FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Dynamic Quality-of-Service Management Under Software-Defined Networking Architectures

Rita Martinho

MASTER DISSERTATION

Master's Degree in Electrical and Computer Engineering

Supervisor: Pedro Miguel Santos, Ph.D.

Co-Supervisor: Luís Almeida, Ph.D.

July 30, 2021

Dynamic Quality-of-Service Management Under Software-Defined Networking Architectures

Rita Martinho

Master's Degree in Electrical and Computer Engineering

Approved in oral examination by the committee:

Chair: Professor Ana Aguiar

External Examiner: Professor Pedro Brandão

Supervisor: Professor Pedro Santos

July 30, 2021

Abstract

The Internet is facing new challenges emerging from new trends in Information and Communication Technologies (ICT) for example, cloud services, Big Data, increased mobile usage etc. Traditional IP networks rely in two design principles that, despite serving as an effective solution in the last decades, have become deprecated and not well fit for the new challenges. First, the control and data plane are tightly embedded in the networking devices and second, the structure is highly decentralized with no centralized point of management. In decentralized architectures, the cost of keeping an updated view of the network status across all nodes in order to make (quasi-) optimal resource management decisions often outweighs the benefits of such strategy. This static and rigid architecture leaves no space for innovation with a consequent lack of scalability. Also, it leads to high management and operation costs. Software-Defined Networking (SDN) emerges from the awareness of network research communities and industrial market leaders of the need to rethink the design of the conventional network. The SDN paradigm provides a platform over which is possible to manage modern networks in a more manageable, cost-effective and adaptable way, that is suited for the dynamic nature of today's applications.

After a thorough review of the State of the Art, we found gaps especially in two main fields: the lack of solutions for OpenFlow-enabled devices that do not implement all OpenFlow protocol features (e.g., the support of OpenFlow queues); and the limited number of real-world implementations often hides technical issues or limitations that are not apparent in simulated scenarios and that we report here. Our solution especially contributes to those fields, presenting an hybrid solution - i.e., a solution that leverages from the SDN potential as well as using other QoS mechanisms.

In this dissertation, a novel SDN-enabled solution, the set of SDN applications *xDynApp* is presented. This set of applications aims to dynamically manage the network's bandwidth resources in order to grant Quality of Service (QoS) guarantees to priority applications. This is achieved by using OpenFlow Meters (a SDN solution), the IEEE 802.1p queuing mechanism (a legacy QoS mechanism) and an *admission control* stage, in the case of explicit QoS reservation.

The solution is implemented using the ONOS SDN controller, with the *xDynApp* set of applications being deployed as ONOS applications by using the ONOS Java API. As for the Forwarding Elements that compose the underlying network it was used Aruba 2930M-JL320A switches. The *xDynApp* ultimately proves to be a solid and working solution: besides fulfilling all its behavioural design aspects, when comparing, under congestion scenarios, to a baseline approach, where there is no SDN-enabled solution (a traditional network), *xDynApp* improves the QoS metrics for the priority application. For example, it improves 14 times the Packet Loss Ratio (%), decreases the Jitter and the Inter Packet Arrival Time time values and proves to grant higher bandwidth values to the priority applications.

Keywords: Quality of Service, Real-Time, Software-Defined Networking

Resumo

A Internet está a enfrentar novos desafios que emergem de novas tendências em Tecnologias de Informação e Comunicação (TIC), como por exemplo, serviços na nuvem, *Big Data*, o aumento do uso de serviços móveis, etc. Redes IP tradicionais baseiam-se em dois principais aspectos que, apesar de terem servido como uma solução eficaz nas últimas décadas, têm-se tornado obsoletos e não adequados para os novos desafios. Primeiramente, o plano de controlo e de dados estão totalmente integrados no mesmo elemento de rede e em segundo lugar, a estrutura é altamente descentralizada com nenhum ponto centralizado de gestão. Em arquiteturas descentralizadas, o custo de manter uma visão atualizada do *status* da rede em todos os nós para tomar decisões (quase) ótimas de gerenciamento de recursos, geralmente supera os benefícios de tal estratégia. Esta estática e rigída arquitetura não proporciona espaço para inovação, o que leva a uma consequente falta de escalabilidade. Além disso, leva a altos custos de gestão e operação. Redes Definidas por *Software* (SDN) emergem da consciência da comunidade científica e dos líderes do mercado industrial da necessidade de repensar o *design* das redes tradicionais. O paradigma de SDN providencia uma plataforma sobre a qual é possível gerir redes modernas de uma forma mais gerenciável, económica e adaptável, que está pronta para a natureza dinâmica dos aplicativos de hoje em dia.

Após uma revisão minuciosa do Estado da Arte, verificaram-se algumas lacunas, especialmente em dois campos principais: a falta de soluções para dispositivos habilitados para OpenFlow que não implementam todos os recursos do protocolo (por exemplo, o suporte de filas OpenFlow); e o número limitado de implementações em redes reais, que muitas vezes esconde problemas técnicos ou limitações que não são aparentes em cenários simulados e que relatamos aqui. A nossa solução contribui especialmente para estes campos, apresentando uma solução híbrida - i.e., uma solução que aproveita o potencial SDN, bem como usa outros mecanismos de QoS.

Na dissertação proposta, uma nova solução baseada em SDN, o conjunto de aplicações *xDynApp*, é apresentada. Este conjunto de aplicações tem com como objetivo gerir dinamicamente a largura de banda da rede de forma a oferecer Qualidade de Serviço (QoS) a aplicações prioritárias. Esta meta é alcançada usando os *Meters OpenFlow* (uma solução SDN), o mecanismo de filas IEEE 802.1p (um mecanismo de QoS de base) e uma fase de controlo de admissão, no caso de reserva explícita de QoS.

A solução é implementada usando o controlador SDN ONOS, com o conjunto de aplicações *xDynApp* a serem criadas como aplicações ONOS, usando a Java API ONOS. Relativamente aos elementos de encaminhamento que compõem a rede, são usados os switches Aruba 2930M-JI320A. As aplicações *xDynApp* provam constituirem uma sólida e funcional solução: para além de cumprirem todos os aspectos de *design* comportamentais, ao comparar, em cenários de congestionamento, a uma solução de base onde não há uma solução habilitada para SDN (uma rede tradicional), o conjunto *xDynApp* melhora as métricas de QoS para a aplicação prioritária. Por exemplo, melhora 14 vezes a taxa de perda de pacotes (%), diminui os valores de tempo de *Jitter* e de Tempo de Chegada Entre Pacotes como também concede valores de largura de banda superiores

para aplicações prioritárias.

Keywords: Qualidade de Serviço, Redes Definidas por *Software*, Tempo Real

Agradecimentos

Com o terminar desta fase da minha vida, tenho necessariamente de agradecer aos meus pais, Cristina e Carlos, por todo o apoio - sem eles, nunca teria alcançado o que alcancei e nem seria a pessoa que sou hoje. Queria também agradecer ao resto da minha família por estarem sempre do meu lado e apoiarem sempre as minhas decisões. Não podia deixar de fazer um agradecimento especial ao meu avô Antero, por ser a minha grande inspiração. É verdadeiramente a pessoa mais bonita e humana que conheço.

Agradeço também a todos os meus amigos, académicos e não-académicos. A vida é realmente feita das amizades que fazemos e os meus amigos foram um pilar fundamental no decorrer destes últimos 5 anos. Uma agradecimento sincero ao meu amigo Hugo Guia por toda a ajuda que me deu no ínicio desta dissertação.

Quem me conhece, sabe que os meus animais de estimação são a minha vida. Por isso, não podia deixar de os mencionar - um beijo com saudades para a minha Kika, que partiu recentemente e que tantas horas passou ao meu lado enquanto eu trabalhava para esta dissertação.

Queria mostrar a minha graditão ao meu orientador Professor Pedro Santos por toda a ajuda prestada. O Professor Pedro mostrou-se sempre prontamente disponível para me ajudar ou para esclarecer alguma dúvida que tivesse. Agradeço especialmente pela exigência, confiança e frontalidade. Por último mas não menos importante, um grande obrigada ao meu co-orientador Professor Luís Almeida por partilhar sempre as suas ideias, conhecimentos e também a sua enorme paixão pela área, que muito admiro.

Rita Martinho

"I am the master of my fate,
I am the captain of my soul."

William Ernest Henley

Contents

List of Figures

List of Tables

Abbreviations

5G	Fifth Generation
AC	Access Category
ACK	Acknowledge
AES	Audio Engineering Society
AP	Access Point
API	Application Programming Interface
ARP	Address Resolution Protocol
AS	Autonomous System
AVB	Audio Video Bridging
BSS	Basic Service Set
CLI	Command Line Interface
CPU	Central Process Unit
CSP	Constrained Shortest Path
CSV	Comma-Separated Values
D-ITG	Distributed Internet Traffic Generator
DiffServ	Differentiated Services
DOM	Document Object Model
DoS	Denial of Service
DSCP	Differentiated Services Code Point
EDCA	Enhanced Distributed Channel Access
EDCF	Enhanced Distributed Coordination Function
EDF	Earliest Deadline First
FE	Forwarding Element
FTT	Flexible Time-Triggered
GMB	Guaranteed Minimum Bandwidth
GUI	Graphical User Interface
HPE	Hewlett Packard Enterprise
HTB	Hierarchical Token Bucket
HTTP	Hypertext Transfer Protocol
HaRTES	Hard Real-Time Ethernet Switching
ICT	Information and Communication Techonlogies
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IT	Information Technology
ITA	Inter-Packet Arrival Time
IntServ	Integrated Services
IoT	Internet of Things

JSON	JavaScript Object Notation
KPI	Key Performance Indicator
LAN	Local Area Network
LARAC	Lagrangian Relaxation Based Aggregated Cost
LLDP	Link Layer Discovery Protocol
LTS	Long Term Support
MAC	Media Access Control
ML	Machine Learning
MPEG-TS	Moving Picture Experts Group-Transport Stream
MPLS	Multi-Protocol Label Switching
MTU	Maximum Transmission Unit
NC	Netcat
NDP	Neighbor Discovery Protocol
NFV	Network Functions Virtualization
NOS	Network Operating System
NP	Non-Priority
OF	OpenFlow
ONF	Open Networking Foundation
OOBM	Out-Of-Band Management
OS	Operating System
OSGi	Open Services Gateway Initiative
OVS	Open VSwitch
P	Priority
PCP	Priority Code Point
PoE	Power Over Ethernet
PLR	Packet Loss Ratio
PSNR	Peak Signal-to-Noise Ratio
Q	Queue
QCI	Quality of Service Class Identifier
QoE	Quality of Experience
QoS	Qualilty of Service
RAM	Random Access Memory
RegEx	Regular Expression
RSVP	Resource Reservation Protocol
RTCP	Real-Time Transport Control Protocol
RTP	Real-Time Protocol
SA	Simulated Annealing
SDN	Software Defined Networking
SIP	Session Initiation Protocol
SLA	Service Level Agreement
SPQ	Strict Priority Queuing
STP	Spanning Tree Protocol
SVC	Scalable Video Coding
TC	Traffic Control
TCP	Transmission Control Protocol
TSN	Time-Sensitive Networking
TSSDN	Time-sensitive Software-defined Network
ToS	Type of Service
UDP	User Datagram Protocol
VIP	Very Important Person

VLAN	Virtual Local Area Network
VLC	Video Lan Client
VoIP	Voice Over Internet Protocol
WAN	Wide Area Network
WLAN	Wireless Local Area Network
WMM	Wi-Fi Multimedia
XML	Extensible Markup Language
WRR	Weighted Round Robin

Chapter 1

Introduction

The Internet's growth reported in the last decades has led to a new type of society: a digital one where a world-wide connectivity is both present and needed. Not only that, but the Internet is facing new challenges emerging from new trends in Information and Communication Technologies (ICT) such as Big Data, cloud services, the increased mobile usage etc [80]. Traditional IP networks, despite enjoying, at the moment, a widespread adoption, are complex and hard to manage, decreasing flexibility and hindering innovation and evolution of the networking infrastructure and new technologies. Nowadays networks also face some challenges on scalability, high availability and high bandwidth purposes. In this context, Software-Defined Networking (SDN) emerges as a new paradigm that promises to solve the aforementioned issues.

SDN is becoming the new technological standard for network management (both wired and wireless), taking a central stage in in the upcoming 5G networks [21], [63]. We are observing a trend in which networking devices are becoming commoditized (i.e., homogenized across models and brands) and most of the hardware/physical functionalities are being moved up to the software layer, allowing for remote or automated control. This more agile and flexible paradigm allows to explore new functionalities, such as dynamic adjustment of Quality of Service (QoS) to meet the applications' needs and available resources.

1.1 Motivation

Real Time Applications, applications where timing is critical - i.e, the information sent by a node should be received within a specific deadline by a receiver node - can be found in many aspects of the world. Deadlines are often classified based on the importance of the related application: e.g, transmission of video frames is liable to suffer larger delays (*soft-deadlines*) when comparing to the adjust of a route in a self-driving vehicle (*hard-deadlines*) or even, the firing of a car's airbag (*firm-deadlines*) [52].

In multimedia applications, users are more demanding regarding their Quality of Experience (QoE). However, in those applications, timing is not as critical as in the latter but information must be received in such a way the user does not perceive the delay (for example, video-conferences).

De facto, solutions for provisioning QoS guarantees for multimedia applications are the main objective of this dissertation but more demanding applications could also be supported. The traditional TCP/IP protocol stack-based Internet architecture does not take the aforementioned issues into account - there is no centralized point controlling, on-the-fly, network resource optimizations that lead to the minimum delay of packets. For example, the TCP protocol ensures packets are received and are received in the correct order but there is no specialized mechanism to guarantee timely delivery. In fact, nowadays networks are based in a best-effort service, where there is no differentiation between packets.

New technologies are targeting those issues, especially in the audio/video domain, such as 802.1X - Audio Video Bridging (AVB). Nevertheless, this dissertation envisions a solution that leverages a widely-used open-source tool for SDN - OpenFlow - and implements an algorithm for dynamic QoS management.

1.2 Challenges and Proposed Solution

The Software-Defined Networking paradigm does not natively provide tools to support Quality of Service, however, due to its programmability and flexibility, it is possible to deploy tools to tackle the mentioned challenge. To reach a solution that provides dynamic QoS management for real-time applications that, in the scope of this work, must correspond to priority traffic, this dissertation comprehends the following steps:

- The setup of a small SDN testbed composed of network switches, managed by an open-source network management software (a SDN controller) and with client terminals that produce/consume the data streams;
- Investigation of how to implement and/or use dynamic QoS policies under the SDN management tool with the SDN southbound protocol OpenFlow;
- Run a scenario of multiple realistic client streams, possibly with conflicting requirements, and evaluate the performance of the SDN-based dynamic QoS management;
- Characterization of the performance and advantages of SDN-based management of dynamic QoS against the behaviour of traditional networks.

Some challenges are also set:

- How to identify priority traffic;
- How to identify the source and the destination of the priority traffic;
- What tools can be used in order to differentiate priority traffic from non-priority one;
- How to manage the network resources (e.g. bandwidth) to give QoS guarantees.

The SDN paradigm has been around for some years and there is already some work developed using this concept to provide QoS guarantees to applications in a more scalable, dynamic way (some of those works can be found in Chapter 2). The proposed solution presents a novel set of SDN applications that identifies priority traffic - either by defining a set of pre-defined traffic characteristics that matches priority ones or by having end-users defining what corresponds to a priority communication. After, the SDN applications proceed to grant QoS guarantees to those priority traffic by managing the available bandwidth on the network. Furthermore, this solution also incorporates an *admission control* stage. The complete design of the set of SDN applications is explained in Chapter 3 and its actual implementation in Chapter 4.

This dissertation is on the scope of the Bosch's project *SafeCities* [2]. This project aims to respond and anticipate the challenges facing modern urban societies, increasingly dependent on technological developments in the fields of remote sensing, data transmission, storage and intelligent processing, supplying the need for safety, privacy, comfort and efficiency.

1.3 Contributions

This dissertation aims to contribute to QoS management in Software Defined Networking Architectures in 4 ways:

1. Development and implementation of two novel mechanisms using SDN tools for (i) enforcing, network-wide, traffic prioritization and (ii) offering QoS reservation (specifically bandwidth) to flows indicated by the user.

2. Usage of a combination of legacy QoS mechanisms and OpenFlow-driven ones, allowing OF-compatible equipment that do not have the full set of OF functionalities, particularity the non-mandatory OF queues, to still be able to offer some level of QoS services;

3. Creation of a protocol that allows to introduce in the algorithm an *admission control* stage;

4. Deployment of the solution over a real, not simulated network testbed and a quantitative and qualitative evaluation of the solution's performance.

1.4 Document Structure

The rest of the present document is structured in Chapters, each one with a specific purpose and divided in different Subsections to allow an organized reading.

Chapter 2 gives not only a summary of SDN concepts and other standards for QoS enforcement but also a comprehensive study of the State of the Art. This study is based on a created taxonomy and converges in a final Subsection which aims to understand the framing of this dissertation on the current State of the Art.

Chapter 3 enforces the characterization of the problem this dissertation addresses, as well as, it provides a complete and detailed presentation of the design of the developed SDN application. Furthermore, why some design choices were made are explained.

Chapter 4 demonstrates how the developed SDN applications were implemented - first, its is explained how it was implemented over the assigned physical testbed and second, how it was software-implemented.

Chapter 5 is where the results obtained in this dissertation can be found. 6 experiments are presented - some correspond to experiments that aim to evaluate the proposed solution, others correspond to experiments that aims to validate the algorithms' behaviour.

Chapter 6 finalizes this dissertation by making the final conclusions one can obtain from the developed work. It also makes a brief summary of the workflow of this dissertation and presents possible future work.

Chapter 2

State of the Art and Background

The present State of the Art is composed of 3 main Sections. Section 2.1 gives an overview on SDN and related concepts that the reader needs to know in order to understand the subsequent sections. In Section 2.2 some standards/models for QoS enforcement in Multimedia Applications are presented. Finally, Section 2.3 gives a detailed and structured analysis of the already developed work on the same sphere of this dissertation, converging in the positioning of the on-work dissertation in the State of the Art.

2.1 Overview of SDN

Regarding the present Section, first, in SubSection 2.1.1 some *status quo* concepts about nowadays computer networks and why they are no longer a feasible solution for many aspects such as, management, scalability and innovation are introduced. Subsection 2.1.2 presents a widely accepted definition for SDN and also a detailed description of the most important SDN layers (in a bottom-up approach). Finally, Subsection 2.1.3 presents some important architectural aspects of the chosen SDN controller that was used during the developed work, the ONOS controller.

2.1.1 Management Approaches in Networking

It is usual to divide computer' networks in three planes according to their functionality:

- The data plane: it is essentially the plane responsible for efficiently carrying/forwarding user traffic (in some literature this plane is also known as *user plane*). This plane also corresponds to the networking devices.

- The control plane: can be viewed as the plane where protocols logically reside i.e., the plane responsible for all activities that are necessary to perform data plane activities such as making routing tables, address assignment, etc.

- The management plane: it is the plane that includes software services comprehending activities related to provisioning and monitoring of the network such as the ones described in the Fault, Configuration, Counting, Performance and Security (FCAPS) framework [56] .

As described in [54], this relationship can be summarized as: the network policies (operations rules) are defined in the management plane, the control plane enforces those policies and the data plane is responsible for the execution of the policies, forwarding the data accordingly. A graphical schematic about this layered view of functionalities planes is depicted in Figure 2.1.

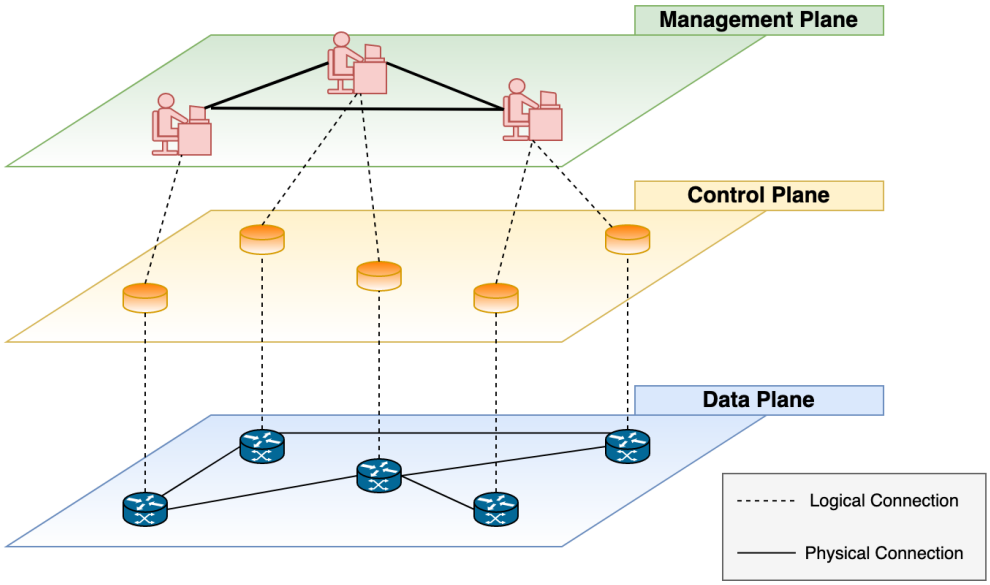

Figure 2.1: Layered View of Functionalities Planes

Conventional IP networks rely in two main aspects that, despite serving as an effective solution in the last decades, have become deprecated. The two main aspects are: the control and data plane are tightly incorporated and embedded in the networking devices and there is no centralized point of management as the whole structured is decentralized. This static and rigid architecture has become responsible for a vertically-integrated, vendor specific industry which leaves no space for innovation and leads to some inertia - for example, the IPv6 (a protocol update) was introduced in 1995 and, twenty-six years later, the transition from IPv4 to IPv6 is not yet complete [54]. Not only these type of architectures hinder innovation and scalability but they also lead to high management and operation costs. Networks misconfigurations or even monitoring become a not trivial problem because network management and debugging are tedious and often frustrating tasks. These problems are often solved with the help of a myriad of specialized components, often called *middleboxes* but they ultimately lead to a growth in the network complexity and crystallization of the current *status quo*.

Software-defined networking and, in a more general fashion, the *programmable networks* paradigm, arises from the awareness of network research communities and industrial market leaders of the need to rethink the design of conventional networks [49]. In fact, one of the major

examples of this new technology adoption in a large-scale network is the B4: Google's private WAN, which connects Google Data centers across the world via SDN. [48]

2.1.2 SDN Definition and Architecture

The Open Network Foundation (ONF) is is a non-profit consortium dedicated to the development, standardization, and commercialization of SDN and they provide a widely accepted definition for this new paradigm [37]:

"*Software-Defined Networking (SDN) is an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today's applications. This architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services. The OpenFlow protocol is a foundational element for building SDN solutions.*"

Kreutz et al. in [54] also define SDN as a network architecture with four pillars:

1. The control and data plane are decoupled and so, control functionality is removed from network devices that become *brainless* packet forwarding elements. Those devices are also often called *whiteboxes*.
2. Forwarding decisions are flow-based, instead of destination-based i.e., all packets of a flow [1] receive identical service policies at the forwarding devices.
3. Control logic is moved to an external entity, the SDN controller or Network Operating System (NOS).
4. The network is *programmable* through software applications that run on the top of the SDN controller interacting with the underlying data plane devices. The authors define this pillar as being the principal proposition of the SDN concept.

Moreover, a SDN architecture comprehends 8 main layers:

1. Network Infrastructure;
2. Southbound Interface;
3. Network Hypervisor;
4. Network Operating System (SDN Controller);
5. Nourthbound Interface;
6. Language-based Virtualization;

[1] Defined as the set of packets that match some filter criterion.

7. Programming Languages;

8. Network Applications.

Figure 2.2 depicts the SDN architecture as well as the layers comprehended in each plane. Of the eight layers of a SDN architecture, it will be analysed in more detail the following layers, as they are the most relevant ones in the scope of this dissertation: Network Infrastructure layer, Southbound Interface layer (with an emphasis on OpenFlow protocol), the SDN controller layer and the Network Applications layer.

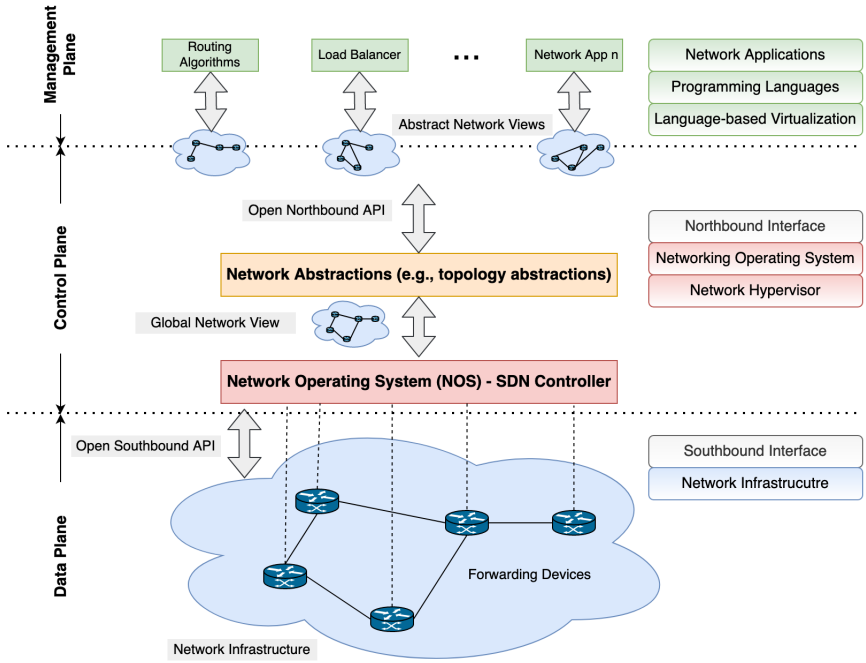

Figure 2.2: SDN Architecture (Left) and SDN Layers (Right), based in [54]

2.1.2.1 Network Infrastructure

Also known as the data plane in the SDN context, it comprehends principally the Forwarding Elements (FE) - components in software [15] or hardware- meaning that those devices simply forward packets without any kind of embedded control or any sort of autonomy. The network's intelligence is moved to the SDN Controller/NOS that becomes the *network brain*. It is important to mention, however, that the infrastructure, itself, is the same as in traditional networks - a set of networking equipments.

One main and important difference between the SDN paradigm and previous network implementation resides in the fact that SDN specifies that the infrastructure should be built on top of an open and standard interface. This represents a pivotal approach since it allows configuration, communication compatibility and interoperability [54], between the Controller and the FE. This

surpasses the difficulty of programming and controlling miscellaneous FE which often are built under closed and vendor-specific interfaces.

2.1.2.2 Southbound Interface & OpenFlow

As described in 2.1.2.1, in a SDN architecture, control and forwarding network equipments are connected via an open and standard interface. In SDN literature this interface is often called *southbound interface* or *southbound API* and acts as the separator of control and data planes.

A wide number of open protocols are available as southbound interface for SDN (such as ForCES, presented by the IETF[2] or POF [59]) but on this brief theoretical review, only one is being revised: **OpenFlow**. As of this writing, it is the most accepted and widely used southbound interface for SDN [57].

OpenFlow is currently on its 1.6 version and the ONF is the responsible entity for its standardization. OpenFlow is a flow-oriented protocol and it is essentially supported both by the *OpenFlow Switch* (the forwarding device) and the OpenFlow-enabled controller. An *OpenFlow switch* has at least 3 components [45],[38] :

1. Flow table(s);
2. Group tables, which perform packet lookups and forwarding;
3. OpenFlow channel, which permits the communication to the aforementioned controller.

Each entry of each flow table has three parts: 1) a matching rule, 2) counters that keep statistics of the matching packets and 3) a set of instructions/actions that are executed when a packet matches a rule. The OpenFlow-based controller can add/delete/update flow-entries in flow tables.

OpenFlow provides a wide set of options for matching rules, such as by switch port, MAC source/destination, VLAN ID, TCP/UDP source or destination port, MPLS label, etc. However, only a few of those are actually mandatory to be supported to a given protocol version. These main matching rules can be combined to define a flow rule.

Regarding the possible actions, it is important to mention the four most relevant:

1. Sending packets to port(s) - physical or logical;
2. Encapsulating a flow's packets and sending to the controller;
3. Sending a packet to a *normal* processing pipeline (i.e. a non-OpenFlow pipeline);
4. Dropping a packet.

When a new packet arrives to the switch, the lookup process starts by verifying if there is any matching rule valid in the very first table. This process is repeated in every flow table (following a priority based on the table position in the tables' pipeline) and can also result in a miss when

[2] A comparison between OpenFlow and ForCes can be found in [61] and [78]

the packet does not match the criteria defined by neither of the flow tables. In a miss occurrence, the packet can be dropped or, in the most common case, sent to the controller or to the normal processing pipeline for a further evaluation.

OpenFlow version 1.3 introduces the meter table implementation. A meter table is made up of meter entries which define per-flow meter i.e. meters are attached directly to flow entries. A flow can specify a meter in its instruction/action set and the meter measures and controls the rate of the aggregate of all flow entries to which it is attached.

A meter is composed of three components:

1. **Meter identifier**: a 32 bit unsigned integer that identifies the meter.
2. **Meter Bands**: a list of meter bands where each meter band states the rate of the band and how to process the packets. A band can be of the type *drop* - which discards the packets and can be used to define a rate limiter band; and the type *dscp remark* which decreases the drop precedence of the DSCP field in the IP header of the packets.
3. **Counters**: updated number of processed packets.

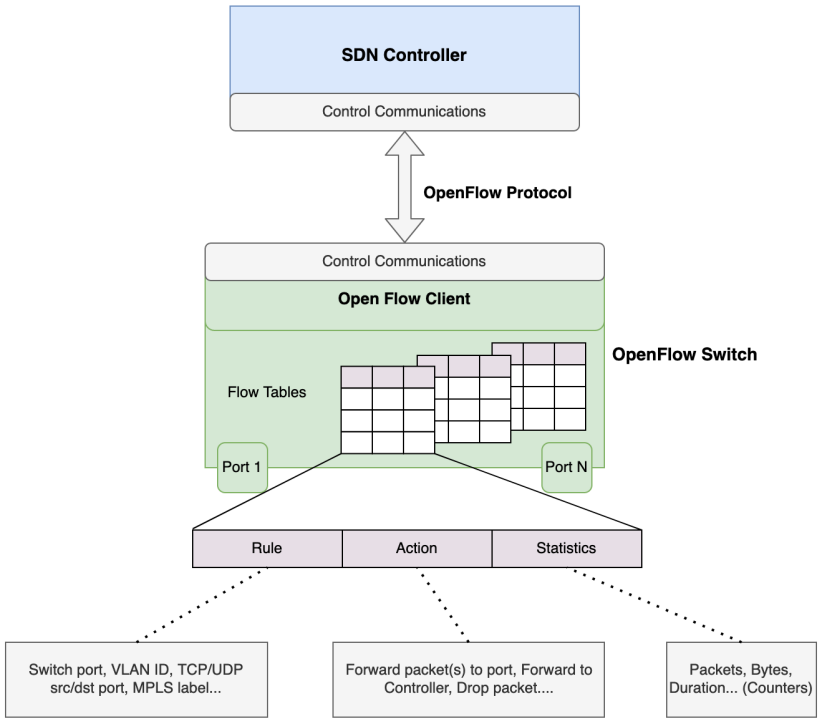

Figure 2.3: OpenFlow Communication and OpenFlow Flow Tables

As depicted in [54] and by way of conclusion, it is possible to say that OpenFlow provides three information sources, all three with origin in the FE and as the destination, the controller:

- Event-based messages, triggered by a port or link change;

- Flow statistics;
- Packet-in messages regarding the last two action described in the actions-list.

2.1.2.3 SDN Controller

The SDN Controller presents a crucial piece of the SDN architecture as it is the principal responsible for a programmatic interface that allows control and configuration of the network based on the policies defined by the network operator. As described before, the controller acts as the *network brain* since it is the logically centralized control entity, serving as the operating system (OS) for the network (NOS). Moreover, the SDN controller is agnostic to the lower-level details of the devices in the layer below (the FE).

As reported in Section 2.1.2.2, the controller communicates with the forwarding elements via a *southbound interface*. Parallelly, the controller communicates with network applications via a *northbound interface*. However, contrary to what happens to the southbound one, there is no widely accepted standard such as OpenFlow. The ONF, in 2013, created a working group focused specifically on northbound APIs and their development. This difficulty of creating a standard interface can be explained by the variety of network applications with diverse requirements and functionalities.

From a architectural point of view, SDN controllers can be divided in two groups:

- Centralized: In this type of architecture, there is a single entity that manages the forwarding elements. Even though representing a simpler solution, this also represents a single point of failure (bottleneck) and can create scalability problems since one controller may not be able to manage a large number of FE.
- Distributed: Representing the opposite of the above architecture type, in a distributed one it is possible to have a centralized cluster of controllers or even a wide number of physically distributed ones. This set of clusters can communicate via a west/east southbound API. This type of architecture obviously leverages some advantages such as being suited for adoption in a small or large scale network, being capable of accommodating requirements of any environment and being more fault-tolerant.

From the logical design point of view, Wenfeng Xia et al. in [80] define the SDN controller as being decoupled into four building components:

1. High Level Language: for SDN applications to define their network operation policies;
2. A rule update process: to install rules generated from the mentioned policies;
3. A network status collection process: to collect network infrastructure information;
4. A network status synchronization process: to build a global network view using network status collected in the controller(s).

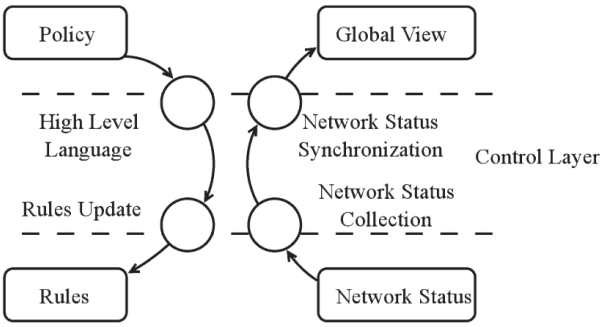

Figure 2.4: Controller Logical Design as depicted in [80]

As it is of paramount importance for any SDN testbed-study the correct choice of the controller (or controllers) in order to meet the study requirements, a brief and structured analysis of the main important controllers on the market is provided in Table 2.1.

Name	Type of Architecture	Northbound API	Program. Language	OpenFlow Support	Interface	Open Source	Platform Support	Documentation
Onix [53]	distributed	Onix API	C++	yes	-	no	-	Limited
Beacon [34]	centralized	ad-hoc API	Java	yes	CLI, Web UI	yes	Linux, MacOS, Windows	Fair
NOX [42]	centralized	ad-hoc API	C++	yes	CLI, Web UI	yes	Linux	Limited
POX [32]	centralized	ad-hoc API	Python	yes	CLI, GUI	yes	Linux, MacOS, Windows	Limited
Maestro [27]	centralized	ad- hoc API	Java	yes	Web UI	yes	Linux, MacOS, Windows	Limited
Open Day Light [76]	distributed	REST, REST-CONF	Java	yes	CLI, Web UI	yes	Linux, MacOS, Windows	Good
ONOS [14], [24]	distributed	RESTful API	Java	yes	CLI, Web UI	yes	Linux, MacOS, Windows	Very Good

Table 2.1: SDN Controllers Comparison

2.1.2.4 Application Layer

Yosr Jarraya et al. in [49] define this layer as embodying two main fields:

- SDN/Network Applications;
- SDN uses cases.

The network applications implement the control-logic that will eventually be translated to data plane commands (dictating the forwarding elements behaviour), fulfilling the network operator needs as they allow the management of specific network functionalities. In SDN literature it is common to group these network applications in five high-level groups:

1. **Traffic Engineering** including: Load Balancing, Traffic Optimization, QoS Enforcement, etc;
2. **Mobility and Wireless** including: Dynamic Spectrum Usage, on-demand Virtual Access Points, seamless mobility through efficient hand-overs, Downlink Scheduling, etc;
3. **Measurement and Monitoring** including: improving the visibility of broadband performance, reduction of control plane overload, etc;
4. **Security and Dependability** including: Access Control, Denial-of-service (DoS) attacks detection and optimization, Traffic Anomaly Detection, etc;
5. **Data Center Networking** including: Live Networking Migration, Eminent Failure Avoidance, Optimization of Network Utilization, etc.

SDN applications can be seen as means to serve a specific use case in a given enforcement. Common SDN uses-cases include: SDN for cloud computing; SDN for Information Content Networking (ICN); SDN for mobile networks (with special relevance to the 5G use case); SDN for network virtualization and SDN in a Network Functions Virtualization (NFV) environment.

2.1.3 The ONOS controller

The Open Network Operating System (ONOS) [14] is an open source SDN controller hosted by the ONF capable of managing and configuring an entire network. ONOS, among other functionalities, provides APIs, abstractions and a complete CLI and GUI. The ONOS kernel and its applications are written in Java as bundles that are loaded into the Karaf OSGi container that runs and is installed as a single JVM - allowing ONOS to run on most of the well known Operating Systems like GNU/Linux or Windows. As the ONF describes, ONOS is indeed an *extensible and modular* controller - its architecture is divided modularly and each module exposes APIs to allow another modules to interact with it (eg. Northbound/Southbound API/REST API). It is important to note that the referred modules are often protocol and vendor-agnostic, allowing ONOS developers to independently extend or improve ONOS. ONOS developers can simply create ONOS applications by using the extensive ONOS Java API [13]. However, for the integration of built applications, developers need to compile and run ONOS source code by using the Bazelisk tool (a wrapper for Bazel) and the ready-to-use script *onos-create-app*, for the application creation.

The SDN high-level, general architecture discussed in 2.1 is very similar to the one exposed in the ONOS wiki as depicted in Figure 2.5 and in Figure 2.6.

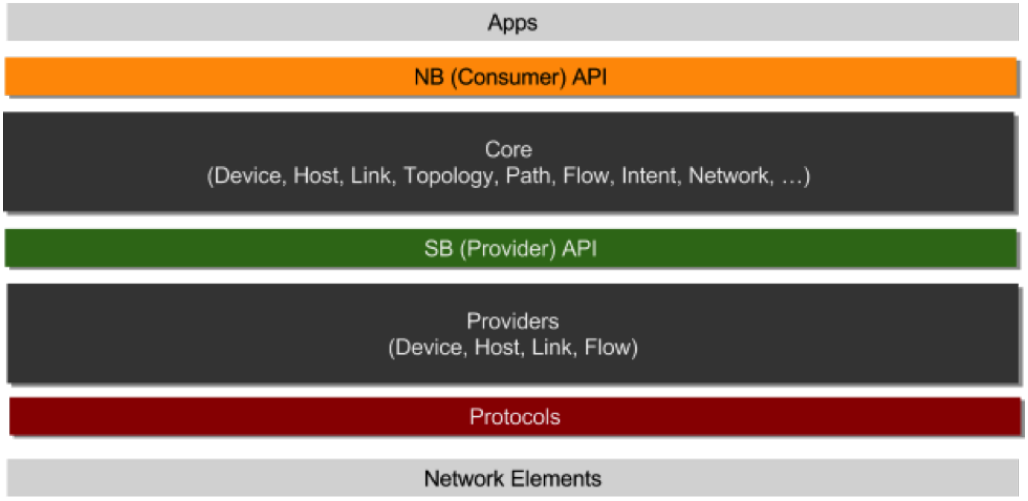

Figure 2.5: ONOS Architecture

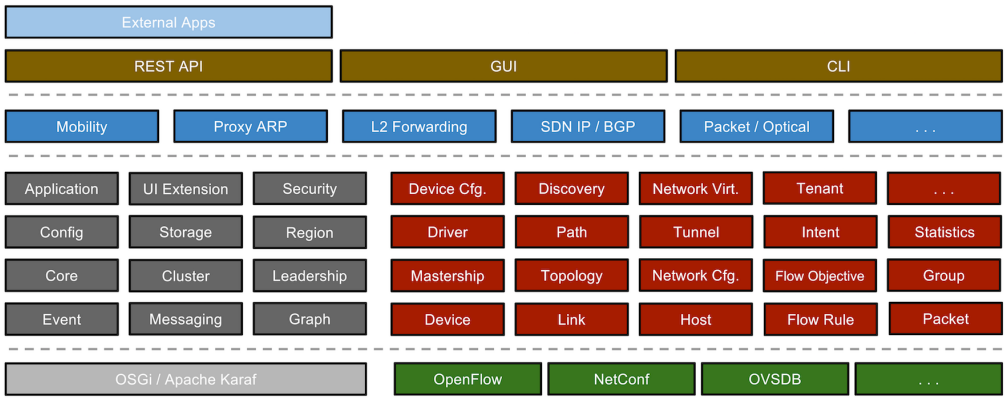

Figure 2.6: ONOS Subsystems

The ONOS system, being composed of layers of functionalities, is mainly divided into *services* and *subsystems*. A service represents a *unit of functionality that is comprised of multiple components* and a subsystem the *collection of components making up the service*. As such, ONOS defines a set of primary services, as described in the ONOS wiki:

- *Device Subsystem*: manages the inventory of infrastructure devices.
- *Link Subsystem*: manages the inventory of infrastructure links.
- *Host Subsystem*: manages the inventory of of end-station hosts and their locations on the network.
- *Topology Subsystem*: manages time-ordered snapshots of network graph views.
- *Path Service*: computes/finds paths between infrastructure devices or between end-station hosts using the most recent topology graph snapshot.

- *FlowRule Susbystem*: manages inventory of the match/action flow rules installed on infrastructure devices and provides flow metrics.
- *Packet Subsystem*: allows applications to listen for data packets received from network devices and to emit data packets out onto the network via one or more network devices.

As it is a well-documented, powerful controller and as it is largely discussed on the web (eg. in various Google groups), ONOS was decided to be the used one throughout the developed work on this dissertation.

2.2 Standards for QoS enforcement in Multimedia Applications

The Quality of Service (QoS) is a measurement of the overall performance of a specific service. QoS, in computer networks, is often quantitatively measured using metrics that can define the network service quality, such as: delay, jitter, bandwidth, packet loss rate, availability etc. It should be highlighted that QoS does not have always the same meaning i.e. different applications require different QoS requirements and thus different QoS models or technologies to provide guarantees. This Subsection aims to briefly review some other already existing technologies or models that guarantee QoS requirements for the multimedia scenario.

2.2.1 IntServ and DiffServ

The Internet Engineering Task Force (IETF) has provided various QoS architectures/models to delivery QoS guarantees to applications that run on top of the traditional networks: *best-effort* ones. A best-effort network specifies that an application can produce traffic without notifying the network but the network does not provide any sort of performance guarantee as every packet is treated the same.

One of those architectures presented by the IETF is the Integrated Services (*IntServ*) model [30]. This model is oriented towards QoS support for individual flows. Resources are explicit reserved (in an end-to-end path fashion) with the help of the Resource Reservation Protocol (RSVP) and every router must store per flow information (*soft sate*). *IntServ* also defines two service classes (besides the best-effort):

- Guaranteed Service: for applications that require a pre-defined packet delay that must be warranted.
- Controlled-Load Service: for applications that are tolerant and that can adapt to occasional packet losses.

This model obviously lacks scalability and arises complexity issues and, because of that, the research community presented a new model: the Differentiated Services (*DiffServ*) [26] which works on a flow-aggregation basis, defined by a small number of classes. This model implements service differentiation through per-hop behaviour (PHB) with each packet carrying indication of

which service class it belongs to - an information provided by the 6-bit differentiated services code point (DSCP) in the 8-bit differentiated services field (DS field) which can be found in the IP header of the packet. Each traffic class may map to different hop-behaviours, for example, it is recommended that the DSCP field 0 maps to the Default Forwarding PHB represented by best-effort traffic, without no requirements. On the other hand, the DSCP field 46 is the recommended one for the classification of the Expedited Forwarding PHB - representing low-loss, low-delay and low-jitter - suitable for real-time services.

Even though this model solves the scalability issue of *IntServ* (there is no need for each router to maintain information about a specific flow, e.g.) it is too simplistic, providing a per-class treatment and not per-flow, which is often desired for applications such as multimedia ones. Given the advantages and disadvantages explained before, it is clear that those architectures are not fully successful at QoS support, serving as mere patches to the traditional networks. The Multiprotocol Label Switching (MPLS) [58] [79] provides a partial solution since it allows for a quick switching (guaranteeing, e.g, delay requirements).

2.2.2 Real Time Protocol and Real Time Control Protocol - RTP/RTCP

The Real Time Protocol (RTP) [46] was first introduced by the Internet Engineering Task Force (IETF) in 1996. RTP is a flexible real-time transport protocol over multiple clients-IP networks using the UDP protocol. It provides end-to-end transport functions suitable for applications such as audio, video or streaming media (video-conferences, television services [3], etc). The aforementioned functions encompass jitter[4] compensation, detection of packet loss and out-of-order delivery.

RTP is often used in conjunction with the Real Time Control Protocol (RTCP). RTP carries the actual media streams while RTCP is the responsible for monitoring statistics of QoS, for providing means for adaptation (flow control) and for providing streams synchronization. RTCP also supplies periodic feedback to all members of an RTP session.

Even though RTP/RTCP has been, over the last decades, a valid solution for applications with strict time-delivery delays [72], [28], [71], it is, however, still highly coupled with traditional IP networks inheriting all the problems of those networks as described in 2.1.1.

2.2.3 Audio Video Bridging - AVB

Audio Video Bridging [6], [67] refers to the set of of IEEE technical standards (*IEEE 802.1Q*) aiming to provide improved synchronization, low-latency, and reliability for switched Ethernet (layer-2) networks. Since 2012, this set of standards belong to the scope of the Time-Sensitive Networking (TSN) task group. AVB allows the creation of a single network for audio, video, and other data like control information, using an AVB-compatible switch. This allows to mix normal network data and audio data in the same network (much like *network slicing*). The AVB

[3] The delivery of television content over Internet Protocol (IP) networks.

[4] Jitter is a statistical variation of the delay in delivering data over a network.

networking offers low-predictable latency (≈ 2ms) for audio in a seven-hop maximum, 100Mbps network [67]. AVB also guarantees reserved bandwidth.

Despite having different focus (SDN being a generic paradigm, AVB a QoS-dedicated solution), some comparisons can be draw: firstly, most AVB solutions are proprietary (requiring specific equipments) and SDN aims to deliver open-source solutions. Secondly, AVB defines a set of QoS solutions for audio only while SDN-based solutions can be deployed to guarantee QoS to any type of traffic. Thirdly, AVB exists in the extension of Local Area Networks (LAN) (ethernet) whereas SDN imposes no restrictions.

2.2.4 IEEE 802.11e Wi-Fi Multimedia - WMM

Wi-Fi Multimedia (WMM) [22] is proposed by the Wi-Fi Alliance as a profile of the 802.11e standard. It essentially provides basic QoS features for 802.11 networks i.e. wireless networks, differing from the technologies mentioned above. WMM is based on Enhanced Distributed Coordination Function (EDCF) and defines 4 Access Categories (AC) - levels of priority - for the Enhanced Distributed Channel Access (EDCA): (i) voice - `AC_VO` (ii) video - `AC_VI` (iii) best–effort - `AC_BE` (iv) background - `AC_BK`. Each access category uses pre-defined 802.11e fields (such as CWMin,CWMax and AIFS) and those fields map to a specific priority.

2.2.5 IEEE 802.1p

The 802.1p refers to the IEEE technique that implements QoS at the media access control (MAC) level. Eight classes of service are defined via the 3-bit Priority Code Point (PCP) that can be found in the Ethernet frame header in a VLAN tagged environment (as per defined by the IEEE 802.1Q standard). There is no particular rule for the way traffic is treated when assigned to a class but IEEE recommends that, for example, the PCP 0 maps to a 0 (lowest) priority and can be used by background traffic whereas the PCP 7 maps to a 7 (highest) priority and can be used by Network Control traffic.

2.3 Challenges in QoS enforcement for SDN Solutions

As explained in 2.1.1, SDN is an emerging paradigm and due to its capabilities depicted in 2.1 it has proven in several studies that it is a strong solution for QoS provisioning [51] [25].

This Section aims not only to study some SDN-based QoS solutions for real-time traffic (with special emphasis to the multimedia scope) in the scientific community, but also to understand the framing of this dissertation in the state of the art.

2.3.1 Research Methodology

Firstly a broad research of papers regarding QoS guarantees enabled by SDN on the main scientific databases (IEEE Xplore, ScienceDirect, Scopus etc) was made. Some importance was given to the

papers that were most cited by others, as well as their references. On a first step, approximately 50 references were collected.

To shorten the above number, it was decided to do a deeper reading of the abstract (to understand what references would be more relevant) and also, it was decided not to integrate on this study of the State of the Art articles that were older than 2010 as this research area is a rapidly growing one, converging in 25 references.

To structure this study, a references' taxonomy was created as shown in Table 2.2 and each component is listed below:

- The year;
- The application scenario;
- The type of solution implemented;
- The most important QoS metrics involved: delay, bandwidth, loss and jitter (the symbol $\surd$ indicates that the reference provides some mechanism or shows results for that specific metric);
- The *closeness level* of each reference regarding this dissertation, on a scale from 1 to 6 and considering the following points:
  - 1 pt : Experiments on a real testbed;
  - 1 pt : Study on the multimedia scope;
  - 1 pt : Some sort of QoS algorithm;
  - 1 pt : Hybrid solution - legacy QoS Mechanisms + SDN mechanisms;
  - 1 pt : Usage of an open-source controller;
  - 1 pt : Focus on delay guarantees;

All references have a common goal: the provisioning of QoS guarantees and so, the *Type of Solution* columns regards to that specific challenge. With respect to the next Subsections, where a more extensive study of each reference is performed, the references are grouped by its application scenario. It was also chosen to restrict to references with a *closeness level* equal or higher of 4, unless, of course, if even though the closeness is lower, important results are presented.

Reference	Year	Application Scenario	Type of Solution	Closeness Level	Delay	Bandwidth	Loss	Jitter
[74]	2016	Multimedia Delivery	AP admission control	5	√	√	-	-
[33]	2012	Multimedia Delivery	QoS Routing	5	√	√	√	-
[29]	2010	Multimedia Delivery	Network Calculus	5	√	-	√	-
[62]	2017	Multimedia Delivery	Crafted SDN application	4	-	√	-	-
[82]	2015	Multimedia Delivery	QoS routing	4	√	-	√	√
[81]	2015	Multimedia Delivery	QoS Routing	4	√	√	√	-
[65]	2014	Multimedia Delivery	QoS Routing	4	√	√	-	√
[77]	2014	Multimedia Delivery	New SDN/OpenFlow architecture	4	√	√	-	-
[70]	2019	Industrial Networks	METRICS architecture	5	√	-	-	-
[75]	2017	Industrial Networks	METRICS architecture	4	√	√	-	√
[60]	2019	Industrial Networks	METRICS architecture	4	√	√	-	√
[41]	2017	Industrial Networks	Network Calculus	2	√	-	√	-
[31]	2018	IoT Networks	QoS Routing	5	√	√	√	√
[68]	2014	IoT Networks	Network Calculus	2	√	√	-	√
[47]	2017	Telecommunication Networks	Machine Learning Algortihm	4	√	√	√	√
[35]	2018	Virtual Private LAN Service	Authentication Module	3	-	√	-	-
[43]	2019	In-Vehicle Networks	TSN & SDN integration	3	√	√	-	-
[44]	2018	Non-Defined: Real-Time Systems	QoS Module	3	-	√	-	-
[23]	2016	Non-Defined: Real-Time Systems	Priority-Adjustment schemes	3	√	-	-	-
[55]	2017	Non-Defined: Real-Time Systems	Heuristic Algorithm	2	√	√	-	-
[40]	2014	Non-Defined: Real-Time Systems	Network Calculus	2	√	√	-	-
[39]	2018	Non-Defined: Real-Time Systems	FlowQoS mechanism	2	√	√	-	-
[73]	2014	Non-Defined: Real-Time Systems	Inter/intra-AS QoS framework	2	-	√	-	-
[50]	2015	Non-Defined: Real-Time Systems	Inter-AS QoS Routing	1	-	√	-	-
[69]	2014	Non-Defined: Real-Time Systems	Crafted SDN application - monitoring	1	-	√	-	-

Table 2.2: References Comparison According to Pre-Defined Taxonomy

2.3.2 Multimedia Delivery

Access Control with ONOS Controller in the SDN Based WLAN Testbed

The research done by Jung Wan Shin et al. in [74] emerges from the authors' realization that most existing studies in the scope of SDN in the wireless domain is residual, as they are basically theoretical assumptions. Therefore the authors present an application to provide QoS guarantees for a high-priority group based on access control by an WLAN AP.

The ONOS controller is chosen because from the analysed open-source controllers, ONOS is capable of "providing scalability, high availability, performance in terms of throughput and latency, easy to support new network services, and providing SDN control for OpenFlow enabled legacy devices". The remaining components of the proposed environment are (i) 3 OpenFlow switches, employed using Raspberry Pi and OpenvSwitch package; (ii) 2 WLAN Access Points, employed using Raspberry Pi and hostapd package and (iii) end stations.

To guarantee the quality of service for high priority users, the authors propose a network application based on four parts:

- *Traffic Monitoring:* The controller collects information and when the resource utilization of the APs is higher than a pre-defined value, the traffic monitoring application sends a message down to the WLAN AP.
- *Flow Table Update Based on order of priority:* The authors use 2 flow tables, one for the access list and another for the deny list. The users are categorized as one of the follow: VIPs, members, black list members and none. Their access is allowed or denied according to a set of rules based on the mentioned pre-defined value of resource utilization (for example, VIP members are granted with full-time access no matter the resource utilization).
- *WLAN AP Control:* Based on the flow tables definitions and rules, the controller has to manage the WLAN AP access mode, via OpenFlow messages.
- *MAC Filtering:*The priority of each user is matched accordingly to its MAC address.

Even though the authors propose an innovative approach to wireless access control using SDN logic, they never show concrete results originated from concrete experiments and so, it can be considered as a lack that may encourage future development.

OpenQoS: An OpenFlow Controller Design for Multimedia Delivery with End-to-End Quality of Service over Software-Defined Networks

Hilmi Elgimez et al. in [33] propose a novel OpenFlow controller design for multimedia delivery with end-to-end QoS support, named OpenQoS and implemented over the FloodLight controller [36]. With OpenQoS, a new prioritization scheme is proposed, based on routing. Multimedia flows are routed based on a QoS routing scheme while other flows are routed based on the best-effort model, the shortest path. This approach does not use resource reservation nor priority queuing and so it allows the adverse effects of QoS provisioning on non-QoS flows (latency and packet loss) to be minimized.

OpenQoS exploits the OpenFlow forwarding mechanism paradigm to identify multimedia flows among the others. Some match fields are considered: the MPLS class header, ToS field in IPv4, etc. The dynamic QoS routing problem is exposed as a Constrained Shortest Path (CSP) problem: the network is represented as a simple graph, composed of nodes and links and the route is found so that it minimizes a cost function (based on delay and congestion at each link). For this route calculation function, the authors proposed the usage of the Lagrangian Relaxation Based Aggregated Cost (LARAC) algorithm.

To test the above explained algorithm, a real testbed is implemented. This testbed is composed of 3 OpenFlow enabled switches (in a triangular shape), one SDN controller and 3 host computers. The authors run several tests, ones using UDP streaming and others using adaptive HTTP streaming. All tests are compared based on the peak signal to noise ratio (PSNR) of the receiver stream with respect to the original raw video. From the experiments, three major conclusions are inducted, revealing the robustness of OpenQoS :

1. OpenQoS working alone with TCP, outperforms the HTTP-based multi-bitrate adaptive streaming, even under congestion scenarios.

2. OpenQoS can guarantee seamless video delivery even when an unreliable transport protocol (UDP) is used.

3. OpenQoS can guarantee full video quality when TCP is used.

The most relevant drawback of OpenQoS is PSNR degradation in high traffic networks, because the rerouting scheme can not work elaborately with the increase of network load.

A QoS-Enabled OpenFlow Environment for Scalable Video Streaming

The work developed in [29] by Seyhan Civanlar et al. exploits the SDN paradigm to guarantee QoS guarantees for video streaming applications.

The authors start by expressing that every QoS routing algorithm must take into account the existence of best-effort traffic and associated congestion. A liner programming problem is used to determine the QoS flow routes assuming entirely lossless QoS traffic patterns. The algorithm imposes that best-effort traffic is always routed on the shortest path while the QoS traffic is possibly rerouted - with strict delay bounds. One important system's feature is the existence of forwarding elements that act as *polices* to make sure the end points conform to their Service Level Agreements (SLAs) stated in the QoS contract, received from the streaming server. These FEs report the QoS health status to the SDN controller - that act accordingly.

The aforementioned algorithm is validated over a real testbed composed of 4 OpenFlow-Enabled forwarders, 1 SVC streaming server, 1 SVC streaming client and 1 SDN controller. The forwarders are positioned in a topology allowing for 2 physically disjoint paths: a shortest one (best-effort one) and a *longer* one, where the QoS traffic is possible to be rerouted. A specific scenario composed of UDP cross traffic, simulating congestion, a base layer traffic (QoS traffic) and enhanced layer traffic (best-effort) is imposed. The results show that it is possible to successfully

reroute the traffic while keeping its QoS guarantees.

Providing of QoS-Enabled Flows in SDN Exemplified by VoIP Traffic

Jannis Ohms et al. in [62] provide a SDN Application to guarantee QoS for real-time services on SDN networks, exemplifying with Voice Over Internet Protocol (VoIP) traffic. The application works by receiving a copy of the SIP traffic on the network, allowing to proactively pushing flows to establish a path for the RTP stream. The application parses the *OK* message of the Session Initiation Protocol (SIP) so it can make a bandwidth estimation and further reservation. The development and testing environment is composed by 2 VoIP phones, one VoIP proxy, one instance of the Floodlight Controller and 3 SDN-enabled switches. The aforementioned application itself, is composed of 4 elements:

1. SIP parser: to convert SIP packets into objects.
2. Message Evaluation: which is responsible for the extraction of the information to allow the bandwidth reservation.
3. Bandwidth Reservation: the actual component that based on the messages provided by the Message Evaluation, estimates the bandwidth and assigns an OpenFlow queue for the call.
4. Flow-Pusher: with the knowledge of the calls and callers' IP, this component creates the full path on top of the topology. After the path is calculated, the required flows get pushed.

A series of experiments is then performed: 2 scenarios where a low priority stream competes with a high priority stream, with no bandwidth reservation (both over UDP and TCP) and 2 scenarios exactly the same as before but with bandwidth reservations provided by the SDN application. Regarding TCP, the reserved bandwidth is totally provided, however, when it comes to UDP, the results show an interesting aspect: the bandwidth could not be provided and, for enabling that to happen, the size of the queues needs to be overestimated and so, this is a field that the authors claim it needs further studying.

Adaptive Routing for Video Streaming with QoS Support over SDN Networks

The work developed by Tsung-Feng Yu et al. in [82] aims to improve the OpenQoS algorithm proposed in [33], suggesting a more flexible algorithm (*adaptive routing approach for video streaming* - ARVS) when comparing to the OpenQoS. Also, the authors focus on the Scalable Video Coding (SVC) standard. The main focus is to improve the performance of QoS-enabled video streaming under various loads of the original path. As in [33], the routing problem is also exposed as a Constrained Shortest Path (CSP).

The algorithm is tested over a simulated network (using Mininet [10]), composed of 30 nodes and 1 SDN controller - Floodlight. The results are presented as comparisons with the OpenQoS algorithm: when the load level of the shortest path is raised to 0.7, the ARVS improves the packet loss rate of base layer packets up to 77.3%. Also, it enhances by 51.4% the coverage under various network loads for all paths. The algorithm, however, does not guarantee the performance of

enhancement layer packets but it can treat them as regular best-effort packets.

HiQoS: An SDN-based multipath QoS solution

Yan Jinyao et al. in [81] propose a QoS scheme named HiQoS which guarantees QoS for different types of traffic by using multiple paths between source and destination together with queuing mechanisms. Furthermore, HiQoS also recovers from link failure very promptly by rerouting traffic from the failed path to a stable one.

The authors define the traffic between 3 sets: (i) video stream with high bandwidth requirements (ii) interactive audio/video with low delay requirements and (iii) normal data streams which is defined as best-effort traffic. This division is based on, for example, source IP address, source port. Then, the different types of packets are forwarded to different queues, according to the available queues' bandwidth. This approach has one advantage and one disadvantage. On the bright side, this allows complete separation of all streams, however, when the total traffic of an application exceeds the mentioned available bandwidth some packet loss may occur. Also, the authors modify the Dijkstra's algorithm by calculating multipaths that possibly satisfy the QoS constrains of the system. When a flow reaches an OpenFlow enabled-switch and if that flow does not have a match entry, the switch sends a request to the controller asking for a path to deliver to that flow. The controller selects the optimal path and push flow entries to the switch for packet forwarding.

The HiQoS scheme is tested on a simulation environment (using Mininet and the Floodlight controller). For comparison purposes, the authors proposed another two schemes: the LiQoS which simulates a traditional network, with no QoS guarantees and MiQoS which is similar to HiQoS but it is a single-path solution. The results show that HiQoS is, by far, the best solution, reducing the delay in data transmission and in servers' response time. Also, the scheme also leads to a throughput (Mb/s) utilization more than 2x higher when comparing to the other solutions. As mentioned before, it also quickly recovers from link failure, improving the overall performance of the system.

Reliable Video over Software-Defined Networking (RVSDN)

Harold Owens II et al. in [65] propose an improved version of the work done in [64] (VSDN) by the same authors. VSDN framework can guarantee delay, jitter and bandwidth requirements and reliability[5]. This requires the path selection process to consider more than one path (whereas VSDN only choose the single-optimal path). The main component of both frameworks is the *routing module*. This routing module uses an heuristic algorithm - A*Prune Algorithm to select paths.

The mentioned framework is integrated into the ns-3 [11] simulator and as such, its experimental setup and results are obtained from the simulator. The results are compared with a regular MPLS architecture and with a VSDN-based on. Moreover, the metric used to access the performance is *Number-of-Requests Serviced by Architecture*. The authors concluded that RVSDN was able to service network requests that required a reliability constraint of 0.999 while the other

[5]The ability of the network to perform to specification per request.

architectures failed at providing guarantees to requests with a reliability greater than or equal to 0.995. Also, in what concerns to the latter constraint, RVSDN was able to provide 31 times more requests when compared with MPLS or VSDN.

SDN control framework for QoS provisioning

Slavica Tomovic et al. in [77] propose a system that is able to differentiate between services and also allows an efficient usage of network resources by means of an automated control. The system is made up of four main blocks:

- *Resource Monitoring*: The SDN controller periodically (with a 3 s interval) sends specific query messages asking for per-flow and per-interface statistics. This allows the controller to know useful information about flow's bandwidth and link load.

- *Route Calculation*: By receiving a list of flows with certain QoS requirements and the information obtained in the Resource Monitoring module, this block basically determines the routes for the different type of traffics. The QoS algorithm calculates the optimal route as the shortest one with the higher available bandwidth, so that the delay is minimized. In addition, in order to not degrade best-effort traffic, the controller can reroute that traffic before congestion occurs.

- *Resource Reservation*: This module is responsible for providing end-to-end bandwidth guarantees for the priority flows by configuring the output queues of the forwarding devices. The traffic is served with a Hierarchical Token Bucket (HTB) scheduling algorithm which grants setting up minimum and maximum rate for each flow.

- *Call Admission Control (CAC)*: This module is left for future work but the main purpose is to reject QoS requests if there is not the enough conditions and send that feedback to the requesting client.

This design is tested over a real testbed made up of 1 SDN controller (POX), a set of clients and server and 6 Linux devices running OpenvSwitch software. Two experiments were conducted and all the results were compared with the traditional best-effort model and a IntServ-like model. In the first experiment, 8 UDP flows are created and the results show that only the proposed model exploits the path diversity and the experienced throughput (Mb/s) of each flow is always quite stable and closed to its requested one. In the second experience, the main objetive was to test the best-effort reroute function - which ended up being well successful as the best-effort traffic is equally distributed. This function is a little bit different from other works on the same area because, usually, the rerouted traffic is the high priority one.

2.3.3 Industrial Networks

Extending OpenFlow with Flexible Time-Triggered Real-Time Communication Services

The METRICS project [9] presents a SDN-based networking solution for Industrial Internet of Things real-time constrained traffic. Luis Silva et al. in [75] present the work developed in the context of the mentioned project.

METRICS uses a logically centralized controller (an Extended OpenFlow Controller) responsible for a real-time capable heterogeneous SDN network (with wired and wireless segments - but on the present paper only the wired is address). This network is expected to handle traffic with real-time and reliability requirements, corresponding to the type of traffic in Industry 4.0 environments. The switches used in the project are based on the Hard Real-Time Ethernet Switching (HaRTES) [3] platforms. Those switches implement the Flexible Time-Triggered (FTT) paradigm [66] providing (i) support for synchronous and asynchronous traffic; (ii) server-based traffic scheduling for the asynchronous traffic; (iii) online stream management without service disruption; (iv) seamless integration of standard Ethernet nodes and (v) traffic policing.

The authors alert to the fact the OpenFlow protocol does not support, by default, time-triggered traffic i.e. real-time traffic. To enable that, the authors provide a real-time add-on (RTOF), which combines the current OpenFlow (at the time, 1.5 version) services, messages and API with a set of RTOF ones. The properties of real-time flows must be specified so that its reservation is possible by the OpenFlow controller. This API basically extends the OpenFlow API by defining specific messages to add, remove and modify real-time stream reservations, get the list of registered real-time streams and get the individual parameters of existing reservations. The authors highlight the fact that a direct communication between the HaRTES platform and the RTOF it is not possible and so they define a Mediator, acting as a translator between the both.

To validate the proposed architecture, experimental tests are conducted with the results showing that real-time traffic can be segregated from the standard and forwarded deterministically, i.e with low delay and low jitter.

A Real-Time Software Defined Networking Framework for Next-Generation Industrial Networks

Luis Moutinho et al. in [60] present a full-featured SDN framework built on top of the work done in [75] and with the same purpose: addressing the challenges in Industry 4.0. The authors claim that the solution presented by Luis Silva et al. lacks of continued QoS guarantees due to the absence of a central admission control whit schedulability analysis.

This framework includes real-time extensions, an Application Programming Interface (API) and a real-time SDN controller. The need for the latter arises from the realization that the available, open-source controllers do not provide the services required to manage networks with strict timeliness requirements. This complete framework is validated in practice with a prototype implementation in a realistic use-case. In fact, with 3 switches and cycles of 250μs, the authors achieve a 1μs jitter on real-time traffic (a good value for Industrial 4.0 applications' requirements) and a reconfiguration time between operational nodes lower than 100 ms.

Real-Time Wireless Data Plane for Real-Time-Enabled SDN

Paulo Ribeiro et al.in [70] present a data plane architecture that extends the abilities of the METRICS project (in particular, the Real Time Reservations Framework (RTOF), an OpenFlow real-time Add-on), integrating support for real-time dynamic reservations on WiFi. This architecture is built on top of 2 components:

1. The Wireless Multimedia (WMM) profile, which enhances the real-time behaviour at medium access.

2. The Linux Traffic Control (TC) to setup dynamic real-time reservations.

At layer 3 input, the authors define a classifier, which separates real-time traffic from non real-time traffic: the latter is forwarded to a FIFO queue and the former suffers a second classification, based on the *priority* parameter of each flow which is then submitted to a Token Bucket Filter and associated queue. A fixed-priorities scheduler (PRIO QDisc) is then employed in order to assure that non real-time traffic is only served in the absence of real-time one - this offers protection against misbehaving nodes. At layer 2, the authors map the real-time traffic to the voice access category (AC_VO), as it is the one with the the possible higher priority and the non real-time traffic to the background category (AC_BK) - the possible lower priority, which provides a effective segregation of both traffics. If the flows at the input of the Linux network stack are already instantiated, nothing is done, else, a *packet-in event* is sent to the SDN controller, which may respond with a new flow message (*add_flow_entry*) in conjunction with a RTOF message (*add_RT_stream*), specifying that it is a real-time flow. The authors then implement a real testbed, consisting of 2 producer nodes: a best-effort traffic one and a real-time traffic producer; and 2 consumer nodes (best-effort and real-time), belonging to the same Basic Service Set (BSS). The authors measure the delay CDFs (Cumulative Distribution Functions) between the end nodes for different scenarios: with no QoS mechanism, only using WMM, WMM+TC and finally, with the SDN integration. The most important result is that the addition of the SDN control caused "not more than a small degradation in the delays" while preserving its advantages.

2.3.4 IoT Networks

An Application-aware QoS Routing Algorithm for SDN-based IoT Networking

Guo-Cin Deng et al. in [31] propose a feasible solution for the problems encountered in the Internet-of-Things (IoT) world: the big data generated can consume network bandwidth leading to congestion and also, IoT applications that need to transfer multimedia data with multiple QoS requirements do not have the appropriate support in traditional IoT architectures. To conquer the above problems, the authors propose an application-aware QoS routing algorithm (AQRA) which evolves the work developed in [68]. The AQRA algorithm encompasses four parts:

- An SDN-based IoT network architecture: An architecture composed by 5 layers: an *application layer*, where IoT applications reside; a *network layer*, composed by OpenFlow

switches; an *edge layer*, containing several edge equipment such as IoT gateways; a *perception layer*, composed by IoT devices or sensors and a *control layer*, where the SDN controller runs the AQRA algorithm.

- Traffic classification: In order to differentiate between high-priority and low priority applications, the AQRA identifies and stores the flow_info message of each flow according to the app_profile message (sent by the IoT application), which indicates The QoS Class Identifier (QCI).
- SA-based QoS routing: The authors use a heuristic algorithm so that the best path for a specific flow with QoS requirements is found. Also, load balancing among paths is achieved by selecting the path which has the maximum available bandwidth (ABW).
- QoS-aware admission control: following the idea of dropping low-priority or medium-priority flows that would probably use the available resources needed for the high-priority ones.

To validate the AQRA algorithm, a simulation environment is created using the Mininet-WiFi emulator. The topology is composed by the Rye controller, 3 application servers, 3 virtual OpenFlow switches, 15 OpenFlow-enabled access points and 45 end devices accessing the network via WiFi. The authors also create 4 types of services: mission critical data transfer, video streaming, voice service and non-mission data transfer. A set of pre-defined QoS requirements is evaluated and then, from the results, a fitness ratio is calculated (flows that meet the QoS requirements/all flows). Concluding, all the services receive proper QoS guarantees - for example, the mission critical data transfer service has a fitness ratio of 1 for all requirements.

2.3.5 Telecommunication Networks

Applying Big Data Technologies to Manage QoS in an SDN

Shashwat Jain et al. in [47] present a work with a first premise as follows: the Telecommunication networks management could be improved if performance data was analyzed and a possible correlation between Key Performance Indicators (KPI) and QoS metrics was made - i.e. a method that creates a set of formulae that quantify the relationship between the KPIs and the metrics. The SDN simulation network was composed of tools such as the Mininet emulator and the POX SDN controller. During each simulation, 24 KPIs (including number of bytes, sent and received; round-trip times etc) were collected from a virtualized network, traffic simulator and IT infrastructure. The data collected was first analyzed using Spearman's algorithm to correlate time series data for each KPI - this correlation is then expressed in a scale from +1 (positive correlation) and -1 (negative correlation). A Machine Learning algorithm (*M5Rules*) was also used to quantify which KPI had the biggest overall impact on the QoS level. The authors performed several tests using a network topology that allowed to simulate multiple flows with varying combinations of clients and servers so that considerable changes in the flow metrics and corresponding KPI value was generated.

2.3.6 In-Vehicle Networks

Software-Defined Networks Supporting Time-Sensitive In-Vehicular Communication

Note: This reference has a closeness level of 3 ("Focus in delay guarantees", "QoS Algorithm" and "Hybrid Solution") but its results should not be omitted since they present relevance in the context of this dissertation.

Timo Häckel et al. in [43] present a solution for communication in in-vehicular communications based in Time-Sensitive Software-Defined Networking (TSSDN) switching, which combines the capacities of Time-Sensitive Networking (TSN) (a standard for real-time support in layer-2 networks, as explained in 2.2) and SDN. This approach implements time-sensitive flows via OpenFlow and TSN streams integration with the SDN controller. The authors decide to simulate a network topology made of 2 clients, 2 TSSDN switches and 1 SDN controllers, with the client 0 generating TSN traffic and client 1 receiving it (path: Client 0 -> TSSDN switch 0 -> SDN controller -> TSSDN switch 1 -> Client 1).

The main goal of this research was to understand whether the addition of SDN logic (and its advantages) could alter the time-sensitive traffic performance already implemented by TSN, i.e. if SDN imposes some extra delay. The authors reached to the conclusion that it is possible to maintain the real-time capabilities even within a topology as previously described. This study represents huge importance in the scope of this dissertation since one of the points to-be studied is to investigate the SDN overhead.

2.3.7 Summary and Discussion

After the comprehensive review of the State of the Art on the area, some conclusions can be drawn. First, the majority of the works show a similar approach: primarily some sort of traffic monitoring is made, which leads to traffic classification. Based on the information gathered, most of the works present a route calculation process. With one or more paths selected as feasible ones, the system often proceeds to *reserve* resources to guarantee QoS requirements. However, few works present some kind of policing to ensure the system is behaving accordingly (which is the case of [29]).

Second, and probably the most important conclusion, is that SDN-based works rarely are tested on a real testbed. In fact, most use Mininet or ns-2/3 simulator as it is a cheaper, easier and more easily scalable solution. Related to that, as the standard Mininet controller is the Floodlight, it becomes the most used and studied SDN controller. One of the main contributes of this dissertation is that a real testbed is used, with all of its challenges and inconveniences but way more close to a real world network.

Third, some works aim to enhance OpenFlow protocol features which is the case of [77], yet OpenFlow is a work-in progress protocol and each version improves the last, causing some works to become obsolete. Also, it is noticeable that some (but few) works take advantage of the operating systems used on the on-tested architecture or others legacy QoS mechanism, such as the work developed by Paulo Ribeiro et al. in [70] where the Traffic Control Linux's tool and the

WMM profile are used. The integration of newly proposed and growing SDN technologies (i.e., OpenFlow, controllers) with legacy or end-host mechanisms may result in relevant and worthy mechanisms to improve service quality.

As depicted in 2.2, only 5 references have a *closeness level* of 5 or more (representing only 20% of the total). Furthermore, only one work ([47]) of the gathered ones apply some sort of machine learning algorithm and it does not present a real-time procedure to access QoS guarantees, it solely relies on a *a priori* analysis to derive QoS practices in a telecommunication network.

Chapter 3

xDynApp Algorithm Design

As mentioned in Chapter 1, Real-Time and multimedia applications need Quality of Service (QoS) guarantees such as low delay, high bandwidth, low jitter, etc. The current *status quo* of networking relies on a best-effort model, i.e. every packet is treated in equal terms, the network does not provide any a native solution that guarantee that data is delivered with some QoS guarantees. Also, network performance characteristics are highly dependent on the network traffic load. Even though some adaptations have been designed (for example, the ones specified in 2.2), they inherit all the problems of *traditional* networks such as low scalability and limited management capabilities. Software-Defined Networking allows the network's intelligence to be moved up to a SDN Controller, decoupling the control plane and the data plane. This emerging paradigm does not natively provide an approach to fulfill QoS requirements. However, due to its agility and flexibility, it is possible to explore and implement new functionalities such as dynamic adjustment of QoS to meet the applications' needs and available resources - the main goal of this dissertation.

This Chapter provides the presentation and explanation of the developed SDN-applications' algorithm design to provide QoS to applications. In Section 3.1 the main tools used in the applications design are presented. In Section 3.2 details the design of the main developed application are introduced. In Section 3.3 the design of another, simpler, created SDN-application is presented. In Section 3.4 the main differences of both application are highlighted. Section 3.5 gives details for design choices. Finally, Section 3.6 gives a short summary of the design details discussed in this Chapter.

3.1 Overview of *xDynApp*

The *xDynApp* is a set of SDN applications created to dynamically and autonomously provide QoS guarantees to applications. *xDynApp* is designed to mostly manage the bandwidth resources in the underlying network but, ultimately, it also improves other QoS metrics such as Jitter and Packet Loss Ratio. The benefits of *xDynApp* are specially relevant in network congestion scenarios.

The *x* in *xDynApp* stands for *Reserve* or *Priority* as the set is subdivided in two different ones: *ReserveDynApp* and *PriorityDynApp*. Both applications, whilst, in a high-level, being designed differently, use the same root tools to provide QoS guarantees. The referred tools are:

1. *OpenFlow Drop Meters*: introduced in OFv1.3, meters allow rate-monitoring of ingress traffic. The network manager can *point* packets of a given flow to the meter and then, the meter can perform some operation based on the rate it is receiving the packets. In the case of the discussed set of applications, the meters act as a rate limiters, dropping packets when the rate is higher than a predefined one.

2. *802.1p Queuing Mechanism*: as described in 2.2, the VLAN PCP can assign a packet priority; that priority can then be mapped to an egress queue of an output port of a switch that can have different behaviours according to what is pre-configured. The *xDynApp* works upon the assumption that higher PCPs have higher priority and packets marked with those PCPs are mapped to one or more queues that have higher Guaranteed Minimum Bandwidth (GMB) (in percentage, regarding the port's bandwidth) for sending egress traffic. On the other hand, lower PCPs have lower priority and are thus mapped to queues that have lower GMB. However, it is important to note that each class of traffic (differentiated by its associated PCP) *always* has a GMB, allowing even lower priority traffic to never *starve*.
   Subsection 3.5.1 details the main reason for this implementation detail.

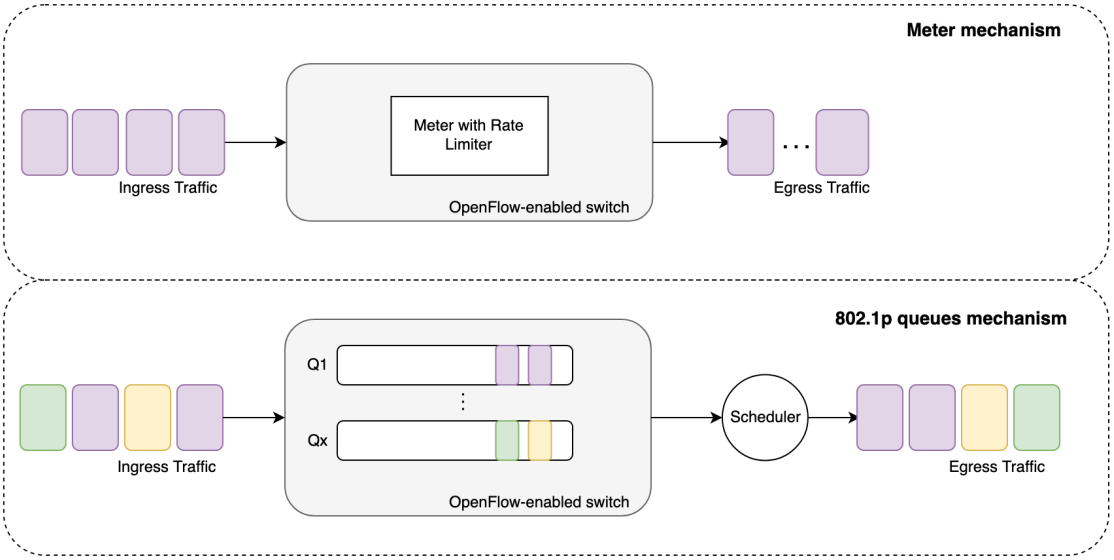

Figure 3.1: xDynApp Tools

With that in mind, *xDynApp* is expected to work in any LAN environment (it can also be scaled to larger networks with minor adjustments) composed of OpenFlow enabled devices that support OFv1.3 or higher and the 802.1p technique. For the developed work and for the remainder of this document, each output port of each switch on the network has two queues, Q1 and Q2, with Q1

having a GMB of 20% and Q2, 80%. However, the system here described could as equally explore a configuration with more than 2 queues.

3.2 ReserveDynApp

ReserveDynApp application presents a SDN-based algorithm to provide QoS guarantees to priority flows. This is accomplished by allowing users of the network to make a *request* to the controller for a guaranteed bandwidth regarding a specific flow, i.e. users define what corresponds to a priority flow and what does not. Since the SDN application has knowledge of the requests, it is possible to apply into the algorithm an *admission control* stage, as it will be later discussed.

The workflow for the process of submitting a request is specified in the following Subsection.

3.2.1 Requesting a Reserved-Bandwidth Flow

1. The end-user sends to the controller via TCP port 54321 a request in a pre-defined XML format. This XML format presumes the indication of the IP address of the user who is sending the request as the source IP of the flow, the IP address of the destination of the flow, the TCP or UDP port to be used, and an estimated value of the necessary bandwidth for the communication (in Mb/s).

   Listing 3.1: XML request structure

   ```
   <?xml version="1.0" encoding="UTF-8"?>
      <requests>
         <hostrequest>
            <from>srcIp</from>
            <to>dstIp</to>
            <port>port</port>
            <estband>estband</estband>
         </hostrequest>
      </requests>
   ```

   It should be highlighted that this end-user $\leftrightarrow$ controller communication is made via a non-OpenFlow channel.

2. The controller and, more specifically, the *ReserveDynApp* receive and process the request. The application then checks for a miscellaneous of syntax and logical errors in the received request. The end-user should have the UDP 54321 port open to receive a feedback from the controller. The available feedback codes and meanings are depicted in Table 3.1.

   The code *200* is particularly only available for a full successful request. The codes in the format *3xx* are for syntax errors: if the XML structure does not correspond to the one depicted in the XML presented in 3.1 or if any of the IPs does not have a valid syntax (e.g., the

Code	Meaning
200	Everything is ok
300	Bad XML strucuture
301	Bad src IP syntax
302	Bad dst IP syntax
400	Unrecognized src IP
401	Unrecognized dst IP
500	Non-valid port value
501	Non-valid bandwidth value
600	Internal Error

Table 3.1: Feedback Codes and Meanings

IP address 4.5.6 or the IP address 172.16.10.465 are not valid). Codes in format *4xx* refer to unrecognized IPs, i.e. the controller does not recognize that IP as being part of the network. Codes in format *5xx* refer to non-valid values of both port and bandwidth: the port must be in the range of 0-65535 [1] and the bandwidth must be in the range of 1-1000 (Mb/s). Code *600* refers to some internal error on the controller side.

If the code equals *200* then the controller stores the request in memory. It should be noted that the request is made using TCP but the feedback is sent using UDP. This difference is explainable by the fact that the end-user always knows the destination of the request but, contrariwise, if the IP source of the request does not have a valid syntax or if the controller does not recognize it, the feedback can not be sent to a specific IP and it is broadcasted.

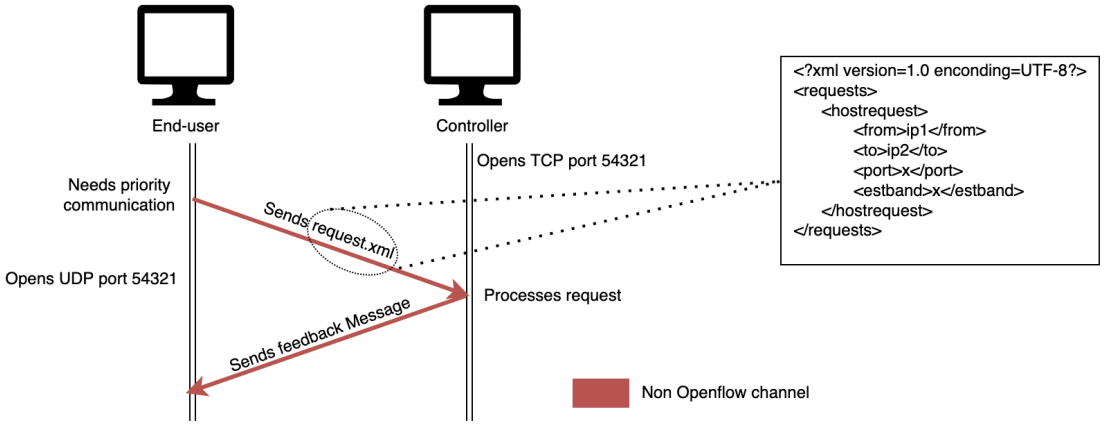

Figure 3.2: Request's Workflow

ReserveDynApp Configuration and Operation Stage

Upon the application start, the algorithm searches for every different physical port speed (e.g. 10 Mb/s or 100 Mb/s) of all ports in every switch of the network. For each different port speed and

[1] Some ports are, of course, reserved but that conflict management is left to the end-user side.

for each switch, a Drop Meter with a rate speed limiter corresponding to 20% of the port speed is installed. This process is only made once in the lifetime cycle of the application and thus, it can be considered a single-time configuration. A high-level view of the *ReserveDynApp*'s algorithm is depicted in Figure 3.4.

ReserveDynApp expects the SDN controller to be collecting every packet that runs on the dataplane. The application starts by dissecting the packet and it checks if there is any stored-in-memory request that matches the content of the packet. Two further actions are possible and explained in the following Subsection.

3.2.2 Treatment of Priority Flows

If there is a stored request, the application enters the *admission control* stage as aforementioned. The algorithm defines that firstly it should be checked if the source and destination are on the same switch. If they are not on the same switch, i.e. there is a path(s) composed of links between the source and destination, the application calculates all *possible* paths between source and destination. For every path and for every in-port or out-port of every link, it is checked if there is available bandwidth. This calculation is as follows:

$$Estimated\_Bandwidth < (Port\_Speed - Current\_Port\_Load) \tag{3.1}$$

Meaning that it is considered that, for a given port, there is available bandwidth if the estimated bandwidth (provided at the request) is less than the physical port speed minus the SDN-*observed* current rate/load. If this checking fails for any port of any link in a path, that path is considered not feasible. If there are more than one path, the next path is tested.[2] If every path is considered not feasible, the application states that the request can not be met and erases it from memory and the packet is treated as a non-priority one. If the source and destination are on the same switch, this process is only made for the in-port and for the out-port. Again, if any port does not have sufficient bandwidth, the request is discarded and the packet treated as a non-priority one.

For the accepted requests (the priority flows), the *ReserveDynApp* proceeds to the flow installation process on the switches involved in the connection between the source and destination. These flows have as *rules* the following criteria (below is an example for a flow that matches UDP packets):

(i) MAC address of source node

(ii) MAC address of destination node

(iii) Byte IP protocol = 17 [3]

(iv) UDP Port

(v) (Physical)In Port

[2] The order of testing starts at the shortest path (in hop count) and so on, being preferred the shortest path possible.
[3] 6 for TCP and 17 for UDP, as defined by the Internet Assigned Numbers Authority (IANA).

and as *actions* the following:

(i) Set VLAN PCP to 0

(ii) Forward packet to a physical output port.

Furthermore, if the packet is using TCP, the exact same procedure is done expect the Set VLAN PCP action, which in the case is set to 7 and, of course, the *rule* port/Byte IP protocol is for TCP.

As stated, the behaviour of *ReserveDynApp* when treating UDP and TCP priority (P) flows is different due to some differences in both protocols and also some observed performance of the underlying network switches. Since TCP packets are mapped to Q2, the application using TCP has already some QoS guarantees. This is not the case of applications using UDP (packets are mapped to Q1) and so, further configurations are needed - which are explained below in the non-priority (NP) flows treatment. More information about the motivation for this difference can be found in Subsection 3.5.5.

Note that setting the VLAN PCP enforces a packet to go to a specific output queue in the output port of the switch. Also, the specification of the physical output port throughout the installation of flows in the involved switches creates the end-to-end connectivity which uses the found feasible path during the *admission control* stage.

Note: As stated in the OpenFlow version 1.3 documentation [16], some *actions* and even some *rules* have pre-requisite rules that should be respected. For example, it is not possible to specify the *rule* (iv) Port without additionally adding the *rule* Byte IP protocol that corresponds to the used TCP/UDP port.

3.2.3 Treatment of Non-Priority Flows

If there is not a stored request - originally or coming from a not accepted request, the application also checks if the sender and destination are on the same switch or not, proceeding to the same procedures as before. However, knowing the path between the source and destination (being the path composed of link(s) or simply an *input port-output port* on the same switch), the application checks if there is an UDP priority flow using one or more joint physical ports of the to-be-installed non-priority flow. Additionally, if there is more than one joint port, the application searches for the one with the *minimum* port speed, i.e. the bottleneck. Subsequently, the application installs flows in the involved switches with the following *rules* criteria:

(i) MAC address of source node

(ii) MAC address of destination node

(iii) (Physical)In Port

and as *actions* the following:

(i) (if there is UDP P flow on the same path) Point to the already installed meter with the rate limiter that corresponds to the 20% of the bottleneck port speed.

(ii) Forward packet to a physical output port.

The functioning of this meter mechanism is highlighted and exemplified in Figure 3.3.

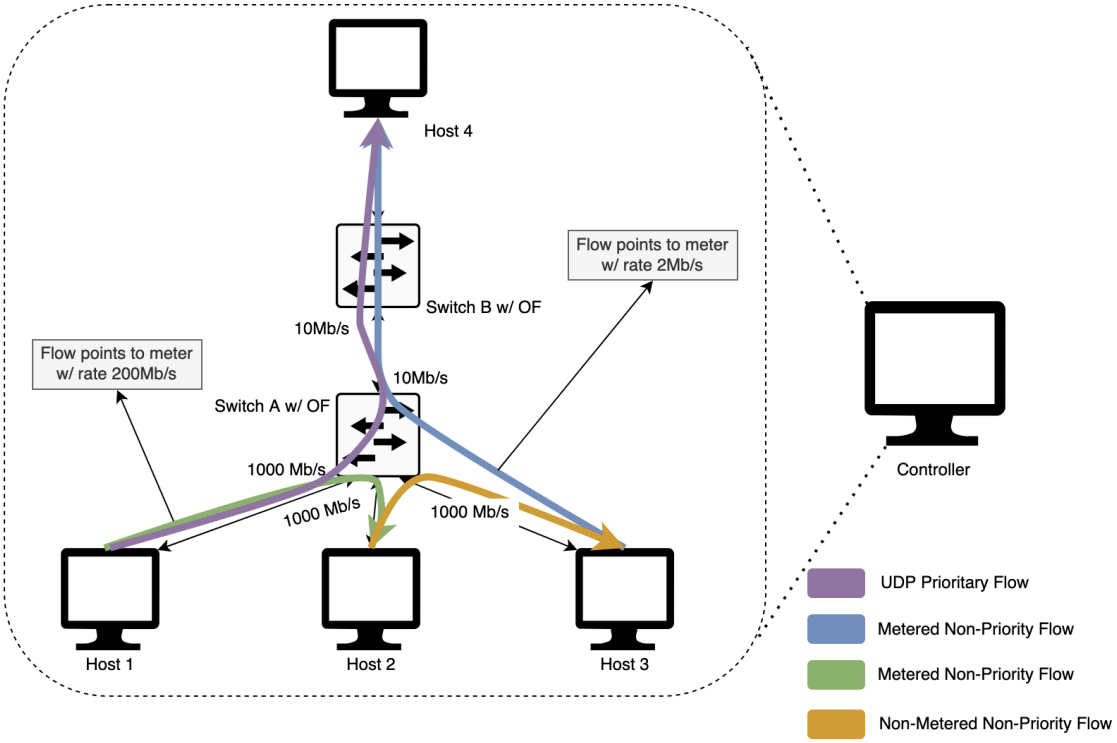

Figure 3.3: Meter Mechanism

As there is a priority UDP flow from Host 1 to Host 4 (purple), every flow that uses one or more joint port of the ones being used by the UDP flow, will point to a meter. The green flow uses a joint port with port speed of 1000 Mb/s meaning that it is pointing to a meter with a rate limit of 200 Mb/s. The blue flow uses more than one joint port, however, the lowest (the bottleneck) has a port speed of 10 Mb/s and thus, it points to a meter with a rate limit of 2 Mb/s. Finally, yellow flow does not have any joint port and so, normal flow rules are installed and it is not pointed to any meter.

If the packet does not have any joint port of the ones used by any UDP priority flow, the application proceeds to simply install flows with the same *rules* criteria as the ones using joint ports but there is only one *action*, the physical output port forwarding. In both cases, the VLAN PCP is not specified but the default value is 0 and thus, the packet will be mapped to Q1.

3.2.4 Rules Installation and Deletion Process

It is important to point out that after the flows installation process, OpenFlow-enabled switches do not forward the packet to the controller as they know what do to with packets that match the *rules* criteria. Notwithstanding, there is a subtle, important difference between the timespan duration of the installed flows. If they correspond to non-priority flows, the algorithm defines that they should only be installed with a timeout of 10 seconds - after 10 seconds, the switch erases from memory the installed flows and the aforementioned process repeats again. If they correspond to priority flows (UDP or TCP), the switch only erases from memory after 20 seconds of not receiving any packet that matches the *rules* criteria. This difference was created to avoid adding unnecessary overhead for priority flows resulting from unnecessary controller-side processing which could lead to a degradation of the flows' QoS. Installing permanently the flows is also not an option as the traffic of every network is always changing its form, causing flow rules to because obsolete. The timeout of priority and non-priority flows is left as management parameter that the network manager can adjust.

Another solution for avoiding the overhead of the flow installation process would be to create and install *a priori* generic and permanent flows for the priority ones, i.e. only using as the *rules* criteria the byte IP protocol and the port, and only setting as *action* the forwarding to the standard treatment at the output port (in OpenFlow terms, forward to the NORMAL output port). This way, an installed flow is always present but unfortunately there is no assurance that the end-to-end shortest path is being used and our solution provides such guarantee, thus motivating the relevance of installing end-to-end dedicated flow rules.

Furthermore, in what regards the installation of priority flows, besides being installed flows that match the priority class in hands, flows that would match the other transport protocol but the same port are also installed. For example, if a packet matches the UDP/5004 combination, flows for the TCP/5004 combination are also installed. This design detail arised from the observation that some applications used both protocols and the same ports for the communication as is the case of the iPerf3 [8] tool when using the UDP mode which begins the communication with a TCP handshake.

Instant of Actual Reservation

One important aspect that should be mentioned is the fact that the *admission control* stage is being made only upon a packet's reception that matches a stored request and it is not being done immediately after the reception of the request. This design choice was made due to the fact that, even though there is a received request, there is no certainty that the end-user will initiate the connection immediately after sending the request. In other words, it is pointless to check the available bandwidth at the time of the request, as conditions may have changed at the instant at which the flow actually starts. The uncertainty about the instant at which the source will start the communication also entails that the controller can not give an immediate feedback to the end-user regarding

the acceptance of the request. Nevertheless, it is possible to give that feedback, *a posteriori*, counting that the end-user maintains some sort of communication method - possibly, like the ones used in the workflow of the request process. For simplicity, the *ReserveDynApp*'s algorithm does not give that feedback. Another option would be to allow the end-user to specify an estimation of the start of the communication, however, our solution is simple and, as it not dealing with time uncertainties, it is more accurate.

3.2.5 Enforced Limits to the Reservation Mechanism

Aggregated Minimum Guaranteed Bandwidth to NP Flows

Finally, as mentioned before but not exactly specified, it was decided as a design option to preserve the QoS of the existing traffic in the network, whether it priority or not - as the request acceptance takes into account the overall current load at the port [4]. The main goal is to not dramatically affect on-going flows with the imposition of a priority flow onto the network, even if they are not priority. A non-priority flow can enter the network without making any request to the controller but, of course, they will never have bandwidth guarantees besides the ones provided by the root tools (802.1p queuing mechanism and meters mechanism).

Aggregated Maximum Guaranteed Bandwidth to P Flows

ReserveDynApp provides the guarantee that a priority flow has its requested bandwidth available throughout its lifetime by the *admission control* mechanism. However, there is a maximum value that can be attributed to priority flows - Aggregated Maximum Guaranteed Bandwidth - i.e. the end-user may ask for more bandwidth than the maximum but there is no guarantees that it would be provided. This case will be highlighted with an example:

A network is composed of ports with a single port speed: 10 Mb/s. A end-user sends a request for a bandwidth of 9 Mb/s. Upon the communication start, the request is accepted as there is available bandwidth at the moment. Later in time, a non-priority flow enters the network and uses a joint port of one the priority flow is using. This non-priority flow is trying to use 5 Mb/s - as the algorithm defines, the non-priority flow is either rate limiter to 2 Mb/s by the meter or the 802.1p queuing mechanism but it still occupies 2 Mb/s, making the priority flow to decrease its used bandwidth to 8 Mb/s (10 Mb/s - 2 Mb/s).

Concluding, if a priority flow have a requested bandwidth that matches the following:

$$(Port\_Speed - Request\_Bandwidth) < Guaranteed\_Bandwidth\_NPFlow \tag{3.2}$$

with

$$NPFlow\_Tried\_Bandwidth >= Guaranteed\_Bandwidth\_NPFlow$$

[4] Subsection 3.5.4 provides further information about this detail.

a priority flow has a guaranteed bandwidth that has as maximum value of:

$$Port\_Speed - Guaranteed\_Bandwidth\_NPFlow \tag{3.3}$$

Note that this imposition arises from the fact that non-priority flows, besides being non-priority, still have some guarantees, otherwise, they would starve in the presence of priority ones. Also, this value should be publicised by the *ReserveDynApp* to the end-user, however, due to the same reasons the design states that the bandwidth checking is only made upon the communication start, the controller does not inform the end-user about this. The controller, after sending the *200* feedback code, could calculate that value based on the ports speed that compose the path between source and destination but there is no certainty that both source-host or end-host are not changing their location between the time instant the request is made and the time instant the communication starts. This detail is especially important for mobility scenarios.

3.2.6 Final Remarks on *ReserveDynApp*

Input Flow Possibilities			**Output Queue Assignment**
P UDP Flow	**P TCP Flow**	**NP Flow**	**Egress Port**
X			UDP P ->Q1
	X		TCP P ->Q2
		X	NP ->Q1
X		X	UDP P ->Q1; NP ->Q1 w/ meter
X	X		UDP P ->Q1; TCP P ->Q2
	X	X	TCP P ->Q2; NP ->Q1
X	X	X	UDP P ->Q1; TCP P ->Q2; NP ->Q1 w/ meter

Table 3.2: Possible States of an Egress Switch Port

ReserveDynApp guarantees QoS requirements for priority applications using an *admission control stage*, the 802.1p priority technique, if the class corresponds to TCP, and using meters to *meter* non-priority traffic that is using joint ports of the ports that UDP priority flows are possibly using, as UDP priority applications do not use the 802.1p technique. Note that the different root mechanisms forces some different behaviours. For example, the queue mechanism allows non-priority flows to access more than 20% of the port's bandwidth if there are TCP priority flows using less than 80% of the port's bandwidth. This is not the case when using meters: if a non-priority flow is metered to 20% of the port's bandwidth it can not have any larger bandwidth access, no matter the percentage the UDP priority flow is using. If there are priority flows being mapped to both queues (and no non-priority flows), the 802.1p queuing mechanism states that the 80%/20% distribution is still valid, however, as there is a *admission control* stage, each priority flow has it bandwidth reserved. Finally, for both methods, if a flow is mapped to a queue and the other queue is empty, it has full bandwidth access.

Upon the application deactivation, all flows, all meters and all stored requests are purged from the network. Furthermore, all possible states of an egress switch port regarding its queues is presented in Table 3.2.

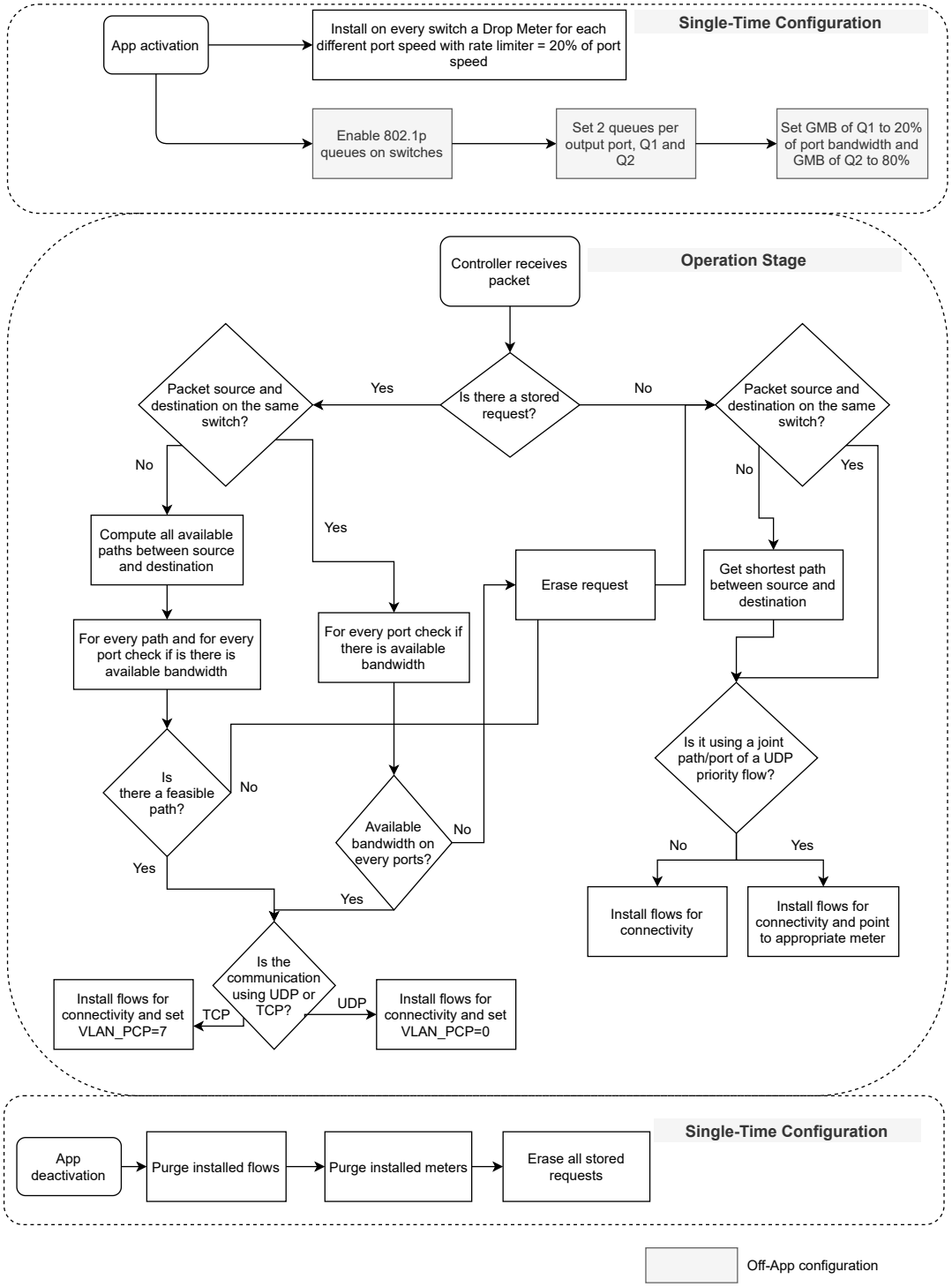

Figure 3.4: *ReserveDynApp* Algorithm

3.3 PriorityDynApp

PriorityDynApp application presents an alternative, simpler, SDN-based algorithm to provide QoS guarantees, when compared to *ReserveDynApp*. Contrariwise to *ReserveDynApp*, there are pre-defined classes of priority applications. These applications (being multimedia or other) can be typically associated with well-known used transport protocol and port - for example, VLC media player [18] RTP streams by using UDP and the default ports 5004 and 5005 (5004 for the data and 5005 for the sender reports). *PriorityDynApp* defines only two classes for priority but the number of classes can be escalated to any number and transport/port (application) combination the network manager considers. In this work, the two pre-defined classes chosen were based on the default stream behaviour of the VLC media player as presented in Table 3.3.

Despite not being designed or implemented, it would be reasonably simple to provide an intuitive mechanism that would allow the network manager to introduce, on-the-fly, more priority combinations. This could be done, for example, using a command line (CLI) of the SDN controller's console.

Application Protocol	Transport Protocol	Port
RTP	UDP	5004 || 5005
HTTP	TCP	8080

Table 3.3: *PriorityDynApp* Classes of Priority

PriorityDynApp Configuration and Operation Stage

Like *ReserveDynApp*, upon the application start, the algorithm also searches for every different port speeds of all ports in every switch of the network. For each different port speed and for each switch, a Drop Meter with a rate speed limiter corresponding to 20% of the port speed is installed. It is, then again, considered a single-time configuration. A high-level flowchart visualization of the *PriorityDynApp* algorithm is provided in Figure 3.5.

PriorityDynApp also expects the controller to be collecting every packet that runs on the dataplane but the main difference from *ReserveDynApp* emanates at this point. *PriorityDynApp* also dissects the packet but, instead of checking if there is a stored request that corresponds to the packet's information, it simply checks for combinations of transport protocols and ports that match the ones depicted in Table 3.3. After that, two further actions are possible and explained below:

- If a packet corresponds to a priority one, the process for priority flows installation is exactly as the one used in *ReserveDynApp* except that, instead of using a pre-defined path, the shortest one between source and destination is used.
- If a packet does not correspond to a priority one, the process for non-priority flows installation is exactly as the one used in *ReserveDynApp* including the meter mechanism which involves checking if a non-priority flow uses a joint port of the ports used by a UDP priority flow.

A 10 seconds timeout for non-priority flows is also chosen. Likewise, the switches erase from memory installed priority flows after 20 seconds of not receiving any packet that matches the *rules* criteria. The installation of priority flows that matches both TCP and UDP protocol is also still applied.

3.3.1 Final Remarks on *PriorityDynApp*

PriorityDynApp also uses the 802.1p queuing and meter mechanism, however does not implement any form of *admission control* as there is no prior request made for priority flows. Although presenting a simpler solution as there is no need for those requests, it also has its drawbacks - it is not possible to guarantee that an alive, running on the system flow, has more QoS guarantees than the ones provided by the root tools mentioned in Section 3.1, which only give guarantees to aggregated flows. This is special relevant, for example, for the case where there is UDP priority flows being mapped to Q2 and TCP priority flows being mapped to Q1. In this case, UDP flows only receive a GMB of 20%, as stated in the 802.1p queuing mechanism, i.e. TCP priority flow has more guarantees than UDP priority flows. *ReserveDynApp* resolves this issue using the *admission control* stage, which gives QoS guarantees per flow.

Upon the application deactivation, all flow and all meters are also removed from the network. Furthermore, as the flows installation is very similar to the one discusses in *ReserveDynApp*, the Table presented in 3.2 is still valid for *PriorityDynApp*.

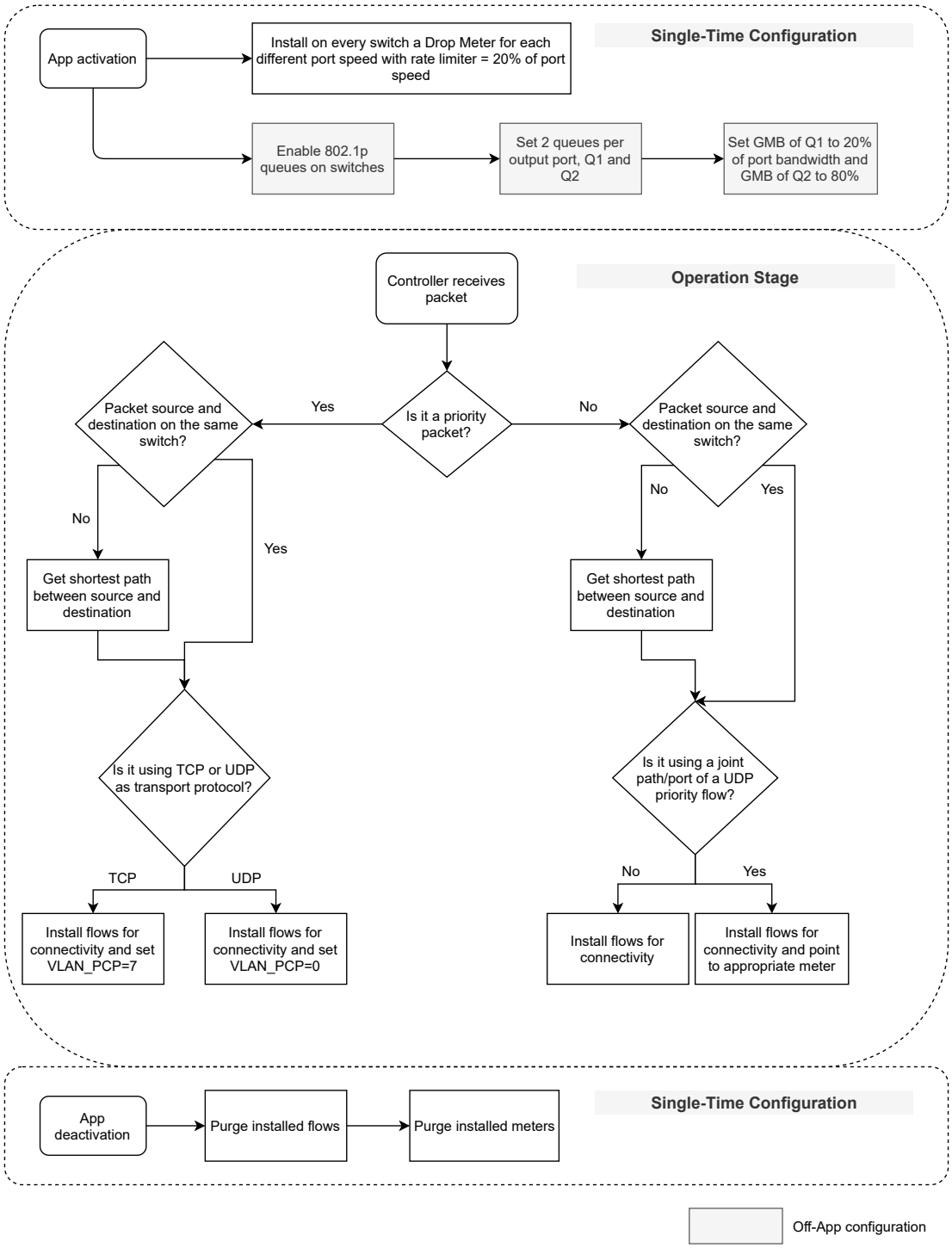

Figure 3.5: *PriorityDynApp* Algorithm

3.4 ReserveDynApp vs PriorityDynApp

ReserveDynApp and *PriorityDynApp* being two similar solutions for the same problem in-hands, have some differences:

1. *PriorityDynApp* is simpler and easier to deploy;
2. *PriorityDynApp* is faster at installing flows as less checks are made;
3. *PriorityDynApp* allows the network manager to decide the priority classes whilst *ReserveDynApp* gives that freedom to the end-user.
4. *ReserveDynApp* guarantees that a priority flow has the requested bandwidth guaranteed while alive (except in the cases discussed in Section 3.2);
5. *ReserveDynApp* provides an additional layer of QoS for non-priority flows (by the *admission control* stage). However, both provide a guarantee of 20% of the total bandwidth available.
6. *PriorityDynApp* and *ReserveDynApp* both use the 802.1p queuing and meter mechanism for QoS deployment but only *ReserveDynApp* performs an *admission control* stage.

	PriorityDynApp	**ReserveDynApp**
Simplicity	√	-
Setup Speed	√	-
Guaranteed Bandwidth Per Flow	-	√
User-Defined Priorities	-	√
Some QoS to non-priority flows	√	√
# Methods for QoS deployment	2	3

Table 3.4: Differences Between *PriorityDynApp* and *ReserveDynApp*

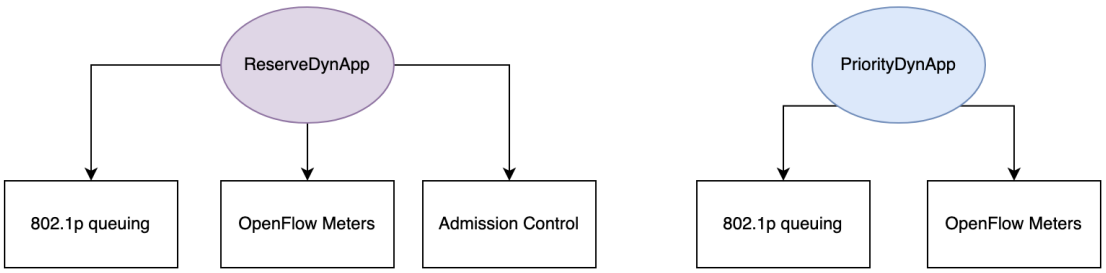

Figure 3.6: Methods for QoS Deployment

3.5 Motivation for Some Design Options

Both *ReserveDynApp* and *PriorityDynApp*, before reaching a final and stable state went through a tuning process as some initial assumptions revealed themselves to not be valid or simply because

the used software or hardware did not meet expectations. This Section aims to present some approaches that were eschewed and consequent detailed choices of design.

3.5.1 OpenFlow queues vs 802.1p queues

The very first idea was to use OpenFlow queues. As specified in [16] an OpenFlow queue can attach to a port and be used to map flow entries on it. These flows that are mapped to the queues, are treated according to the queues' configuration. For example, OpenFlow version 1.3 states that a queue must have a set of proprieties, including:

- *min_rate*: a property that defines a guaranteed minimum data rate for a queue. This allows for a specific traffic prioritization against others.
- *max_rate*: a property that defines a maximum data rate for a queue. If the actual data rate is higher than the *max_rate* the switch will drop or delay the packets to gratify the *max_rate*.

OpenFlow queues are specified within the OpenFlow protocol, nevertheless, the protocol does not handle the queue management on switches, that process must be done outside of its scope. Also, the OpenFlow specifications does not state as mandatory that an OpenFlow-compatible switch must have support for OpenFlow queues; this decision is up to the switch vendors. Most software switches, like Open vSwitch (OVS), support this feature but not every hardware-switch vendors provide it - which is the case of the switches assigned for this project, the Aruba 2930M 24G PoE+ 1-slot Switch (JL320A). This information can be confirmed here [4]. For this reason, it was decided to use the built-in 802.1p queues and the OpenFlow VLAN PCP setting action to interact with them.

3.5.2 Interaction with queues

Regarding the chosen way to interact with built-in queues, the early idea was to set the DSCP field of the IP header. Aruba switches (at least from ArubaOS higher than the 16.03 version) present a mapping between the DSCP field value and the 802.1p tag (and therefore, the mapping to a queue). Unfortunately, after considerable effort in trying to enable this mechanism, we observed that priorities would not be set in this manner. This probably can be explainable because the in-between switches connections correspond to only layer 2 communications and the DSCP field is set a layer 3 level. It was then defined that it was to be used the VLAN PCP marking.

3.5.3 Buggy Reporting of Meter Activation

The integration of the meter mechanism was also not straighforward. When using the chosen SDN controller (ONOS) to install meters, the switch thrown error messages with the error code = METER_EXISTS. The installation process seemed to be faulty and this mechanism was about to be dropped, however, and with the help of Google Groups and nice contributors [5], it was realized

[5]ONOS discuss - Google Group, "Meter Error". https://groups.google.com/a/onosproject.org/g/onos-discuss/c/Og8_71ah_Ss/m/z0f7ZnUMAwAJ, Online, accessed 03 June 2021.

that the meters were *de facto* installed and operational - the switch fails to report the existence of the meters on the MULTIPART_REPLY OpenFlow messages, i.e. the meter regarding communication from the controller to the switches is 100% functional but not from the switch to the SDN controller. This issue, already noticed by HPE support, is on pending state.

3.5.4 SDN *observed* rate

The process of retrieving the SDN *observed* rate aforementioned was firstly thought to be made *per-flow* instead of *per-port*. The main idea was to only accept a request if the request bandwidth did not interfere with the used priority flows' bandwidth. However, not only was decided to not interfere with all type of traffic but also, when trying to obtain *per-flow* statistics using the ONOS's FlowStatisticService, the methods threw empty values for every rate and for every flow. This behaviour is explainable either by an erroneous communication (as the meter example) or some ONOS bad implementation. Nevertheless, the Aruba switch seems to be returning all mandatory flow statistics to the ONOS controller as, for example, the number of packets matched, the duration etc, as demonstrated in Figure 3.7 in the ONOS GUI. This issue was not found reported on the web and so, the root cause is yet to be discovered.

Figure 3.7: ONOS Receiving Valid Flow Statistics

3.5.5 TCP and UDP treatment differentiation

In what regards the treatment differentiation between UDP and TCP priority flows, it can be explained by the observed behaviour of UDP flow under the 802.1p queuing mechanism, specifically marked with a high PCP and hence mapped to Q2.

If the application that is using UDP as the Transport Protocol is using IP fragmentation [6] (which is the case of iPerf3 in UDP mode and also the MPEG-TS standard - very common in video streaming) and if the rate is close to the maximum rate possible for that port (with or without congestion), there is a high number of dropped packets seen at the receiver. This, of course, is explainable by the fact that the queue is full and dropping packets. Because the application is using IP fragmentation, one fragment being dropped (lost) corresponds to a complete packet loss. However, this behaviour is not observed when a similar UDP flow is mapped to Q1. One possible

[6] IP fragmentation corresponds to the process of breaking packets into smaller fragments so that the resulting fragments can pass through a link which has a smaller maximum transmission unit (MTU) than the original packet size.

explanation is if the sizes of the Aruba 802.1p queues are not the same, being the Q2 smaller than Q1. Aruba documentation regarding the queues' size was not found.

As the 802.1p queuing mechanism did not work in all instances of UDP traffic, it was decided to use the meter mechanism to grant QoS to UDP P flows.

Experimental Evidence

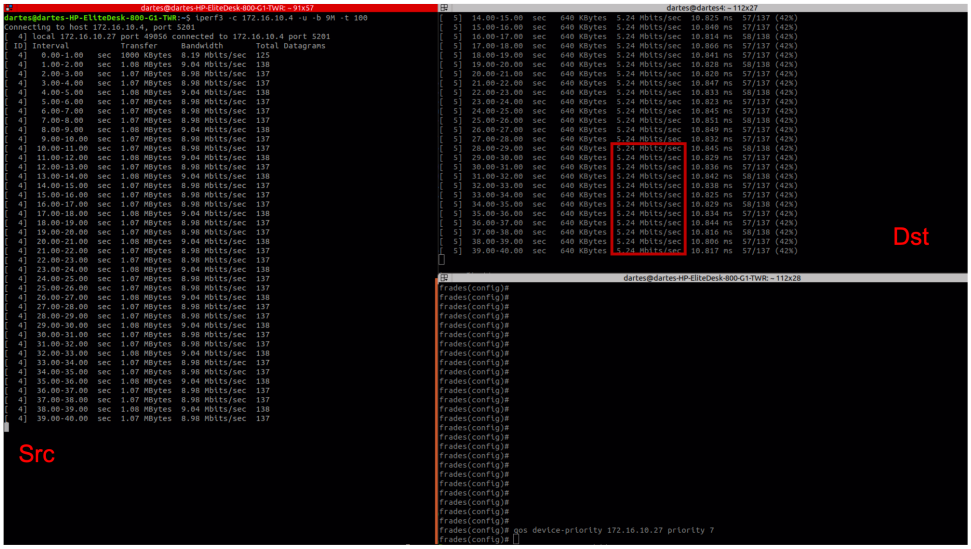

Figure 3.8: UDP Priority Flow Being Mapped to Q2, Left: Host 172.16.10.27 Right Top: Host 172.16.10.4 Right Bottom: Aruba Switch CLI

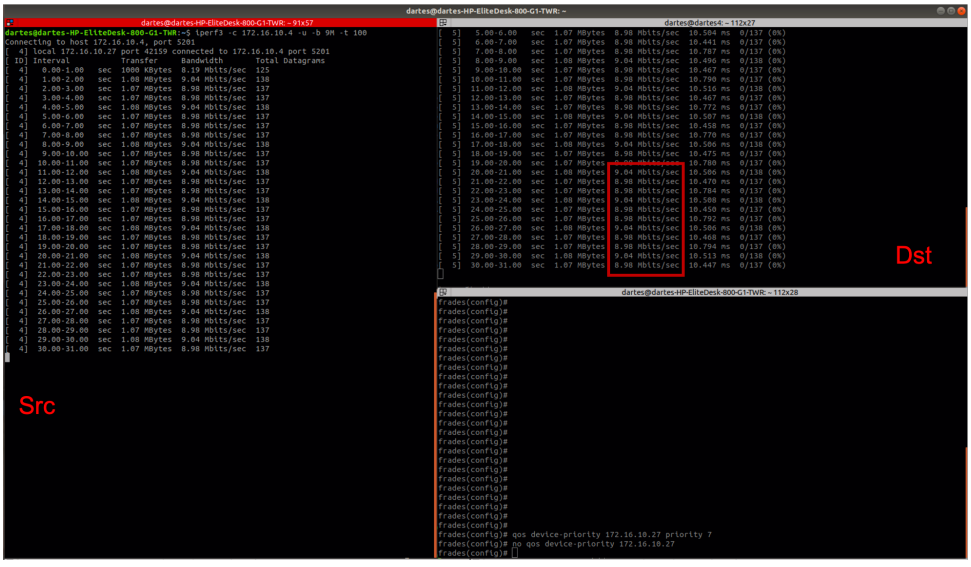

Figure 3.9: UDP Priority flow Being Mapped to Q1, Left: Host 172.16.10.27 Right Top: Host 172.16.10.4 Right Bottom: Aruba Switch CLI

With a scenario of 2 hosts, one being the sender of UDP traffic (172.16.10.27) and another the receiver (172.16.10.4) and both located in two different and connected switches (specifically, 172.16.10.27 is connect to the switch with *frades* as the hostname), if the UDP traffic is being

mapped to Q2 as depicted in Figure 3.8 (Aruba CLI command, giving priority 7 to 172.16.10.27) and being sent at a rate of 9 Mb/s (the port speeds are set to 10 Mb/s), the receiver only sees more or less 5.24 Mb/s. If the UDP traffic is being mapped to Q1, the rate at the sender side and at the receiver side is roughly the same - Figure 3.9.

3.6 Final Remarks

In this Chapter, it was described the designed SDN applications to give QoS guarantees to applications.

The main takeaways of this Chapter are:

- Two SDN applications were created towards the same goal in-hands.
- *ReserveDynApp* provides a solid, robust solution to access the goal. *PriorityDynApp* is a simpler solution but does not have the same advantages of the first.
- *ReserveDynApp* provides QoS guarantees *per flow* and *PriorityDynApp* only provides QoS guarantees *per aggregate of flows* - the priority ones.
- Both applications use as root tools *OpenFlow drop meters* and *802.1p queuing mechanism*.
- Packets are mapped to a specific queue due to its VLAN PCP that exists in the 802.1Q tag in a Ethernet frame.
- UDP and TCP priority flows are treated differently due to the reasons aforementioned.
- For the request reception and processing, it was created a protocol that defines feedback codes and their meaning (regarding the communication of the end-users with the SDN controller).
- In *ReserveDynApp*, because there is a request stage, there is the possibility to apply *admission control* for the priority flows and maintain the existing flows' quality.
- *PriorityDynApp* only defines two classes of priority but it can be extensible to any number - this decision is left to the network manager.

Chapter 4

xDynApp Implementation

This Chapter focuses on explaining how the set of the *xDynApp* SDN applications is implemented - Section 4.1 explains how some design details were software-implemented using the ONOS Java API.

The set of *xDynApp* SDN applications were implemented over the ONOS Controller for the reasons mentioned in 2.1.3. Regarding the machine that hosts the ONOS software, it is required that it has, at least, 16GB of RAM and as Operating System GNU/Linux or MacOS for x86_64 architectures (further information can be found here [12]). Accordingly, for the implementation of *xDynApp* it was used a host machine running Ubuntu 18.04.5 LTS with Intel® Core™ i7-4790 CPU @ 3.60GHz × 8 processor and 16GB of RAM.

As the Forwarding Elements (FEs) that composed the underlying network, it was used the OpenFlow enabled switches Aruba 2930M 24G PoE+ 1-slot Switch (model JL320A) [1].

Section 4.1 provides an overview of the *xDynApp* implementation. From Section 4.2 to Section 4.10, it is discussed how some software parts of the system were implemented- it is explained how some Java structures were used to store in memory relevant information and it is also explained how several ONOS services were used to deploy both SDN applications.

Finally, Section 4.11 gives a short summary of the implementation details discussed in this Chapter.

Some prerequisites need to be meet for the operation of the *xDynApp*, namely:

- Ensure some ONOS applications are pre-installed;
- Add pre-installed flow rules on the switches that compose the network;
- Configure 2 output traffic 802.1p queues at each port;
- Usage of OpenFlow version 1.3.

Further details about the prerequisites can be found in Appendix A.

4.1 Implementation Overview

ONOS provides a way for developers to create and deploy SDN applications. Developers only need to compile and run the ONOS source code and have other dependencies (as, for example, python3) installed on the host machine. As such, the *xDynApp* applications were implemented as ONOS applications.

A ONOS application is presented and created as a Karaf OSGi [1] *component* which is contained within a bundle. ONOS applications, besides leveraging from the modularity provided by the OSGi and as described in 2.1.3, also provides to its developers a modular and extensive API.

The version of the ONOS Java API chosen was the 2.5.1 [13], as it is, at the time of writing, the most recent one. The ONOS controller, through its API, provides a series of *services* and *subsystems* to interact with the underlying network. *xDynApp* uses 8 ONOS provided services as depicted in Figure 4.1.

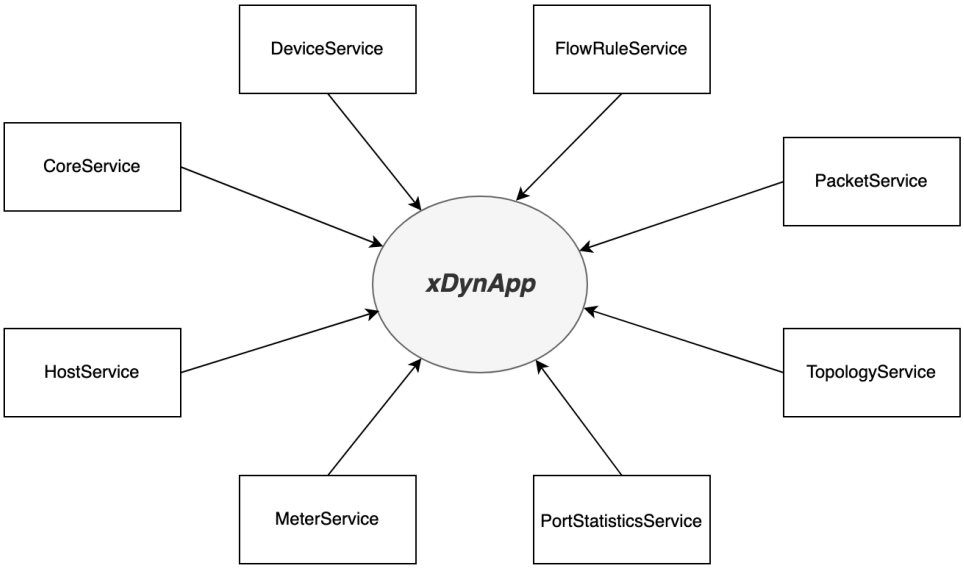

Figure 4.1: ONOS Services Used By *xDynApp*

The following Subsections are presented following the natural workflow of the steps (implicit or explicit) of the *ReserveDynApp* and *PriorityDynApp* algorithm, accordingly to Figures 3.4 and Figure 3.5, respectively.

4.2 App Activation and App Deactivation

Each component (each ONOS application) has available lifecycle event notifications, as for example, the *@Activate* and *@Deactivate* OSGi annotations.

[1] OSGi is a Java framework for developing and deploying modular software programs and libraries. [17]

The *@Activate* annotation is used to notify the component that it is now loaded, resolved and ready to provide service. The activate method (which is proceeded by the *@Activate* annotation) defines some setup procedures upon the application activation. Contrary, the *@Deactivate* annotation, followed by its deactivate method allows the ONOS application developer to define prior steps before the application deactivation.

Regarding the *xDynApp* set of applications, upon the activation of the application, the following procedures are done:

- The created application to the ONOS core using the *Core Service* is registered.
- A packet processor as a director - i.e. this processor can *handle* a packet - with the help of the *Packet Service* is added.
- A Flow Rule Listener using the *FlowRule Service* is added. A listener is a Java entity capable of perceiving changes in the system.
- The meters installation on the network switches is triggered.
- The receiver thread is started (only valid for the *ReserveDynApp*).

Note: Further information about the aforementioned entities and methods is provided in the following Subsections.

```
@Activate
protected void activate() {

    //installing RequestDynApp into ONOS core
    appId = coreService.registerApplication("RequestDynApp");
    //adding a packet processor to process incoming packets
    packetService.addProcessor(packetProcessor, PacketProcessor.director(1));
    // add listener
    flowRuleService.addListener(flowListener);
    // installing meters on the underlying switches
    meterPusher();
    //starting receiver thread for host requests
    receiver.start();

    log.info("Started RequestDynApp");
}
```

Figure 4.2: *ReserveDynApp* Activate Method

Naturally, upon the deactivation of the application, the opposite logical procedures are deployed, as for example: the removal of the Packet Processor from the ONOS system or the removal of the *FlowRule Listener*. Also, all application-installed flows and meters are purged from the switches.

4.3 Request Listener

As specified in the *ReserveDynApp* algorithm design (3.2), the end-users request for a priority communication by sending a XML file-request to the controller which is listening in the 54321 TCP port. This process is achieved by a controller-side thread that opens the connection (the receiver thread aforementioned).

After receiving the file, the *DocumentBuilderFactory* acts as a parser to produce a Document Object Model (DOM) object from the XML content. From the DOM, the necessary information is retrieved and manipulated as simple strings. It is in this step that the checks mentioned in 3.2 are taken. For example, checking that an IP corresponds to a valid hosts involves calling the method *getHostsByIp(IpAddress ip)* from the Host Service. Furthermore, the below Regular Expression (RegEx) is used in order to validate a valid IP syntax:

```
^(([1-9]?[0-9]|1[0-9][0-9]|2([0-4][0-9]|5[0-5])).){3}([1-9]?[0-9]|1[0-9][0-9]|2([0-4][0-9]|5[0-5]))$
```

If there are no errors found, the request is stored into a MultiHashMap Java structure. The map *key* corresponds to the sender IP address and the *value* corresponds to an ArrayList storing the remaining information.

The controller sends the feedback by opening another connection but, this time, as a UDP socket.

4.3.1 Implementation Examples

For the scope of this dissertation, a bash script (named *pushRequest.sh*) was created to be used as the user-side software implementation for sending the XML request. The script simply asks for the user to type in the desired request details, creates an XML document with that information, sends that using a Netcat connection and opens another for feedback reception.

For demonstration purposes, in a testbed composed of 2 hosts (as depicted in Figure 4.3), host 1 wishes to priority communicate with host 2 and so sends a request to the controller.

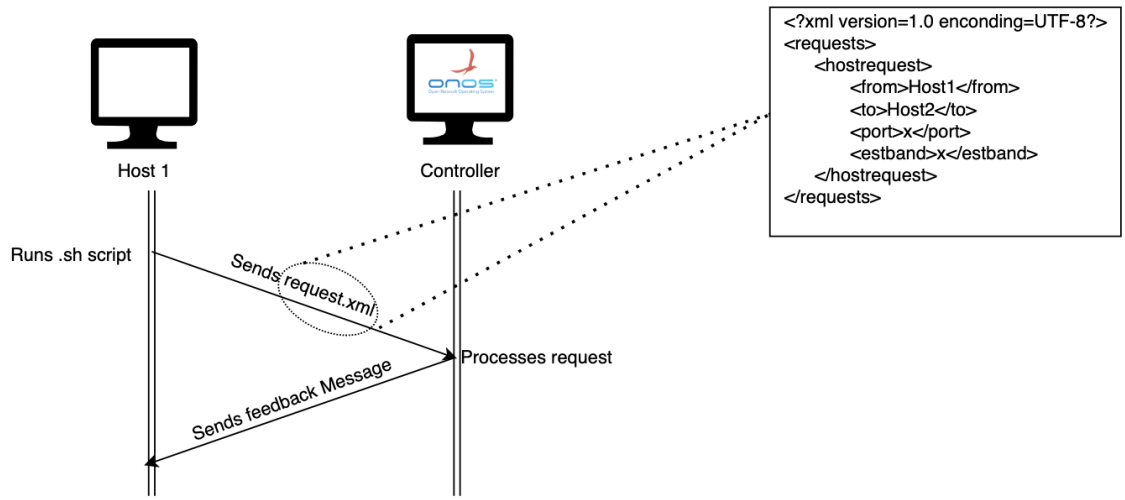

Figure 4.3: Example Testbed and Workflow - Request Listener

The following Figures demonstrate some of the possible errors Host 1 can make and the feedback provided by the controller through the developed *ReserveDynApp*, as defined in Table 3.1.

200 - Everything is OK

Host 1 sends a request to the controller with all fields being valid. As demonstrated on the right-side of Figure 4.4, the user (dartes at host machines dartes4) runs the *pushRequest.sh* script, fills the user prompt with the information desired, opens the connection and, after a few seconds, receive the feedback from the controller.

Left-side of Figure 4.4 shows the controller log file where the network administrator can track relevant information. In this case, the *ReserveDynApp* writes to the log file, as an INFO, that a certain request was received and if it was accepted (meaning, code 200). Moreover, it also writes the current stored requests.

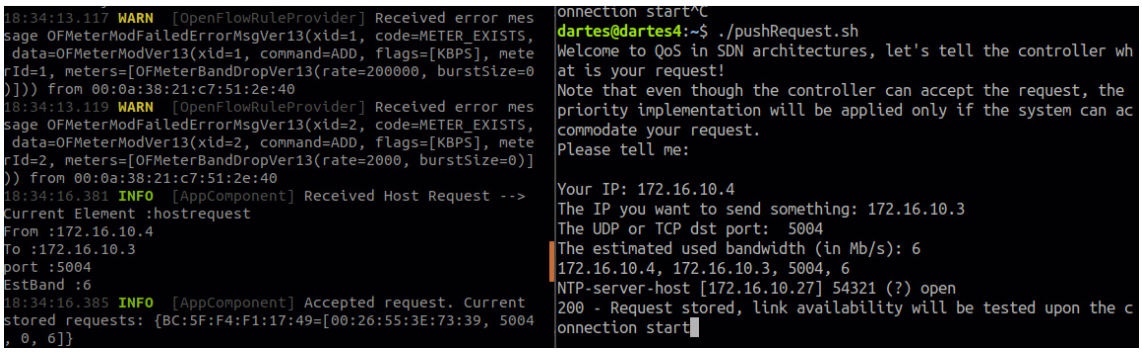

Figure 4.4: Example of a 200 Feedback Code - Left-Side: Controller Log File, Right-Side: Host Sending Request

300 - Bad XML structure

The host sends a XML file request without the specification of the estimated bandwidth and re-

ceives the feedback error 300, meaning that there is a problem with the pre-defined request structure.

```
dartes@dartes4:~$ ./pushRequest.sh
                    Welcome to QoS in SDN architectures, let's tell the controller what is your request!
Note that even though the controller can accept the request, the priority implementation will be applied only if the system can
accommodate your request.
Please tell me:

Your IP: 172.16.10.4
The IP you want to send something: 172.16.10.3
The UDP or TCP dst port:  5004
172.16.10.4, 172.16.10.3, 5004
SDN [172.16.10.27] 54321 (?) open
300 - There is some problem with your request's strucuture
```

Figure 4.5: Example of a 300 Feedback Code

400 - Unrecognized source IP

Host 1 incorrectly writes the source IP as being 172.16.10.6. Even though this is valid a IP in terms of syntax, the IP is not set to any host on the SDN-network. In this case, as the source IP was not correctly specified, the controller does not known a specific IP to send the feedback and so, the feedback code 400 is broadcasted.

```
dartes@dartes4:~$ ./pushRequest.sh
                    Welcome to QoS in SDN architectures, let's tell the controller what is your request!
Note that even though the controller can accept the request, the priority implementation will be applied only if the system can
accommodate your request.
Please tell me:

Your IP: 172.16.10.6
The IP you want to send something: 172.16.10.3
The UDP or TCP dst port:  1234
The estimated used bandwidth (in Mb/s): 5
172.16.10.6, 172.16.10.3, 1234, 5
SDN [172.16.10.27] 54321 (?) open
400 - Controller does not recognize the source IP
```

Figure 4.6: Example of a 400 Feedback Code

4.4 Meter Installation

The process for the meter installation starts with the application retrieving all the available devices (switches) from the underlying network using the *Device Service*. For each discovered device, the application proceed to retrieve all its ports entities (all switch interfaces). For each device and for each port, the port speed is stored in a Java HashSet - note that, the HashSet does not allow duplicate elements and so, for each device, all different available port speeds (the HashSet replaces the repeated values) are obtained. For each device/list of different port speeds combination, a band of type *DROP*, with 20% of the port speed as the rate limit is created. Using the Meter Service, a meter request for the specified device with the pre-created band is created and submitted. This method is depicted in Figure 4.7 and it also represents the first stage of both flowcharts of Figures 3.5 and 3.4.

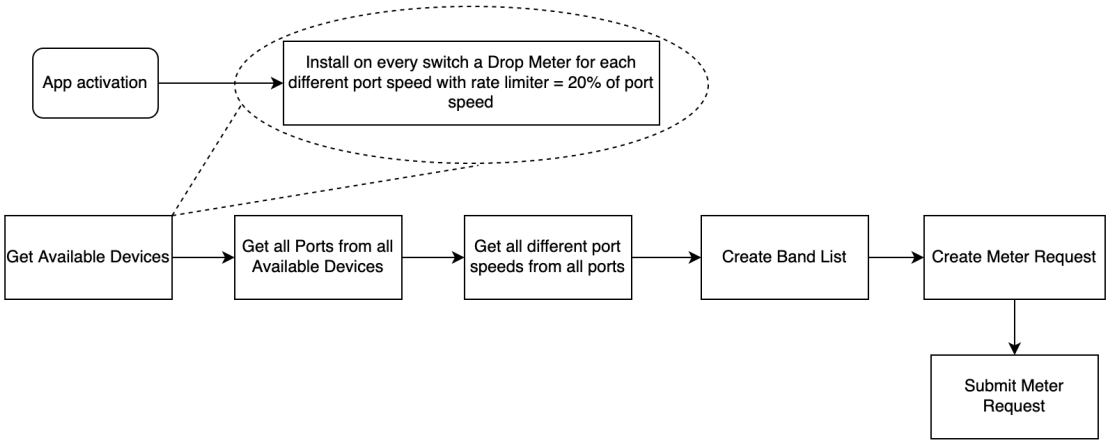

Figure 4.7: Meter Installation

4.5 Packet Processing

According to the pre-installed flow rules on the switches that are on the scope of the network controlled by the SDN controller (ONOS), as before mentioned, all packets are being collected by the controller, i.e. every packet that goes through a switch (and the switch does not know how to deal with the packet) is forwarded to the controller. As aforementioned, at the beginning of the *xDynApp*'s lifecycle, a *Packet Processor* is added into the ONOS core. This processor includes a *process()* method which allows developers to get access to the packet itself. With that in mind, after receiving a packet, *PriorityDynApp* and *ReserveDynApp* work differently as depicted in Figure 4.8.

- *ReserveDynApp* checks, by analysing the different packet layers, if it has the characteristics that match a stored request by analysing the MAC of source and destination and also the used UDP or TCP port. This search is done by checking for these fields in the MultiHashMap that stores the host requests.

- *PriorityDynApp* checks, by analysing the different packet layers, if it has the pre-defined characteristics of a priority class. First, it checks if it is using UDP or TCP as a Transport Protocol, and after it checks for the priority ports that correspond to the used Transport Protocol.

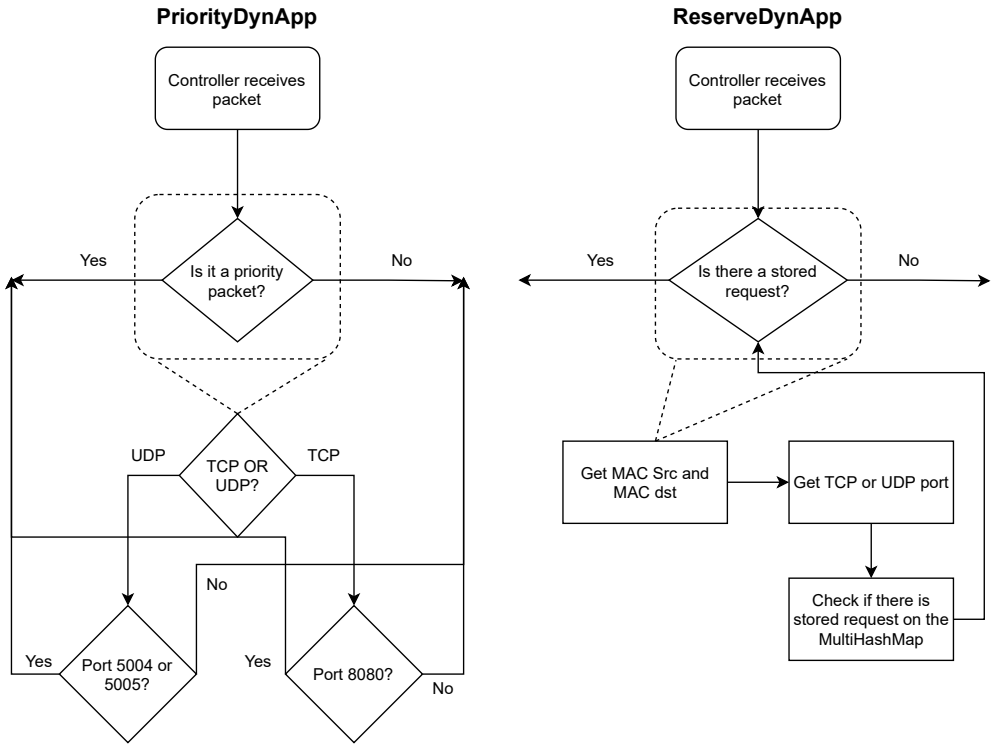

Figure 4.8: Packet Processing Implementation

4.6 Location Checking

Whether it is a priority packet or not, it is checked if the source and destination are on the same switch - if there is the need to simply forward the packet to its switch output port or if there is the need for computing the available paths between source and destination.

Using the *Host Service*, it is possible to get the *Host* entity for both source and destination. The *Host Java Interface* gives a method for retrieving the *Host* location, i.e. the *device entity* (FE) it is connected to and the device used port. With that information, the device of the source and the device of the destination are compared. If they are different, source and destination are on different switches, if the devices are equal and the port is different, they are different hosts on the same switch. This method is depicted in Figure 4.9.

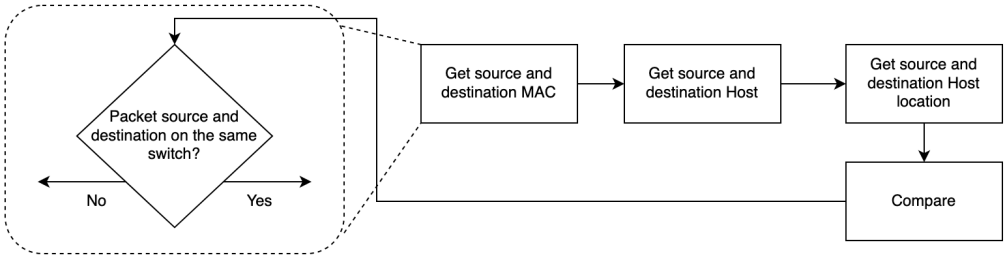

Figure 4.9: Location Checking Implementation

4.7 Paths Discovery

If a source *Host* and the destination *Host* are on different switches, there is the need to calculate the available paths between them in order to successfully implement the connectivity. This process is accomplished using the *getPaths()* method from the *Topology Service*. This method only requires as an argument the current Topology graph (which can be obtained, in real-time with the *currentTopology()* method from the same service), the source device and the destination device.

The *getPaths()* method uses Dijkstra's algorithm to find all the shortest paths between source and destination, computed in terms of hop-count. A Set Java structure of paths where a path is composed of its List of links and a weight, which is an aggregation of the weights of the links the path consists of, are returned. For each link, it is possible to obtain the source and destination device and port. This method is depicted in Figure 4.10.

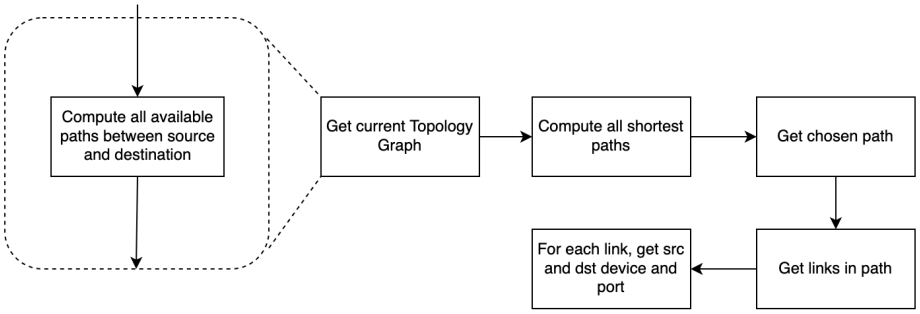

Figure 4.10: Paths Discovery Implementation

4.8 Bandwidth Checking

Regarding the *ReserveDynApp* design, for each request, its feasibility is checked - i.e. if there is available bandwidth to accommodate the request. This procedure is done by using the *load()* method of the *Port Statistics Service*. This method requires as argument the device and port in question and returns a *Load Entity* which has a *rate()* method that returns the current observed rate (in bytes/s) on the port.

4.9 Joint Paths Discovery

The meter mechanism for giving QoS to priority UDP flows depends on knowing the paths of all present UDP priority flows so they can be compared to the path used by a non-priority flow that is about to be installed.

When a UDP priority flow is identified and when the the path discovery process takes place, two structures are created:

1. A MultiHashMap that has as *key* an index (which is a global integer value) and as *value* a List of information about the UDP flow: MAC source, MAC destination and port.

2. A MultiHashMap that has as *key* an index (the same used in the MultiHashMap used to store the information) and as *value* a List of all the devices and ports that the UDP priority flow is using.

When a non-priority flow rules are about to be installed, and, again, when the the path discovery process takes place, each port of its used path is compared to each port of every UDP priority flows used ports. As mentioned in Chapter 3, the smallest port speed of the joint ports is chosen to be the reference to the meter that is acting as rate limiter.

However, even though it is possible to know with simple data structures about the existence of a UDP priority flow, it is not possible to effectively know if the UDP flow is *still present* in the system. For that, a *Flow Rule Listener* that monitors the removal of flow rules is implemented. Upon each event detection, it is possible to know what flow rules were removed and compare the matching field of the flow rule with the information previously stored in the MultiHashMap. If they correspond, both entries of both MultiHashMap are deleted (by its key, which is equal). This way, it is assured that a non-priority flow's path is being compared to valid UDP priority paths. This method is depicted in Figure 4.11.

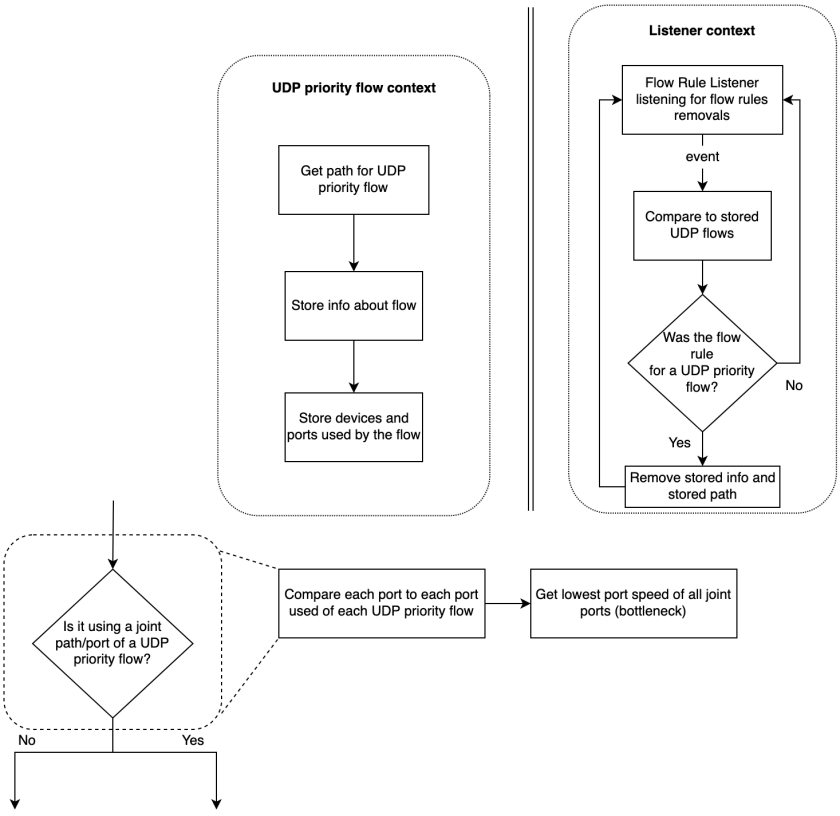

Figure 4.11: Joint Paths Discovery Implementation

4.10 Flow Rules Installation

The final step of either algorithm after receiving a valid packet is the installation of flow rules on the underlying switches. Those flow rules follow the *rules* and *actions* mentioned in Sections 3.2 and 3.3.

The very first step of the flow rule installation is to create the *rules* and *actions*. The earlier is accomplished by using the *DefaultTrafficSelector builder* which creates a *TrafficSelector* entity. As the name states, this entity *selects* some traffic according to the pre-defined *rules*. The latter is accomplished by using the *DefaultTrafficTreatment builder* which creates a *TrafficTreatment* entity - correspondingly, this entity *treats* the traffic that match the *TrafficSelector* according to the pre-defined *actions*.

Even though the selector and the treatment are defined, it is still missing some properties of the flow rule. The creation of the flow rule is performed by the *DefaultFlowRule builder* and encompasses the following:

- Definition of what ONOS application is installing this flow rule. Whenever a ONOS application is submitted into the ONOS core (as described in 4.2), an application ID that identifies the application is created.
- Definition of the desired *TrafficSelector*.
- Definition of the desired *TrafficTreatment*.
- Definition of the flow rule priority (integer value) within its table. A packet is confronted with a flow rule (with its matching field - i.e. the selector) by a descending order of the flow rule's priority in the flow table of each device. It was decided that non-priority flows have a *129* priority and priority flows have a *130* priority. The decision for the higher value of priority flows is that a packet that matches the *rules* of a priority flow always matches the *rules* of a non-priority one. This insures that, if there is a priority flow installed, a packet that matches it, is treated by it.
- The definition of the type of timeout. Non-priority flows are defined to have a HardTimeout of 10 seconds meaning that imperatively they are uninstalled after 10 seconds. Priority flows have a IdleTimeout of 20 seconds, meaning that the flows are only uninstalled after 20 seconds of not having packets that match the criteria. This observed matching is done by the ONOS controller that is obtaining from the switches, the flows statistics.
- Finally, the definition of the flow rule recipient device, i.e. the device that this flow rule is to be installed.

Ultimately, the actual process of submitting the Flow Rule into the system is the responsibility of the *applyFlowRules(FlowRule)* method from the FlowRule service. This method is depicted in Figure 4.12 and in Figure 4.13.

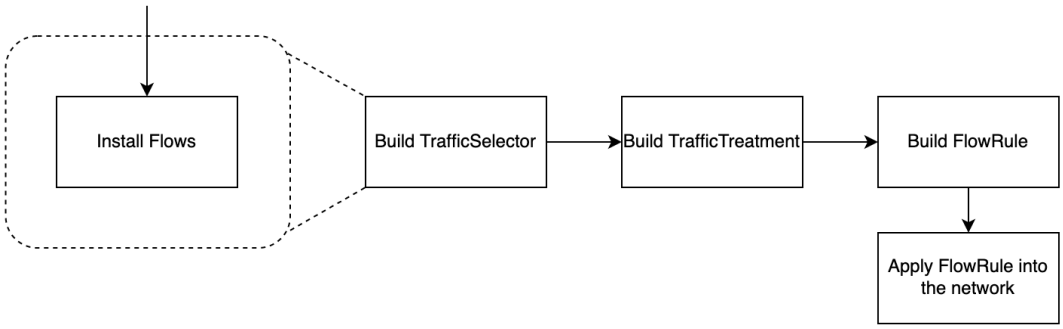

Figure 4.12: Flow Rules Installation

```
selector1 = DefaultTrafficSelector.builder()
        .matchEthSrc(src).matchEthDst(dst).matchEthType(Ethernet.TYPE_IPV4).matchInPort(inPort).matchIPProtocol((byte)17).matchUdpDst(port).build();

output1 = DefaultTrafficTreatment.builder().setVlanPcp(bytePcpUDP).setOutput(outPort).build();

flowRuleService.applyFlowRules(DefaultFlowRule.builder()
        .fromApp(appId)
        .withSelector(selector1)
        .withTreatment(output1)
        .withPriority(130)
        .withIdleTimeout(20)
        .forDevice(deviceId)
        .build());
```

Figure 4.13: Example of a UDP Priority Flow Rule Installation

4.11 Final Remarks

In this Chapter it was granted how both the *ReserveDynApp* and *ReactiveDynApp* were implement - first, in a hardware point of view and second, in a software point of view.

The main takeaways of this Chapter are:

- ONOS provides a simple, interactive and intuitive way to manage a SDN-based network. Even though the integration with the Aruba switches was not subtle due to some Aruba unsupported actions, it was easy to overcome that issue using the ONOS REST API to manually install flow rules.
- The Aruba OS provides built-in QoS support that ultimately ended up being integrated with the SDN solution.
- The Aruba OS also provides an intuitive OpenFlow configuration framework and has complete available information in [4].
- The set of the *xDynApp* applications were built using the ONOS ability to allow developers to easily deploy SDN applications by using its Java API. *xDynApp* uses 8 ONOS services from its API.
- ONOS Java API highlights the benefits of SDN by granting a series of Java methods, e.g. by having a network topology view, there are methods that calculate the paths between source

and destination using different algorithms or even the ability to dissect the received packet layer by layer.

- A thread was used to create a separated non-OpenFlow channel to allow end-hosts to communicate non-OpenFlow messages with the controller.

Chapter 5

xDynApp Evaluation and Validation

In this Chapter, it is presented the results obtained from all the experiments performed, in order to understand how the *xDynApp* SDN applications operate under different scenarios and if their behaviour corresponds to what was desired. In Section 5.1 the used testbed and specifications about the different components are explained. Section 5.2 lists and briefly describes the tools used throughout the experiments set development. In Section 5.3 how the experiment set is organized is explained. From Section 5.4 to Section 5.6 every experiment taken is presented. For each experiment, the statement, how the experiment is implemented using the testbed, the results and discussion of the results are granted. Finally, Section 5.7 gives a short and broader summary of the main results.

5.1 Experiment Setup

As mentioned in Chapter 1 one of the main goals of this dissertation was to implement and test a SDN-algorithm over a real, not-simulated network. For that, three OpenFlow-enabled switches were used and 1 simple switch was also part of the testbed. Furthermore, four host machines were used. The testbed implementation for the experiments is depicted in Figure 5.1 and explained below.

The simple switch connects one port to the FEUP's network, another to one interface of the controller machine and the remaining connect the Out-of-Band management (OOBM) interface of each switch. Furthermore, the four hosts belong to the same private network - 172.16.10.0/24. It should be highlighted that the controller machine has two network interfaces: one connects to the host's network, being assigned with the 172.16.10.27 IP address; the other connects to the simple switch and has a FEUP IP address, 192.168.106.129. The first was created so that the controller could have a direct non-OpenFlow communication with the hosts, acting as an extra host machine and the latter is the used interface to communicate with the OpenFlow-enabled switches and it also provides connection to the Internet via FEUP's network. More information about each machine is provided in Table 5.1.

As OpenFlow-enabled switches, it was chosen to use the Aruba 2930M 24G PoE+ 1-slot Switch (model JL320A) as mentioned in Chapter 4. More information about each switch is provided in Table 5.2.

Note that, even though the testbed is presented in the configuration of Figure 5.1, that does not necessarily means that the configuration is fixed throughout the experiments - e.g. in some, only two OpenFlow enabled switches are used and the host location is often interchangeable. Moreover, for each experiment a preview of the logical testbed is given.

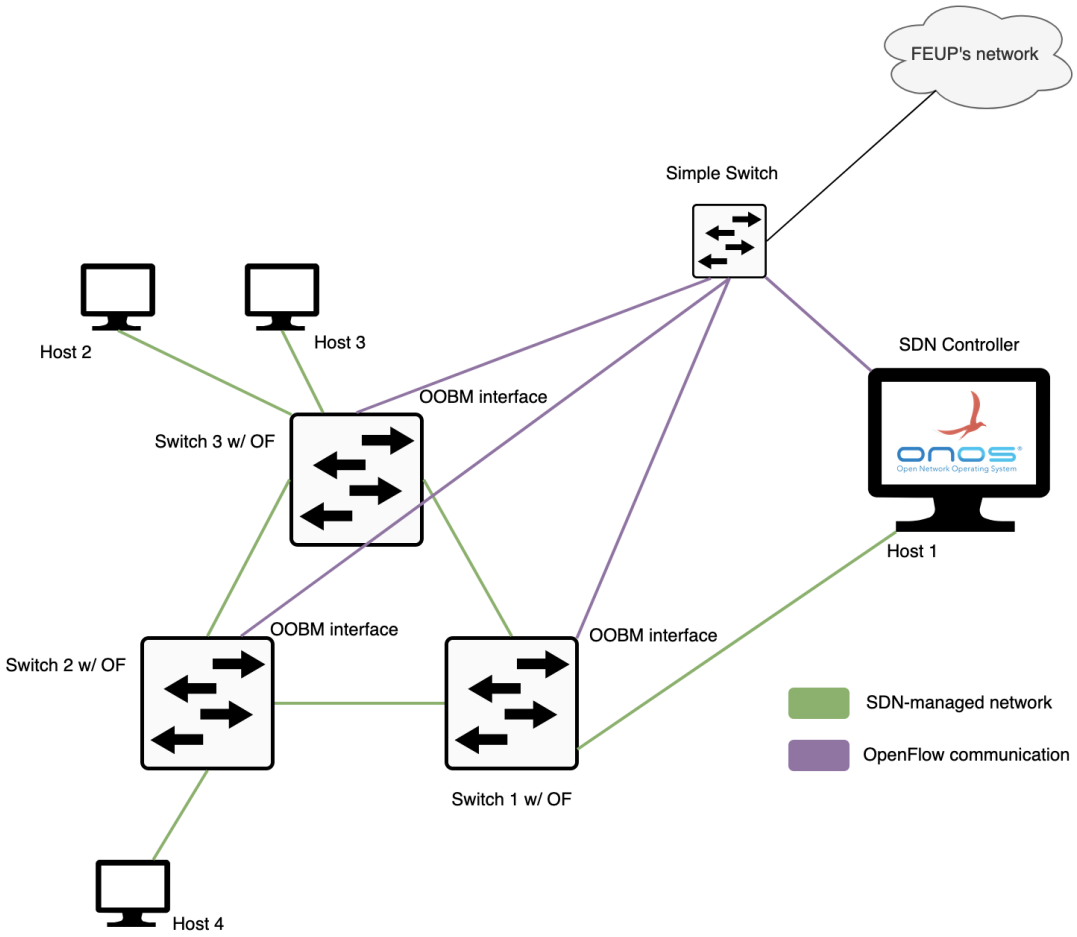

Figure 5.1: Testbed Used in the Evaluation and Validation Phase

5.2 Traffic Generation and Networking Monitoring Tools

For the experiment set development it was used 4 main tools for different purposes: VLC Media Player [18], Wireshark [20], iPerf/iPerf3[8] and Aruba OS port-speed commands.

VLC Media Player

	Operating System	Operating System type	Processor	RAM	FEUP's IP Address	Private Network Address
Controller Machine	Ubuntu 18.04	64-bit	Intel® Core™ i7-4790 CPU @ 3.60GHz × 8	16 GB	192.168.106.129	172.16.10.27 (= Host 1)
Host 2 Machine	Ubuntu 18.04	64-bit	Intel ® Core™ 2 Quad CPU Q9300 @ 2.50GHz × 4	4 GB	-	172.16.10.2
Host 3 Machine	Debian 10	64-bit	Intel ® Core™ 2 Quad CPU Q9400 @ 2.66GHz × 4	8 GB	-	172.16.10.3
Host 4 Machine	Debian 10	64-bit	Intel ® Core™ i5-4570 CPU @ 3.20GHz × 4	8 GB	-	172.16.10.4

Table 5.1: Specifications of Used Controller/Host machines

	Model	OOBM IP Address
Switch 1	Aruba 2930M 24G PoE+ 1-slot Switch (model JL320A)	192.168.106.124
Switch 2	Aruba 2930M 24G PoE+ 1-slot Switch (model JL320a)	192.168.106.126
Switch 3	Aruba 2930M 24G PoE+ 1-slot Switch (model JL320A)	192.168.106.128
Simple Switch	Aolynk DR814	-

Table 5.2: Specifications of Used Switches

VLC Media Player is a free and open-source media player software and it also functions as a streaming media server/client. It is developed by the VideoLan Project. Particularly for this dissertation, it was used its capability for streaming multimedia as a use case, especially for the *PriorityDynApp* as it has pre-defined priority classes. VLC Media Player gives miscellaneous options to stream multimedia, as, for example, using RTP/MPEG Transport Stream or HTTP. It also offers a variety of video and audio codecs - H.264, H.265, MP3 etc.

iPerf/iPerf3

iPerf/iPerf3 is a open-source software tool to perform several network performance measurement tests on IP networks. However, for this work, it was used its capabilities to deploy a source for congestion traffic i.e. traffic that does not contain any information and whose only purpose is to try to fill the channel or as a source of a priority flow. iPerf works in a client-server model and can create data streams between both ends - unidirectionally or bidirectionally. The data streams can be customizable to be either TCP or UDP. It is also possible to define the Maximum Transmission Unit (MTU) size (option *-l* on the command syntax), the rate of transmission (flag *-b*) and the time

interval it is alive (flag *-t*).

It was chosen iPerf3 (the most recent version) when TCP streams were used and iPerf when UDP streams were used. This difference relies on the fact that UDP iPerf3 streams are generated firstly with a TCP handshake between client and server whereas iPerf uses 100% pure UDP streams.

Wireshark

Wireshark is a network protocol analyzer tool capable of live-capturing packets and displaying its TCP/IP information with fine detail (using the pcap API). Wireshark is used for the most varied purposes like network troubleshooting, security examination, protocol debugging and it is also used as an academic teaching tool. The data to be analyzed can be inspected (live or offline) under several available filters e.g. source IP/destination IP, TCP/UDP port, protocols, TCP sessions, time etc. The offered granularity enables complete studies and statistics retrieval which can be outputted to XML, PostScript, Comma Separated Values (CSV) and plain text files.

Even though Wireshark's goal is not to provide high-level user-friendly analysis, it offers a complete analysis for RTP streams under its Telephony menu. That tool and others were used to get QoS metrics for different experiments - further information can be found when suitable throughout the following Section.

Aruba OS commands

For some experiments it is necessary to have different port speeds for different ports. This is accomplished by using an Aruba CLI command. As an example, below it is demonstrated how it is used to set the interface(port) 1 to 10 Mb/s using full-duplex mode:

```
interface 1 speed-duplex auto-10
```

5.3 Overview and Organization of Experiments

The subsequent analyzed experiments are of two types:

1. **Validation:** the *xDynApp* functionalities and capabilities are tested to validate that they are working correctly and as expected.
2. **Evaluation:** the behaviour of *xDynApp* is evaluated under QoS metrics.

Also, the experiments are divided in categories regarding what is being validated/evaluated as explained below:

- **Validation of Core Functionalities:**
  - **Experiment 1:** Bandwidth Guarantee Upon Reservation
  - **Experiment 2:** Rate Limit Non-Priority Flows

  - **Experiment 3:** Dynamic Flow Update On P/NP Traffic Changes

- **Routing and Bandwidth Reservation Across Entire Route:**

  - **Experiment 4:** Path Reservation in Presence of a Single-Path
  - **Experiment 5:** Path Reservation in Presence of Multiple Paths

- **Impact on QoS Metrics:**

  - **Experiment 6:** Evaluate QoS Metrics with Priority Flows

Furthermore, some experiments are exclusively thought and performed for either *ReserveDynApp* or *PriorityDynApp*. Others, as both applications use the same underlying mechanisms, for both. The experiments are demonstrated following the plan:

- **Statement**: a concise statement explaining the experiment purpose proceed by the specification of the experiment type.

  - (if Validation type) **Success Condition**: a concise statement explaining the desirable behaviour.
  - **Application:** which *xDynApp* is being analyzed.

- **Implementation:** the description of the implementation process and the possible necessary workflow of the experiment.

- **Results and Observations:** where appropriate results are presented as well as its observations and discussion.

5.4 Validation of Core Functionalities

5.4.1 Experiment 1: Bandwidth Guarantee Upon Reservation

Statement

> Validation - *Validate application ability of guaranteeing requested bandwidth to a priority flow while alive.*

Application: *ReserveDynApp*.
Success Condition: a priority flow has its requested bandwidth constant even if a non-priority flow is using a joint path/port.

Implementation

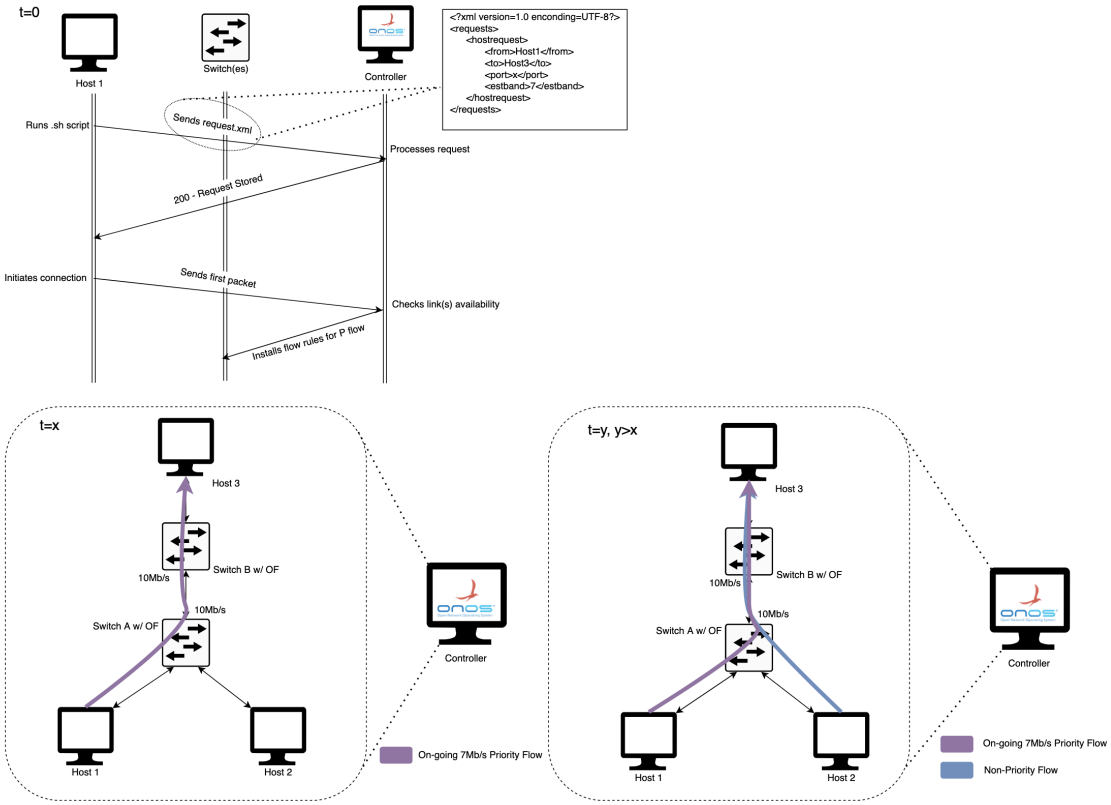

Figure 5.2: Up: Experiment 1 Workflow, Down: Experiment 1 Logical Testbed

As depicted in Figure 5.2, Host 1 wishes to perform a priority 7 Mb/s communication with Host 3 - for that sends to the controller a request with the following fields:

Listing 5.1: XML request structure

```
<?xml version="1.0" encoding="UTF-8"?>
   <requests>
      <hostrequest>
         <from>Host1Ip</from>
         <to>Host3IP</to>
         <port>5004</port>
         <estband>7</estband>
      </hostrequest>
   </requests>
```

The request is correctly formatted and so the controller stores the request. The ports that connect both involved switches are set to use 10 Mb/s as port speed. When Host 1 starts the actual communication (exemplified by a TCP/UDP iPerf3 stream) there is no traffic using the path between Host 1 and Host 3 and so there is available bandwidth on the bottleneck queue (egress queue of output port of switch A) - the application proceeds to install flow rules for a priority (P) flow. After a while, a non-priority (NP) TCP iPerf3 flow (from Host 2 to Host 3) enters the system

using the joint ports demonstrated in Figure 5.2. This non-priority flow is trying to have access to the maximum channel capability (in the case, 10 Mb/s) which is accomplished by setting the *-b 0* flag on the iPerf command.

Results and Observations

The main metric to be analyzed is the received bandwidth at Host 3 from both Host 1 and Host 2 - this way, it is possible to validate that the requested bandwidth is provided. This metric is presented using the built-in I/O Wireshark (running on Host 3) graph ([19]) where it is possible to obtain the visualization of the bandwidth (in bits/sec) against time. Moreover, for complete results, it was used both a UDP priority flow and a TCP one.

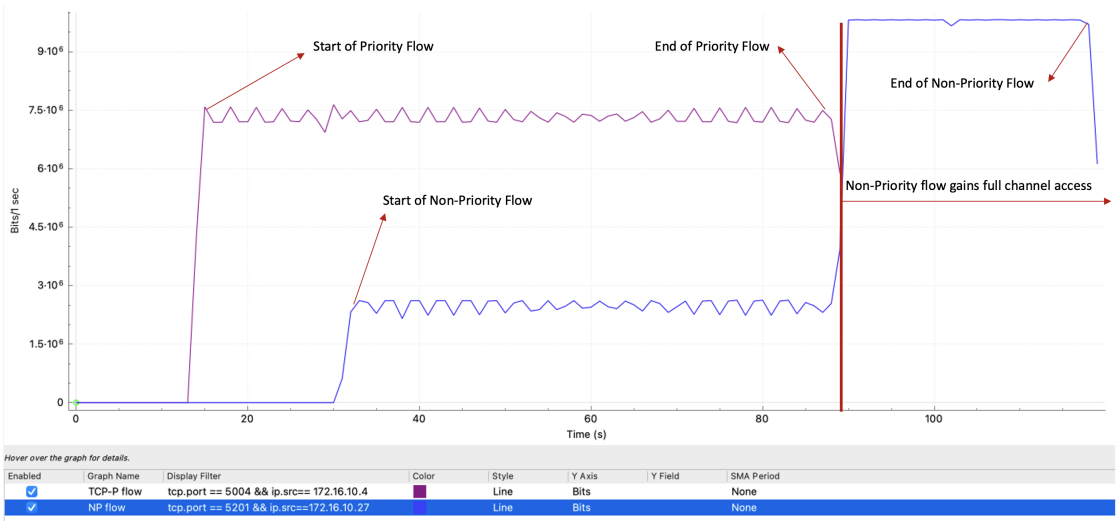

Figure 5.3: Experiment 1 - A TCP Priority Flow Has Guaranteed Bandwidth

As presented in Figure 5.3, the priority flow (represented in purple) starts at about t=17 seconds and it is using an approximate bandwidth of 7 Mb/s, as announced. At t=30 seconds, the non-priority flow (represented in blue) enters the network. Validating the *ReserveDynApp* expected behaviour, the bandwidth of the TCP priority flow is kept constant even after the non-priority flow starts. Immediately after the priority flow termination, the non-priority one starts, gaining access to all the channel capacity because it was using the *-b 0* iPerf option, as aforementioned.

Figure 5.4: Experiment 1 - A UDP Priority Flow Has Guaranteed Bandwidth

As presented in 5.4, the priority flow (represented in green) starts at about t=25 seconds and it is using an approximate bandwidth of 7 Mb/s, as announced. At t= 45seconds, the non-priority flow (represented in black) enters the network. Validating the *ReserveDynApp* expected behaviour, again, the bandwidth of the UDP priority flow is kept constant even after the entry of the non-priority flow. Some seconds after the priority flow termination, the non-priority one starts gaining access to all the channel capacity because it was using the *-b 0* iPerf flag, as aforementioned.

One difference, although not strictly related to the experiment, should be mentioned: the time interval that elapses between the end of the P flow and the NP flow starts using the whole channel bandwidth. The TCP priority scheme is accomplished by the 802.1q queuing mechanism (using a WRR scheduler) that means that if there is no packet in Q2, all packets of Q1 are immediately served, thus justifying the immediate response of the non-priority flow in Figure 5.3. The UDP priority scheme works using the meter mechanism. After the UDP flow termination, it has to elapse a timeout of 20 seconds so that the UDP priority flow rule is uninstalled. Only after that and after the expiration of the already-installed non-priority flow is the application capable of recognizing that there is no more a UDP priority flow on that path and that the non-priority one does not need to point to a meter.

5.4.2 Experiment 2: Rate Limit Non-Priority Flows

Statement

Validation - *Validate application capability to properly meter non-priority traffic.*

Application: *PriorityDynApp* and *ReserveDynApp*.

Success Condition: If a non-priority flow is using one or more common ports of those that a UDP priority flow is using, then the application recognizes it and installs flow rules that point to a

meter with the rate limit of 20% of the joint port that has the lowest port speed. If the non-priority flow is not using any joint port then the flow rules point to no meter.

Implementation

For the implementation of experiment 2, 4 end-hosts are used. As depicted in Figure 5.5, there is a UDP priority flow from Host 1 to Host 4, passing through Switch A and Switch B. For demonstration purposes, there are 3 additional non-priority flows all with different treatments:

- From Host 3 to Host 4 there is a NP flow that uses 3 joint ports of the ports the UDP P flow is using - the one(s) with the lowest port speed has 10 Mb/s and thus, the NP flow rules point to a meter with rate limit = 2 Mb/s (which is 20% of 10 Mb/s).
- From Host 1 to Host 2 there is a NP flow that uses 1 joint port of the ports the UDP P flow is using with a port speed of 1000 Mb/s and so, the NP flow rules point to a meter with rate limit = 200 Mb/s (which is 20% of 1000 Mb/s).
- From Host 2 to Host 3 there is a NP flow that does not use a joint port of the ports the UDP P flow is using. That flow is not pointing to any meter.

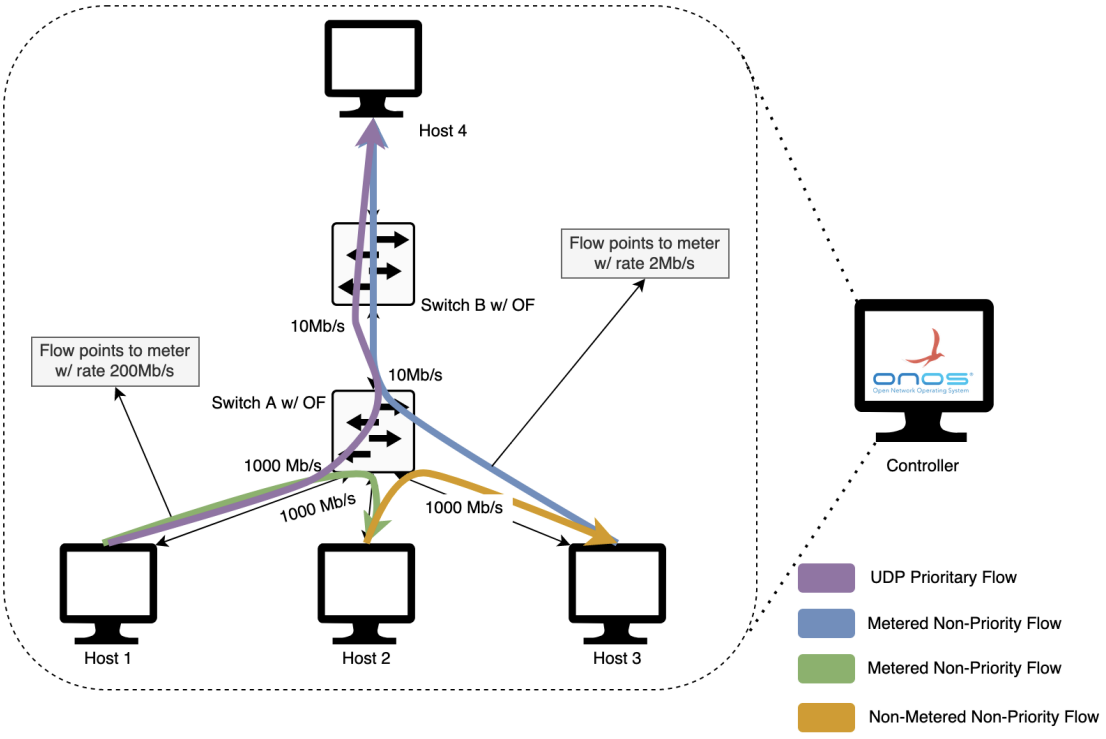

Figure 5.5: Experiment 2 - Logical Testbed

Results and Observations

The experiments results are obtained using the ONOS built-in real-time load per link display provided by the ONOS GUI. Also, all flows are generated using iPerf. The non-priority flows are TCP streams with the iPerf *-b* option enabled, meaning that they are trying to use all the channel's bandwidth.

Even though this experiments is valid for both *PriorityDynApp* and *ReserveDynApp*, the results were obtained using the latter. The mapping between IP address and experiment 2 host names is as follows:

- Host 1: 172.16.10.4
- Host 2: 172.16.10.27
- Host 3: 172.16.10.2
- Host 4: 172.16.10.3

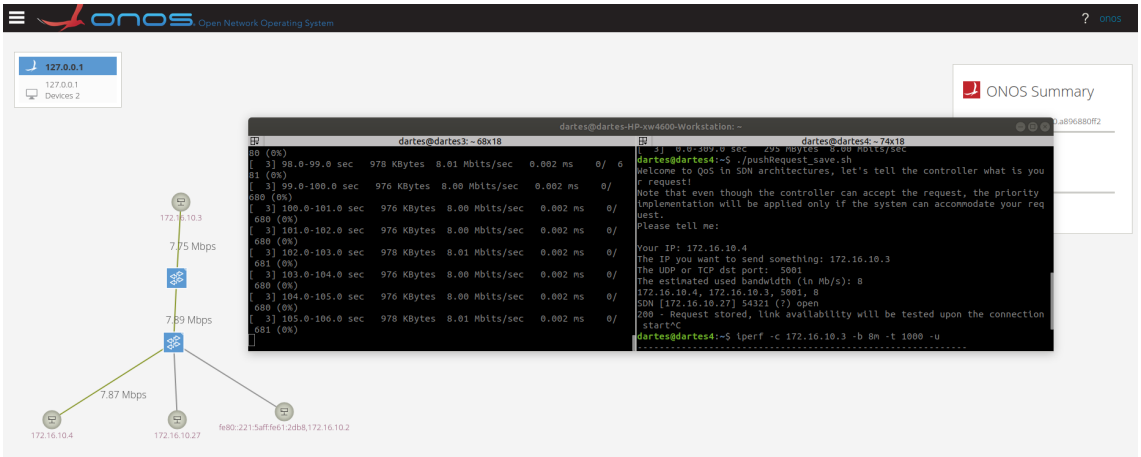

Figure 5.6: Experiment 2 - UDP Priority Flow

As demonstrated in Figure 5.6, Host 1 (*dartes4*) sent a request for a priority flow with Host 4 as the destination and with an estimated bandwidth value of 8 Mb/s (right side terminal window). When the actual communication starts, the request is accepted as there is available bandwidth on the path that connects both hosts.

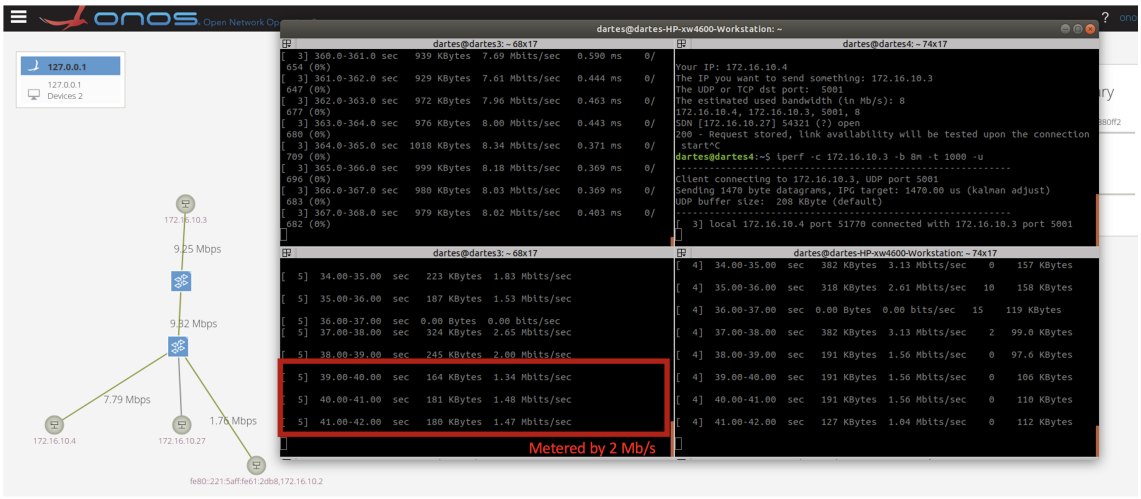

Figure 5.7: Experiment 2 - NP Flow Metered by 2 Mb/s

When Host 3 communicates (with no-priority) with Host 4, as it was using a joint port with 10 Mb/s as port speed (the lowest one), its rate is limited to 2 Mb/s, as seen in Figure 5.7.

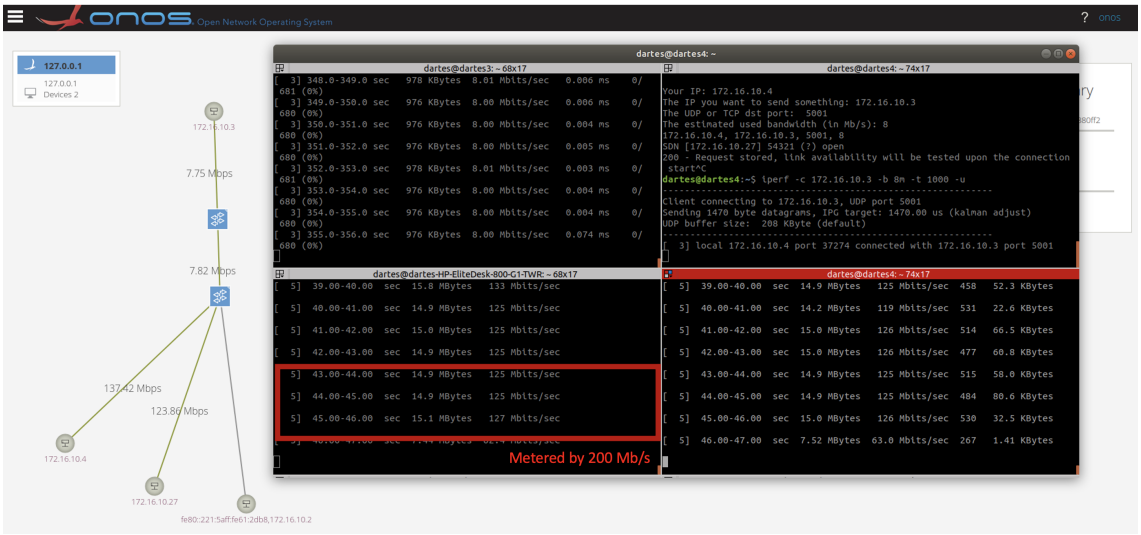

Figure 5.8: Experiment 2 - NP Flow Metered by 200 Mb/s

When Host 1 communicates (with no-priority) with Host 2, as it was using a joint port with 1000 Mb/s as port speed, its rate is limited to 200 Mb/s as seen in Figure 5.8.

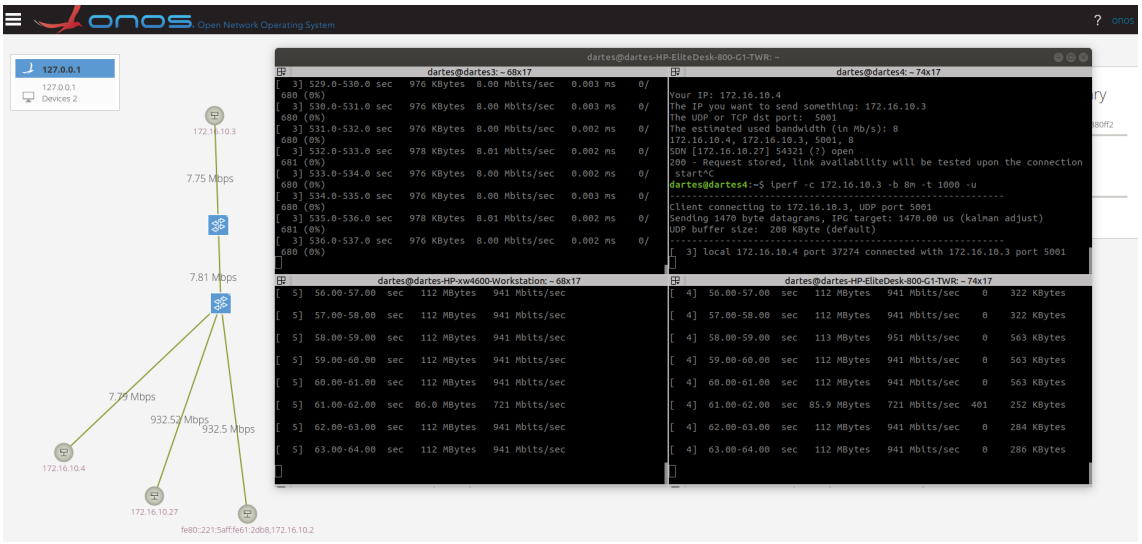

Figure 5.9: Experiment 2 - NP Flow Not Netered

As depicted in Figure 5.9, when Host 2 non-priority communicates with Host 3, as it is not using a joint port, it is not metered by any meter and it uses all the channel's bandwidth it can use.

5.4.3 Experiment 3: Dynamic Flow Update On P/NP Traffic Changes

Statement

> Validation - *Validate application ability of perceiving changes in priority flows as depicted in Figure 5.10.*

Application: *PriorityDynApp*.

Success Condition: the application correctly identifies that an end-host changed the priority of the sending traffic and proceeds to install correct flow rules.

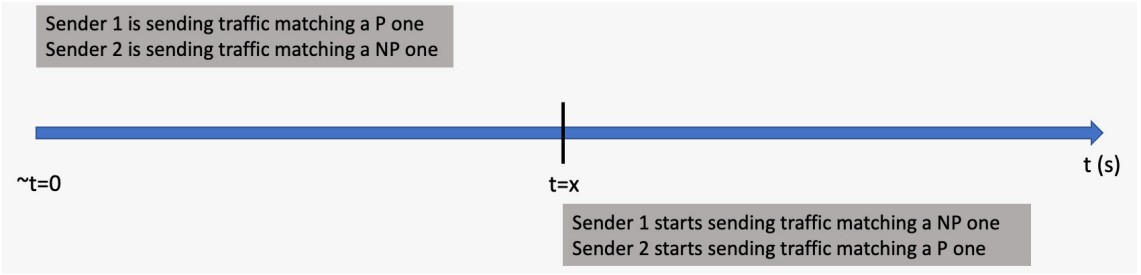

Figure 5.10: Experiment 3 - Workflow

Implementation

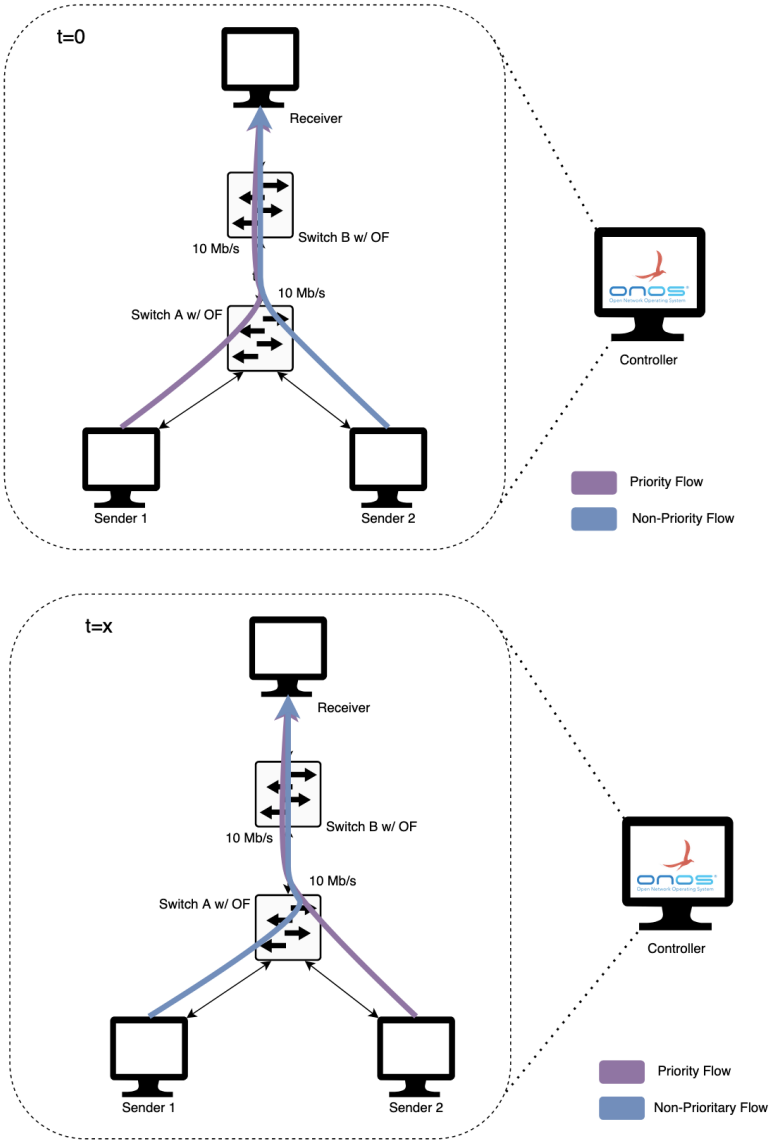

Figure 5.11: Experiment 3 - Logical Testbed

As demonstrated in Figure 5.11, at t=0, Sender 1 is sending traffic that matches a priority one (in the case, it was also used the VLC Media Player and the same video) and Sender 2 is sending traffic that matches a non-priority one (again, a flow composed of 3 TCP streams). At t=x, the scenario is exactly the opposite - Sender 2 is sending priority traffic and Sender 1 non-priority one. Again, the non-priority flows are always started before the priority ones and those are always started after a stable state of the first. The high-level steps are:

1. Sender 2 starts sending non-priority (NP) traffic.

2. Sender 1 starts sending priority (P) traffic.

3. Sender 1 stops sending P traffic.
4. Sender 2 stops sending NP traffic.
5. Sender 1 starts sending NP traffic.
6. Sender 2 starts sending P traffic.
7. Sender 2 stops sending P traffic.
8. Sender 1 stops sending NP traffic.

Results and Observations

RTP Priority Stream

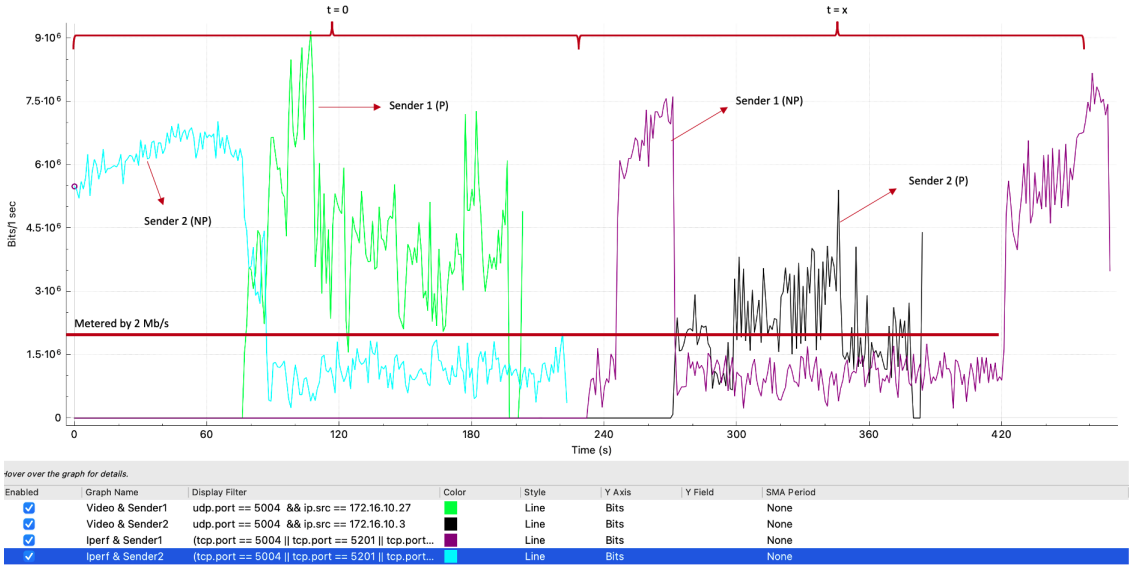

Figure 5.12: Experiment 3 - RTP Priority Stream Results

The main metric to analyse for this experiment, is the bandwidth on the Receiver side regarding both Senders. This metric is possible to obtain using the Wireshark I/O graphic tool.

The application's ability to perceive changes is possible due to the timeout mechanism described in Chapter 3 and this ability is demonstrated in Figure 5.12, which presents the priority a RTP (UDP) stream has. The blue and green lines correspond to the t=0 moment in time and the purple and black lines correspond to the t=x moment. Sender 2 starts sending non-priority traffic and almost immediately after Sender 1 starts sending priority traffic, Sender 2 substantially drops the sending rate. Sender 1, now represented by the purple line, starts sending non-priority traffic and later, Sender 2, being represented by the black line starts sending priority one - the exact same priority mechanism happens as in the t=0 moment, i.e. the flow rules were not propagated in time but instead, the controller installed appropriate flow rules for each moment.

The different bandwidth values for both the priority flows are explained by the fact that each host (Sender 1 and Sender 2) have different computational resources with impact on the streaming potential.

HTTP Priority Stream

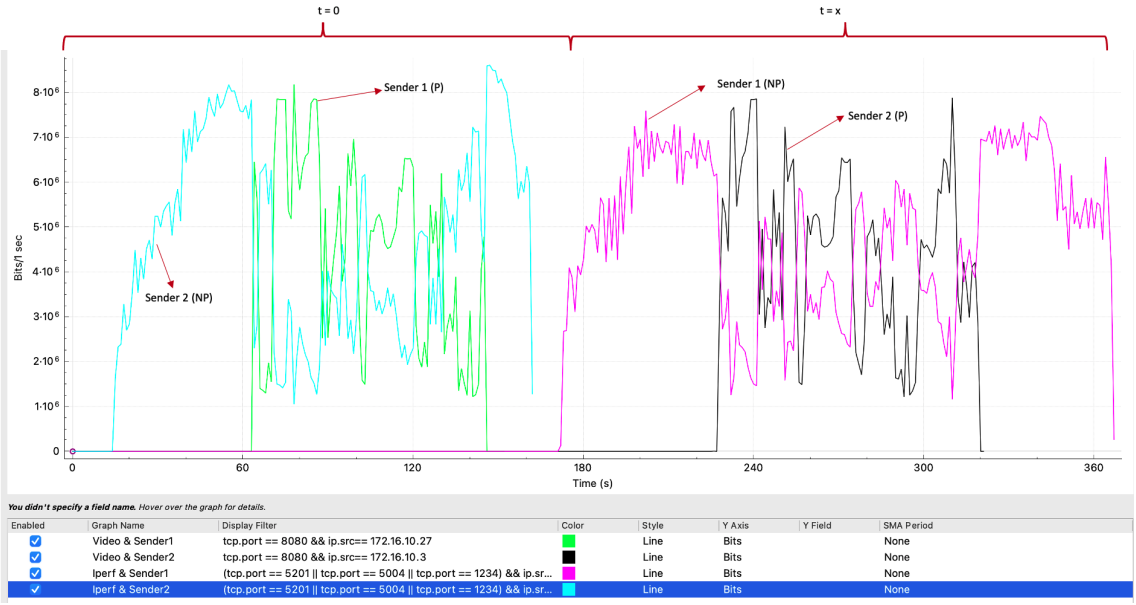

Figure 5.13: Experiment 3 - HTTP Priority Stream Results

When the priority flow corresponds to a TCP stream (Figure 5.13), even though it still has priority over the non-priority flow, the results are less visually obvious: the non-priority flow is not metered as it was the case of Figure 5.12, instead, the NP flow complements the bandwidth used by the TCP P flow. As discussed in Chapter 3, the 802.1p queuing mechanism used in this case allows the NP flow to gain access to more than the Guaranteed Minimum Bandwidth *if* the priority flow is also not requesting its minimum.

5.5 Routing and Bandwidth Reservation Across Entire Route

5.5.1 Experiment 4: Path Reservation in Presence of a Single-Path

Statement

> Validation - *Validate application capacity to check if, upon bandwidth reservation request and in a single-path environment, there is available bandwidth on the path that connects Sender and Receiver.*

Application: *ReserveDynApp*.

Success Condition: If there is not sufficient bandwidth, the controller denies the request and proceeds to install flow rules for a non-priority type of flow. Contrariwise, if there is sufficient bandwidth, the controller accepts the request and installs flow rules for a priority type of flow.

Implementation

As specified in the *success condition* statement, this experiment must be implemented for 2 scenarios:

- The first being the scenario where there is a on-going 8 Mb/s NP flow from Sender 1 to Receiver 1 (Figure 5.14). Sender 2 wishes to communicate (with priority) with Receiver 2 using a rate of 6 Mb/s and sends a request to the controller, which accepts it. Since both these flows share a sufficient port of 10 Mb/s (the bottleneck), when Sender 2 initiates the connection, there is not available bandwidth on the path (10 Mb/s - 8 Mb/s < 6 Mb/s) and so the controller installs flow rules for a non-priority flow.
- The second being the scenario where there is a on-going 2 Mb/s NP flow from Sender 1 to Receiver 1 (Figure 5.15). Sender 2 wishes to priority communicate with Receiver 2 using a rate of 5 Mb/s and sends a request to the controller, which accepts it. Since both these flows use a joint port of 10 Mb/s (the bottleneck), when Sender 2 initiates the connection, there is available bandwidth on the path (10 Mb/s - 2 Mb/s > 5 Mb/s) and so the controller installs flow rules for a priority flow.

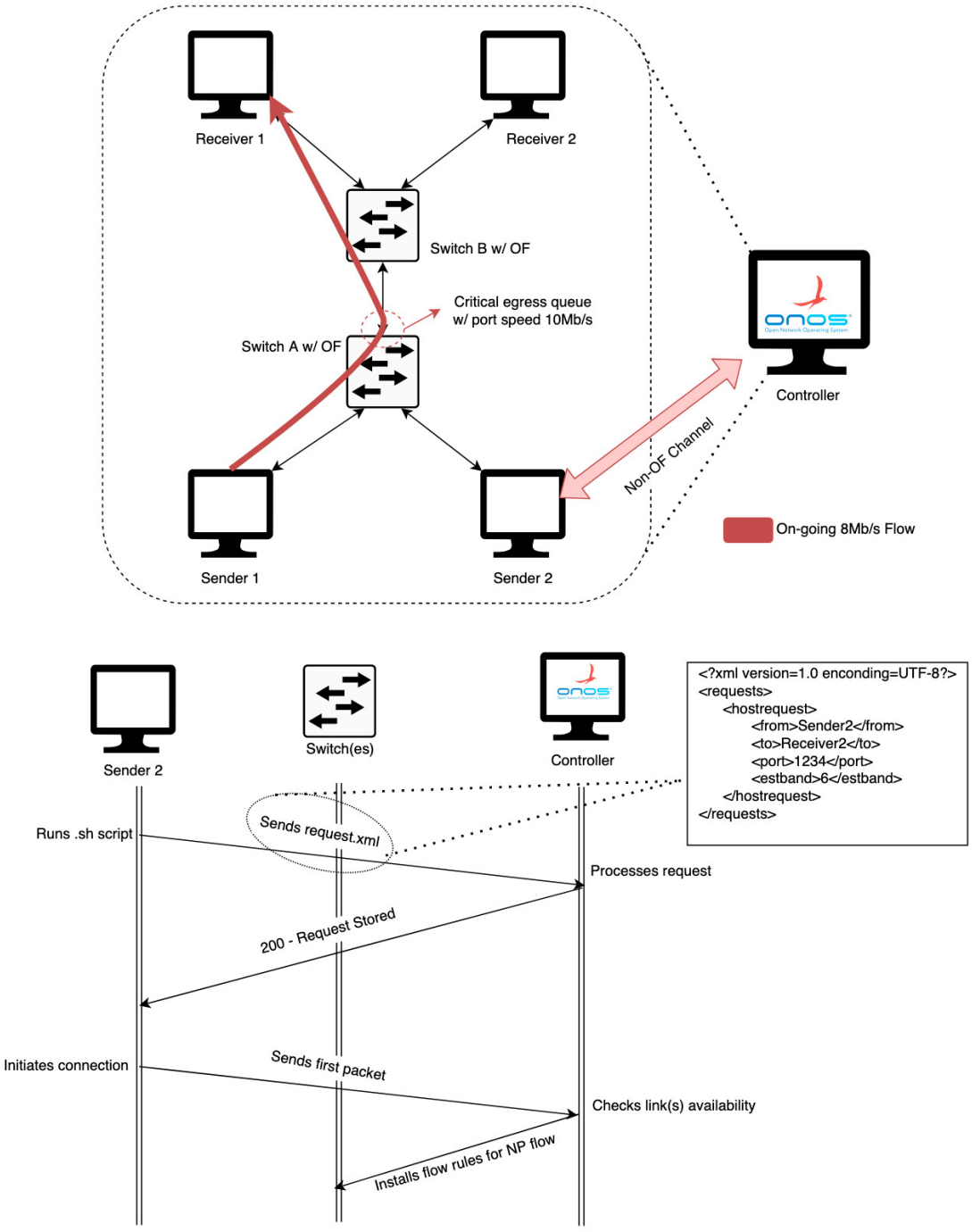

Figure 5.14: Experiment 4 - Logical Testbed and Workflow - Single Path - NP Flow Installation

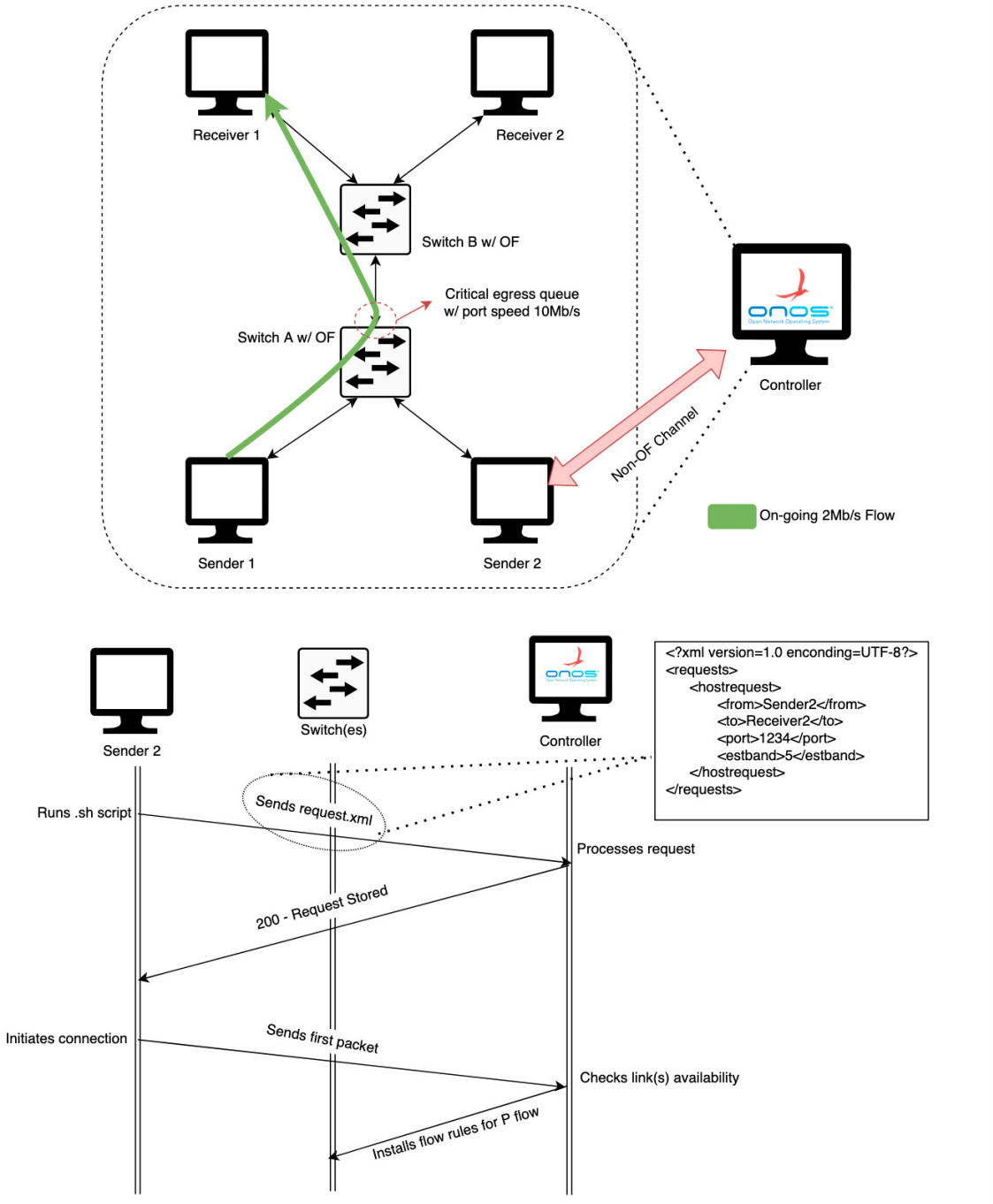

Figure 5.15: Experiment 4 - Logical Testbed and Workflow - Single Path - P Flow Installation

Results and Observations

For the inspection of this experiment results, as it was the case of experiment of SubSection 5.4.2, the ONOS capability to provide to the network administrator the real-time load at each link by using the topology environment of the ONOS GUI is used. Also, all flows are iPerf streams, being it priority or not. The mapping between IP address and experiment 4 names is as follows:

- Sender 1: 172.16.10.27
- Receiver 1: 172.16.10.3
- Sender 2: 172.16.10.4
- Receiver 2: 172.16.10.2

First scenario - NP Flows installation

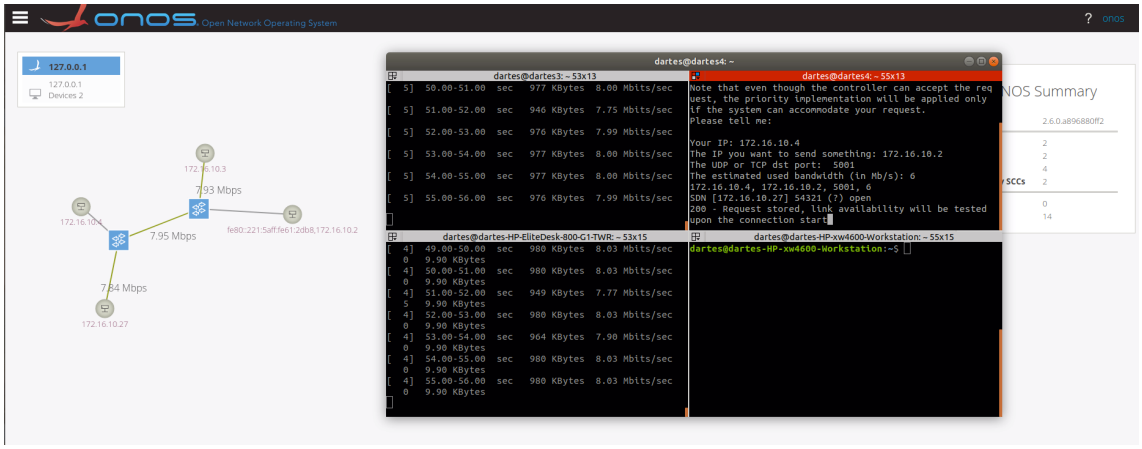

Figure 5.16: Experiment 4 - On-going 8 Mb/s NP flow

As depicted in Figure 5.16, from Sender 1 to Receiver 1 there is an approximated NP 8 Mb/s flow (this is confirmed by the left up and down terminal window). Also, it is possible to see at the right top terminal window that Sender 2 (*dartes4*) sent a request for a priority communication with Receiver 2 with an estimated bandwidth of 6 Mb/s.

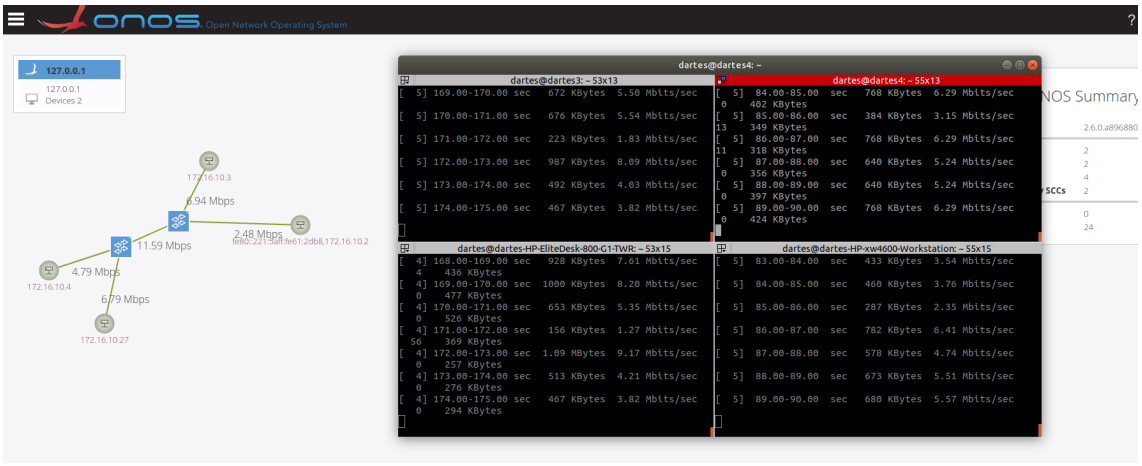

Figure 5.17: Experiment 4 - Controller Installs NP Flow Rules

As demonstrated in Figure 5.17, when Sender 2 is communicating with Receiver 2, as there is no available bandwidth in the path, flow rules for NP flows are installed. Thus, it does not have bandwidth guarantees, and the end-hosts are forced to divide the available bandwidth.

First scenario - P Flows installation

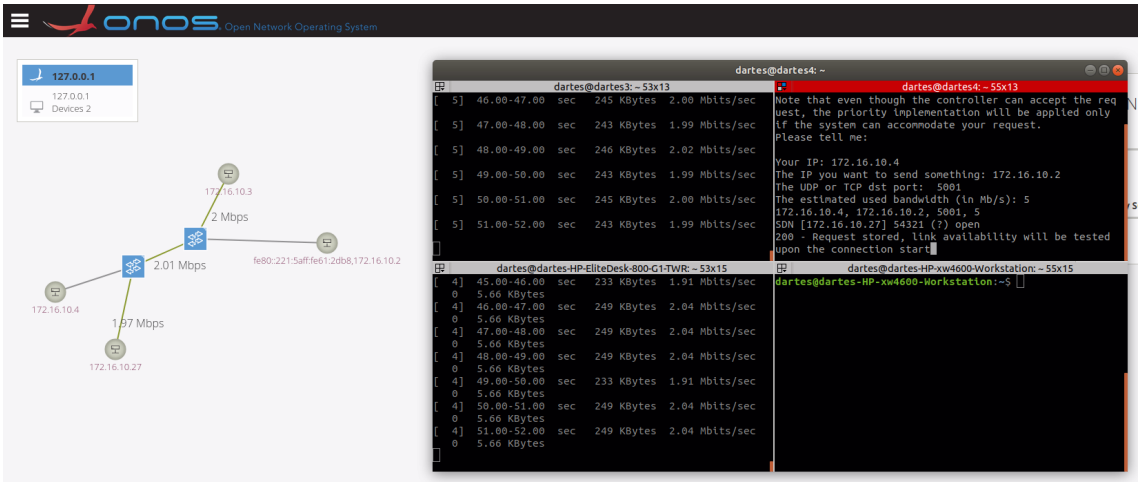

Figure 5.18: Experiment 4 - On-going 2 Mb/s NP flow

As depicted in Figure 5.18, from Sender 1 to Receiver 1, there is an approximately 2 Mb/s NP flow (this is confirmed by the left top and down terminal window). Also, it is possible to see at the right up terminal window that Sender 2 (*dartes4*) sent a request for a priority communication with Receiver 2 with an estimated bandwidth of 5 Mb/s.

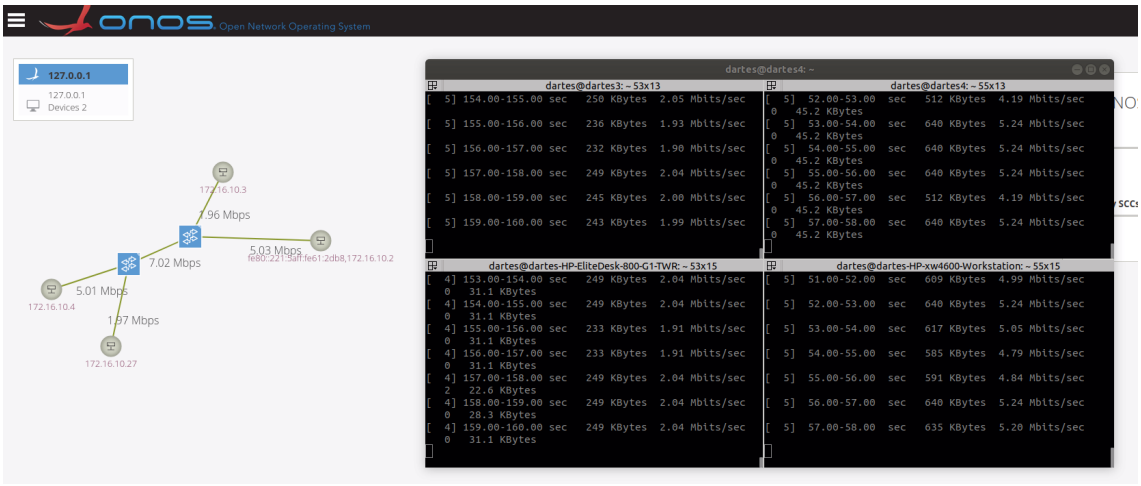

Figure 5.19: Experiment 4 - Controller Installs P Flow Rules

As demonstrated in Figure 5.19, when Sender 2 is communicating with Receiver 2, as there is available bandwidth in the path, flow rules for P flows are installed and the communication has guaranteed bandwidth. This is confirmed by checking the flow Table for any switch involved were it is possible to see flow rules for priority flows - last line of Figure 5.20.

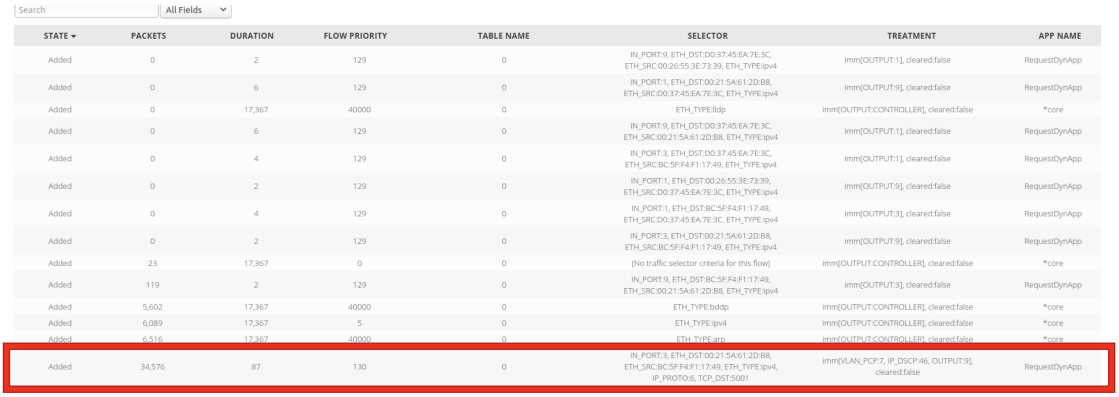

Figure 5.20: Experiment 4 - Installed P Flow on the Switch Flow Table

5.5.2 Experiment 5: Path Reservation in Presence of Multiple Paths

Statement

> Validation - *Validate application capacity to check if, upon bandwidth request and in a multi-path environment, there is available bandwidth in any path that connects Sender and Receiver.*

Application: *ReserveDynApp*.

Success Condition: If there is not sufficient bandwidth on the shortest path (hop-count), the controller checks if the alternative paths have available bandwidth and so on - then, the controller

installs priority flow rules for the first path that is feasible. Contrariwise, if there is sufficient bandwidth on the shortest path , the controller installs flow rules for the priority flow using the shortest path.

Implementation

For the deployment of a multi-path scenario it is necessary to have, at least, 3 switches. For the experiment, 3 switches are connected in a triangular topology. In a non-SDN environment, all 3 switches must be running the Spanning Tree Protocol (STP) to avoid infinite loops, which would block specific ports. However, as in the SDN-enabled solution - each path taken by a packet is pre-defined, the STP is disabled in all switches (and it also allows to have multipaths for this specific architecture).

As specified in the *success condition* statement, this experiment must be implemented for 2 scenarios:

- There is a on-going 8 Mb/s NP flow from Sender 1 to Receiver 1 (Figure 5.21) using the shortest path (path A) between those end-hosts, as per defined by the application algorithm. Sender 2 wishes to reserve bandwidth to communicate with Receiver 2 using a rate of 8 Mb/s and sends a request to the controller, which accepts it. Regarding the shortest path, both these flows use a common port of 10 Mb/s (the bottleneck) and so, that path is marked as not feasible as there is not available bandwidth. With that information, the controller proceeds to check the feasibility of the second shortest path (path B) - as there is no on-going flow using that path, there is available bandwidth. The controller then installs priority flow rules using path B for connectivity purposes.

- There is a on-going 2 Mb/s NP flow from Sender 1 to Receiver 1 (Figure 5.22) using the shortest path (path A) between those end-hosts, as per defined by the application algorithm. Sender 2 wishes to communicate with priority with Receiver 2 using a rate of 2 Mb/s and sends a request to the controller, which accepts it. Regarding the shortest path, both these flows use a joint port of 10 Mb/s (the bottleneck), nevertheless, as the estimated bandwidth on the request is less than the bandwidth available (10 Mb/s - 2 Mb/s > 2 Mb/s), the controller marks the shortest path as being feasible (and does not make any further checkings) and proceeds to install flow rules using that path for connectivity purposes.

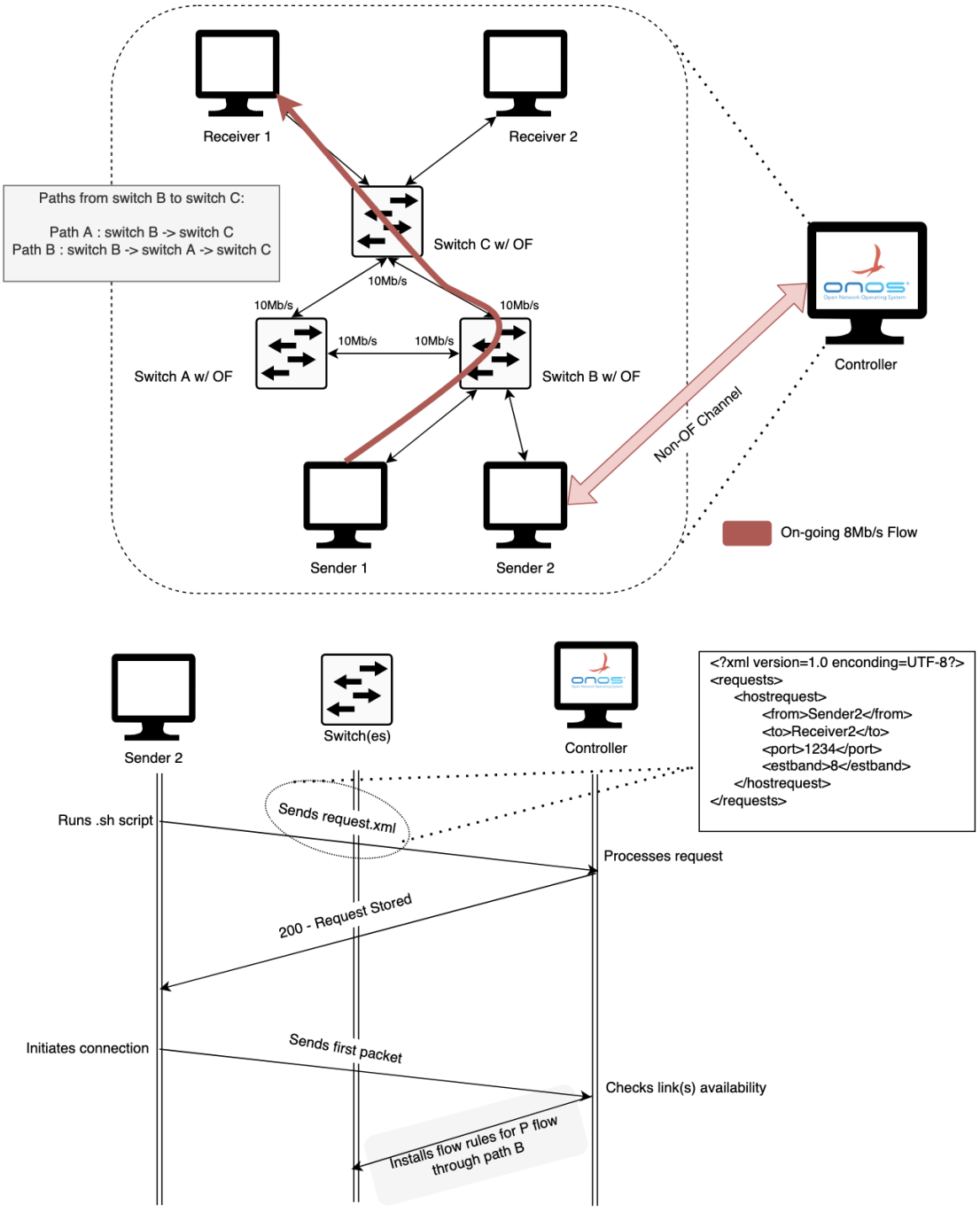

Figure 5.21: Experiment 5 - Logical Testbed - Multi Path - Using 2nd Shortest Path (Path B)

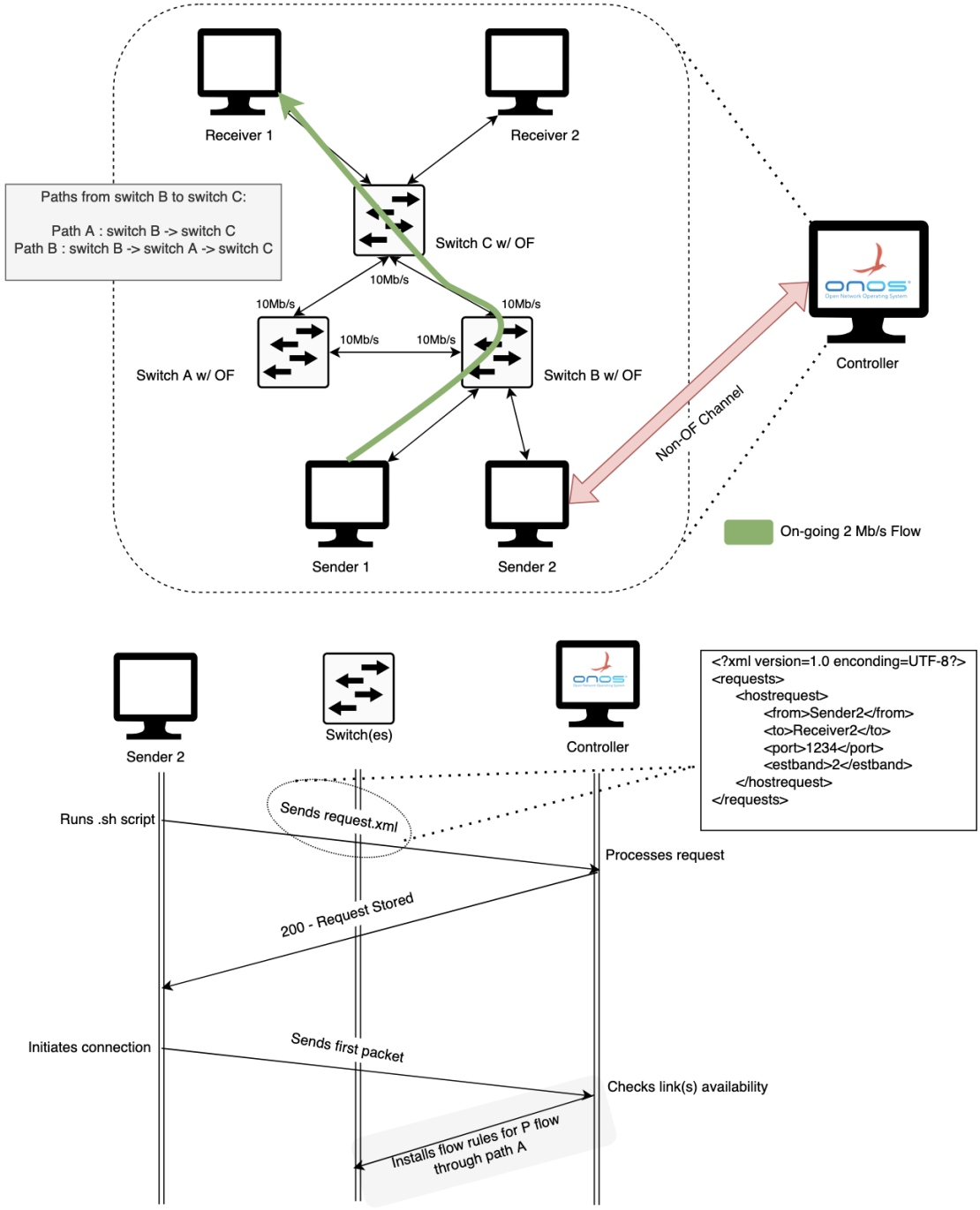

Figure 5.22: Experiment 5 - Logical Testbed - Multi Path - Using Shortest Path (Path A)

Results and Observations

Following the behaviour of experiment 5.4.2 and 5.5.1, this experiment results are also obtained using the ONOS built-in real-time load per link display. Again, all flows (priority or non priority) are iPerf streams. The mapping between IP address and experiment 5 names is as follows:

- Sender 1: 172.16.10.27

- Receiver 1: 172.16.10.4
- Sender 2: 172.16.10.3
- Receiver 2: 172.16.10.2

First Scenario - Usage of second shortest path

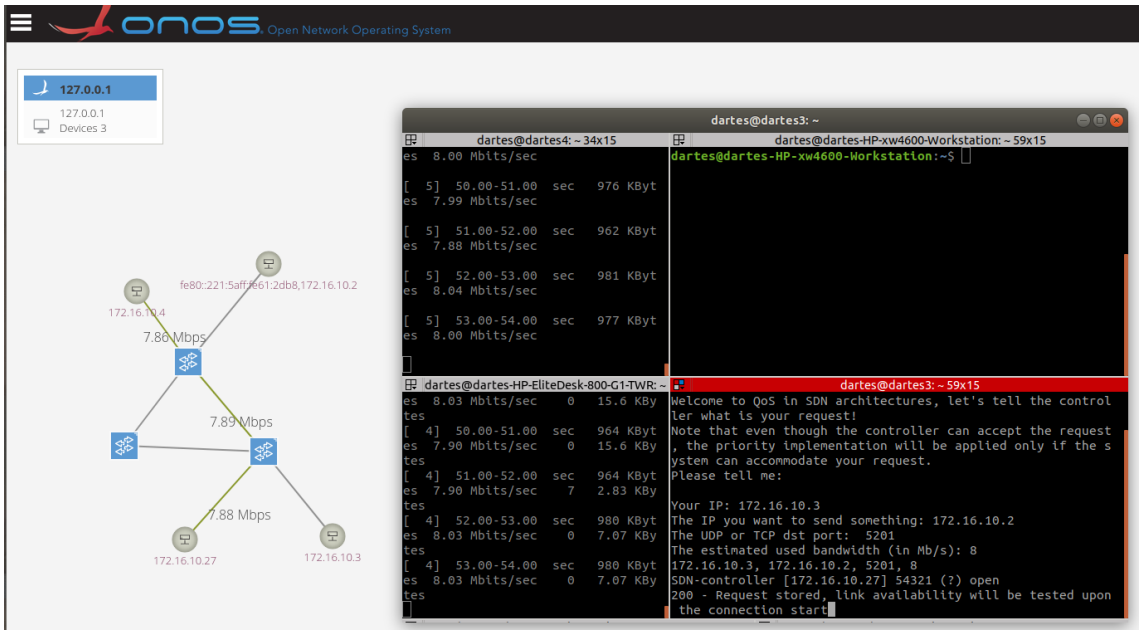

Figure 5.23: Experiment 5 - On-going 8 Mb/s NP flow

As depicted in Figure 5.23, from Sender 1 to Receiver 1 there is an approximated NP 8 Mb/s flow (this is confirmed by the left up and down terminal window) using the shortest path between Switch B and Switch C (path A). Also, it is possible to see at the right down terminal window that Sender 2 (*dartes3*) sent a request for a priority communication with Receiver 2 with an estimated bandwidth of 8 Mb/s.

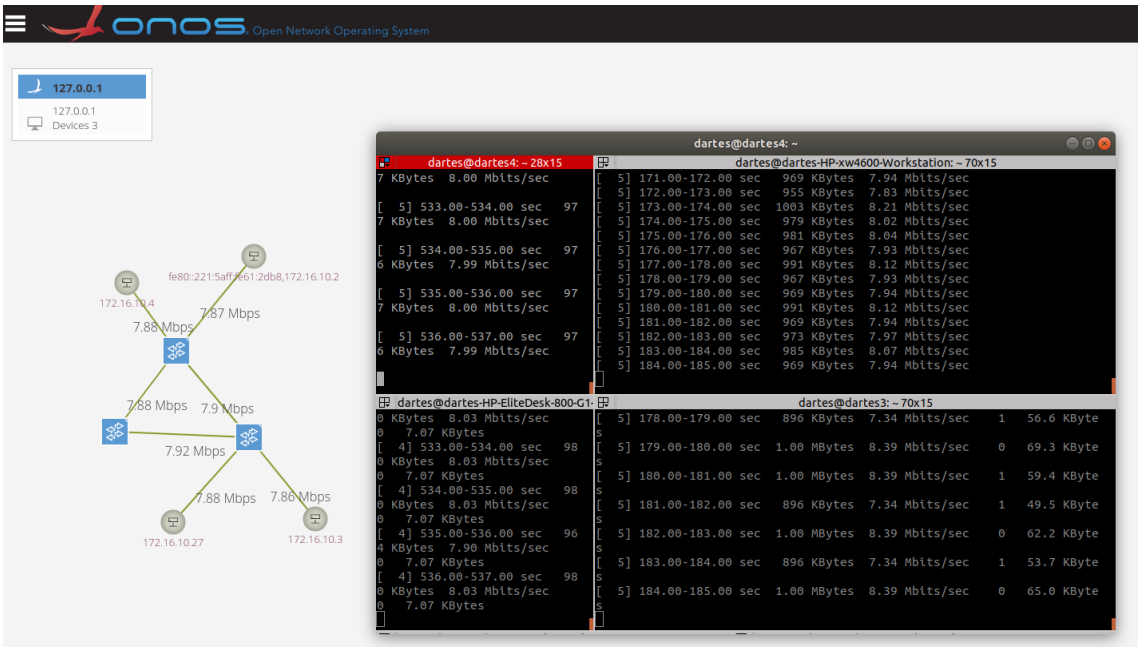

Figure 5.24: Experiment 5 - Controller Installs P Flow Rules for Path B

As exhibited in Figure 5.24, when Sender 2 is communicating with Receiver 2, as there is not available bandwidth on the shortest path (path A) from the non-priority flow occupation, the controllers proceeded to check the feasibility of the second shortest path (path B) which have available bandwidth. Thus, priority flow rules are installed using path B in order to accommodate Sender's 2 request.

Second Scenario - Usage of shortest path

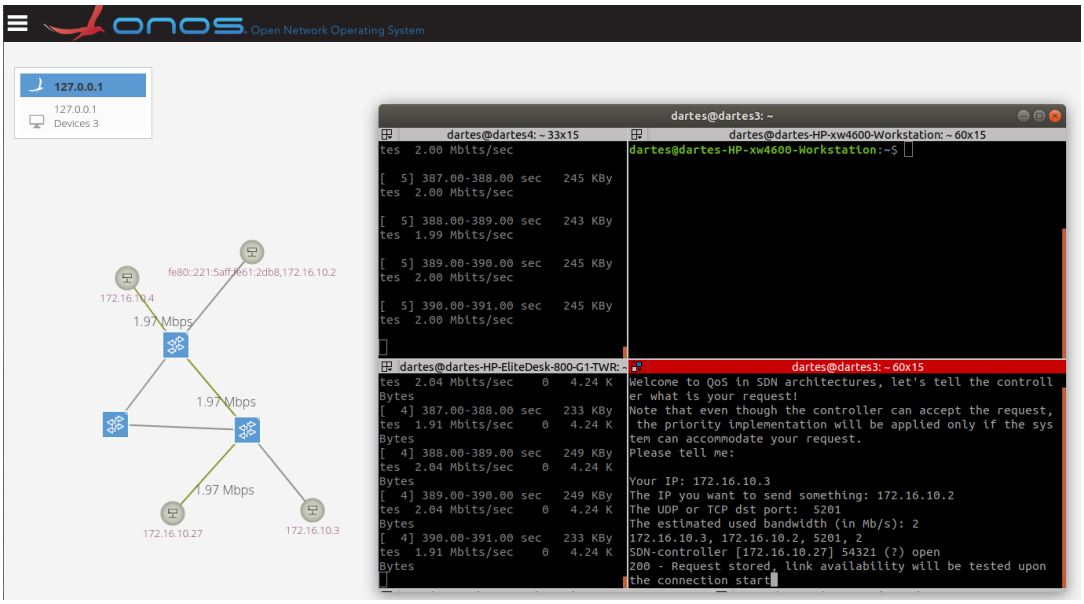

Figure 5.25: Experiment 5 - On-going 2 Mb/s NP flow

As depicted in Figure 5.25, from Sender 1 to Receiver 1 there is an approximated NP 2 Mb/s flow (this is confirmed by the left up and down terminal window) using the shortest path between Switch B and Switch C (path A). Also, it is possible to see at the right down terminal window that Sender 2 (*dartes3*) sent a request for a priority communication with Receiver 2 with an estimated bandwidth of 2 Mb/s.

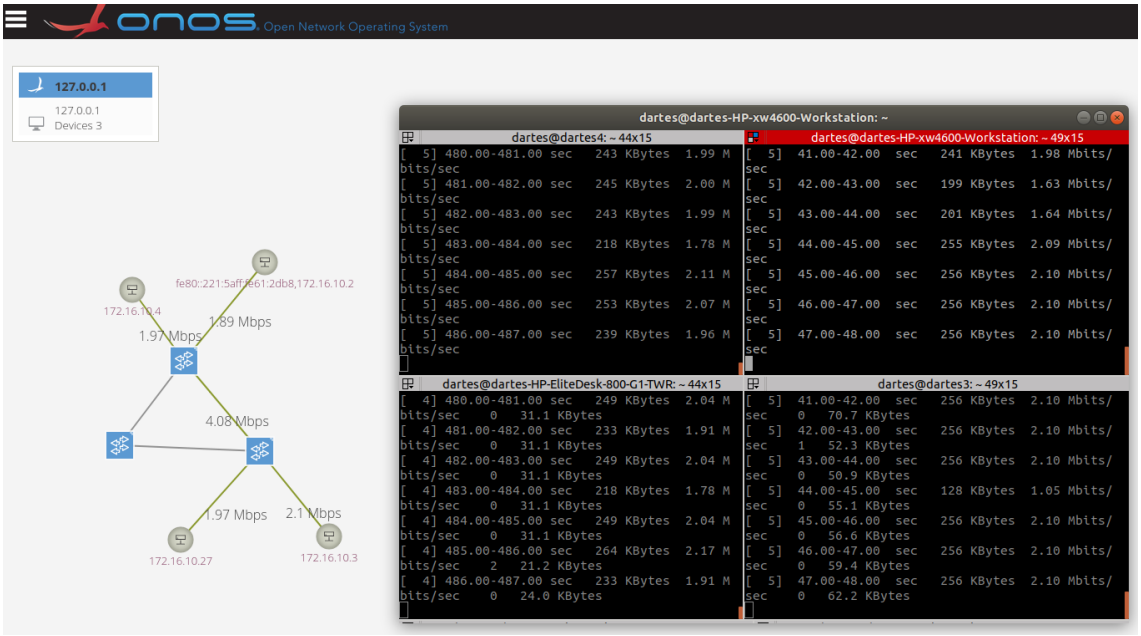

Figure 5.26: Experiment 5 - Controller Installs P Plow Rules for Path A

As exhibited in Figure 5.26, when Sender 2 is communicating with Receiver 2, as there is available bandwidth on the shortest path (path A), the controller proceeds to install priority flow rules using path A in order to accommodate Sender's 2 request. Note that the link that connects Switch B and Switch C is aggregating both flows' rate (~2 + ~2).

5.6 Impact on QoS Metrics

5.6.1 Experiment 6: Evaluate QoS Metrics with Priority Flows

Statement

> Evaluation - *Evaluate application ability to provide QoS to priority applications under congestion scenarios*

Application: *PriorityDynApp* and *ReserveDynApp*.

Implementation

For the implementation of the experiment, the priority flow is used as the use-case of the streaming of a video. For the effect, as mentioned at the beginning of this Section, the VLC Media Player

is used. Therefore, one host is acting as the Video Sender (server of the stream) and another is acting as the Video Receiver (client of the stream). Another two hosts are acting as the iPerf server and client, i.e. another two hosts are performing congestion, particularly in the port that connects switch A to switch B (which is set to a port speed of 10 Mb/s), as depicted in Figure 5.27.

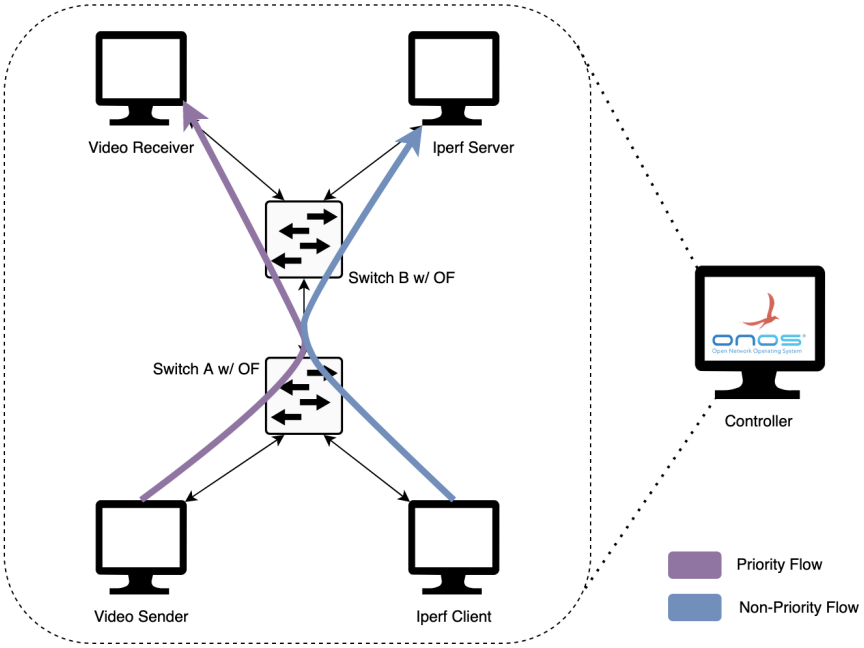

Figure 5.27: Experiment 6 - Logical Testbed

The video has a 177,1 MB size and a duration of 5 minutes and 14 seconds. Under a non-SDN-enabled network and with no congestion, the streaming of the video uses approximately 5 Mb/s. The non-priority flow is logically composed of 3 TCP streams with iPerf option *-b 0* enabled. The number 3 is specifically chosen to ensure that the priority flow, in normal conditions, does not have the necessary bandwidth available (4 streams sharing the same 10 Mb/s interface), i.e., there is congestion. Furthermore, the non-priority flow is always started before the priority one and the latter is always started after the non-priority flows have reached a stable state in terms of channel bandwidth division. The video codec chosen was the H.264 + MP3 (MP4) and the VLC Media Player is using a buffer with 1.5 seconds size (default value).

For simplicity, in this experiment, only the *PriorityDynApp* is used, as it automatically recognizes VLC streams as priority flows. Nevertheless, both the purpose and the results of this experiment are still valid for the *ReserveDynApp* as the on-test mechanisms (802.1p queuing and meters) are root tools of both applications.

Results and Observations

For completeness, this experiment is evaluated when the priority flow is using UDP and when is using TCP which is accomplished by using RTP/MPEG-TS (port 5004) and HTTP (port 8080), respectively. Moreover, it was decided to perform only one time each experiment, as it is not

expected that the experiment outcomes will vary considerably under different initial conditions, e.g. the start of the NP flows (of course, with the restriction that they have already reached the stable state aforementioned) or even the duration of the video.

Ultimately, this experiment can be divided into 8 scenarios as presented in Figure 5.28. First, it is divided into the UDP and TCP case scenario. Second, it is divided into a scenario with an without congestion. Third, it is divided into a scenario where the *xDynApp* is used and when there is no SDN-enabled solution. The addition of the no congestion scenario and the addition of the no SDN scenario allows the *xDynApp* to have a baseline scenario to be compared. Also, all sub-experiments corresponds to the last duration of the video to be streamed.

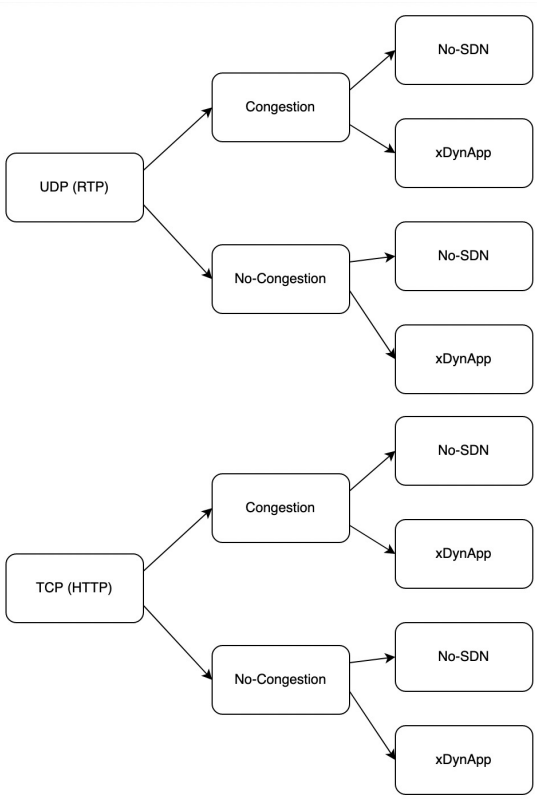

Figure 5.28: Experiment 6 - Logical Tree

As UDP QoS metrics are used the ones depicted in Table 5.3 and below described. Some of the metrics are automatically provided by the Wireshark RTP stream analysis (e.g. delta, PLR, jitter), while others are *manually* obtained (e.g. bandwidth measurement is obtained at the Conversations Wireshark Menu).

UDP QoS Metrics
Delta (ITA)
Jitter
PLR (%)
Sequence Errors
Bandwidth (Kb/s)

Table 5.3: UDP QoS Metrics

- **Delta or Inter Packet Arrival Time (ITA):**is the time difference between the arrival of one packet and the arrival of the previous packet. This metric works at the network layer and reflects the packet arrival at the capture interface where it is timestamped.

- **Jitter:** is the deviation from true periodicity of a presumably periodic signal, in relation to a reference clock signal, i.e. it is the delay variation of the packet delivery. Wireshark calculates jitter according to the RFC 3550 (RTP) [7] - using RTP timestamps.

- **Packet Loss Ratio (%):** the percentage of lost packets when compared to the expected total value of received packets. Wireshark subtracts, modulo 65535, the last RTP sequence number from the first and then compares that value to the number of packets received within the UDP stream.

- **Number of Sequence Errors:** as RTP marks every packet with a sequence number, Wireshark checks if it is out of order. It is different from PLR as a wrong sequence may just happen because of a out-of-order delivery (which tends to happen more in a congestion scenarios).

- **Bandwidth (Kb/s):** the number of bits sent per second from server to client.

Note that there is no actual *latency* (the time it takes for some data to get to its destination across the network) metric as it requires an extremely precise synchronization between Sender and Receiver which can not be met with such a small (really low transmissions delays) network as the one used in this experiment set.

As TCP QoS metrics the ones depicted in Table 5.4 and below described are used. Wireshark does not natively provided any HTTP stream analysis and so, all the metrics were *manually* obtained from Wireshark.

TCP QoS Metrics
Delta (ITA)
Bandwidth (Kb/s)
Out-of-order segment (%)
Previous segment not captured %)
Duplicated ACK (%)
Retransmission (%)
Fast Retransmission (%)

Table 5.4: TCP QoS Metrics

- **Delta or Inter Packet Arrival Time (ITA):** the same as UDP.
- **Bandwidth (Kb/s):** the same as UDP.

The following 5 metrics are in percentage regarding the total number of packets from server to client:

- **Out-of-order-segments (%):** a segment is marked as out-of-order if the TCP sequence is not in its natural order, meaning that when Wireshark analysis the sequence number it is not the expected next sequence number.
- **Previous segment not captured (%):** Wireshark marks a packet as not captured if it detected a gap in the TCP sequence numbers.
- **Duplicated ACK (%):** Duplicated acknowledgments are sent from the Receiver to the Sender if the Receiver does not receive a packet with the sequence number it was expecting. [1] Duplicate ACKS are part of the TCP congestion mechanism called Fast Retransmission, if Sender receives an *x* number of Duplicate ACKS, it proceeds to retransmit the asked packet.
- **Retransmission (%):** after sending a packet, the Sender starts a retransmission timer. If it does not receive a TCP ACK before the timer expires, the Sender will assume the packet has been lost and will retransmit it.
- **Fast Retransmission (%):** a packet that was retransmitted due to the TCP Fast Retransmission mechanism.

As TCP provides reliable transmission, there is no *de facto* metric for packet loss, however, the previous metrics can be seen as an *indicator* that some packets were lost due to congestion (being dropped in some queue). The first two does not necessarily mean that there is packet loss because packets from the same TCP flow, in a large network, can use different paths which can lead to different delays - however, in this experiment, they provide good metrics for packet loss as

[1] As defined in the TCP Protocol, e.g. if Receiver receives a packet with segment number = 1 it replies with an acknowledgment number = 2 and Sender, in natural conditions, reply with a packet with a sequence number = 2.

there is only one path from source to destination.

RTP without congestion

When using a RTP stream and in a no-congestion scenario, the following metrics are obtained for the priority flow:

	NoSDN	xDynApp
Max Delta (ms)	356,94	523
Mean Delta (ms)	2.05	2.09
Max Jitter (ms)	29.33	41.67
Mean Jitter (ms)	1.93	1.97
PLR (%)	1.33	1.16
#Seq Errors	72	55
Bandwidth (Kbits/s)	5341	5228

Table 5.5: Obtained Metrics for RTP Without Congestion

For the following BoxPlot graphs, it was chosen to exclude the outliers as they correspond to values numerically distant from the rest of the data. Also, maximum values are already presented throughout the tables.

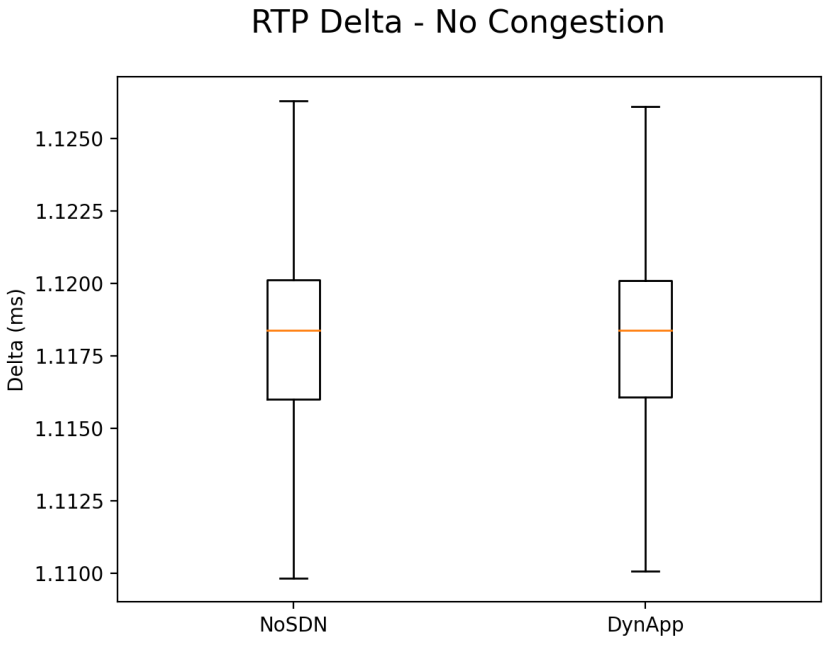

Figure 5.29: Experiment 6 - RTP Without Congestion - BoxPlot for Delta Values

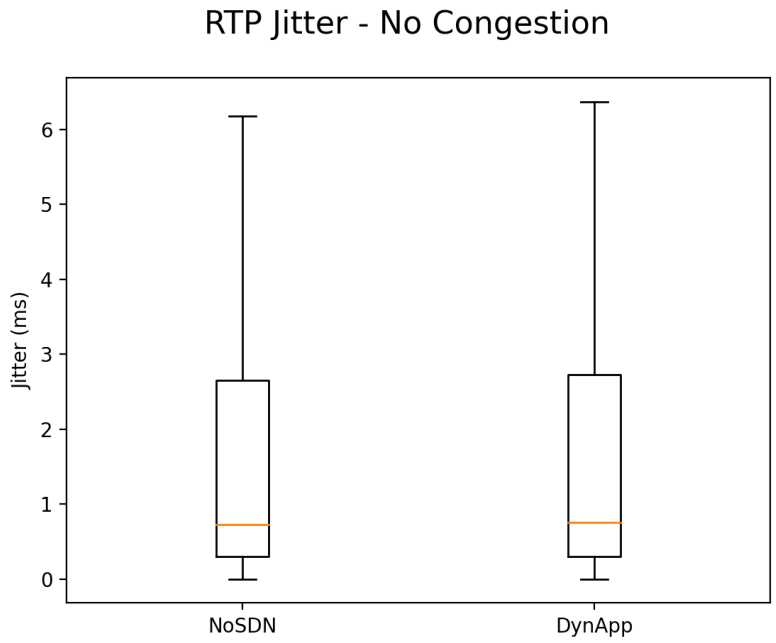

Figure 5.30: Experiment 6 - RTP Without Congestion - BoxPlot for Jitter Values

As one can observe, in Table 5.5, when there is no congestion, *xDynApp* shows approximately the same behaviour when comparing to a scenario where the video is streaming using a network with no SDN-solution enabled. Even though some higher maximum values (regarding delta and jitter) are obtained with the *xDynApp*, both mean values are very similar. This similarity is enforced in Figure 5.29 and 5.30 where the data distribution is very much alike in both cases. This also ultimately means that the overhead introduced by the *xDynApp* - the switch first forwards a packet to the controller, the controller then proceeds to install flow rules - note that this does not mean that there is a degradation of the flow's QoS.

RTP with congestion

When using a RTP stream and in a congestion scenario, the following metrics are obtained for the priority flow:

	NoSDN	**xDynApp**
Max Delta (ms)	3527.16	346.26
Mean Delta (ms)	2.40	2.10
Max Jitter (ms)	334.87	31.05
Mean Jitter (ms)	2.39	1.97
PLR (%)	17.25	1.16
#Seq Errors	644	114
Bandwidth (Kbits/s)	4537	5206

Table 5.6: Obtained Metrics for RTP With Congestion

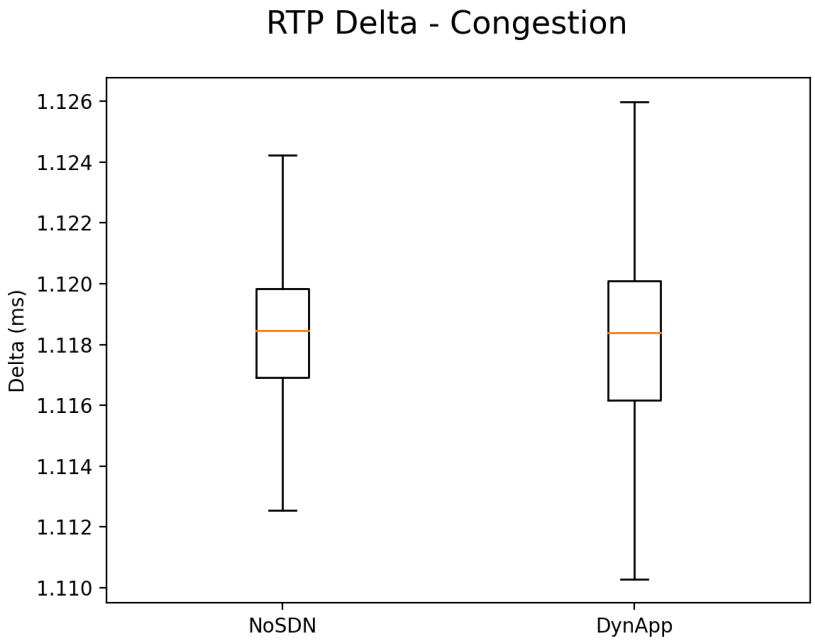

Figure 5.31: Experiment 6 - RTP With Congestion - BoxPlot for Delta Values

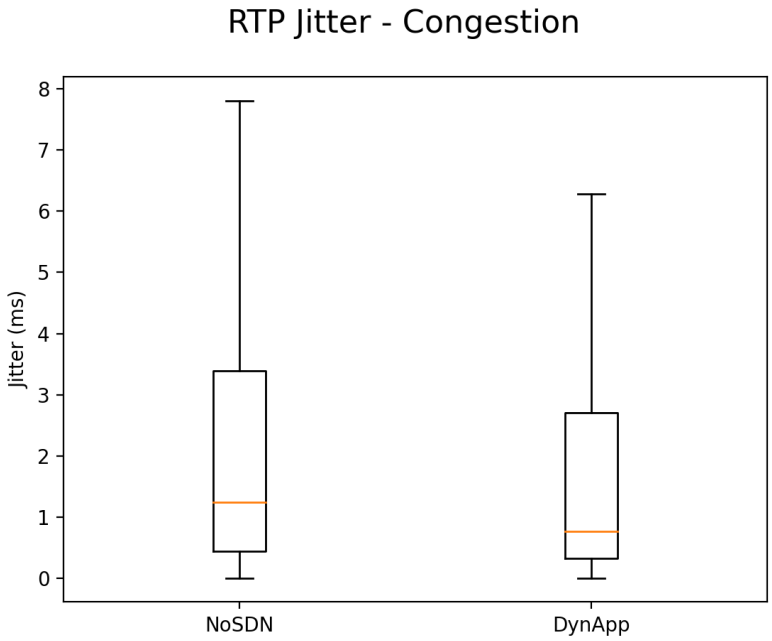

Figure 5.32: Experiment 6 - RTP With Congestion - BoxPlot for Jitter Values

The benefits of *xDynApp* are highlighted under congestion scenarios, as expected. All analysis metrics are better for the *xDynApp* case. In a non-SDN scenario, the PLR is 17.25% and with *xDynApp* is of 1.16% which represents a 14x better value. Moreover, both delta and jitter mean

values are lower. Note that the bandwidth obtained for the *xDynApp* under congestion and under no congestion is very similar.

The data distribution for the delta values (Figure 5.31) shows that, even though *xDynApp* experiences higher delta values (as visible in its higher top whisker), it also presents a smaller mean and median value. The distribution of the jitter values (Figure 5.32) shows that the difference between the maximum and the minimum value (excluding outliers) is smaller and with and inferior median for the *xDynApp* which is relevant as applications such as video streaming require stable and small jitter.

Realizing that for the no-SDN case, the delta maximum value presents an unusual value and by plotting the delta values with no outlier absorption we observe that it corresponds to an isolated value and so that sample should not be used as a reliable source of comparison.

HTTP without congestion

When using a HTTP stream and in a no congestion scenario, the following metrics are obtained for the priority flow:

	NoSDN	**xDynApp**
Max Delta (ms)	1168.61	1409.25
Mean Delta (ms)	2.57	2.58
Out-of-Order Segment (%)	0	0.007
Previous Segment Not Captured (%)	0.009	0.06
Duplicate ACK (%)	0.57	1.96
Retransmission (%)	0.04	0.06
Fast Retransmission (%)	0.003	0.06
Bandwidth (Kbits/s)	4692	4679

Table 5.7: Obtained Metrics for HTTP Without Congestion

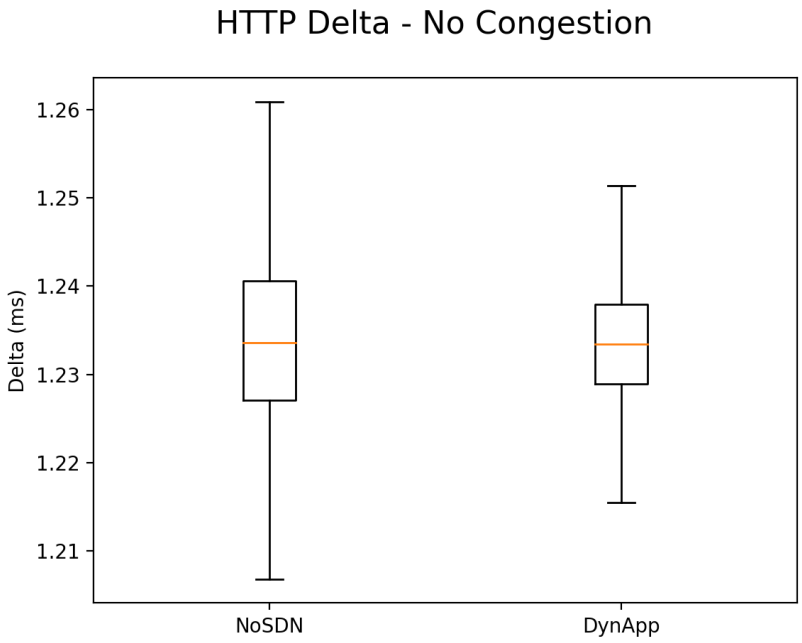

Figure 5.33: Experiment 6 - HTTP Without Congestion - BoxPlot for Delta Values

As already obtained in the RTP case, the HTTP scenario without congestion shows similar results between the no SDN use case and with *xDynApp*, especially in which regards the delta (with the mean and median value extremely close) and bandwidth values. The results show, once again, that the *xDynApp* does not introduce notable aggravations to the stream quality.

HTTP with congestion

When using a HTTP stream and in a congestion scenario, the following metrics are obtained for the priority flow:

	NoSDN	**xDynApp**
Max Delta (ms)	7927.46	1506.20
Mean Delta (ms)	4.84	2.65
Out-of-Order Segment (%)	0.14	0.001
Previous Segment Not Captured (%)	0.40	0.08
Duplicate ACK (%)	10.20	1.78
Retransmission (%)	0	0.05
Fast Retransmission (%)	0.31	0.05
Bandwidth (Kbits/s)	2497	4530

Table 5.8: Obtained Metrics for HTTP With Congestion

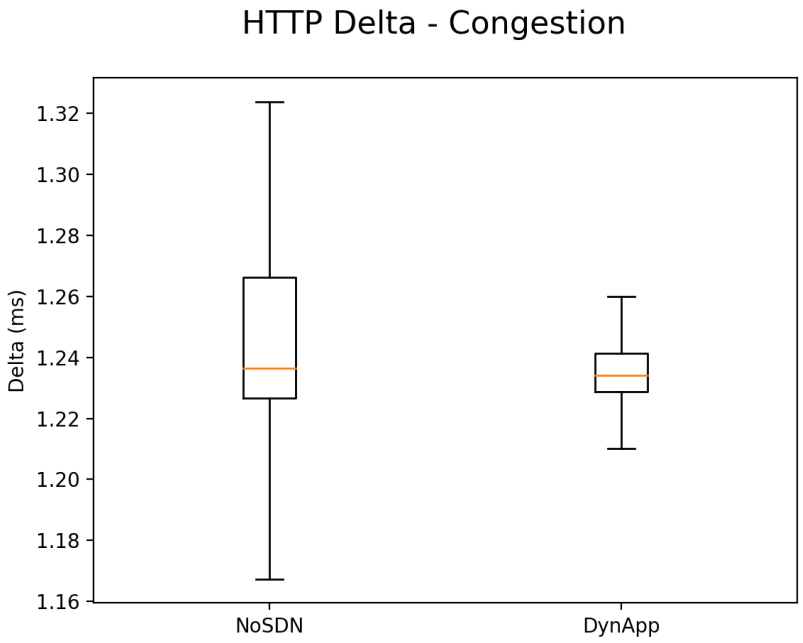

Figure 5.34: Experiment 6 - HTTP With Congestion - BoxPlot for Delta Values

The benefits of *xDynApp* managing the network's bandwidth resources are also notable for a TCP stream. As presented in Table 5.8, all metrics show better results for the *xDynApp* case except the percentage of Retransmission marked packets. However, this can be explained by the higher percentage of Fast Retransmissions packets - the time it takes for a packet to be retransmitted within the Fast Retransmission algorithm is often smaller than the time necessary to the Retransmission timeout elapse. The obtained bandwidth under the scope of *xDynApp* and with congestion is almost twice the value under the use case of no SDN-enabled solutions. Regarding the delta values, it is also notable in Table 5.8 and in Figure 5.34 that the values tend to be smaller and less dispersed.

5.7 Final Remarks

The main takeaways of this Chapter are:

- Firstly and as demonstrated in experiment 6 (5.6.1), it should be highlighted that the *xDynApp* does not introduce a notable worsening of a stream's QoS under no congestion when compared to a non-SDN solution (traditional network). Although, that may be not the case for all architectural scenarios - in this experiment set, the physical distance from the switches to the machine hosting ONOS is very small, in fact, they are connected on the same switch. Larger distances can introduce higher propagation delays and the impact of the need to transmit OpenFlow messages before the actual start of the flow can be more determinative.

Nevertheless, this procedure, for each priority flows happens only once. For congestion scenarios, all benefits of *xDynApp* are accentuated, using the meter mechanism or the 802.1p queues.

- Secondly, all validation experiments were successful proving that *xDynApp* fulfils its design proposal.

Chapter 6

Conclusion

This dissertation aimed to understand how the Software-Defined Networking paradigm could help to grant Quality of Service guarantees to applications running on the SDN managed network. As stated in Chapter 2, SDN, being the concept or paradigm of a centralized network management, does not offer *per se* QoS natively; it just offers a platform on top of which QoS mechanisms can be built and offered. The proposed solution (explained in detail in Chapter 3) englobes two SDN applications (the *xDynApp* applications) that dynamically manage the network's bandwidth for granting QoS: first, the most robust and complex application, *ReserveDynApp*, allows end-users to *request* for a guaranteed bandwidth regarding a specific communication; the second, *PriorityDynApp*, a simpler solution, determines pre-defined classes of priority (higher probability of receiving QoS guarantees) - some applications (defined by its Transport Protocol and port) are declared to be priority and others non-priority. Both developed SDN applications use two root tools to achieve the desired management ability: the OpenFlow meter mechanism (a SDN solution) and the IEEE 802.1p queuing mechanism (a non-SDN solution). Moreover, because in *ReserveDynApp* there is a *request stage*, it is possible to deploy an *access control* mechanism which allows to maintain the existent flows quality.

The proposed solution is implemented using the ONOS controller and the SDN applications are deployed as ONOS applications (using the ONOS Java API). As for the Forwarding Elements (switches), Aruba Switches were used. As explained in Section 3.5 and as in any design and development process, some self established design options ended up being circumstantial - we observed that OF queues, not being mandatory, end up not being implemented in many OF-enabled equipment, which happened to be the case of the switches that had been assigned to this work. Notwithstanding, we believe that most of the complications found throughout the work that led to some design choices also ultimately led to more sturdy solutions. As for the case of the queues, it was found a solution that works for both cases: for OpenFlow-enabled switches, the solution works for switches that support or do not support the OpenFlow queues.

As already discussed in Chapter 5, the *xDynApp* proves to be a solid and working solution to give QoS guarantees to priority applications by dynamically managing the network's bandwidth. We believe that the thought set of experiments presented in the aforementioned Chapter constitutes

a complete and valid set upon which is possible to deduce conclusions. For the series of *validation* experiments, it is possible to conclude that all the proposed design functionalities are validated - e.g. the meter mechanism properly rate limits non-priority flows. The *evaluation* results also present important aspects to take into account:

- *xDynApp* does not worsen the QoS of a flow when comparing to a scenario when there is no SDN solution at all (a traditional network). However, as already mentioned in Chapter 5, that may not be the case for all architectural scenarios.
- *xDynApp* when comparing to a non SDN solution and with a scenario where there is traffic causing congestion, improves the QoS metrics for the priority flow.

After a profound review of the State of the Art, we concluded that even though throughout the years, several works in the same area have been developed, few used real testbeds (most of them are implemented and simulated using Mininet network emulator). Furthermore, from the set of 25 analyzed references, few implement a hybrid solution - i.e. a solution that leverages from the SDN potential as well as using legacy QoS mechanism.

This dissertation followed the ensuing workflow:

1. An investigation of the State of the Art in order to understand the already developed work in the SDN field.
2. The configuration of a SDN-enabled network. That includes the configurations of the Aruba switches, the installation and configuration of the ONOS SDN controller and the configuration of each host machine.
3. A comprehensive analysis of the ONOS Java API potentials.
4. A study of the problem and the proposal of a new approach to solve it - the *xDynApp* applications set. This includes finding limitations of the software and hardware tools and implement solutions to overcome those.
5. The design of a relevant set of experiments to assure that it would be possible to deduce valid conclusions regarding the performance of the *xDynApp*.
6. The deployment of the mentioned experiments and the analysis of the resulting information.

6.1 Future Work

The SDN paradigm proved to be powerful and even though our solution has reached a stable state we believe it is possible to improve it. Some of the potential future work that could be done includes:

- Implement a policing mechanism for the *ReserveDynApp*. Our solution highly depends that the application trust the estimated bandwidth requested by an end-user (this value is used in

the *admission control* stage). There is no entity (a possible *policer*) analysing the on-going traffic and making sure every communication complies to its requested bandwidth and not more. This *policer* could apply measures to misbehaviour end-users.

- When testing if there is available bandwidth on the path, we chose to test the port load (aggregation of P and NP flows) but another solution could be to analyse the on-going bandwidth of P flows and make further decisions based on that, i.e., this solution would only preserve the quality of the on-going P flows. This, of course, requires the possibility to obtain per-flow statistics.

- Regarding *RequestDynApp* and as already mentioned, it could be possible to extend the developed protocol for the request processing by giving more detailed feedback.

- Regarding *PriorityDynApp* and as already mentioned, it could be possible to allow the network manager to add, on-the-fly, classes of priority. This could be deployed by using the ONOS possibility to implement application specific ONOS CLI commands.

- One of the first thought goals for this dissertation was to use of Machine Learning (ML) under a SDN environment. Even though that goal was not accomplished, we believe that it would be interesting to apply ML to dynamically identify priority traffic.

- Another goal not accomplished was the integration of a solution for wireless scenarios. Our solution is intrinsically designed for low mobility scenarios but a SDN-solution could be used to improve, for example, the handover process in mobility ones.

- As for the experiments set, it could be added a Section where our solution is compared with another approaches - e.g. compare our mechanism to check available bandwidth on a path with the AVB used mechanism.

Appendix A

Prerequisites for *xDynApp* Operation

This Appendix is divided in two Sections: Section A.1 describes the necessary ONOS configurations so that *xDynApp* can properly work; Section A.2 explains the necessary Aruba switches configuration to allow the 802.1p queuing mechanism and also the OpenFlow communication with the ONOS controller.

A.1 ONOS setup

To allow the correct and expectable functioning of the SDN-network and thus, the correct functioning of the *xDynApp* applications, there is a set of ONOS built-in applications that need to be installed and activated:

- **Default Drivers**: a suite of default drivers.
- **Host Location Provider**: provides host discovery and location (within the network) to the ONOS core by eavesdropping on the Address Resolution Protocol (ARP) and the Neighbor Discovery Protocol (NDP) packets.
- **LLDP Link Provider**: provides link discovery to the ONOS core bye eavesdropping on the Link Layer Discovery Protocol (LLDP) control packets.
- **OpenFlow Agent**: OpenFlow agent application for virtualization subsystem.
- **OpenFlow Base Provider**: provides the base suite of device, flow and packet providers that rely on the OpenFlow protocol to interact with network devices.
- **OpenFlow Provider Suite**: suite of the OpenFlow base providers bundled together with ARP/NDP host location provider and LLDP link provider.
- **Optical Network Model**: ONOS optical information model.
- **Proxy ARP/NPD**: the ProxyARP application responds to ARP or NDP requests on behalf of a target host if the target host is known to ONOS.

- **Tunnel Subsystem**: an extension for the tunnel subsystem.
- **Virtual Network Subsystem**: an extension for the virtual network subsystem.
- (**ONOS GUI2**: not exactly mandatory but allows the network administrator to have a GUI presentation of ONOS which is desirable.)

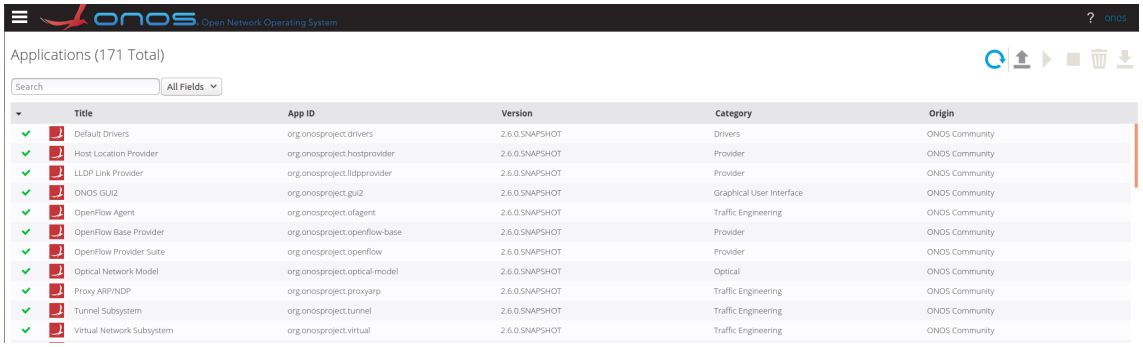

Figure A.1: ONOS Setup Built-in Apps

Some of these applications dynamically try to install flow rules on the network switches to assure that they can properly function. These *default* flow rules are presented in table A.1.

Selector/Rule	Treatment/Action
ETH_TYPE:arp	OUTPUT: CONTROLLER, wipeDeferred
ETH_TYPE:lldp	OUTPUT: CONTROLLER, wipeDeferred
ETH_TYPE:bddp	OUTPUT: CONTROLLER, wipeDeferred

Table A.1: *Default* Flow Rules

However, the Aruba switches do not support the wipeDeferred (instructions to clear the deferred instructions set) action and, upon flow rules attempt of installation, they all stay in PENDING_STATE with the switches returning the error OFBadInstruction with the code= UNSUP_INST. The correct installation, without the wipeDeferred action, can be made using the ONOS REST API and the Swagger UI. These flow rules must be presented and installed in the pre-defined, valid JSON format. This manual action is only necessary to be done once everytime those applications are installed into the ONOS core.

Additionally, *xDynApp* requires two more *default* flow rules to be installed, converging in the five necessary flow rules presented in table A.2.

A.2 Switches Setup

As for the involved switches (Aruba 2930M 24G PoE+ 1-slot Switch - model JL320A), some requirements are also set. As mentioned in Section 3.1, the *xDynApp* algorithm presupposes that

Selector/Rule	Treatment/Action
ETH_TYPE:arp	OUTPUT: CONTROLLER
ETH_TYPE:lldp	OUTPUT: CONTROLLER
ETH_TYPE:bddp	OUTPUT: CONTROLLER
ETH_TYPE: ipv4	OUTPUT: CONTROLLER
(empty criteria)	OUTPUT: CONTROLLER

Table A.2: *xDynApp* Mandatory *Default* Flow Rules

the network elements are ready to handle the 802.1p queuing mechanism. This can be achieved by the following Aruba CLI commands:

- `qos queue-config 2-queues`

  The default number queues per output port is 4. This command defines that there are only 2, Q1 and Q2 and that the bandwidth distribution is 20% of GMB to Q1 and 80% to Q2 (default configuration). The used scheduling mechanism is Weighted-Round-Robin (WRR) [5].

- For every interface/port:

  ```
  qos trust dot1p
  ```

  This command defines that the switch should trust the 802.1p priority information in the incoming and outcoming traffic, i.e. there is no remarking.

Also, as the *xDynApp* is a SDN application, it requires that the used SDN controller and the FEs can communicate via some southbound protocol, which, in the case, as it as being mentioned, is the OpenFlow protocol. The Aruba switch configurations for OpenFlow are depicted in Figure A.2 and explained below.

```
openflow
   controller-id 1 ip 192.168.106.129 controller-interface oobm
   instance "1"
      member vlan 10
      controller-id 1
      version 1.3 only
      connection-interruption-mode fail-standalone
      table-num policy-table 0
      enable
      exit
   enable
   exit
```

Figure A.2: Aruba Switch OpenFlow Configuration

`controller-id 1 ip 192.168.106.129 controller-interface oobm`

The above command defines that there is a controller with the id *1*, that is using the 192.168.106.129 IP address and that can be found in the *OOBM* (out-of-band management) interface. This way, OpenFlow and actual data packets are completely separated, as advised on the Aruba documentation.

```
instance "1"
```

At this point, it is possible to define an OpenFlow instance to be in *aggregation mode* or *virtualization mode*. When an OpenFlow instance is defined to be in *aggregation mode*, that instance includes all VLANs except the management VLAN and the OpenFlow controller VLANs. The *virtualization mode*, which is the mode selected, allows non-OpenFlow VLANs and VLANs that belong to OpenFlow instances to be configured on the switch. It also enables the independence of different OpenFlow instances within the OpenFlow context. Both modes were tried but when using the *aggregation mode*, ONOS discovered links between switches were only unidirectional and when using the *virtualization mode*, the links were bidirectional and so it was decided to use the latter.

```
member vlan 10
```

As the instance is defined to be in *virtualization mode*, members need to be added - in the case, the switch is prepared to communicate OpenFlow information about hosts that are part of VLAN 10.

```
controller-id 1
```

The instance is using the controller with *1* as id.

```
version 1.3 only
```

The Aruba Switch 2930M series only support OpenFlow version 1.0 and 1.3 and so it was decided to use the newer as it has more features (ONOS controller also supports OFv1.3).

```
connection-interruption-mode fail-standalone
```

The above command defines that if the OpenFlow instance loses connection with the controller, legacy switching and routing functions handle packets of new flows. Existing flows of the OpenFlow instance are removed.

```
table-num policy-table 0
```

This OpenFlow instance besides having only one table, it has the number 0. Furthermore, both the instance and the OpenFlow context need to be enabled.

References

[1] Aruba switches 2930m series. https://www.arubanetworks.com/products/switches/access/2930m-series/, Online, accessed 04 June 2021.

[2] Bosch - projeto e inovação - safecities. https://www.bosch.pt/a-nossa-empresa/bosch-em-portugal/projetos-de-inovacao/safe-cities.html, Online, accessed 21 June 2021.

[3] HARTES project. http://hartes.av.it.pt/, Online, accessed 23 January 2021.

[4] HPE arubaos-switch openflow v1.3administrator guide for 16.03. https://support.hpe.com/hpesc/public/docDisplay?docLocale=en_US&docId=emr_na-c05365339, Online, accessed 03 June 2021.

[5] HPE documentation - qos queues configuration. https://techhub.hpe.com/eginfolib/networking/docs/switches/RA/15-18/5998-8155_ra-2620_atmg/content/ch04s08.html, Online, accessed 08 June 2021.

[6] IEEE 802.1: 802.1BA - Audio Video Bridging (AVB) Systems. https://www.ieee802.org/1/pages/802.1ba.html, Online, accessed 09 January 2021.

[7] IETF - RFC 3350 - RTP: A transport protocol for real-time applications. https://datatracker.ietf.org/doc/html/rfc3550, Online, accessed 18 June 2021.

[8] iPerf - the ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/, Online, accessed 01 June 2021.

[9] METRICS project. http://www.av.it.pt/metrics/, Online, accessed 23 January 2021.

[10] Mininet: An Instant Virtual Network on your Laptop (or other PC) - Mininet. http://mininet.org/, Online, accessed 01 February 2021.

[11] ns-3 | a discrete-event network simulator for internet systems. https://www.nsnam.org/, Online, accessed 01 February 2021.

[12] ONOS developer guide. https://wiki.onosproject.org/display/ONOS/Developer+Guide, Online, accessed 07 June 2021.

[13] ONOS java api 2.5.1. http://api.onosproject.org/2.5.1/apidocs/, Online, accessed 31 May 2021.

[14] ONOS wiki page. https://wiki.onosproject.org/display/ONOS/ONOS, Online, accessed 31 May 2021.

[15] Open vSwitch. https://www.openvswitch.org/, Online, accessed 08 January 2021.

[16] Openflow version 1.3 specification. https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf, Online, accessed 03 June 2021.

[17] OSGI working group. https://www.osgi.org/about/, Online, accessed 06 June 2021.

[18] VLC media player. https://www.videolan.org/, Online, accessed 31 May 2021.

[19] Wireshark - the I/O graphs window. https://www.wireshark.org/docs/wsug_html_chunked/ChStatIOGraphs.html, Online, accessed 22 June 2021.

[20] Wireshark web page. https://www.wireshark.org/, Online, accessed 09 June 2021.

[21] Sherif Abdelwahab, Bechir Hamdaoui, Mohsen Guizani, and Taieb Znati. Network function virtualization in 5G. *IEEE Communications Magazine*, 54(4):84–91, 2016.

[22] Wi-Fi Alliance. Wi-Fi Alliance ® Technical Committee Technical Specification (with WMM-power save and WMM-admission control). pages 1–25, 2012.

[23] Namwon An, Taejin Ha, Kyung-joon Park, and Hyuk Lim. Dynamic Priority-Adjustment for Real-Time Flows in Software-Defined Networks. *2016 17th International Telecommunications Network Strategy and Planning Symposium (Networks)*, pages 144–149, 2016.

[24] Pankaj Berde, William Snow, Guru Parulkar, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, and Pavlin Radoslavov. Onos: Towards an Open, Distributed SDN OS. *HotSDN 2014 - Proceedings of the ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking*, pages 1–6, 2014.

[25] Ahmed Binsahaq, Tarek R. Sheltami, and Khaled Salah. A Survey on Autonomic Provisioning and Management of QoS in SDN Networks. volume 7, pages 73384–73435. Institute of Electrical and Electronics Engineers Inc., 2019.

[26] S. Blake, D. Black, M. Carlson, and E. Davies. IETF - RFC 2475: An Architecture for Differentiated Services. 1998.

[27] Zheng Cai, Alan L Cox, and T S Eugene Ng. Maestro: A System for Scalable OpenFlow Control. Technical report.

[28] Chen Chen, Shalinee Kishore, and Lawrence V. Snyder. An innovative RTP-based residential power scheduling scheme for smart grids. In *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, pages 5956–5959, 2011.

[29] Seyhan Civanlar, Murat Parlakisik, A. Murat Tekalp, Burak Gorkemli, Bulent Kaytaz, and Evren Onem. A QoS-enabled openflow environment for scalable video streaming. *2010 IEEE Globecom Workshops, GC'10*, pages 351–356, 2010.

[30] D. Clark, R. Braden, and S. Shenker. IETF - RFC 1633: Integrated Services in the Internet Architecture: an Overview. 1994.

[31] Guo Cin Deng and Kuochen Wang. An Application-aware QoS Routing Algorithm for SDN-based IoT Networking. *Proceedings - IEEE Symposium on Computers and Communications*, 2018-June:186–191, 2018.

[32] POX Manual Current Documentation. Installing POX. https://noxrepo.github.io/pox-doc/html/, Online, accessed 09 January 2021.

[33] Hilmi E. Egilmez, S. Tahsin Dane, K. Tolga Bagci, and A. Murat Tekalp. OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks. *2012 Conference Handbook - Asia-Pacific Signal and Information Processing Association Annual Summit and Conference, APSIPA ASC 2012*, 2012.

[34] David Erickson. The Beacon OpenFlow controller. In *HotSDN 2013 - Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.

[35] Joakim Flathagen, Terje M. Mjelde, and Ole I. Bentstuen. A combined Network Access Control and QoS scheme for Software Defined Networks. In *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2018*. Institute of Electrical and Electronics Engineers Inc., nov 2018.

[36] Project FloodLight. FloodLight Controller. https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview, Online, accessed 24 January 2021.

[37] Open Network Foundation. SDN Definition. https://opennetworking.org/sdn-definition/, Online, accessed 03 January 2021.

[38] Open Network Foundation. OpenFlow Switch Specification Version 1.3.0 (Wire Protocol 0x04). *Current*, 0:1–36, 2012.

[39] R. Gokilabharathi and P. Deenalakshmi. Efficient Load Balancing to Enhance the Quality of Service (QOS) in Software Defined Networking (SDN). In *Proceedings of the 2nd International Conference on Trends in Electronics and Informatics, ICOEI 2018*, pages 1046–1051. Institute of Electrical and Electronics Engineers Inc., nov 2018.

[40] Jochen W. Guck and Wolfgang Kellerer. Achieving end-to-end real-time Quality of Service with Software Defined Networking. *2014 IEEE 3rd International Conference on Cloud Networking, CloudNet 2014*, pages 70–76, 2014.

[41] Jochen W. Guck, Amaury Van Bemten, and Wolfgang Kellerer. DetServ: Network models for real-time QoS provisioning in SDN-based industrial environments. *IEEE Transactions on Network and Service Management*, 14(4):1003–1017, dec 2017.

[42] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, jul 2008.

[43] Timo Hackel, Philipp Meyer, Franz Korf, and Thomas C. Schmidt. Software-defined networks supporting time-sensitive in-vehicular communication. In *IEEE Vehicular Technology Conference*, volume 2019-April. Institute of Electrical and Electronics Engineers Inc., apr 2019.

[44] Md Alam Hossain, Mohammad Nowsin Amin Sheikh, Shawon S.M. Rahman, Sujan Biswas, and Md Ariful Islam Arman. Enhancing and measuring the performance in Software Defined Networking. *International Journal of Computer Networks and Communications*, 10(5):27–40, sep 2018.

[45] Fei Hu, Qi Hao, and Ke Bao. A survey on software-defined network and OpenFlow: From concept to implementation. In *IEEE Communications Surveys and Tutorials*, volume 16, pages 2181–2206. Institute of Electrical and Electronics Engineers Inc., apr 2014.

[46] V. Jacobson, R. Frederick, S. Casner, and H. Schulzrinne. RTP: A Transport Protocol for Real-Time Applications. https://tools.ietf.org/html/rfc3550, Online, accessed 01 February 2021.

[47] Shashwat Jain, Manish Khandelwal, Ashutosh Katkar, and Joseph Nygate. Applying big data technologies to manage QoS in an SDN. *2016 12th International Conference on Network and Service Management, CNSM 2016 and Workshops, 3rd International Workshop on Management of SDN and NFV, ManSDN/NFV 2016, and International Workshop on Green ICT and Smart Networking, GISN 2016*, pages 302–306, 2017.

[48] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. *SIGCOMM 2013 - Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 3–14, 2013.

[49] Yosr Jarraya, Taous Madi, and Mourad Debbabi. A survey and a layered taxonomy of software-defined networking. In *IEEE Communications Surveys and Tutorials*, volume 16, pages 1955–1980. Institute of Electrical and Electronics Engineers Inc., apr 2014.

[50] Murat Karakus and Arjan Durresi. A scalable inter-AS QoS routing architecture in software defined network (SDN). *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*, 2015-April:148–154, 2015.

[51] Murat Karakus and Arjan Durresi. Quality of Service (QoS) in Software Defined Networking (SDN): A survey. In *Journal of Network and Computer Applications*, volume 80, pages 200–218. Academic Press, feb 2017.

[52] Hermann Kopetz. Real-time systems: Design principles for distributed embedded applications. In *Kluwer Academic Publishers*, volume 34, page 142, 1997.

[53] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010*, 2019.

[54] Diego Kreutz, Fernando M. V. Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-Defined Networking: A Comprehensive Survey. volume 103, pages 14–76. Institute of Electrical and Electronics Engineers Inc., jun 2014.

[55] Rakesh Kumar, Monowar Hasan, Smruti Padhy, Konstantin Evchenko, Lavanya Piramanayagam, Sibin Mohan, and Rakesh B. Bobba. End-to-End Network Delay Guarantees for Real-Time Systems Using SDN. In *Proceedings - Real-Time Systems Symposium*, volume 2018-Janua, pages 231–242. Institute of Electrical and Electronics Engineers Inc., jan 2018.

[56] Andrzej Kwiecień and Karol Opielka. Management of industrial networks based on the FCAPS guidelines. In *Communications in Computer and Information Science*, volume 291 CCIS, pages 280–288, 2012.

[57] Adrian Lara, Anisha Kolasani, and Byrav Ramamurthy. Network innovation using open flow: A survey. *IEEE Communications Surveys and Tutorials*, 16(1):493–512, mar 2014.

[58] Bruce S. Davie Larry L. Peterson. Computer Networks: a system approach. pages 354–368. Elsevier, 2012.

[59] Shengru Li, Daoyun Hu, Wenjian Fang, Shoujiang Ma, Cen Chen, Huibai Huang, and Zuqing Zhu. Protocol Oblivious Forwarding (POF): Software-Defined Networking with Enhanced Programmability. *IEEE Network*, 31(2):58–66, 2017.

[60] Luis Moutinho, Paulo Pedreiras, and Luis Almeida. A Real-Time Software Defined Networking Framework for Next-Generation Industrial Networks. *IEEE Access*, 7:164468–164479, 2019.

[61] Bruno Astuto A. Nunes, Marc Mendonca, Xuan Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys and Tutorials*, 16(3):1617–1634, 2014.

[62] Jannis Ohms, Olaf Gebauer, Nadiia Kotelnikova, Diederich Wermser, and Eduard Siemens. Providing of QoS-Enabled Flows in SDN Exemplified by VoIP Traffic. Technical Report March, 2017.

[63] J. Ordonez-Lucena, P. Ameigeiras, D. Lopez, J. J. Ramos-Munoz, J. Lorca, and J. Folgueira. Network slicing for 5G with SDN/NFV: Concepts, architectures and challenges. *arXiv*, (May):80–87, 2017.

[64] Harold Owens and Arjan Durresi. Video over software-defined networking (VSDN). *Proceedings - 16th International Conference on Network-Based Information Systems, NBiS 2013*, pages 44–51, 2013.

[65] Harold Owens, Arjan Durresi, and Raj Jain. Reliable video over software-defined networking (RVSDN). *2014 IEEE Global Communications Conference, GLOBECOM 2014*, pages 1974–1979, 2014.

[66] Paulo Pedreiras and Luis Almeida. The flexible time-triggered (FTT) paradigm: An approach to QoS management in distributed real-time systems. *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2003*, 2003.

[67] PreSonus. An Introduction to AVB Networking. https://www.presonus.com/learn/technical-articles/an-introduction-to-avb-networking, Online, accessed 09 January 2021.

[68] Zhijing Qin, Grit Denker, Carlo Giannelli, Paolo Bellavista, and Nalini Venkatasubramanian. A software defined networking architecture for the internet-of-things. *IEEE/IFIP NOMS 2014 - IEEE/IFIP Network Operations and Management Symposium: Management in a Software Defined World*, 2014.

[69] Daniel Raumer, Lukas Schwaighofer, and Georg Carle. MonSamp: A distributed SDN application for QoS monitoring. *2014 Federated Conference on Computer Science and Information Systems, FedCSIS 2014*, 2:961–968, 2014.

[70] Paulo A. Ribeiro, Liudas Duoba, Rui Prior, Sergio Crisostomo, and Luis Almeida. Real-Time Wireless Data Plane for Real-Time-Enabled SDN. In *IEEE International Workshop on Factory Communication Systems - Proceedings, WFCS*, volume 2019-May. Institute of Electrical and Electronics Engineers Inc., may 2019.

[71] Omar Said, Yasser Albagory, Mostafa Nofal, and Fahad Al Raddady. IoT-RTP and IoT-RTCP: Adaptive protocols for multimedia transmission over internet of things environments. *IEEE Access*, 5:16757–16773, jul 2017.

[72] Thomas Schierl, Karsten Grüneberg, and Thomas Wiegand. Scalable video coding over RTP and MPEG-2 transport stream in broadcast and IPTV channels - Seamless content delivery in the future mobile internet. *IEEE Wireless Communications*, 16(5):64–71, oct 2009.

[73] Sachin Sharma, Dimitri Staessens, Didier Colle, David Palma, Joao Goncalves, Ricardo Figueiredo, Donal Morris, Mario Pickavet, and Piet Demeester. Implementing quality of service for the software defined networking enabled future internet. In *Proceedings - 2014 3rd European Workshop on Software-Defined Networks, EWSDN 2014*, pages 49–54. Institute of Electrical and Electronics Engineers Inc., dec 2014.

[74] Jung Wan Shin, Hwi Young Lee, Won Jin Lee, and Min Young Chung. Access control with ONOS controller in the SDN based WLAN testbed. *International Conference on Ubiquitous and Future Networks, ICUFN*, 2016-Augus:656–660, 2016.

[75] Luis Silva, Pedro Goncalves, Ricardo Marau, Paulo Pedreiras, and Luis Almeida. Extending OpenFlow with flexible time-triggered real-time communication services. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, pages 1–8, 2017.

[76] The Linux Fundation. OpenDaylight. https://www.opendaylight.org/, Online, accessed 09 January 2021.

[77] Slavica Tomovic, Neeli Prasad, and Igor Radusinovic. SDN control framework for QoS provisioning. *2014 22nd Telecommunications Forum, TELFOR 2014 - Proceedings of Papers*, pages 111–114, 2014.

[78] Tina Tsou, Xingang Shi, Jing Huang, Zhiliang Wang, and Xia Yin. Analysis of Comparisons between OpenFlow and ForCES. 2012.

[79] Arun Viswanathan, Ross Callon, and Eric C. Rosen. IETF - RFC 3031 - Multiprotocol Label Switching Architecture. 2001.

[80] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, and Haiyong Xie. A Survey on Software-Defined Networking. In *IEEE Communications Surveys and Tutorials*, volume 17, pages 27–51. Institute of Electrical and Electronics Engineers Inc., jan 2015.

[81] Jinyao Yan, Hailong Zhang, Qianjun Shuai, Bo Liu, and Xiao Guo. HiQoS: An SDN-based multipath QoS solution. *China Communications*, 12(5):123–133, 2015.

[82] Tsung Feng Yu, Kuochen Wang, and Yi Huai Hsu. Adaptive routing for video streaming with QoS support over SDN networks. *International Conference on Information Networking*, 2015-Janua:318–323, 2015.