored inside the Floodlight controller. In our mathematical model we take into account the available bandwidth, the packet loss, and the delay for calculating the new path according to the network status. The available bandwidth parameter is continuously calculated for each switch and stored into the *Weighted Map* structure, as explained in Chapter 4. Since it is not simple to get a precise amount of packet loss and delay for each link, these values are predefined in the *Weighted Map* for each test. Hence, it will be an interesting challenge to integrate in our architecture an additional module to continuously monitor and

calculate both the packet loss and the delay of the network. Finally, this is the starting point for the implementation into a real world scenario, such as the UCLA campus.

6.1 Future Work

In this section, we want to highlight a few interesting challenges that concern SDN and OpenFlow. Specifically, it is important to give a generic overview about the possible integration between the SDN paradigm and other interesting technologies.

Many users have access to the Internet via wireless networks, especially in a campus network. Thus, the size and complexity of wireless networks and heterogeneous systems connected to them is constantly evolving and increasing. However, since SDN and OpenFlow were conceived in a wired scenario, it is important to have accurate network information and a good control of the packet flow in a wireless scenario. Moreover, many other issues have to be taken into account when we move from the wired to the wireless environment, especially when different technologies such as Wi-Fi, WiMAX, and LTE need to cooperate together. Hence, in this direction, the natural next step could be the extension of the architecture presented in this thesis to deal with the new challenges offered by Wireless SDN [54]. In this case, to manage the resources and paths as best as possible, it is indispensable to continuously retrieve precise network information directly from the different wireless devices. When we are able to collect these types of information, then it will be possible to decide not only which the best path is that packet flows have to take, but also which the best interface is that clients have to use at a given moment to reach the best QoS. Therefore, OpenFlow can be exploited to trigger a seamless vertical handoff, according to the Media-Independent Handover protocol, to achieve a high-mobility connectivity. We plan to exploit our solution to manage the handover between different network technologies. In this scenario, our model can be run with different parameters, depending on the communication protocol, and can provide optimal results to be implemented on the campus network.

It is also particularly interesting to analyze and try to solve new challenges

in Vehicular Ad Hoc Networks (VANETs) by means of SDN and OpenFlow. Some of the major challenges for communication in VANETs are very high mobility, dynamically changing topology, sparsely located nodes and very short duration of communication. SDN can work in conjunction with VANETs to proactively set the best path to improve the communication among vehicles and Access Points (APs) [55].

Moreover, since the Information Centric Networks (ICNs) paradigm is considered as the candidate replacement for the IP protocol, an integration between ICN and SDN [56] will probably be necessary to build the future networks.

In addition, the enormous growth in multimedia contents, that amount to approximately 51% of all the consumers Internet traffic data in 2011 [57], is likely to have a great impact on the network. This aspect implies further efforts to achieve a good QoE in terms of bandwidth requirements, latency constraints, and packet losses. Especially the video streaming is an increasingly popular way to consume media contents, and the Adaptive Video Streaming is becoming an emerging delivery technology which aims to increase the user satisfaction and maximize the connection utilization. By using a SDN technology in conjunction with OpenFlow and exploiting the control plane to retrieve information and manage the switch, it will be feasible to reach the best video streaming QoE possible. Furthermore, we can use two different approaches to reach the goal of enhancing the QoE, depending on the network capabilities. The first approach, on the one hand, is to exploit the network side to verify the possibility to change a specific flow path when the network is suffering congestion or when there is a high packet loss. The new path is chosen according to the particular type of video service that is streamed. Thus, to make this approach viable, we can improve the Multi-criteria approach involved in the presented architecture. The second solution, on the other hand, exploits the user side when the network cannot find a better path. In this case, we can communicate with the client to adjust the video streaming bit rate, by means of MPEG DASH, to keep the quality of the video as high as possible from the user point of view.

Appendix A

Python Code

A.1 Network Topology Configuration

```
#!/usr/bin/python

"""
Example for creating the Mininet topology depicted below.
It is possible to customize the link parameters.

(multimedia client) h3------s1-------s2------s3-------h1 (video streaming server)
                            | \    / | \    / |
                            |  \  /  |  \  /  |
                            |   \/   |   \/   |
                            |   /\   |   /\   |
                            |  /  \  |  /  \  |
                            | /    \ | /    \ |
   (medical client) h4------s4-------s5------s6-------h2 (medical server)

Francesco Ongaro, May 2014
francesco.ongaro@studio.unibo.it
www.ongarofrancesco.com
"""

from mininet.cli import CLI
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.link import TCLink
from mininet.util import dumpNodeConnections
from mininet.log import lg


```

```
29 if __name__ == '__main__':
30
31     lg.setLogLevel( 'info')
32
33     net = Mininet( controller=RemoteController, link=TCLink) #listenPort is
           necessary for adding flow entry
34     net.addController('c0', ip='192.168.2.252', port=6633) # wlan0 @ HOME
35
36     #HOST constructor
37     H1 = net.addHost('h1', mac='aa:aa:aa:aa:aa:01') #VIDEO STREAMING SERVER
38     H2 = net.addHost('h2', mac='aa:aa:aa:aa:aa:02') #MEDICAL SERVER
39     H3 = net.addHost('h3', mac='aa:aa:aa:aa:aa:03') #MULTIMEDIA CLIENT
40     H4 = net.addHost('h4', mac='aa:aa:aa:aa:aa:04') #MEDICAL CLIENT
41
42     #SWITCH constructor;
43     S1 = net.addSwitch('s1', listenPort=6634)
44     S2 = net.addSwitch('s2', listenPort=6635)
45     S3 = net.addSwitch('s3', listenPort=6636)
46     S4 = net.addSwitch('s4', listenPort=6637)
47     S5 = net.addSwitch('s5', listenPort=6638)
48     S6 = net.addSwitch('s6', listenPort=6639)
49
50     #LINK constructor
51     net.addLink(H3, S1, delay='1ms', loss=0, bw=10)
52     net.addLink(H4, S4, delay='1ms', loss=0, bw=10)
53
54     net.addLink(S1, S2, delay='1ms', loss=0, bw=10)
55     net.addLink(S1, S5, delay='1ms', loss=0, bw=10)
56     net.addLink(S1, S4, delay='1ms', loss=0, bw=10)
57     net.addLink(S4, S2, delay='1ms', loss=0, bw=10)
58     net.addLink(S4, S5, delay='1ms', loss=0, bw=10)
59
60     net.addLink(S2, S3, delay='1ms', loss=0, bw=10)
61     net.addLink(S2, S6, delay='1ms', loss=0, bw=10)
62     net.addLink(S2, S5, delay='1ms', loss=0, bw=10)
63     net.addLink(S5, S3, delay='1ms', loss=0, bw=10)
64     net.addLink(S5, S6, delay='1ms', loss=0, bw=10)
65
66     net.addLink(S3, H1, delay='1ms', loss=0, bw=10)
67     net.addLink(S3, S6, delay='1ms', loss=0, bw=10)
68     net.addLink(S6, H2, delay='1ms', loss=0, bw=10)
69
70     # Start network
71     net.start()
72
73     print "*** Dumping host connections"
74     dumpNodeConnections(net.hosts)
75
76     print "*** Testing network connectivity"
77     net.pingAll()
```

```
78
79      print "*** Hosts are running"
80      print "*** Type 'exit' or control-D to shut down network"
81
82      CLI( net )
83      net.stop()
```

Bibliography

[1] I. Faynberg, L. R. Gabuzda, T. Jacobson, and H.-L. Lu, "The development of the wireless intelligent network (WIN) and its rela/tion to the international intelligent network standards," *j-BELL-LABS-TECH-J*, vol. 2, pp. 57–80, Autumn 1997.

[2] Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks," white paper, Open Networking Foundation, Palo Alto, CA, USA, Apr. 2012.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[4] "OpenFlow Switch Specification Version 1.4.0." `https://www.opennetworking.org`.

[5] "Project Floodlight documentation." `http://www.projectfloodlight.org/documentation/`.

[6] J. Gustafsson, G. HeikkilÃd', and M. Pettersson, "Measuring multimedia quality in mobile networks with an objective parametric model.," in *ICIP*, pp. 405–408, IEEE, 2008.

[7] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*. USA: Addison-Wesley Publishing Company, 5th ed., 2009.

[8] "ITU-T P.800. Methods for subjective determination of transmission quality - Series P: telephone transmission quality; methods for objective and subjective assessment of quality," Aug 1996.

[9] U. It, "ITU-T recommendation G.114," tech. rep., International Telecommunication Union, 1993.

[10] J. Färber, "Network Game Traffic Modelling," in *Proceedings of the 1st Workshop on Network and System Support for Games*, NetGames '02, (New York, NY, USA), pp. 53–57, ACM, 2002.

[11] T. Szigeti and C. Hattingh, *End-to-End QoS Network Design: Quality of Service in LANs, WANs, and VPNs (Networking Technology)*. Cisco Press, 2004.

[12] "Internet Engineering Task Force." `http://www.ietf.org/`.

[13] R. Braden, D. Clark, and S. Shenker, *Integrated Services in the Internet Architecture: An Overview*. United States, 1994.

[14] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, *An Architecture for Differentiated Service*. United States, 1998.

[15] E. Rosen and Y. Rekhter, *BGP/MPLS VPNs*. United States, 1999.

[16] 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*, Hotnets-IX, (New York, NY, USA), pp. 19:1–19:6, ACM, 2010.

[17] "Open Networking Foundation." `https://www.opennetworking.org`.

[18] B. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, and T. Turletti, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks," *Communications Surveys Tutorials, IEEE*, vol. PP, no. 99, pp. 1–18, 2014.

[19] K. Terplan and P. A. Morreale, *The Telecommunications Handbook*. A CRC handbook, Taylor & Francis, 2000.

[20] A. T. Campbell, I. Katzela, K. Miki, and J. Vicente, "Open Signaling for ATM, Internet and Mobile Networks (OPENSIG'98)," *SIGCOMM Comput. Commun. Rev.*, vol. 29, pp. 97–108, Jan. 1999.

[21] A. Doria, F. Hellstrand, K. Sundell, and T. Worster, "General Switch Management Protocol (GSMP) V3." RFC 3292 (Proposed Standard), June 2002.

[22] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden, "A survey of active network research," *Communications Magazine, IEEE*, vol. 35, pp. 80–86, Jan 1997.

[23] D. L. Tennenhouse and D. J. Wetherall, "Towards an Active Network Architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 81–94, oct 2007.

[24] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A Clean Slate 4D Approach to Network Control and Management," *SIGCOMM Comput. Commun. Rev.*, vol. 35, pp. 41–54, Oct. 2005.

[25] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 105–110, July 2008.

[26] R. Enns, "NETCONF Configuration Protocol." RFC 4741 (Proposed Standard), December 2006.

[27] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern, "Forwarding and Control Element Separation (ForCES)." RFC 5810 (Proposed Standard), March 2010.

[28] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, (New York, NY, USA), pp. 1–12, ACM, 2007.

[29] "OpenFlow Switch Specification Version 1.0.0." `https://www.opennetworking.org`.

[30] A. Lara, A. Kolasani, and B. Ramamurthy, "Network Innovation using OpenFlow: A Survey," *Communications Surveys Tutorials, IEEE*, vol. 16, pp. 493–512, First 2014.

[31] M. P. Fernandez, "Comparing OpenFlow Controller Paradigms Scalability: Reactive and Proactive," *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, vol. 0, pp. 1009–1016, 2013.

[32] "Project Floodlight." `http://www.projectfloodlight.org/floodlight`.

[33] D. Erickson, "The Beacon Openflow Controller," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, (New York, NY, USA), pp. 13–18, ACM, 2013.

[34] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow Software-Defined Networks," in *NOMS*, pp. 1–8, 2014.

[35] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and performance evaluation of an OpenFlow architecture," in *Teletraffic Congress (ITC), 2011 23rd International*, pp. 1–7, Sept 2011.

[36] K. Phemius and M. Bouet, "Monitoring latency with OpenFlow," in *Network and Service Management (CNSM), 2013 9th International Conference on*, pp. 122–125, Oct 2013.

[37] H. Egilmez, B. Gorkemli, A. Tekalp, and S. Civanlar, "Scalable video streaming over OpenFlow networks: An optimization framework for QoS routing," in *Image Processing (ICIP), 2011 18th IEEE International Conference on*, pp. 2241–2244, Sept 2011.

[38] H. E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp, "OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks," in *Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific*, pp. 1–8, IEEE, Dec. 2012.

[39] H. Egilmez, S. Civanlar, and A. Tekalp, "An Optimization Framework for QoS-Enabled Adaptive Video Streaming Over OpenFlow Networks," *Multimedia, IEEE Transactions on*, vol. 15, pp. 710–715, April 2013.

[40] S. Civanlar, M. Parlakisik, A. Tekalp, B. Gorkemli, B. Kaytaz, and E. Onem, "A QoS-enabled OpenFlow environment for Scalable Video streaming," in *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, pp. 351–356, Dec 2010.

[41] P. Georgopoulos, Y. Elkhatib, M. Broadbent, M. Mu, and N. Race, "Towards Network-wide QoE Fairness Using Openflow-assisted Adaptive Video Streaming," in *Proceedings of the 2013 ACM SIGCOMM Workshop on Future Human-centric Multimedia Networking*, FhMN '13, (New York, NY, USA), pp. 15–20, ACM, 2013.

[42] H. Liu, Y. Hu, G. Shou, and Z. Guo, "Software Defined Networking for HTTP video quality optimization," in *Communication Technology (ICCT), 2013 15th IEEE International Conference on*, pp. 413–417, Nov 2013.

[43] A. Ishimori, F. Farias, E. Cerqueira, and A. Abelem, "Control of Multiple Packet Schedulers for Improving QoS on OpenFlow/SDN Networking," in *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, pp. 81–86, Oct 2013.

[44] K. Piamrat, C. Viho, J. Bonnin, and A. Ksentini, "Quality of Experience Measurements for Video Streaming over Wireless Networks," in *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, pp. 1184–1189, April 2009.

[45] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving High Utilization with Software-driven WAN," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, (New York, NY, USA), pp. 15–26, ACM, 2013.

[46] "ITU Telecommunication Standardization Sector." `http://www.itu.int/en/ITU-T/Pages/default.aspx/`.

[47] R. M. Karp, "Reducibility Among Combinatorial Problems," in *Complexity of Computer Computations* (R. E. Miller and J. W. Thatcher, eds.), pp. 85–103, Plenum Press, 1972.

[48] "AMPL, A Mathematical Programming Language." http://www.ampl.com/.

[49] "IBM CPLEX Optimizer." http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[50] "Mininet network emulator." http://mininet.org/.

[51] "VLC Media Player." http://www.videolan.org/vlc/.

[52] "Big Buck Bunny movie." http://www.bigbuckbunny.org/.

[53] "Apache FtpServer." http://mina.apache.org/ftpserver-project/.

[54] C. Chaudet and Y. Haddad, "Wireless Software Defined Networks: Challenges and opportunities," in *Microwaves, Communications, Antennas and Electronics Systems (COMCAS), 2013 IEEE International Conference on*, pp. 1–5, Oct 2013.

[55] I. Ku, Y. Lu, E. Cerqueira, R. Gomes, F. Ongaro, and M. Gerla, "Towards Software-Defined VANET: Architectures and Services," in *Accepted for the Med-Hoc-Net 2014*, June 2014.

[56] S. Salsano, N. Blefari-Melazzi, A. Detti, G. Morabito, and L. Veltri, "Information Centric Networking over SDN and OpenFlow: Architectural Aspects and Experiments on the OFELIA Testbed," *Comput. Netw.*, vol. 57, pp. 3207–3221, Nov 2013.

[57] CISCO, "The Zettabyte Era: Trends and Analysis," 2012.