

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Software-Defined Network for Internet of Things

Francisco Luís Araújo Fraga



Master in Electrical and Computers Engineering

Internal Supervisor: Prof. João Manuel Couto das Neves

External Supervisor: Dr. Waldir Aranha Moreira Júnior

July 17, 2019

Resumo

É claro que o constante crescimento do tráfego global da Internet deve ser compensado com soluções mais rápidas, simples, escaláveis e flexíveis. Tendo isto em conta, é importante perceber o quão facilmente as Redes Definidas por Software (*Software-Defined Networking (SDN)*) podem simplificar este problema quando comparadas com as redes convencionais. As SDNs representam uma tecnologia em crescimento que foi desenhada para melhorar a eficiência, flexibilidade e acelerar a introdução de novos serviços com suporte de escalabilidade, representando a solução perfeita para realizar gestão de recursos e orquestrar serviços de uma forma inteligente.

Tendo em conta a enorme quantidade de dispositivos conetados, o processo de evolução da tecnologia e as suas características, é inevitável não reparar no existente crescimento considerável do nível de tráfego de Internet. Assim sendo, a segurança e direção dos fluxos de dados da rede devem ser considerados como tópicos críticos de debate. O mundo de pesquisa deve então garantir um bom comportamento das desvantagens do mundo da Internet das Coisas (*Internet of Things (IoT)*) (movimento, segurança, programabilidade e gestão da informação), sendo nesse contexto em que as SDN se podem revelar a ser uma boa solução. Devido a uma separação por parte das SDN dos planos de dados e controlo, uma combinação de ambas as tecnologias tem potencial para ser considerada uma revolução no mundo IoT e conseqüentemente deve ser feita uma análise das várias hipóteses existentes. Porque o IoT está constantemente a evoluir e permite um comportamento mais sólido, cada vez mais casos de uso estão a ser considerados. Tendo em conta que tecnologias em crescimento e as Next-Generation Networks (NGN) dão cada vez mais importância ao factor eficiência, é necessário que o mundo da pesquisa encontre soluções que tenham em atenção os recursos que cada situação específica exige.

Esta dissertação de mestrado foca-se em resolver os problemas de gestão que as fatias de rede dos vários casos de uso podem causar, sendo estas criadas com uma ferramenta resultante da combinação de SDN com IoT. Porque cada fatia considerada vai lidar com um caso de uso em específico e características distintas, diferentes recursos vão ter que ser especificados de modo a ser possível obter o nível de Qualidade de Serviço (*Quality of Service (QoS)*) pretendido. Com vista a atingir este fim, diversos cenários e modelos de QoS vão ser considerados, implementados e testados, considerando várias métricas e como elas se relacionam com a solução proposta.

Abstract

It is clear that the growth on global Internet traffic must be somehow compensated with faster, simpler, scalable and more flexible solutions. With that in mind, it is important to understand how easily SDN can simplify this problem when compared to traditional networks. SDN is an in growing and potential topic designed to increase efficiency, flexibility and accelerate the introduction of new services with the support of scalability, representing the perfect way of performing resource management and intelligent service orchestration.

Having in mind the huge amount of connected devices and its technology and characteristics evolution, it is inevitable to notice that a considerable continuous growth in the Internet generated traffic is happening. In light of this, the security threatening and direction of data network flow should be considered as crucial points to debate. The researching world should be aware that it is mandatory to guarantee a good behaviour on IoT downside characteristics (movement, safety, programmability and information management) and it is there when SDN turns out to be profitable. Because of the SDN separation between data and control planes, these technologies coupling can prove to be a turning point to the IoT world itself and the consideration and analysis of potential solutions should be taken into account. Due to the constant IoT evolving, more use cases are being considered as it provides a more solid behaviour when dealing with them. Since emerging technologies and NGN care about efficiency, they demand for the researchers to find solutions regarding the resources each individual situation requires.

This Master Thesis project focuses on solving multiple use cases slices management problems, created on a SDN coupling with IoT framework. Because each considered slice deals with a specific use case and each one of them has different characteristics, distinct amounts of resources will be needed in order to provide the solution required QoS level. In order to do so, distinct scenarios and QoS models will be studied, implemented and tested in the created SDN environment, aiming to analyse several metrics how they relate with the proposed solution.

Acknowledgements

I would like to acknowledge Prof. João Neves for accepting to supervise me and thank him for all his teaching, suggestions and patience throughout this Master Thesis process. I would also like to thank Dr. Waldir Moreira for giving me the chance to develop this work at Fraunhofer AICOS and helping me to fully understand the scope and requirements of it. It is important to express gratitude to Dr. Antonio Júnior for providing me insights regarding SDN and pointing me to the right direction in times of uncertainty. I want to thank Faculdade de Engenharia da Universidade do Porto for providing me the opportunity to take advantage of its facilities and all its community knowledge.

I find important to deeply thank all my family for all the unconditional support. Special remarks go to my father for teaching me to be who I am and providing me with buttress when hard times strike, my mother for encouraging me to follow my dreams and kick starting my love for Mathematics and Engineering, my brother for being a life companion and providing me with his everyday counselling availability.

I want to express my gratitude to my girlfriend, Mafalda, for representing a foundation and keeping up with me in the majority of my educational process.

I want to thank Gonçalo for being a partner since my very beginning at FEUP and also sharing with me all late nights studying.

Concluding, I want to thank my friends for their support and providing me with moments of happiness and joy. To all of you, I express my sincere gratitude.

Francisco Luís Araújo Fraga

*“Sempre chega a hora em que descobrimos que sabíamos
muito mais do que antes julgávamos.”*

José Saramago

Contents

1	Introduction	1
1.1	Thesis Scope	1
1.2	Motivation	2
1.3	Main Goal and Specific Objectives	2
1.4	Document Structure	3
2	Literature Review	5
2.1	Software-Defined Networking	5
2.1.1	Control Plane	6
2.1.2	Data Plane	8
2.1.3	Network Virtualisation	9
2.2	Internet of Things	10
2.2.1	Coupling with SDN	10
2.2.2	Message Queuing Telemetry Transport	12
2.3	SOFTWAY4IoT	14
2.3.1	Description	14
2.3.2	Architecture	16
2.3.3	Considered Wireless Technologies	17
2.4	Use Cases	19
2.4.1	Digital Farming	19
2.4.2	Industry 4.0	20
2.5	Performance Architecture	20
2.5.1	Performance Mechanisms	21
2.5.2	Service-Level Agreements	23
2.5.3	Policies	23
2.5.4	Quality of Service in SDN	23
3	Proposed Solution	25
3.1	Solution coupling with SOFTWAY4IoT	25
3.1.1	Implementation machine specifications	26
3.1.2	VM-Application	27
3.1.3	VM-Manager	28
3.1.4	Host	29
3.2	Data Plane networking	30
3.2.1	A LoRa physical interface	30
3.2.2	The new SDWI module	31
3.2.3	Resultant network architecture	34
3.3	QoS Manager	36

3.3.1	Module Placement	36
3.3.2	Module Functioning	37
3.3.3	QoS Scenarios	41
3.4	Related Work	43
4	Testing and Obtained Results	45
4.1	Testing preparation	45
4.1.1	Testing automation and methodology	46
4.1.2	Test map	47
4.2	Best Effort	47
4.2.1	Maximum achievable bandwidth	47
4.2.2	Multiple clients	48
4.2.3	Multiple Slices	52
4.3	Traffic Shaping and Traffic Dropping	53
4.3.1	Flows analysis and Protocols comparison	53
4.3.2	Two slices traffic policing	56
4.4	Per-Flow QoS testing	57
4.4.1	One slice IntServ	58
4.4.2	Two slices IntServ	59
4.5	DiffServ QoS Testing	63
4.5.1	Prioritisation	63
4.5.2	Two slices DSCP marking	65
4.5.3	Two slices DSCP matching with prioritisation	68
5	Conclusions and Future Work	73
5.1	Conclusions	73
5.2	Future Work	74
A	Source Code	77
A.1	Tests Container Dockerfile	77
A.2	Code to create 50 Iperf3 servers (Running on container)	78
A.3	Function that starts an Iperf3 client and handles the test result (Host)	80

List of Figures

2.1	Traditional network nodes in contrast with SDN.	6
2.2	Software-Defined Networking architecture.	7
2.3	Network Slicing visualised.	9
2.4	Relationship between Network Virtualisation, SDN, NFV and IoT.	11
2.5	Considered IoT network architecture.	12
2.6	MQTT protocol visualised.	13
2.7	SOFTWAY4IoT interaction with different devices	15
2.8	SOFTWAY4IoT dealing with multiple wireless technologies.	15
2.9	SOFTWAY4IoT components architecture.	16
3.1	SOFTWAY4IoT implementation and its relation with the SDN architecture. . . .	26
3.2	VM-Application System Architecture.	28
3.3	VM-Manager System Architecture.	29
3.4	Host System Architecture.	30
3.5	Dragino LoRa shield used as access point.	31
3.6	BLE client/server relationship.	32
3.7	Profiles, Services and Characteristics hierarchy.	33
3.8	LoRa network architecture and its relation with SOFTWAY4IoT.	34
3.9	Overall network architecture.	35
3.10	QoS Module placement.	37
3.11	Comparison between policing and traffic shaping.	38
3.12	Typical Open vSwitch packet flowchart.	40
3.13	HTB configuration example.	41
3.14	Per-Flow QoS congestion point.	42
3.15	Diffserv QoS DSCP marking and different congestion conditions.	42
4.1	Performed tests overview.	47
4.2	Maximum SDN4IoT achievable bit rate when utilising TCP and UDP protocols. .	48
4.3	Total bit rate of TCP and UDP protocols resultant from a growing client number.	49
4.4	Average clients bit rate of TCP and UDP protocols resultant from a growing client number.	49
4.5	CPU utilisation when gradually increasing clients in TCP and UDP protocols. . .	50
4.6	Clients total bandwidth on the Iperf3 non default TCP sized packets test.	51
4.7	Clients average bandwidth on the Iperf3 non default TCP sized packets test. . . .	51
4.8	CPU utilisation when gradually increasing clients in TCP non default sized packet.	52
4.9	Comparison of two Slices total bandwidth.	52
4.10	Clients average bandwidth on the two Slices test.	53
4.11	Traffic policing TCP and UDP flows comparison.	54

4.12	Traffic shaping TCP and UDP flows comparison.	55
4.13	Server perspective UDP flows on traffic policing and traffic shaping.	56
4.14	Two slices policing implementation total bandwidth.	56
4.15	Two slices policing implementation aggregated bandwidth.	57
4.16	One slice IntServ implementation queues total bandwidth.	59
4.17	Queues average bit rate on the one slice IntServ test.	59
4.18	Two slices IntServ - Slice 1 aggregated bandwidth.	60
4.19	Two slices IntServ - Slice 2 aggregated bandwidth.	60
4.20	Two slices IntServ - Total achieved bandwidth comparison.	61
4.21	Two slices IntServ - Slice 1 Queues bandwidth.	61
4.22	Two slices IntServ - Slice 2 Queues bandwidth.	62
4.23	Two slices IntServ - Average clients bandwidth per Queue.	62
4.24	Prioritisation test queues implementation.	64
4.25	Prioritisation test queues limit and total bandwidth.	64
4.26	Prioritisation test distribution of queues bandwidth.	65
4.27	Two slices DSCP marking - Slice 1 aggregated bandwidth.	66
4.28	Two slices DSCP marking - Slice 2 aggregated bandwidth.	66
4.29	Two slices DSCP marking - Total achieved bandwidth in each slice.	67
4.30	Two slices DSCP marking - Slice 1 Queues bandwidth.	67
4.31	Two slices DSCP marking - Slice 2 Queues bandwidth.	68
4.32	Two slices DSCP marking - Average clients bandwidth per Queue.	68
4.33	Two slices DSCP matching with prioritisation - Slice 1 queues implementation.	69
4.34	Two slices DSCP matching with prioritisation - Slice 2 queues implementation.	69
4.35	Two slices DSCP matching with prioritisation - Both slices total bandwidth.	70
4.36	Two slices DSCP matching with prioritisation - Slice 1 total bandwidth.	70
4.37	Two slices DSCP matching with prioritisation - Slice 2 total bandwidth.	71

List of Tables

2.1	OpenFlow main pros and cons.	8
2.2	A Comparison between DiffServ and IntServ.	22
3.1	Considered flow entry characteristics.	39
4.1	One Slice IntServ implementation details.	58
4.2	Two Slices IntServ implementation details.	60
4.3	Two Slices IntServ implementation details.	65

Acronyms and Abbreviations

ACK	Acknowledge Packet
API	Application Programming Interface
ATT	Attribute Protocol
BLE	Bluetooth Low Energy
CBQ	Class-based queuing
DSCP	Differentiated Services Code Point
FCMan	Fog Computing Manager
FIFO	First in first out
GATT	Generic Attribute Profile
GPS	Global Positioning System
HTB	Hierarchy Token Bucket
HTML5	Hypertext Markup Language 5
IoS	Internet of Systems
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
LoRa	Long Range
M2M	Machine-to-Machine
MAC	Media Access Control
MQTT	Message Queuing Telemetry Transport
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NFV	Network Functions Virtualisation
NGN	Next-Generation Networks

NOS	Network Operating System
NSMan	Network Security Manager
OSI	Open Systems Interconnection
OVS	Open vSwitch
Qdiscs	Queueing Disciplines
QoE	Quality of Experience
QoS	Quality of Service
RED	Random Early Detection
REST	Representational State Transfer
RMA	reliability, maintainability, and availability
RSVP	Resource Reservation Protocol
SDN	Software-Defined Networking
SDR	Software-Defined Radio
SDWI	Software-Defined Wireless Interfacing
SFQ	Stochastic Fairness Queueing
SLA	Service-Level Agreement
SMan	Slice Manager
SOFTWAY4IoT	SOFTware defined gateWAY and fog computing for Internet of Things
TBF	Token Bucket Filter
TC	Linux Traffic Control
TCP	Transmission Control Protocol
TDD	Time Division Duplex
TLS	Transport Layer Security
ToS	Type of Service
UUID	Universally Unique Identifier
VIMan	Virtual Interface Manager
VIX	Virtual air Interface of technology X
VLAN	Virtual Local Area Network
VM	Virtual Machine
WebGM	Web-based Gateway Manager

WFQ Weighted fair queuing

Wi-Fi Wireless Fidelity

WRED Weighted RED

Chapter 1

Introduction

This chapter provides an introduction regarding this master thesis work that addresses the topic of "Software-Defined Network for Internet of Things". It starts by presenting the scope of the proposed work, clarifying under which circumstances the thesis was proposed. It is then followed by the author motivation that is considered to be one important topic regarding this type of work. To conclude, the chapter lists the main goal and the specific objectives considered to achieve the proposed work, followed by an overview of the document structure.

1.1 Thesis Scope

Having gained strength from its promise of a facilitated centralised control over distributed network architectures, SDN is an ever-increasing topic of research [1] and is a broad term that is applied to a variety of approaches to network design [2]. This technique aims to provide a more simple, flexible and scalable network management. In contrast with usual networks, its architectures take advantage of network programmability allowing them to not fully rely on hardware.

Network slicing is a virtualisation capability based on SDN that allows multiple logical networks to operate on top of a common shared physical infrastructure. Thus, this solution enables turning traditional networking infrastructures into scalable and flexible ones, providing each device the choice of running on the slice that better fits its needs [3]. With the continuous growth of IoT, the need of having an easily programmable network that supports several scenarios is getting stronger.

IoT is composed of devices capable of sensing/actuation, communication and processing, that could be used to improve our daily life. However, the majority of production scenarios takes advantage of several nodes that are deployed under different circumstances. In light of that, it is crucial to consider how to enhance the system devices interoperability and consequently try to handle the necessity of distinct nodes resources sharing and usage [4].

1.2 Motivation

The need of supporting heterogeneous devices and technologies in communication is enlarging with the value and size growth in software and hardware. The industry is looking for a solution that enables an easier way for network connections with flexibility. IoT standardisation has been considered one of the main challenges to their deployment [4]. As the implementation of this technology exceeds the boundaries of the conventional networking solutions, SDN allows accessing the resources of such conventional networks considering today's networking service demands [5].

Because of the multitude of existing use cases, there is a clear need for a clever solution of network segmentation thought in order to be able to support particular and distinct scenarios. As each use case has specific characteristics and requirements, different network function capabilities need to be defined according to the application context. This comes with the need of being the most adaptable as it can be and provides the possibility to considerably improve the efficiency of performance requirements.

Since IoT and SDN are in the NGN context, despite all advantages that they can represent, inherent technology weaknesses do exist [6]. Since IoT is considered to be a growing technology, new and distinct devices are constantly appearing in order to be connected. Because SDN represents a flexible solution that simplifies those devices networking adaptation process, IoT downsides can be compensated providing this emerging technology a bigger slice of applications in the industrial and farming world. With the use of slices, SDN is able to easily provide those applications isolated network connectivity environments, allowing them to even adapt to their devices heterogeneity when it is required to do so.

This thesis work addresses this problem, taking the chance to understand and calibrate the requirements of each use case and application scenario to an SDN-based slice. The objective is to combine the standardisation support that IoT demands with the flexibility that SDN slicing can provide, taking advantage of its ability to easily adjust network resources when handling with distinct flows and services.

1.3 Main Goal and Specific Objectives

The main goal of this thesis work is to **develop a SDN-based solution considering multiple slices for multi-purpose scenarios**. In order to achieve that, I defined the following set of specific objectives that helped me develop the proposed solution and achieve the planned main goal:

Preparation of workbench based on SOFTWARE defined gateWAY and fog computing for Internet of Things (SOFTWAY4IoT) - During this period a familiarisation with the tool is expected to happen. In this thesis work it is essential to fully understand the characteristics of the project and its biggest advantages, making sure that it is taking the most of it.

Understanding SDN and the definition of slicing and scenario requirements - Since SDN and slicing in particular are a critical part of the problem, a detailed research about those topics is considered to be mandatory. With that being said, a set of articles will be read in order to achieve a strong theoretical background and define the best way to approach the problem. After that, a definition of the use cases is necessary in a way that a preliminary requirements study will be carried on.

Development of QoS slice managing module - Once familiarised with both, the theoretical aspects of the problem and the tool that will be used to solve it, a module development will be prepared. Accordingly, an exploitation of the characteristics that SDN, slicing and SOFTWARE4IoT can provide will help to develop a new component capable of managing the different slices.

Use cases implementation - One slice will be created for each use case that is intended to be implemented. This activity aims to use all the developed work with the purpose to solve a practical and real world problem, always considering the different aspects that it can imply. The preliminary use cases to take into consideration are Industry 4.0 and Digital Farming. Industry 4.0 focuses on the establishment of intelligent products and production processes. “In future manufacturing (often called smart manufacturing), factories have to cope with the need of rapid product development, flexible production as well as complex environments” [7]. Digital farming is applying precision location methods and decision quality agronomic information to illuminate, predict, and affect the continuum of cultivation issues across the farm [8].

Exhaustive tests of performance - In order to get an idea of the quality of the developed solution, a set of exhaustive independent tests will be performed under each created slice. This activity comes with the need of a guarantee that the proposed project constraints were met and the solution is working as expected, providing the user with the expected QoS level.

Understand requirements of each slice - After the tests period an evaluation of the preliminary study of the project requirements is needed. Based on the performed tests and each use case characteristics, a QoS implementation is expected to be chosen in order to relate it with the best solution to be applied under each slice.

1.4 Document Structure

This document is structured as follows. Chapter 1 pays particular attention to the thesis context, motivation and proposed main goals. Chapter 2 presents the literature review relevant for the understanding the topics addressed in the thesis and that offered the background knowledge for the development of the proposed work. Chapter 3 introduces this Master Thesis problem characterisation, going over works also related to the thesis scope. Furthermore, Chapter 4 describes and analyses the obtained results from the performed exhaustive tests. Finally, Chapter 5 draws

conclusions regarding the proposed solution and its relation with the performed tests, taking the chance to make a proposition of future work.

Chapter 2

Literature Review

The Literature Review of this Master Thesis provides the fundamental theoretical background knowledge and definitions on the topics related to the proposed work. The first section is intended to give relevant clarifications and standard assumptions regarding SDN. It is then followed by an introduction to IoT and how it couples with SDN. Furthermore, the chapter gives details on considered use cases. The chapter is concluded with background on performance requirements and how they can be applied to SDN.

2.1 Software-Defined Networking

Currently, SDN can be described as a networking paradigm where the control plane is chosen to be dissociated from the data plane. Because of that, a centralised control of the network is possible to occur and enabling the Controller to have a global network view where network applications run atop a Network Operating System (NOS) [1]. This technology deals primarily with the Data link, Network and Transport layers of the Open Systems Interconnection (OSI) model, what translates on running independently of the application level, with a logically centralised network sustained by a physical infrastructure.

In contrast with traditional switches, where there is little control over the corresponding forwarding tables, the use of a software centralised controller solution separates the routing experience from the device itself, eradicating potential complexities of dealing with individual network devices. This introduces the idea of dividing the data and control planes, as it allows a more fluent overall experience with the correspondent network protocols and inevitably introduces several benefits when compared to traditional networks: Flexible centralised control; Increase of network reliability; Stronger security; Decrease in complexity by means of automation; Modular control over the network; Simplified network management; Support of a scalable network; Improved user experience; Programmable network services.

Figure 2.1 illustrates the differences between both a conventional network node and a SDN one.

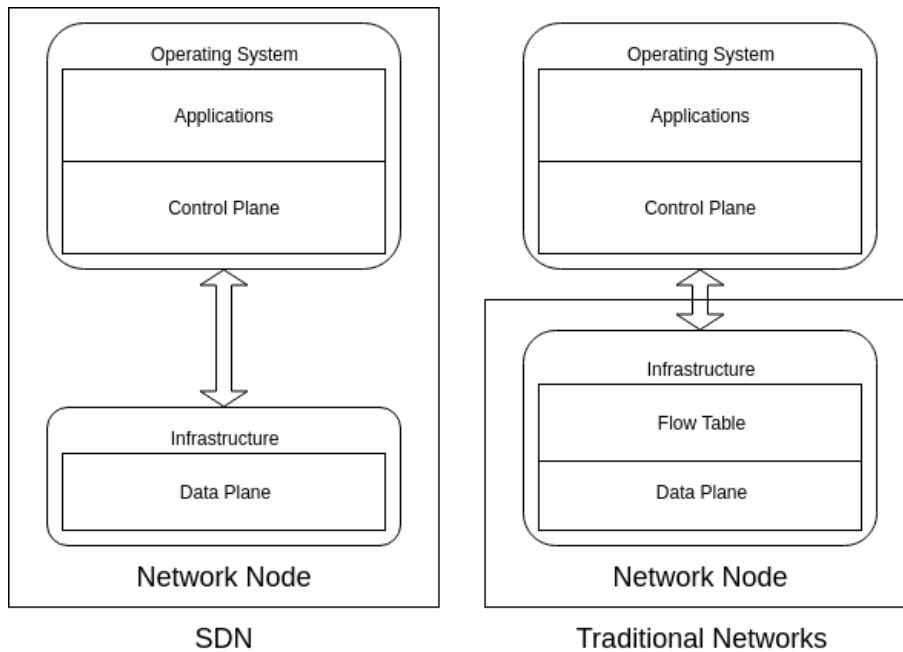


Figure 2.1: Traditional network nodes in contrast with SDN.

As it can be noticed, when dealing with SDN the control logic is transferred from the network node into the controller, whereas in the conventional approach each individual network node has its own control and data plane management [9].

2.1.1 Control Plane

As mentioned before, the fundamental aspect of the SDN world relies on programmability and with that in mind, it is possible to have an entire network of individual network devices (when individually programmed) to perform actions or be orchestrated as one single entity. The control plane, also named as NOS is responsible for abstracting the SDN physical state as an entire network perspective to a controller instance running in the Application layer [1]. This plane has the main duty of dealing with network management, having the ability to perform a continuous network reconfiguration if needed. Since the lack of standardisation represents one of the biggest issues regarding the topic [4], it is important to clarify and resolve potential incompatibilities as early as possible having in mind that each assumption has its own level of integration and community acceptance. With that being said, the NOS is mainly composed of three interfaces, also called Application Programming Interface (API).

Figure 2.2 shows two NOS, each one composed of a controller and its respective interfaces.

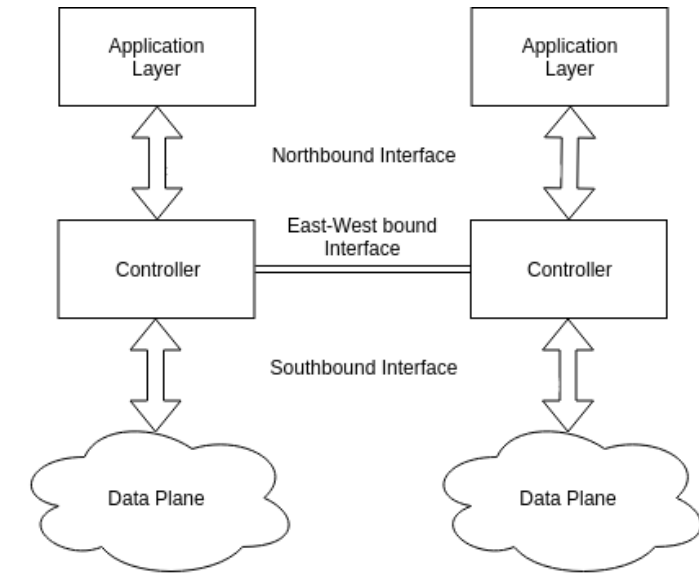


Figure 2.2: Software-Defined Networking architecture.

Most interface solutions were developed under an open-source policy, which has proven to be one of the most attractive aspects regarding the SDN acceptance in the researching community. Summarising, the Northbound API is responsible for performing any necessary communication between the control side of applications and the controller itself. On the other hand, the Southbound interface lets the controller exchange important instructions with the data plane. To conclude, the East-West bound API gives controllers the chance to share coordination policies and protocol related negotiations, which allows them to share a common perspective over the network.

For a better understanding, the mentioned APIs are further described in the next sections following a top-down approach.

2.1.1.1 Northbound Interface

As stated before, through this API, applications can orchestrate (or program) the data plane to perform complex tasks such as traffic engineering, topology discovery, QoS, load balancing, security policy enforcement, firewalls, delay and jitter management, and much more [1]. Docker containers [10] have been proven to be useful, as they provide a solid controller behaviour with the ability to run any combination of applications. Depending on the application instance that is running, a different functionality such as load balancing or security will be available to bring into action. It becomes clear that a controller is only useful when applications are running on top of it, and despite all potential advantages, it seems that the main drawback resides on the lack of available applications that meet with network operators requirements. However, there is still a need to allow the Control Plane to orchestrate all running applications. In order to solve that problem, Open vSwitch (OVS) [11] provides a solution that enables the use of network virtualisation along with containers, allowing them even to communicate in several modes [12].

2.1.1.2 East-West bound Interface

The controllers interactions that will allow sharing information within a SDN is handled via the east-west interface and can also be utilised for server-to-server or server-to-controller communication [1]. This API represents the bridge between different controllers within differing network domains. When dealing with large networks, it might be interesting to perform a segmentation into multiple sub-networks, each one running its own NOS controller. That is when this interface is favourable to use, allowing each NOS to coordinate packet routes and automate network decisions, thus offering a constant global network view update via full mesh connectivity.

2.1.1.3 Southbound Interface

The Southbound API provides the interface between controllers and switches using protocols like OpenFlow [13] to update flow table entries in switches [1], establishing the bridge between a Controller and the Data Plane. This interface enables high-level applications to interact with the network hardware (either physical or virtualised), what demands flexibility availability in order to easily adopt new control methods. For this thesis purpose, despite the lack of some researchers support, OpenFlow will be considered as the standard solution for this API and it might be important to present its pros and cons:

Table 2.1: OpenFlow main pros and cons.

Advantages	Disadvantages
Network devices and functions generalisation.	Not so good network status communication.
Network Operating System concept.	No default method to establish communication channels between the controller and data plane.
Distributed state management techniques.	No valid default mechanism that allows a full device management.

Since NGN require for the ability of network edges to adapt to new requirements and on the other hand, it demands the network core to perform a fast enough packet forwarding, several controller solutions that meet with these characteristics can prove to be helpful. Hence, flexibility is a key enabler for these technologies and allows OpenFlow to easily adapt to edge requirements [1].

2.1.2 Data Plane

The data plane is mainly responsible to assure the communication between the input and output ports is done in a proper way. This plane is formed by either physical or virtualised devices, where a flow-based progress is executed by modules that belong to the control layer. Its greatest current challenge is to become the most possible hardware constraints independent as it can be. Some examples of possible composing network elements are switches, routers or wireless access points where all of them can be virtualised and are interconnected by transmission media in order to

form a network. Having in mind the assumed standardisation taken for the Southbound interface, it is important to know the existent switches types in an OpenFlow network. Pure switches fully depend on the controller in order to forward decisions while the Hybrid ones support traditional networks operations and the corresponding protocols.

Because one of the IoT principles is to allow the connectivity of a huge number of devices, it might be interesting to inspect the data plane limitations that are related with scalability. Since this plane main related constraints reside on memory and CPU utilisation, the researcher's interest involves efficient operations of switching devices and optimising utilisation of transmission media. Memory usage and CPU utilisation grow with the network scale. In case of a big scale network, a significant memory space is required, otherwise continuous hardware upgrades are required to happen. This would avoid packet dropping or repeatedly directing packets to the controller, whose function would constantly be to determine how to process them, resulting in degradation of controller performance [9]. This scenario would harshly limit the system.

2.1.3 Network Virtualisation

With SDN continuous development, the convenient integration with Network Functions Virtualisation (NFV) became popular with time. As its name suggests, NFV allows for the virtualisation of hardware devices always assuring the passing by traffic is correctly flowing by means of SDN. Via direct programming or overlays, the SDN controller helps to set up appropriate virtual networks by monitoring the network topology and configuring the network ports accordingly [1]. As previously stated, when SDN chooses to divide the control plane from the data one, logical and physical networks are being split.

Mainly due to Network Virtualisation, the physical network is expected to perform traffic forwarding using traditional routing protocols, whereas the virtual network deals with network services, configuration programs and even access control lists. This allows for the creation of the so called *slices*, that represent nothing more than logical network partitions. Because Network Slicing is an important topic addressed in this Master Thesis, Figure 2.3 [14] provides a visual representation of this concept.



Figure 2.3: Network Slicing visualised.

Network slicing is a powerful virtualisation capability and one of the key capabilities that enables flexibility, as it allows multiple logical networks to be created on top of a common shared physical infrastructure. The greater elasticity brought about by network slicing helps to address the cost, efficiency, and flexibility requirements imposed by different characteristics scenarios [15]. These characteristics are crucial in order to correctly deploy a solution that meets the different slices and their use cases requirements.

2.2 Internet of Things

IoT can be defined as connectivity and interaction between objects/sensors/devices which allows for the deployment of an intelligent environment. Because of that, it enables a mind set modification in the direction of making possible a device to connect with any object in its proximity environment[5]. The produced amount of data by this technology is somehow restricting the evolution path of the Computer Networks world as it demands solutions with different characteristics from the conventional ones. Due to the nature of IoT network elements and services, the use of traditional network characteristics brings unwanted limitations on the researching world [4]. As it is of great interest to take advantage of IoT, it would be profitable to find a technology that meets its characteristics. Since the SDN world complies with the IoT architecture and both technologies are convenient when integrated together, agreeing with each other requirements, their combination seems to have great potential.

2.2.1 Coupling with SDN

Fortunately, the SDN paradigm can potentially be applied to IoT as an enabler for numerous SDN-based IoT applications. Yet, these applications also come with their own challenges. For instance, SDN requires greater abstractions to achieve the required levels of service and security needed to provide full potentiality to IoT infrastructures [1]. Additionally, as observed in SDN and because both represent emerging technologies, the lack of standardisation is the potential main drawback of IoT. Despite this, due to the topic constant researching and rapid technology arising, this problem is expected to be a diminishing problem.

It is now clear that IoT, as a NGN can prove to be a powerful technology when applied in integration with some of the previous topics. Since SDN was developed having in mind its affiliation with NFV, Network Virtualisation and the Internet itself, Figure 2.4 illustrates how the aforementioned technologies relate to IoT.

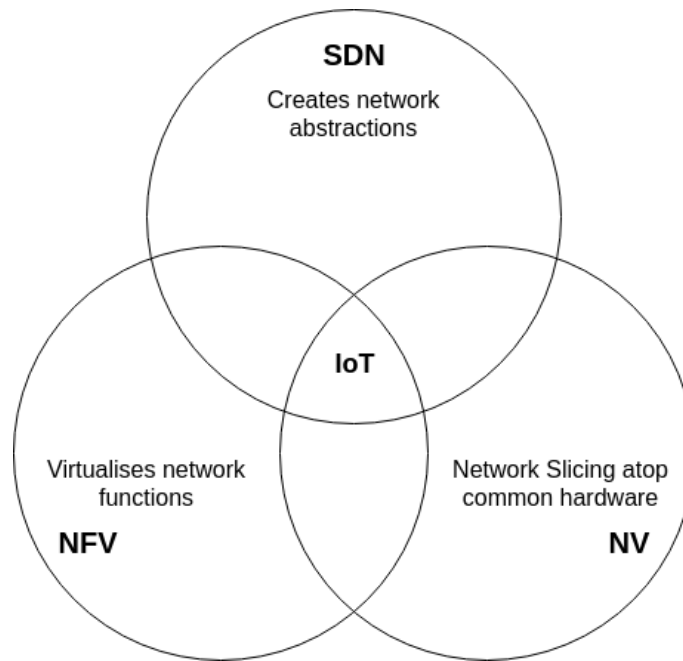


Figure 2.4: Relationship between Network Virtualisation, SDN, NFV and IoT.

Due to the SDN compliance with network functions virtualisation and network slicing, IoT is able to meet requirements and compensate most of its downside characteristics. Hence, as previously mentioned, this technologies are considered to empower IoT when compared its use with traditional networks.

2.2.1.1 Architecture

The more exciting thing the researching world finds in IoT coupling with SDN is the possibility for the same set of sensor nodes to support several applications from multiple developers across the same shared physical infrastructure via software only [1]. The SDN Northbound API enables for an IoT network orchestration which puts aside the limitations that conventional networks imply. As previously stated, because IoT represents an emerging technology, there are several versions regarding its architecture standards with distinct levels of detail. It is then necessary to make assumptions and in order to apply it to the SDN context, this work considers that IoT is composed of three layers as shown in Figure 2.5.

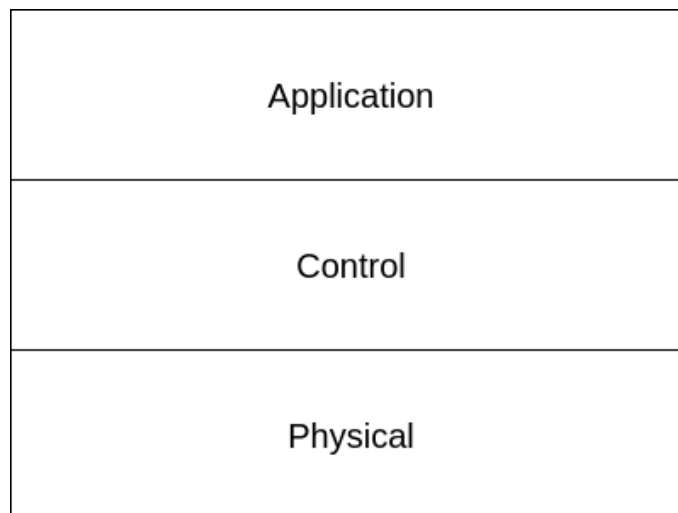


Figure 2.5: Considered IoT network architecture.

Application Layer - IoT applications exist mainly due to network abstractions. Consequently, the control and physical layers assure the correct functioning of this layer and are hidden from the application developer.

Control Layer - The control layer refers to the controller, taking advantage of the South-bound API in order to be able to manage devices from the physical layer and maintaining applications connection through the Northbound Interface. In addition, it provides potential network sharing and QoS-aware routing of the flowing data that from the core network towards its eventual clients [1].

Physical Layer - This layer is designed to be strictly composed of hardware. Having the SDN paradigm in mind, the physical layer devices are not provided with decision-making power, being this a task that only concerns the control layer.

2.2.2 Message Queuing Telemetry Transport

The Message Queuing Telemetry Transport (MQTT) is an open source, lightweight and adjustable network protocol that is considered by many the standard for IoT communications [16]. As its name suggests, it is a messaging protocol that supports asynchronous communication between devices. This enables this protocol to decouple the message sender and receiver in both space and time, transforming it in a good solution for unreliable network environments. Opposing to what its name suggests, this protocol does not takes advantage of messaging queues, using instead a publish and subscribe model which was built on top of the Transmission Control Protocol (TCP)/IP stack [17]. MQTT design principles focus on minimising network bandwidth and device resource requirements while also attempting to ensure reliability and some degree of assurance of

delivery. These principles also turn out to make the protocol ideal of the emerging Machine-to-Machine (M2M) or IoT world of connected devices, and for mobile applications where bandwidth and battery power are a serious parameter to have in mind [18].

Since MQTT takes advantage of a publish and subscribe model there are two fundamental entities in its network, a message broker and a number of clients. The broker is a server responsible to receive all messages from the clients and forward them (if needed) to destination clients. On the other hand, a client can be an IoT sensor or an application that processes IoT data. Because this protocol messaging system is organised by topics, the application developer has the flexibility to specify that certain clients can only interact with certain messages (a given client will only receive a message if he is subscribed to its topic). We can then consider to exist three stages in the MQTT messaging system. It begins when the client connects to the broker, subscribing to any message topic. This connection can be a raw TCP/IP connection or an encrypted Transport Layer Security (TLS) connection for sensitive messages. Afterwards, the client publishes content by sending the message and topic to the broker. Finally, the broker then forwards the message to all clients that subscribe to that topic. A visual representation can be seen in Figure 2.6 [17] below.

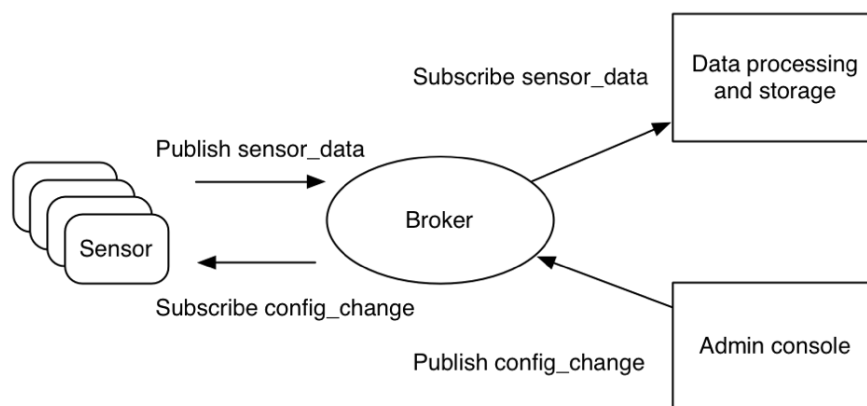


Figure 2.6: MQTT protocol visualised.

The protocol has a simple header to specify the message type, a text-based topic, and an arbitrary binary payload. The application can use any data format for the payload, such as JSON or XML, only requesting the destination clients ability to parse it [17].

2.3 SOFTWAY4IoT

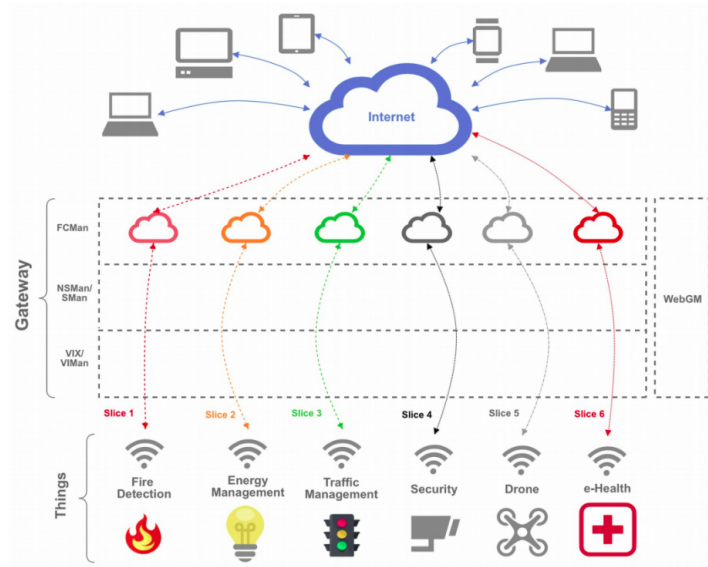
Considering the need for a tool capable of performing network slicing by means of SDN, this work considered a solution that matches both IoT and SDN requirements and enables the implementation of the proposed Digital Farming and Industry 4.0 use cases. Having this in mind, this section briefly presents the basis for this Thesis Work, the SOFTWAY4IoT [19].

SOFTWAY4IoT addresses four main problems: i) provide support for multiple wireless technologies for IoT; ii) minimise the impact of choosing a wireless technology for IoT not yet consolidated; iii) mitigate potential risks related to the network security of the connected devices; and iv) support connectivity of the devices to the infrastructure for data collection and processing, typically a cloud computing infrastructure.

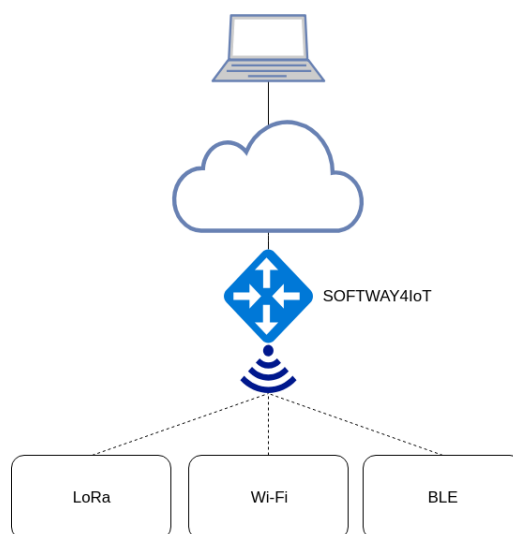
This centralised gateway enables management, monitoring, multiple gateways orchestration and considers heterogeneous devices with different hardware and software requirements. This is achieved by means of Software-Defined Networking and this dissertation takes advantage of the characteristics of SOFTWAY4IoT, using it to create the necessary slices for the target use cases. Despite the fact that SOFTWAY4IoT supports multiple wireless technologies, this Thesis Work considers only Wireless Fidelity (Wi-Fi), Long Range (LoRa) and Bluetooth Low Energy (BLE), typical technologies found in the Digital Farming and Industry 4.0 settings.

2.3.1 Description

As it can be seen by the nomenclature, SOFTWAY4IoT represents a gateway modelled by SDN and Software-Defined Radio (SDR). The solution chooses to interact with IoT devices by means of the cloud and Fog Computing, doing it by creating virtual network slices that will accommodate a certain service or device in compliance with its resources. Fog Computing can be described as decentralised computation process where its memory and processing power reside between the data source and the cloud, it is considered to be a powerful method because it distributes its computational effort through the network edge. Due to the solution architecture, the application is given the power of choosing where from to handle the produced data. It can be available from either a (public or restricted) cloud or a fog infrastructure. In Figure 2.7 [19], a visual clarification regarding the tool purpose can be observed:

Figure 2.7: *SOFTWAY4IoT* interaction with different devices

Because conventional networks do not take advantage of SDR, each wireless considered technology would need a dedicated gateway or physical interface in order to be able to perform communication. As SDR allows for multiple technologies to operate in the same gateway and even add/remove these possibilities by means of software only, the process of wireless technologies dealing is then considered to be simplified. Figure 2.8 depicts that. It should be mentioned that SDN4IoT does not consider SDR. However, as it will be demonstrated on Chapter 3, this Master Thesis reinvented this *SOFTWAY4IoT* module in order to be able to access the considered gateway through all proposed technologies and without taking advantage of SDR.

Figure 2.8: *SOFTWAY4IoT* dealing with multiple wireless technologies.

2.3.2 Architecture

The SOFTWAY4IoT solution resides on IoT corporative communication fully software based, virtualised and integrated with both possibilities of cloud and fog computing. Besides the gateway, this phase will include a centralised management interface designed to deal with monitoring and slices orchestration. This solution along with the gateway will be open-sourced, generic and flexible to the point where several communication IoT components can be replaced by the complete version [19]. Because SOFTWAY4IoT was developed with a modular approach, multiple components can be used and executed in several equipment. It is important to go into the tool architecture details, that can be found in Figure 2.9 [19] below:

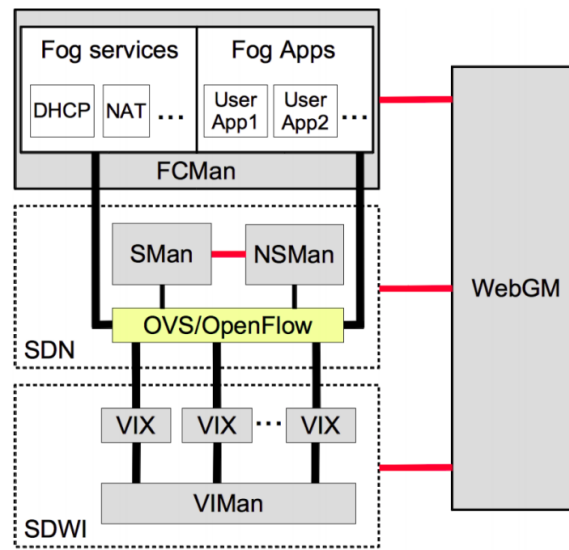


Figure 2.9: SOFTWAY4IoT components architecture.

Likewise the previous sections, each module will be considered in a top-down approach.

Web-based Gateway Manager (WebGM)

This web interface is directly connected to all SOFTWAY4IoT modules. It provides the network administrator with the ability to perform a careful management of the tool functionalities and components, it can be seen as good choice to accommodate the third OpenFlow downside presented in Table 2.1. The used programming languages for this component development were Flask [20] (backend), PostgreSQL [21] (database), Python [22], Hypertext Markup Language 5 (HTML5) [23], Javascript [24] and AngularJS [25] (frontend).

Fog Computing Manager (FCMan)

It allows for the virtualised computational resources management, providing both users and the infrastructure itself to run and isolate (if needed) several services or applications.

These users and infrastructure services are all instantiated in a Virtual Machine (VM) and in isolated Docker [10] containers. The software behind this module mainly relies on Python, Flask and the Representational State Transfer (REST) API.

SDN

This module main concerns are related with the network components configuration, representing the Control Layer. It is split in two major components, where Slice Manager (SMan) is the main responsible for the slice performing, isolating each IoT device by considering its general use case characteristics. On the other hand, the Network Security Manager (NSMan) addresses the basic network security needs such as Network Address Translation (NAT), firewall and intrusion detection. It is exploiting [OpenFlow](#) as its South-bound interface and its chosen controller solution was Ryu [26], mainly due to the OVS [11] (a proposed virtual switch) and OpenFlow compatibility issues. The module development programming language is Python and it takes advantage of the REST API.

Software-Defined Wireless Interfacing (SDWI)

Because SDWI resides on the bottom of the *SOFTWAY4IoT* architecture, it is responsible for handling the transmission and reception of the several considered wireless technologies signals. It takes advantage of SDR and similarly to the previous module it can be divided in the Virtual Interface Manager (VIMan) and Virtual air Interface of technology X (VIX) components. VIX responsibilities reside on performing the communication regarding a certain technology and assuring its modulation and codification processes are correctly performed. Due to the modular development approach of the tool, each new VIX component can be inserted and implemented in the gateway without the need of performing any changes in the remaining modules. Regarding VIMan, its main duty is to manage the physical layer of the gateway taking care of the actual signal transmission and reception in each VIX component respective frequency. It can be seen as a hypervisor of the SDR physical interface, implementing a transmission module and a multiple frequency band reception module. Because VIMan has the ability of interface abstracting, the VIX components only have the need to interact with a virtual interface and never a physical one. This module was developed under the GNURadio [27] development kit and its software relies on C++ [28] and Python.

2.3.3 Considered Wireless Technologies

Since the intended solution is dependent on a communication network, it is important to deeply understand its communication characteristics. *SOFTWAY4IoT* relies on IoT wireless technology, leveraging radio waves that follow a specific protocol based on the design intent of the IoT system. Radio protocols are used by devices to transport data to cloud platforms where a physical or wired connection does not exist. There are many different protocols to choose with characteristics varying in areas of power consumption, physical size, travel distance, data size, and transport

technology availability [29]. Despite the need to consider several factors when choosing the right radio technology, there are also many choices available for whatever the user requirements may be. This subsection is intended to walk through the considered wireless technologies, giving the reader the chance to learn about each used flavor and its corresponding category.

2.3.3.1 LoRa

As it can be seen by its nomenclature, LoRa represents a long range IoT radio technology. The main reasons why LoRa represents a good choice when utilised in the IoT world are: i) Long Range data communications; ii) Low power consumption; iii) Good behaviour on indoor environments; iv) Provides secure data transmission; v) Operates under the unlicensed <1 GHz frequency range [29]. Because it is important for its devices to not interfere with each other communications, this technology takes advantage of a spread spectrum allowing for its capacity increase and coding gain over narrow band communications. LoRa typical data rates go from 0.3 kbit/s to 50 kbit/s and allow for a range up to 15 km. Due to this technology characteristics it is not ideal for situations where large amount of data, high data rates, low latency and tight synchronisation are required [30].

2.3.3.2 Wi-Fi

Wi-Fi can be categorised as a medium range radio technology solution, where its signal is only perceivable for receiver devices within a 100 m radius of the transmitter position. This technology is based on the 802.11 standards and it was firstly designed in order to represent an alternative for traditional wired ethernet communications. All this wireless technology networks are Time Division Duplex (TDD) based systems. This leads to a competition between the access point and client devices in order to use the same channel. Due to the technology characteristics, Wi-Fi networks rely on a half duplex communication. This means that in order for given station to be able to transmit, it has to detect that no other device is using the channel. While transmitting, the Wi-Fi station is not allowed to receive information, making it impossible to detect a collision. All its transmissions have an associated acknowledgement. Hence, if a station happens to not receive an acknowledgement, it presupposes an occurring collision and retries after a random waiting time interval [29]. Considering the 802.11ac standard, this technology is capable of performing transmission data rates around 1300 Mbit/s and its frequency band resides on 5 GHz [31].

2.3.3.3 BLE

This solution is in the short range radio technologies category. Due to its 30 m signal range and low data rate it requires low energy in order to establish connections. In contrast with the traditional Bluetooth, BLE offers improvements in latency, power consumption, and data robustness which imply reduced effective data rates, maxing out at several hundred kbit/s [29]. Despite this, BLE has the potential to support many IoT wireless technology applications and because most of smartphones, laptops and operating systems have the ability to use it, a considering advantage

over many competing IoT wireless technologies is believed to exist. Similarly to the conventional Bluetooth solution BLE uses the 2.4 GHz radio frequency, however, they differentiate by BLE exploiting a simpler modulation system.

2.4 Use Cases

Use cases can be comprehended as a scenario analysis used to achieve a given system related goals. Because this work is in need for a real-world implementation, nothing is better than defining use cases in order to test the proposed solution. Each considered use case defines an application context, which implies the need for its characteristics analysis in order to find the perfect requirements each corresponding networking slice will contain, aiming to maximise the desired QoS. The following section aims to provide a background on both Digital Farming and Industry 4.0.

2.4.1 Digital Farming

Because agriculture can be considered to be one of the most complex systems in the world, there is a general ambiguity built up around Digital Farming. As previously stated, this topic is about precision location, real-time sensing and processes, and the generation of decision quality agronomic information across the continuum of the cultivation cycle. It is also about scalability, end-to-end processes, and generalised operational delivery to the agricultural community [8]. The technology location section takes full advantage of the Global Positioning System (GPS) system and represents the way how all the product geo-location services manage to perform sensing, identification, predictive decision making, and action.

Furthermore, the decision quality information is timely and within the decision loop of the event. It is delivered by advanced sensors, descriptive models, and predictive algorithms that provide the required insight to the agronomic issue. Tomorrow it will be about the sensed, objective, predictive power, and precision of the digital ecosystem [8]. Because the cultivation process depends hardly on decision making, it is essential to have well-timed information counselling regarding each action a farmer should take. The agriculture quality can be compromised due to the lack of essential information that should be provided, and that is why it is essential for Digital Farming to intervene in this process.

Digital Farming already is a reality in some areas: for instance, GPS guidance systems for controlled traffic farming, site-specific fertilisation or plant protection measures as part of a complete production/input cycle using proprietary cloud-based connectivity. This being said, automated data processing and completely integrated, harmonised networks still present a not-so-distant future for agriculture and agricultural machinery. It is then important to guarantee that some requirements are met. Considering the use case characteristics and because the main advantage of Digital Farming is the significantly larger information and knowledge-base obtained, it might be interesting to assure a permanent availability and performance capacity within the overall network itself [32].

2.4.2 Industry 4.0

Industry 4.0 nomenclature is due to being considered the fourth manufacturing revolution already occurred and it promotes the computerisation of manufacturing. It takes advantage of digitisation, adopting computers automation in all its processes and exploiting cutting-edge technologies which revolutionise the world of production. Since Industry 4.0 relies on sensors and machinery connectivity, NGN and in particular IoT seems to be a key component of its infrastructure assigned to handle all the generated data in a cloud environment and helping in general internal operations.

When computers were introduced in Industry 3.0, it was disruptive thanks to the addition of an entirely new technology. Now, and into the future as Industry 4.0 unfolds, computers are connected and communicate with one another to ultimately make decisions without human involvement. A combination of cyber-physical systems, IoT and the Internet of Systems (IoS) make Industry 4.0 possible and allow for smart-factories to be a reality [33]. It is mandatory to have transparency on this revolution generated information, that inevitably will allow to make appropriate decisions throughout all points of the production process, it will be possible to identify solutions for potential problems and improve inefficient systems that might exist.

Since one of the main goals of Industry 4.0 is the capability of performing automated decisions throughout all points of the production process, it is then necessary to have a good level of the requirements for capacity and delay. As time is considered to be a very important matter in Industry, all automated processes and information exchange need to be as fast as possible in order to contribute to the smart factory time efficiency.

2.5 Performance Architecture

Since one of the proposed objectives is based on understanding each slice performance requirements and proceed with calibration, it is important to determine what are the parameters that will define the solution QoS. It is then necessary to provide a performance definition in the computer networks context and how it is related with its architecture. Performance is the set of levels for capacity, delay, and reliability, maintainability, and availability (RMA) in a network. It is usually desirable to optimise these levels, either for all (user, application, and device) traffic flows in the network, or for one or more sets of traffic flows, based on groups of users, applications, and/or devices.

In support of performance in the network, a performance architecture is the set of performance mechanisms to configure, operate, manage, provision, and account for resources in the network that support traffic flows. The performance architecture shows where these mechanisms are applied within the network, and the sets of internal and external relationships between this and other component architectures [34]. A mandatory step to take when developing this work solution architecture will be to consider its main objective. It is then necessary to discuss the developing

goals for performance, doing it throughout the next subsections and entirely based on [34], which provides a solid and detailed explanation regarding this topic.

2.5.1 Performance Mechanisms

There are several performance mechanisms available, the ones that will be taken in consideration are: QoS; Scheduling; Traffic Management; Prioritisation; Queuing; Service-Level Agreement (SLA); Policies.

The Performance Architecture consists in determining which of the above mechanisms apply to the considered network and understand where each of them may be applied. The key is to perform a combination in order for them to provide the means to identify traffic flow types, measure their temporal characteristics, and take various actions to improve performance for individual flows, groups of flows, or for all flows in the network [34].

2.5.1.1 Quality of Service

Quality of Service, or QoS, is described as determining, setting, and acting upon priority levels for traffic flows. QoS is usually associated with Internet Protocol (IP) but is used here to define a class of mechanisms that provision and apply priority levels in multiple layers in the network [34].

Considering traffic based on IP only, there are two standard QoS types that are intended to have distinct perspectives over the network services:

DiffServ - Also called differentiated services, perform a network traffic behaviour analysis and aggregates traffic flows under a per-hop approach. IP packets are marked in the Type of Service (ToS) byte for IPv4 or in the traffic class byte in IPv6, giving intermediary network devices information regarding their prioritisation status, allowing them to determine the corresponding performance. The packets are then attributed a class of traffic flow that will be used to determine its allocated resources in an aggregated approach, what allows for scalability. Because this service does not require to store information regarding each individual flow, the amount of necessary resources to save state information is considerably reduced.

IntServ - Integrated services individually support traffic flows considering an end-to-end basis. It defines values and mechanisms for allocating resources to flows across the end-to-end path of the flow and it is closely tied to the flow nature of a network, by placing importance on supporting a flow at every network device in the end-to-end path of that flow [34]. Contrasting with Diffserv, this type of services may be expensive to use, because it requires support on the network devices and resources for each individual flow. Similarly to DiffServ, it also requires a signalling mechanism in order to share its flow requirements and necessary resource allocations across the end-to-end path, which is provided by Resource Reservation Protocol (RSVP).

Both services can be used in either an individual or combined approach, having the possibility to be applied in a specific area of the network or the entire network itself. The table below [34]

gives the reader a chance to compare both services in terms of scalability, granularity of control and scope of control.

Table 2.2: A Comparison between DiffServ and IntServ.

Function/Feature	Differentiated Services (DiffServ)	Integrated Services (IntServ)
Scalability	Scalable to Large Enterprise of Service-Provider Networks	Limited to Small or Medium-Size Enterprise Networks
Granularity of Control	Traffic Aggregated into Classes	Per-Flow or Group of Flows
Scope of Control	Per Network Device (Per-Hop)	All Network Devices in End-to-End Path of Flow

Since all types of resource control can rely on DiffServ and IntServ, the next section is designed to go through them.

2.5.1.2 Resource Control

Resource Control is a very interesting approach when it comes to provide network performance. The resource control mechanisms this work addresses are Scheduling, Traffic Management, Prioritisation and Queuing. These mechanisms are usually implemented in network devices such as routers and switches but can also be applied to the network as stand-alone hardware, such as in vendor-specific admission control and traffic management devices [34].

Prioritisation - Because there are situations where network entities clash for network resources and the network itself has a limited amount of them to offer, it is necessary to establish a prioritisation which determines who gets resources first, and how much they get. This mechanism is the process of determining which user, application, device, flow, or connection gets service ahead of others, or gets a higher level of service [34]. Since we are considering a performance architecture, there are two perspectives being taken into account:

Single-level - Capacity, delay and RMA are optimised globally.

Multi-level - Capacity, delay and RMA are optimised for one or more groups of traffic flows, based on groups of network entities.

Traffic Management - It can be defined by both the ability to perform an access control on network resources (admission control) and the power to modify traffic flows performance (traffic conditioning). Admission control takes advantage of priority levels in order to interfere with the network access, having the chance to grant, deny or delay it based on the traffic relative priority. On the other hand, traffic conditioning refer to classifiers, meters, markers, shapers, and droppers executed on network packets. Summarising, classifiers operate based on the packets IP header, marking it (if relevant) with a priority level. Metering is measuring the temporal performance characteristics of a traffic flow, including traffic rates

and burst sizes [34]. Shapers have the ability to change a traffic performance characteristic and droppers have the assignment of discarding it when it is desirable.

Scheduling - This mechanism determines in which order each traffic flow is transmitted, doing it by means of priority levels. It can be applied to several intermediary network devices, provided by either network management or its QoS implementation. The combination of QoS, prioritisation, traffic management, and scheduling provides a comprehensive set of mechanisms that can be applied across a network to achieve various performance levels for traffic flows [34].

Queuing - Can be addressed as a storage mechanism of IP packets waiting for being processed. Naturally, there are several queuing approaches and each one was designed in order to meet a particular goal in packet processing. Some examples of interesting queuing mechanisms are First in first out (FIFO), Class-based queuing (CBQ), Weighted fair queuing (WFQ) and Weighted RED (WRED).

2.5.2 Service-Level Agreements

SLA represents a formal contract between a provider and user that define the terms of the provider's responsibility to the user and the type and extent of accountability if those responsibilities are not met. Due to the natural characteristics of this work addressed problem and despite the several ways of the SLA appliance, it makes sense to be defined as an agreement between the network management and its network users (use cases) [34]. Naturally, examples of the SLA performance elements are bandwidth, data rate, upstream and burst tolerance. The SLA can also be used in order to specify capacity, delay, and RMA levels for various types of users, their applications, and devices. A basic, or best-effort are examples of distinct profiles, giving the user the chance to analyse and choose what service suits him better.

2.5.3 Policies

Policies are formal or informal sets of high-level statements and rules about how network resources (and, therefore, performance) are to be allocated among users [34]. Because policies combine the perspective of the expected network behaviour with network devices performance requirements, their purpose relies on creating or managing one or more performance goals. Summarising, policies are able to describe what network, computing, storage, or other resources are available to users, when resources are available, or which users are permitted to access certain resources [34].

2.5.4 Quality of Service in SDN

Because this work is dealing with SDN, it is fundamental to discuss QoS in its context. Recently, QoS through SDN is gaining a wide interest of researchers and they meet via flow routing, resource reservation, monitoring mechanisms, queue management and scheduling, and some other

QoS-oriented mechanisms [35]. The SDN controller facilitates the status detection and dynamic optimisation of the network performance. Virtual switches can be programmed through the controller by setting OpenFlow rules on each switch to forward specific packets [36]. In addition, it is important to mention the OVS support of QoS, which is able to perform both policing and traffic shaping on a given existent switch in the network. Because OpenVSwitch and OpenFlow have support for QoS, they become powerful tools when used together in the SDN world.

2.5.4.1 OpenFlow

Since this Master Thesis takes advantage of OpenFlow, being considered the standard protocol of the Southbound API for not only this, but most of the solutions that take advantage of SDN, it might be important to understand its role in a QoS context. Although SDN and OpenFlow couple support some limited QoS capabilities it allows the system to obtain per-flow QoS control in a more scalable and flexible way compared to the traditional network architectures.

OpenFlow enables network operators to use several routing algorithms (rather than the typical shortest path) within the controller to generate forwarding tables that govern distinct and isolated flows (such as the QoS flows in the data plane). Also, dynamic routing of flows are viable by controllers due to decoupling of control and forwarding functions of devices. These abilities, per-flow and dynamic routing, allow network administrators to come up with more QoS-motivated routing mechanisms for their networks [35].

Chapter 3

Proposed Solution

This chapter is intended to thoroughly provide both a visual and written clarification regarding the problem this Master Thesis is going to solve. The first section describes how the solution is coupled with **SOFTWAY4IoT**, giving details about its functioning and architecture at several levels. Furthermore, the second section gives an insight on the mechanism behind the connection of all IoT devices, taking the chance to refer to the sensors information processing and visualisation. Section number three provides information regarding the considered QoS techniques, going into detail on how they were implemented. The final section is where the solution is applied to the considered use cases, relating the QoS application to each use case characteristics.

3.1 Solution coupling with **SOFTWAY4IoT**

As previously mentioned, **SOFTWAY4IoT** is a gateway capable of network segmentation that takes advantage of software capabilities for doing so. Its architecture is composed by four modules that can be executed in different machines.

It is crucial to address the implemented setup, going into detail on how **SOFTWAY4IoT** and its modules were running and put together in order to have a fully functioning SDN gateway. Therefore, its implementation was divided throughout three machines: Host; VM-Application; VM-Manager. Once more, a top-down approach will be used with the purpose of addressing the functionality of each machine and mentioning their purpose on the standard architecture found in figure 2.2. VM-Application can be described as the machine that stands between the cloud and **SOFTWAY4IoT** itself, being responsible for high level tasks. It represents the FCMan module and consequently the Application Layer of the previously mentioned SDN architecture. VM-Manager main purpose is to run the SDN module, containing Ryu controller, with the additional task of hosting WebGM and all its administration capabilities, including the services which run on other machines. Hence, in the NOS perspective, this machine can be seen as the controller API. Both VMs run on the Host in a virtualisation environment with the help of Xen hypervisor [37], and all their components run inside Docker [10] containers, guaranteeing the previously mentioned isolation of services and applications. In contrast with VM-Application, the Host has

the responsibility of performing the lower level services and therefore represents the Data Plane. The SDWI service runs on the Host and it communicates with the VMs by means of OVS, which takes advantage of OpenFlow to update flow table entries in switches (the considered standard solution for the Southbound Interface mentioned on 2.1.1.3). A schematic of the implemented setup and how it fits in the NOS architecture is depicted in Figure 3.1.

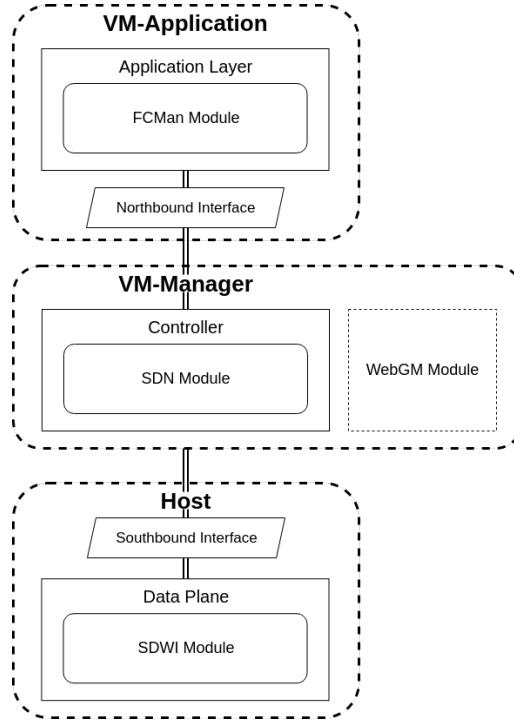


Figure 3.1: SOFTWARE4IoT implementation and its relation with the SDN architecture.

Hence, it is safe to say that the SOFTWARE4IoT implementation respects the standard SDN architecture, relying on open-source projects and taking advantage of modularity in order to stand up as a whole system.

It might now be easier to understand the role each created machine has on the gateway and how it relates with SDN. However, it is necessary to go in detail on how they function and consequently play their role on the SOFTWARE4IoT environment. Because each machine is directly related to one or several modules, a brief insight on this was already given on subsection 2.3.2.

3.1.1 Implementation machine specifications

Because this Master Thesis represents a virtualisation related work, it is important to clarify the implementation machine specifications and the version of the core utilised tools.

Both the implementation and all experiments were run on a DELL Precision 3630 computer with an hex-core Intel i7-8700K processor, containing 12 thread running at 3.70 GHz (clock base) and able to reach 4.70 GHz on turbo mode. Additionally, it was using 32 Gbyte of RAM (type

DDR4-SDRAM) which clock speed was 2666 MHz. Furthermore, the used Ethernet controller was an Intel I219-LM with 1 Gbit/s capacity and its wireless interface was an Intel-AC 9260 which took advantage of the "iwlwifi 4.19.0-0.bpo.2-amd63" driver. To finalise, the computer was running a 64-bit Debian GNU/Linux version 9.9 (stretch) as Operative System, utilising the "4.19.0-0.bpo.2-amd64" Kernel.

The installed Xen Hypervisor version was "4.8.5". Due to the *SOFTWAY4IoT* developers counselling, both machines VMs were running with 3 Gigabytes of RAM, where VM-Application took advantage of 4 Gbyte of Swap Memory while VM-Manager was utilising 1 Gbyte. Both machines were using the same OS as the one in Host computer. In conclusion, the Open vSwitch installed version was "2.6.2", whereas the considered OpenFlow version is "1.3". The SDN module was under control of Ryu version "4.30".

3.1.2 VM-Application

Since this VM is intended to represent the application layer, it has the main role of running applications with the purpose of dealing with the information coming from the bottom of the architecture stack (Data Plane). If it chooses to process information itself, then data is being processed between the cloud and the source device, and therefore Fog Computing is occurring. On the other hand, applications can choose to strictly redirect that information to the cloud, where it would later be computed and consequently Fog Computing takes place.

VM-Application has two physical interfaces in its perspective, which (since they merely represent OVS bridges) are in reality virtual ones. For a matter of convention let's call them "eth0" and "eth1". Interface "eth0" has the main role of providing applications connectivity to the controller and consequently any device available in the Data Plane. On the other hand, "eth1" is the interface that connects to the cloud, being responsible to pass information if any application chooses to process information by means of Cloud Computing.

All applications run inside separated docker containers which are automatically created when the order is given by WebGM. This guarantees information isolation and the ability to provide each application different connectivity environments if needed. One of the reasons Docker containers and services are so powerful is that you can connect them together, or connect them to non-Docker workloads. The deployed applications do not require to be aware that they are running on Docker, or whether their peers are also Docker workloads or not, managing them independently from the platform they choose to use [38].

In order to facilitate information exchanging between containers and VM-Application, *SOFTWAY4IoT* chose to use OVS (Northbound Interface), allowing the use of multiple Virtual Local Area Network (VLAN) on a single bridge. Of course this will result in a more complex and virtual-oriented system, but also allows for an incredibly adaptable and scalable system. Hence, each new slice will be represented by a VLAN in a newly created OVS switch and both "physical" interfaces will be bridged by "br-int" (eth0) and "br-ext" (eth1). We must remind ourselves each slice can have more than one running application, whose connectivity is handled by its OVS bridge with a newly running port. Figure 3.2 depicts the VM-Application system and networking scheme.

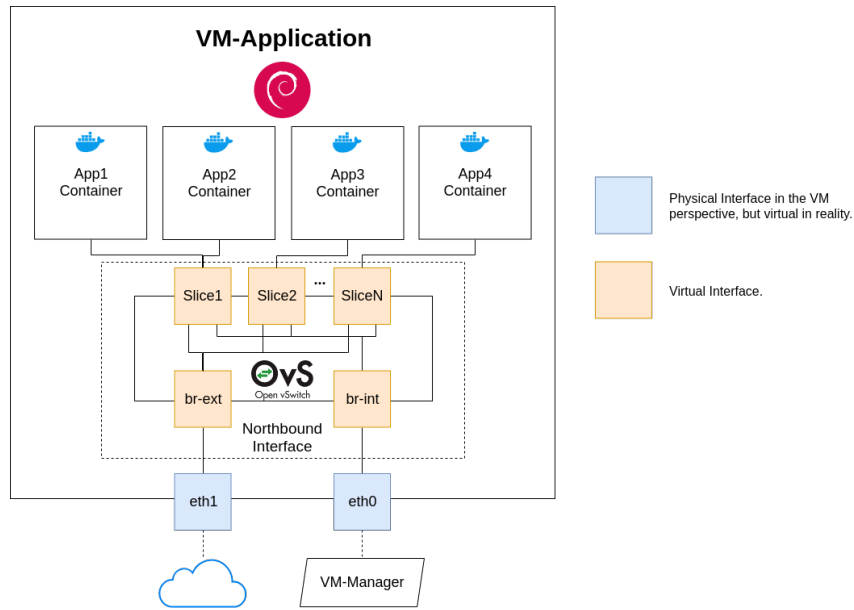


Figure 3.2: VM-Application System Architecture.

3.1.3 VM-Manager

VM-Manager is the most complex machine used in the SOFTWAY4IoT implementation. It plays the role of the core of the considered SDN system and allows for the administration features to take place. It is essentially divided in the WebGM and SDN modules, which are segmented in several services and will be examined throughout this subsection. As stated in the previous Chapter, and following the SOFTWAY4IoT ideology, its components are implemented on docker containers.

WebGM is the web interface that enables any network administrator to manage all gateway functions. Its services are WebGM-Frontend, WebGM-Backend and a Database. WebGM-Frontend hosts the actual web server which is used as presentation layer and interacts directly with the administrator. WebGM-Backend deals with the required APIs that allow to perform any kind of actions at a SDN level, storing the necessary information on the PostgreSQL Database. This module greatly enhances the SOFTWAY4IoT ease of administration actions.

As mentioned in subsection 2.3.2, the SDN module has the duty to run Ryu Controller, SMan, NSMan and runs a container completely dedicated to be an API to FCMan. As a matter of fact, this module was modified in order to support the Ryu QoS capabilities, which take advantage of a REST API and allows rules insertion in the network environment.

Because this machine was not in need for a scalable private Network, it uses a Linux native bridge and virtual interfaces in order to access each container. Similarly to VM-Application, it has Debian as an Operative System, but only uses one physical interface in the machine perspective. This interface is called "eth0" and provides the connection to both VM-Application and the Host machine. Figure 3.3 provides an insight of the VM-Manager architecture.

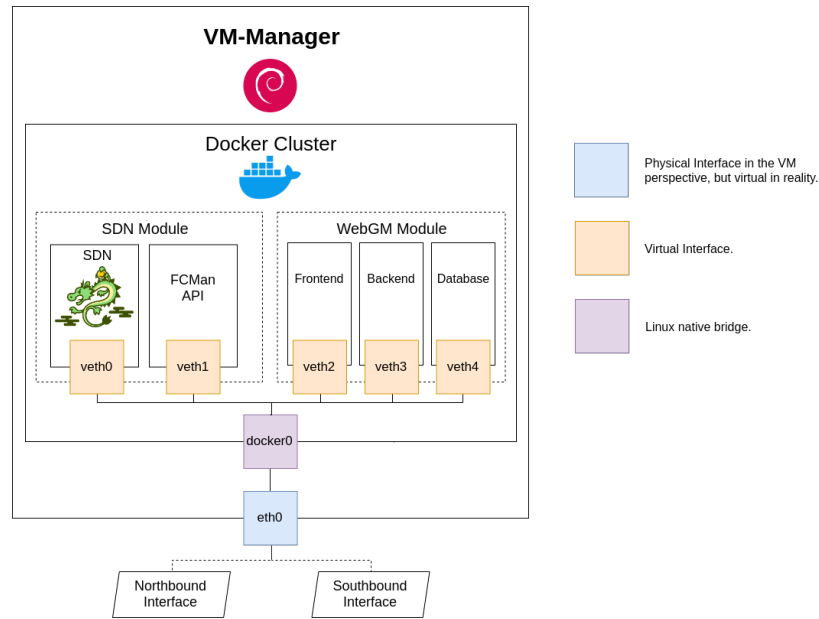


Figure 3.3: VM-Manager System Architecture.

3.1.4 Host

The Host is the foundation of the whole system. This computer is running Xen hypervisor, which is responsible for the execution and management of the previous virtual machines. In order for the inter-machine communication to take place, it takes advantage of OVS and consequently OpenFlow which function as the system Southbound interface. In addition to that, it is running a modified SDWI module and tries to take the best of virtualisation to perform its actions.

Regarding its networking architecture, it has two physical interfaces. The Ethernet "eno1" is responsible to provide Internet access, both to the Host machine and VM-Application through the OVS system. Besides of that, the wireless interface (that supports both, Wi-Fi and BLE) is intended to enable gateway connectivity to devices whose wish to communicate via those technologies. In order for the VMs to be reachable, OVS uses its bridging capabilities being "eno1" now represented by "br0", where "wlp4s0" and the SDWI Module are bridged by "br1", accessing both systems through OVS virtual interfaces.

In addition, as previously stated, the Host runs the SDWI module. In light of reasons that will later be explained, this module was modified to not take advantage of SDR, attending only to the technologies it will need. It is now divided in two services (VI-BLE and VI-LoRa), which no longer require VIMan to properly function. Figure 3.4 provides a visual clarification of the Host system.

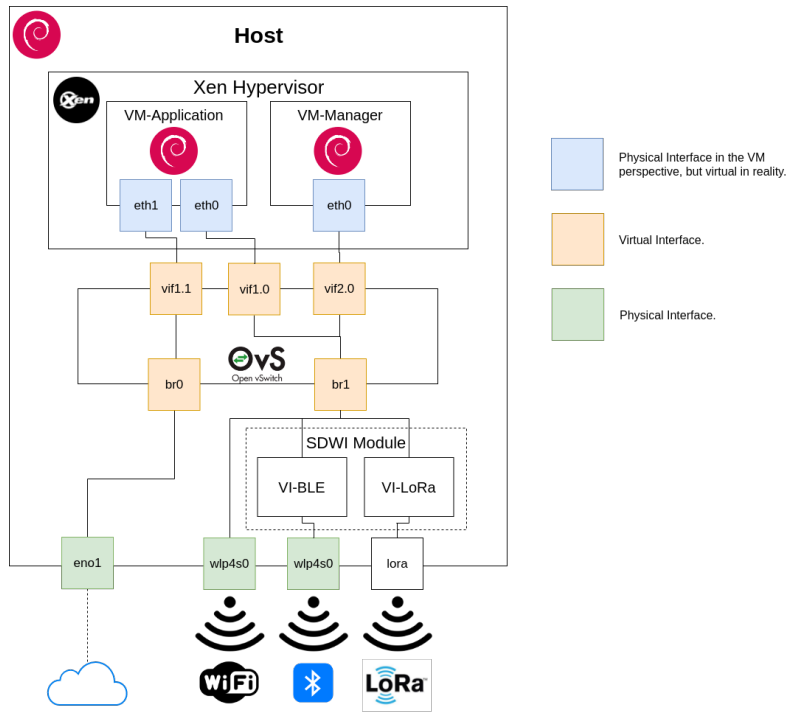


Figure 3.4: Host System Architecture.

Because the Host does not contain a compatible physical interface, it is understandable that LoRa might still be a question mark at this point. However, it will be explained in detail on the next section, which was designed to, not only but also, go through how VI-BLE and VI-Lora are able to handle their responsibilities.

3.2 Data Plane networking

As already mentioned on subsection 2.3.3, one of this work primary objectives is to support several wireless technologies, namely Wi-Fi, BLE and LoRa. In order to do so, the Gateway requires to handle the reception of their signals, interpret them and redirect to the above architecture layers. Additionally, it is necessary to consider that the raw BLE and LoRa technologies may signify problems in the information exchanging process. Those problems can be solved by means of a renovated SDWI module and only the LoRa physical interface is left for the gateway to be able to deal with the necessary radio services.

3.2.1 A LoRa physical interface

Because this work objectives do not require to handle information in a wireless technologies independent way, the radio communication system components are not dealt through software implementations. In other words, due to the limited amount of wireless technologies to deal with, SDR is not considered to be a priority. This way, technologies physical interfaces are responsible

for proceedings such as filtering, modulation/demodulation and codification. Despite of that, there is still a problem with the use of some considered radio technologies.

Now, as seen in the Host System Architecture (Figure 3.4), it is clear that the machine wireless interface supports BLE and can be used to fetch data from this technology end devices. On the other hand, LoRa still needs a physical interface ready to receive information which will be provided from this technology devices. For that reason, an Arduino Uno [39] device along with a Dragino LoRa Shield [40] was used as information transceiver, redirecting it to the SDWI module through its serial port. In brief, the shield was chosen due to its configuration flexibility, Arduino compatibility and low power consumption, representing a good choice for the technology access point.

It was necessary to use RH_RF95 [41], a driver which enables the reception of datagrams via a LoRa capable radio transceiver, which is also capable to configure its signal power and operating frequency. The device can be seen in Figure 3.5 [40].



Figure 3.5: Dragino LoRa shield used as access point.

After receiving and interpreting LoRa related data, this device is able to pass it to Host via USB cable which is represented by nothing more than a serial port in the Arduino perspective.

3.2.2 The new SDWI module

It is important to notice *SOFTWAY4IoT* uses IP when it comes to the Network Layer of the OSI Model. Therefore, having in mind BLE and LoRa protocol stack lack of IP, those technologies devices would not be able to properly communicate with the gateway. Hence, the new version of the SDWI module is required to perform the received packets IP encapsulation and their redirection to the assigned OVS bridge. For that reason, two new components were created, VI-BLE and VI-LoRa.

3.2.2.1 VI-BLE

All BLE devices use Generic Attribute Profile (GATT), what can be translated in APIs offered by operating systems which support this technology likely being based around its concepts [42]. GATT is built on top of the Attribute Protocol (ATT), establishing common operations and a framework for the data transported and stored by this generic data protocol [43]. In order to better understand how VI-BLE works, it is necessary to inspect the profile terminology, which works around server/client relationship and has essentially five important concepts: Client; Server; Profile; Services; Characteristics.

Because BLE focuses on low power consumption, their devices are not willing to send information by themselves and need to be requested for doing so. Hence, the GATT client (VI-BLE on Host) will have the responsibility to start a transaction on the GATT server (a given peripheral device), which suggests a connection interval that the client will respect in order to inspect if any new data is available. This concept is illustrated in Figure 3.6 [44], which in light of the previous proceeding, named the GATT client as "master" and as "slave" the GATT server. Due to the nature of BLE, each peripheral device can only be connected to one central device (Host). However, a central device can be connected to multiple clients, which is perfect for this work scope.

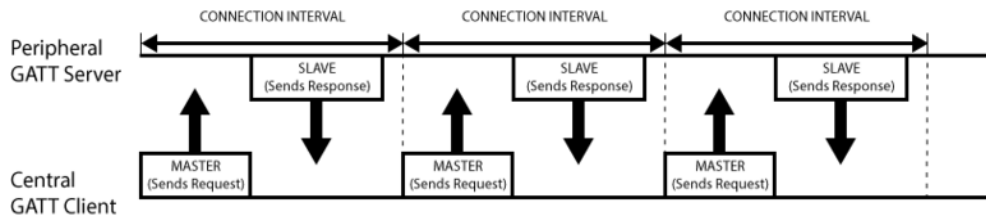


Figure 3.6: BLE client/server relationship.

Transactions are high level objects based on the Profile, Services and Characteristics concepts. It is possible to divide these nested concepts on an hierarchical scheme. Profiles are intended to represent a group of Services. On the other hand, Services are used to segment data into logic entities, where specific information on them are called characteristics. A Service can have one or more Characteristics, and each one distinguishes itself from other services by means of a unique numeric ID called a Universally Unique Identifier (UUID) [44]. Characteristics are then the lowest level concept of GATT, being the position in memory where actual data is stored. Similarly to Services, they are also represented by UUIDs. Figure 3.7 performs a visual clarification on the referred hierarchy.

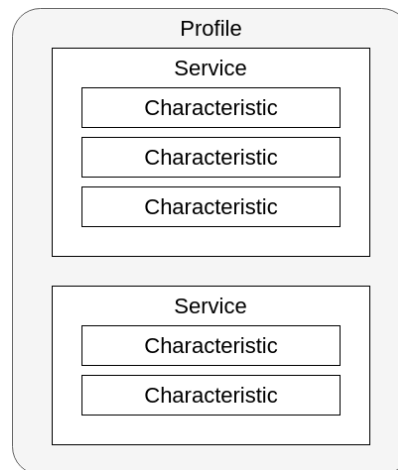


Figure 3.7: Profiles, Services and Characteristics hierarchy.

VI-BLE takes advantage of GATT and one of its Python modules[45] in order to fetch information from the intended BLE devices, using the host bluetooth adapter for doing so. Because it might be interesting to filter the devices VI-BLE wants to connect to, a JavaScript Object Notation (JSON) file is kept containing a list of Media Access Control (MAC) addresses. Additionally, that file has a related IP address to each assigned device, which is used to create a Linux virtual interface and consequently send the fetched information to the Gateway upper layers by means of MQTT.

3.2.2.2 VI-LoRa

The used LoRa network architecture is deployed in a star topology in which nodes, gateway, network servers, and application servers are placed. A node collects sensor records and sends them to a gateway by using single-hop wireless communication with LoRa modulation. The gateway is connected to a central network server and acts as a bridge between the node and the network server, converting Radio Frequency signals to IP packets. The network server determines which data belongs to which node and removes duplicated data, redirecting it to the Application Layer. Hence, the applications that reside on the top of the SDN architecture are used to collect and analyse data from nodes [46].

As it might be noticed, in this work solution context, the LoRa network Application Server is represented by the running applications on the corresponding VM. Additionally, because the component VI-LoRa joins forces with Arduino LoRa Shield in order to perform not only data interpretation, but also packets IP conversion. Because of that, it is responsible to function as both Network Server and Gateway in the technology network architecture. To conclude, nodes are the end devices whose information is fetched from. The LoRa network architecture and its relation with SOFTWAY4IoT can be seen in Figure 3.8.

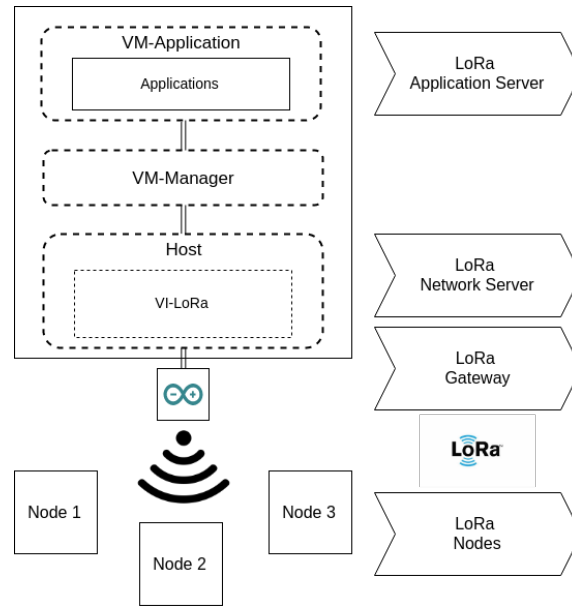


Figure 3.8: LoRa network architecture and its relation with SOFTWAY4IoT.

To summarise, the utilised Arduino LoRa Shield device is able to interpret data by itself and contrarily to BLE devices, LoRa nodes do not need to be stimulated in order send the fetched sensors data. Hence, when a node connects to SOFTWAY4IoT both IP and MAC addresses are generated and kept in a small database which resides inside the SDWI module. After receiving the sensors information, under the same approach of VI-BLE, VI-LoRa uses MQTT in order to exchange data with applications.

3.2.3 Resultant network architecture

All the solution components were already inspected, including the renewed SDWI module which takes a crucial role on the gateway support for the proposed wireless technologies. It is clear that most networking on the solution VMs take advantage of the Openflow protocol in order to properly function. Additionally, it might be interesting to understand which protocols the implementation takes advantage to communicate. In light of this, this subsection is intended to provide an overview on the solution network architecture and to inspect the used protocols on each link. The system protocols overview is illustrated in Figure 3.9.

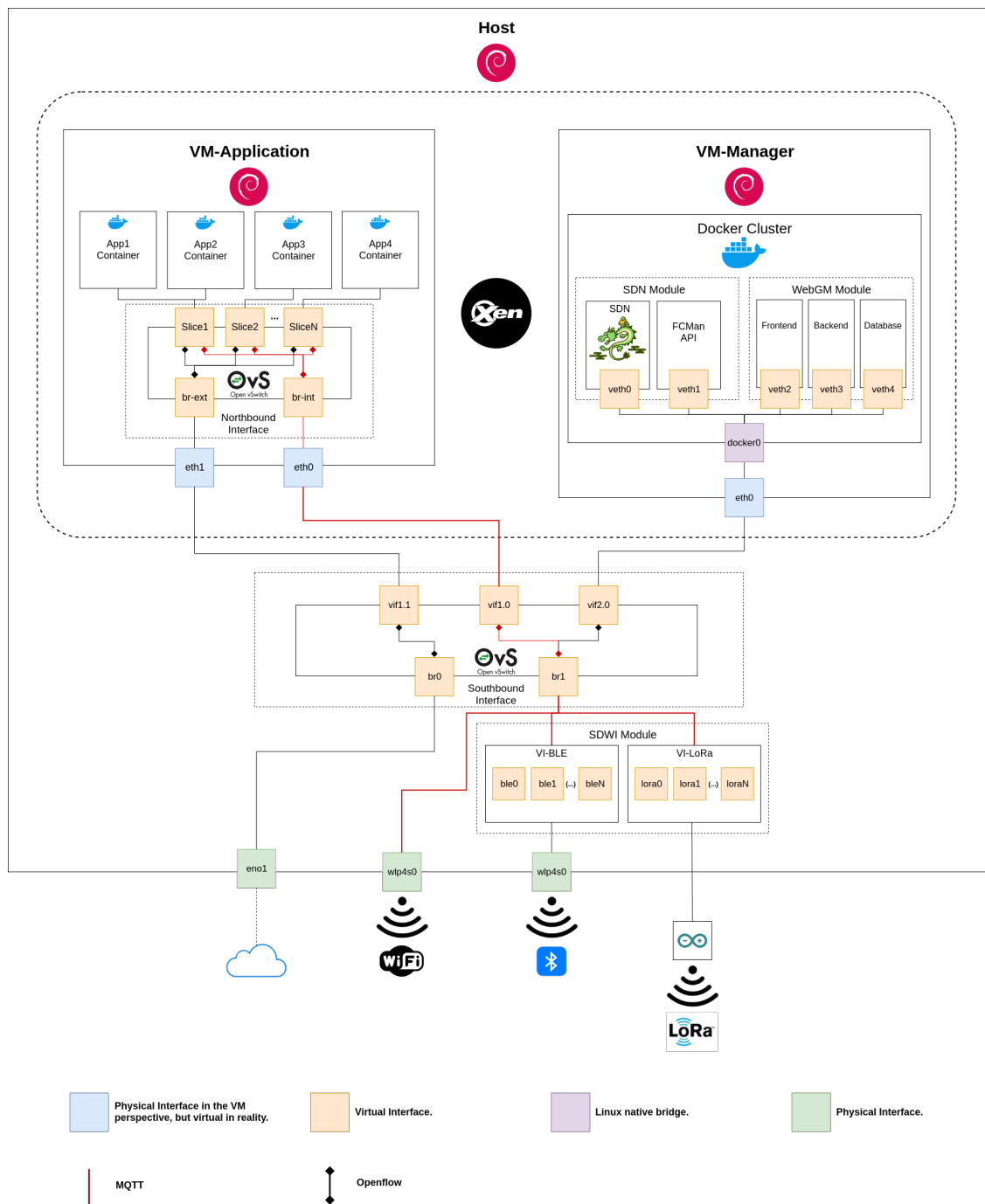


Figure 3.9: Overall network architecture.

Summarising, the Host is responsible for dealing with the Data Plane related tasks of the system architecture. If devices choose to use Wi-Fi in order to access the gateway, traffic enters directly to the system Southbound Interface. On the other hand, in case of BLE and LoRa devices, data firstly goes through the SDWI Module in order to be encapsulated. All those communication processes take advantage of the OpenFlow and MQTT protocols.

VM-Manager is responsible to run the administration features by means of Docker containers. It provides lodging for the SDN Module, which includes the Ryu Controller and FCMan API. Additionally, a web interface with the purpose of facilitating the network administrator actions is hosted on this machine. Furthermore, this important component can communicate with the VM-Application components through the system Northbound Interface.

VM-Application is responsible for dealing with the slice related networking, assigning each one to a newly created OVS switch. Additionally, it has the task to run all slice applications on separated Docker Containers. All those actions are orchestrated and supervised by VM-Manager, where the controller resides.

3.3 QoS Manager

This document already went in detail on how the proposed gateway was implemented. At this point, it is safe to say that a fully functioning SDN gateway is running and capable of fetching data from all proposed wireless technologies devices. Additionally, those devices can be assigned to different network slices if needed. However, all of them are being given the same connectivity conditions, what might not be the ideal scenario if their production environment is not the same. Hence, the solution is in need of a component that would be capable to provide each slice different QoS levels in order to better fit its use case application. As a mean to solve this problem, a new module by the name of QoS Manager was developed. Like mentioned on subsection 2.1.1.1, the Northbound API can program the data plane to perform QoS and because of that, VM-Application is the machine where the QoS Manager module will be placed.

Like shown on section 2.5, there are different QoS models and definitively several aspects to attend regarding its implementation. This section is intended to go through them and how they relate with the SDN scenario.

3.3.1 Module Placement

The current solution has several SDN related running components which are capable of setting QoS levels, namely Ryu Controller [47], OVS [48] and Openflow [49]. Because the original SDN module was modified, the network administrator is now able to perform REST API requests in order to install QoS rules. It must be noticed that despite all those requests, rules are in reality imposed by OVS and Openflow. Hence, the Ryu REST API enables the user to perform requests in order to enable an easier QoS implementation at an OVS and Openflow level. However, because the SDN core resides on the controller, the administrator will only be able to apply QoS on OVS switches that are connected to Ryu Controller.

Now, taking a closer look to the VM-Application architecture and how it relates to OVS. It is clear that all information that comes from the bottom layers passes through the VM "eth0" interface and consequently "br-int" too. Each flow is then divided on "br-int", taking the direction of the bridge that belongs to the destination slice. On that bridge, several ports are placed where some are meant to provide connection to their application containers.

On the other side, it is important to understand that policing can only be applied on traffic which is entering a given OVS switch. Additionally, it should be noticed that "br-int" is connected to any container bridge by means of a patch port. OVS documentation claims that bridges which are connected through patch ports behave like a single one [50], what translates in the Northbound Interface to be represented by only a switch in the QoS Manager perspective. It is important to mention that this could be overcome if it was chosen to implement those rules utilising the Linux Traffic Control tool. As it will be inspected, OVS itself takes advantage of Linux Traffic Control (TC) in order to perform its QoS implementations. However, this would implement features that are not currently supported by OVS, what does not represent this Master Thesis scope.

With that in mind, considering the current implementation, policing will only be applied on the port that is meant to provide connection to the lower layers of the gateway architecture ("br-int" of VM-Application). Hence, this kind of traffic limitation will affect slices at a single-level perspective and their traffic will be split in order to respect those limits. Contrarily, traffic shaping QoS rules are going to be applied in ports that allow traffic to egress the Northbound API environment, making sure this module will actuate in both an application and slice independent way. This can be visualised in Figure 3.10.

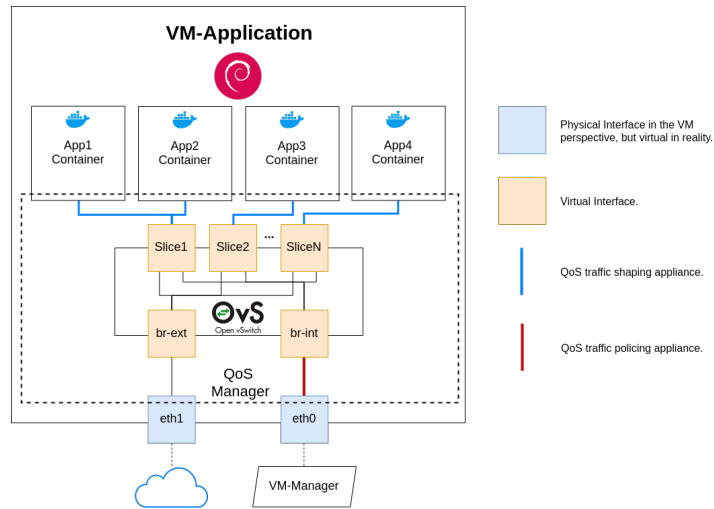


Figure 3.10: QoS Module placement.

3.3.2 Module Functioning

QoS Manager is considered to be an important puzzle piece in this work context. This leads to an importance of going into detail on how this module is able to apply QoS rules. As mentioned in the previous subsection, while Ryu provides a built-in REST API that enables some QoS rules application, those are actually implemented by OVS and OpenFlow features. Because of that, despite of having different roles, it might be interesting to explain how those tools are put together

in order to implement the major QoS models characteristics. Additionally, each tool will be inspected with the purpose of elaborating how their QoS implementation architecture can be related to the QoS Manager module.

3.3.2.1 OpenVSwitch role

As previously mentioned in this document (subsection 2.5.4), there are mainly two ways of providing QoS with OVS. For traffic that ingresses into a switch, OVS has the ability of performing policing [48], which is the action of dropping packets whenever the allocated network resources have been exceeded. On the other hand, when traffic is egressing from a switch, OVS supports traffic shaping. Unlike policing, shaping will slow down the packet transmission or reception instead of dropping the packet, inserting them in queues waiting to be dispatched [51]. The implementation is depicted in Figure 3.11 [52].

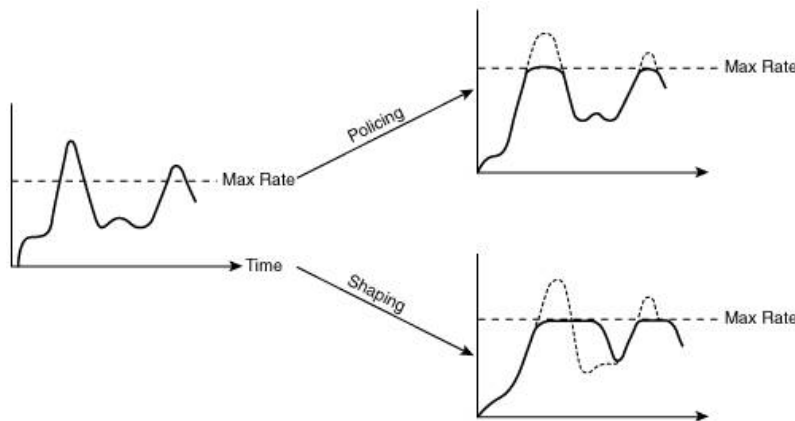


Figure 3.11: Comparison between policing and traffic shaping.

It is important to mention that OVS does not implement QoS by itself, taking advantage of some Linux QoS features previously built in its Kernel. Hence, in order to understand what this tool is capable of doing, it is necessary to firstly comprehend what features Linux QoS supports. With that in mind, it might be interesting to talk about the TC utility.

3.3.2.2 Linux Traffic Control

Linux TC is a very useful utility that has a wide range of applicability and is available in common Linux distributions. As the name suggests, this technology has traffic control related features [53], which rely on Shaping, Scheduling, Policing and Dropping. In order to fully understand this technology it is necessary to go through some interesting concepts (Queueing Disciplines (Qdiscs), Classes and Filters) which take important roles on the Linux TC implementation.

Qdiscs are the basis of traffic control. Whenever the Linux Kernel has the need to send a packet to a certain interface, it is enqueued to the Qdisc that is assigned to it. After this step the

Kernel attempts to get the most packets it can from that Qdisc, and when it does so, it proceeds delivering them to the network adaptor driver [54].

There are two kinds of Qdiscs: classful and classless. It is then possible for a Qdisc to contain classes, which were implemented to function like branches on a tree. Therefore, a class may have assigned other classful or classless queue discipline. This allows the Linux TC to prioritise certain kinds of traffic, trying to process and dequeue a given discipline before another.

Filters are used by classful Qdiscs to return a verdict regarding in which class a packet should be queued. Because classes are allowed to have subclasses and traffic needs to be classified, filters are called in order determine which class should be assigned to it.

There are several types of both classless and classful Qdiscs. On the classless side, we have FIFO, Random Early Detection (RED), Stochastic Fairness Queueing (SFQ) and Token Bucket Filter (TBF). Classful Qdiscs are CBQ, Hierarchy Token Bucket (HTB) and PRIO. This work and OVS take advantage of HTB, which has shaping capabilities based on TBF and has the ability of classes prioritisation. TBF is a good choice when it comes to limit traffic to a configured maximum rate, dealing especially well with big amounts of bandwidth. HTB works by implementing a well composed link sharing hierarchy of classes with an emphasis on conforming to existing practices. It facilitates guaranteeing bandwidth to classes, while also allowing specification of upper limits to inter-class sharing [54].

3.3.2.3 OpenFlow role

OpenFlow contains interesting features that can prove to be essential for the QoS Manager good functioning. Because this module switches take advantage of this protocol, inherently contain three main components: Flow tables; Group table; OpenFlow channel. Flow tables and the group table are usually put together and allow to inspect and forward packets to a given port. On the other hand, the OpenFlow channel is used by Ryu controller in order to apply any rules by means of the OpenFlow protocol. In light of this, the controller can choose to insert, update and remove flow entries from flow tables, in a reactive (based on receiving packets) or proactive way. Hence, flow entries are put together in order to form a flow table and mainly consist on several characteristics. The OpenFlow flow entries structure is shown in Table 3.1.

Table 3.1: Considered flow entry characteristics.

Matching Fields	Timeouts	Priority	Counters	Set of instructions
-----------------	----------	----------	----------	---------------------

Matching fields are used to successfully identity packets within a flow. The ones considered in this work context are the ingress switch port and the packets headers. Priority sets the order of policy application upon flow entry. Because of that, if two flow entries simultaneously match a given packet, the one with higher priority will be used. Counters are incremented when a packet matches a flow entry. The set of instructions defines what actions should be performed. Concluding, timeouts set the maximum idle amount of time that a flow entry can prevail in a given switch.

Since an OpenFlow switch can have multiple flow tables, it takes advantage of a per-table packet processing. The packets matching process is firstly applied in the initial one, progressing to the following if they exist and the predefined set of instructions chooses to do so. Furthermore, if that flow table happens to have a matching entry, the corresponding set of instructions is applied and executed in case of no flow table is left to perambulate. On the other hand, if there is no predefined entry in the table its destination is specified by the settings that are provided by a table-miss rule or dropped in case of absence of it. Figure 3.12 illustrates how packets are handled at the Openflow protocol level.

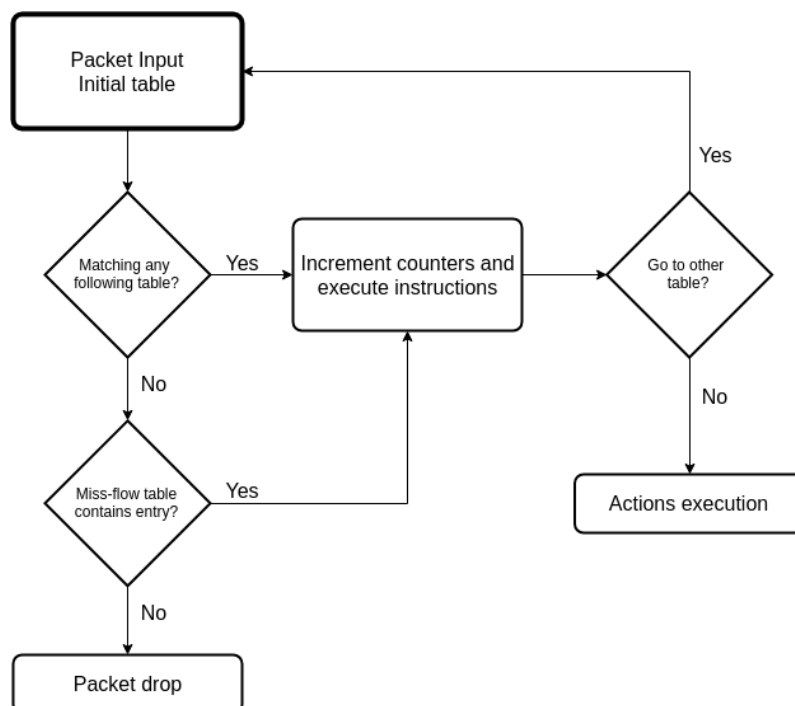


Figure 3.12: Typical Open vSwitch packet flowchart.

The above flowchart is important to understand the role of OpenFlow in the QoS Manager module. Due to its matching capabilities it is possible to distinguish packets based on its headers and its ingress port. This can prove to be essential in flows identification, where the IntServ model is applied. Additionally, because in order to perform traffic classification DiffServ uses a 6 bit Differentiated Services Code Point (DSCP) within the differentiated services octet, services differentiation is also possible to occur. Furthermore, based on the performed matching, packets can be redirected to any OVS created queue and obey to distinct queue characteristics.

Hence, because this module takes advantage of OVS and Openflow, it is able to implement the major QoS models which can be calibrated in order to attend several scenarios. In order to aid the understanding of what both technologies are able to do when joining forces, an example configuration can be found in Figure 3.13.

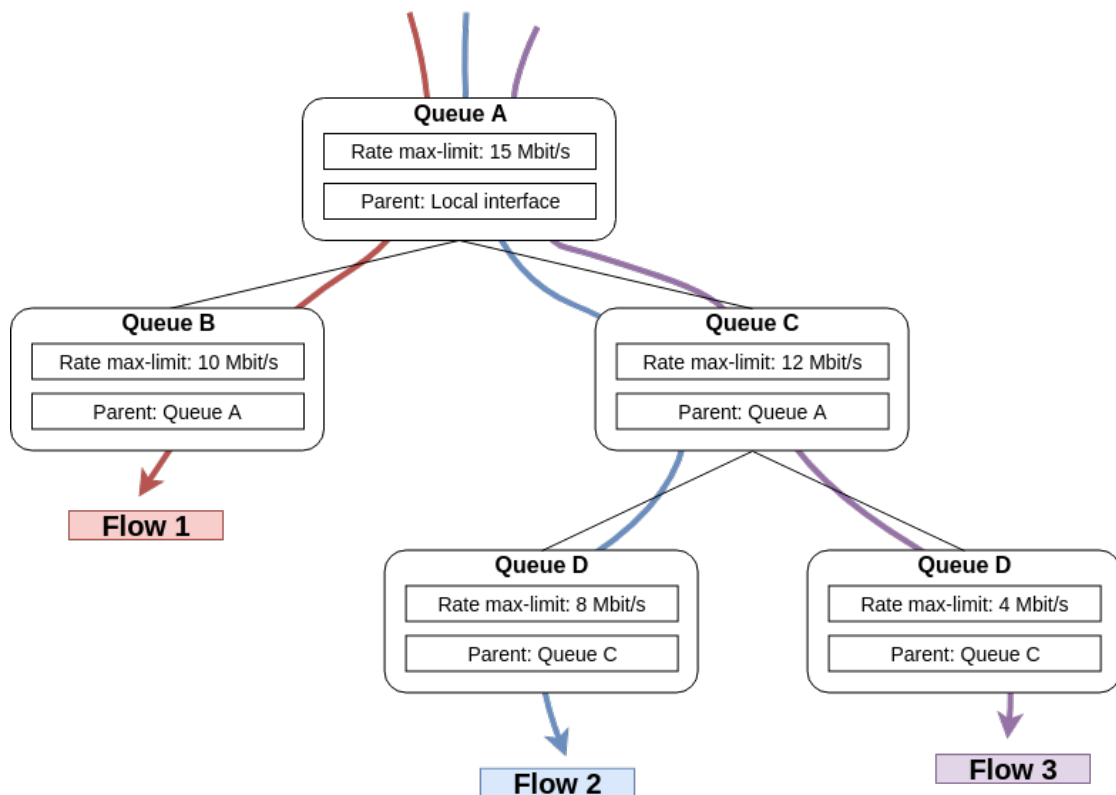


Figure 3.13: HTB configuration example.

3.3.3 QoS Scenarios

Due to the nature and complexity of real-world use cases, allied with the flexibility that SDN can provide, there are some interesting QoS scenarios that can occur on this work gateway context.

It must be reminded that this section scope is to go into detail on the solution QoS implementations capabilities and how they relate to the major QoS Models, whose characteristics already were discussed on Section 2.5.

3.3.3.1 Per-Interface Policing

In this scenario, QoS functionalities are implemented on the gateway by using traffic policing on the ingressing ports of a switch. Rules are setup to limit the transmission rate on a given interface by setting the ingress policing rate and ingress policing burst. This is a simple form of QoS which drops packets received in excess of the configured maximum bandwidth.

Due to its simplicity, policing is usually less accurate and less effective than egress QoS. For its implementation it uses a "token bucket" approach, where the size of the bucket corresponds to ingress policing burst. Initially the bucket is set as full. When a packet is received, its size is converted to tokens and compared to the number of tokens currently in the bucket. In the eventuality of the required number of tokens being available, they are removed and the packet is

forwarded. Otherwise, the packet is dropped. On the other hand, when the bucket is not full, it is refilled with tokens at the previously configure maximum rate. Mainly because policing is applied with an ingress approach, all created slices will be influenced in a similar manner. Because of that, each slice assigned switch will share an equal part of the total bit rate that is flowing in the gateway.

3.3.3.2 Per-Flow QoS

Per-Flow QoS adds Queue settings and rules to reserve network bandwidth, taking advantage of egress traffic shaping for doing so. Hence, the target interface will not only be limited to a specific bandwidth, but instead distinct known applications or services can be assigned a bit rate using IntServ classification [51]. Traffic shaping is configured using the QoS and Queue OVS tables and uses Openflow in order to identify and separate services on the solution SDN environment. Figure 3.14 depicts the Per-Flow QoS congestion point.

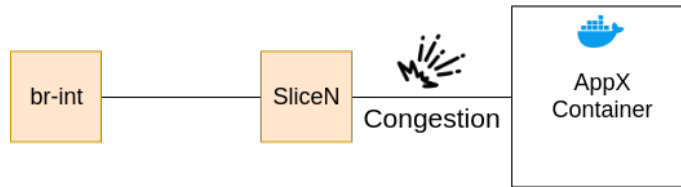


Figure 3.14: Per-Flow QoS congestion point.

3.3.3.3 DiffServ QoS

While Per-Flow QoS is able to have a good behaviour as communication flows increase, the flow entries number which are set for each OVS switch also enlarge. Hence, the Per-Flow QoS is considered not to be scalable. Contrarily, DiffServ QoS divides flows into the several QoS classes at the domain entrance (br-int) and applies DSCP marking, matching them with the associated queues in each slice OVS switch. Therefore, packets will be forwarded according the first six bit of ToS field in its IP header, and realises QoS [55]. Figure 3.15 clarifies the solution DiffServ QoS congestion point.

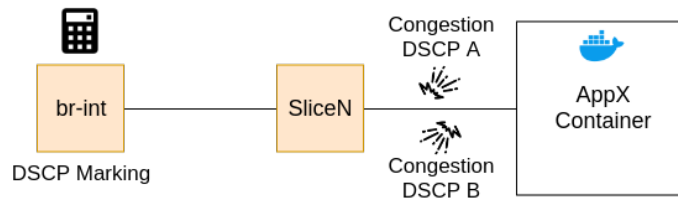


Figure 3.15: Diffserv QoS DSCP marking and different congestion conditions.

3.4 Related Work

This section provides an insight on previously concluded SDN and QoS related works, specifying on how SDN4IoT plans to mitigate their downsides.

Previous work on this topic include that of M. I. Alipio et al. (2016) [49], where the implementation of traffic policing is achieved by setting priority in the IP header of the packet, discarding any kind of traffic shaping and using OpenFlow 1.0 for doing so. Additionally, this work chooses to use a set of RaspberryPi in order to create the SDN related switches and only consider the OVS capabilities for testing purposes.

Furthermore, R. Durner et al. (2015) [56] study dynamic QoS mechanisms for flows in an OpenFlow-based network, focusing their results in prioritisation and bandwidth guaranteeing, which pay only attention to both classless and classful queueing disciplines and do not consider DSCP traffic marking. Durner realised that the use of OpenFlow on different switches develops a variety of testing results, which is a detail that SDN application developers must consider when implementing QoS concepts in the network. Moreover, the single considered transport layer protocol is TCP, discarding any type of UDP-based traffic for testing purposes.

W. Hsu et al. (2016) [57] go into detail on a proposed architecture for QoS/Quality of Experience (QoE) mapping and application adjustment on the SDN context, taking advantage of Ryu Controller and OVS-based OpenFlow switches for doing so. This work objective is to implement and design an application capable of perceiving the users QoE in order to redirect that information for the Internet Service Provider. With that valuable information it is possible to adjust the network rules, providing an higher overall QoS. However, the only considered rate-limiting approach is the OpenFlow Meter Table [58], discarding OVS capabilities regarding the QoS rules implementation domain.

Concluding, A. O. Adedayo and B. Twala (2017) [51] address both major QoS models, not taking into consideration the IoT real world on the testing methodology. These authors work utilises Ryu Controller REST API for being able to apply QoS rules in the OpenFlow OVS-based switches. Additionally, the majority of tests use Mininet [59] in order to emulate the virtualised network, not inspecting the impact multiple clients might have in the tested network. To terminate, this work does not address the concept of network slicing.

Hence, SDN4IoT aims to mitigate the above weaknesses, developing a complete solution considering traffic shaping, traffic dropping, prioritisation and traffic marking [3]. It performs several exhaustive tests in order to analyse the viability of SDN related QoS scenarios comprehended in the IoT world. Each IoT scenario is then represented by a slice where different applied rules and the use of distinct transport layer protocols are addressed.

Chapter 4

Testing and Obtained Results

In the previous chapter a detailed background was provided on how SDN4IoT was implemented, being now safe to say that it is ready to accommodate devices that make use of the aforementioned wireless technologies. Additionally, the two major QoS models were related with this gateway, illustrating on how they theoretically could be implemented in the solution context. Because it might be crucial to understand what are the advantages and drawbacks of SDN4IoT, this chapter presents the performance tests executed to validate it.

In light of this, the first section describes which routine and tools were used in testing preparation. Furthermore, section two discusses best effort testing and how it can relate with some network characteristics. Additionally, third section provides details on the results of raw traffic policing and shaping. Fourth section analyses the results of some IntServ QoS scenarios and application. At last, section five performs an analysis of the Diffserv QoS implementation on this work SDN environment.

4.1 Testing preparation

Since MQTT can have two flavours when it comes to the OSI model Transport layer, TCP and UDP were inspected and compared when used in the same scenario. Additionally, because there are several scenarios to consider, this chapter appropriately analyse two metrics. Those are bandwidth and the overall CPU percentage utilisation (when relevant to consider).

As mentioned on subsection [2.5.1.2](#), the addressed resource control mechanisms are Scheduling, Traffic Management, Prioritisation and Queuing. Prioritisation and Scheduling are only considered on a Multi-level approach. Additionally, all kinds of traffic management are addressed, allowing for admission control when implemented by OVS and OpenFlow. Concluding, as referred on subsection [3.3.2.2](#), queues are applied by OVS which takes advantage of the Linux Traffic Control utility for doing so. The type of the addressed queue mechanism is HTB, which contains TBF related shaping abilities.

It is important to state that the performance tests aim to analyse the implemented gateway SDN capabilities. Hence, all of them discarded the bottleneck that may be imposed by the wireless technologies information exchange and only consider drawbacks resultant of SDN4IoT softwarisation and virtualisation capabilities.

4.1.1 Testing automation and methodology

In order to automate the execution of administration features that WebGM provides, a python script was developed which purpose is to take advantage of both modules FCMan and SDN REST API. By doing so, it is possible to create slices, run containers and register the required IoT devices in an automated fashion every time a test is performed.

All clients ran in the Host machine. Because end devices (things) must be somehow represented, several SDWI-like linux virtual interfaces were created with a given IP and random MAC addresses. As a result of that, having in mind IoT characteristics, all the realised tests can be related with the implemented solution scalability potentialities.

With "virtual end devices", it is now possible to test its connectivity by binding the process to the intended network interface. However, in order to better analyse the solution performance, the injection of stress through traffic generation is needed. Iperf3 represents the perfect tool for doing so. It performs active measurements of the maximum achievable bandwidth on IP networks, with the support of tuning several client parameters and fetching the Server connection statistics [60]. The grounding of the whole testing module was Python, which took advantage of its language Iperf3 module. Due to this module lack of support for fetching the server statistics output (which is an actual feature in the original Linux tool), its source code was modified in order to be able to do it at a Python level. Additionally, with the purpose of giving the community a contribute, the changes were already submitted via pull request to the Github official repository [61], which is currently waiting to be accepted. It is clear that the usage of Iperf3 will always use a small portion of the machine CPU in order to generate traffic and perform measurements. However, after the observation of this phenomenon typical resources usage, the aforementioned Iperf3 related CPU utilisation portion is insignificant and can be discarded.

The remaining piece of the puzzle will reside on top of the gateway network architecture. Each client is in need of an Iperf3 server waiting to be connected and ready to report its statistics results. Hence, VM-Application Docker containers were running in order to accommodate them, resulting in traffic flowing through all the SDN architecture layers. Each server was accessible through the same IP address, but different ports.

Summarising, this Master Thesis investigates the following OpenFlow and OVS capabilities: i) Traffic dropping; ii) Traffic shaping; iii) Traffic scheduling; iv) Traffic prioritisation; v) Per-flow bandwidth management; vi) Traffic markers and classifiers.

4.1.2 Test map

In order to provide an overall perspective of the executed tests, a map was elaborated. Its objective is to relate all the performed tests with the addressed QoS Models and OVS/OpenFlow implementations.

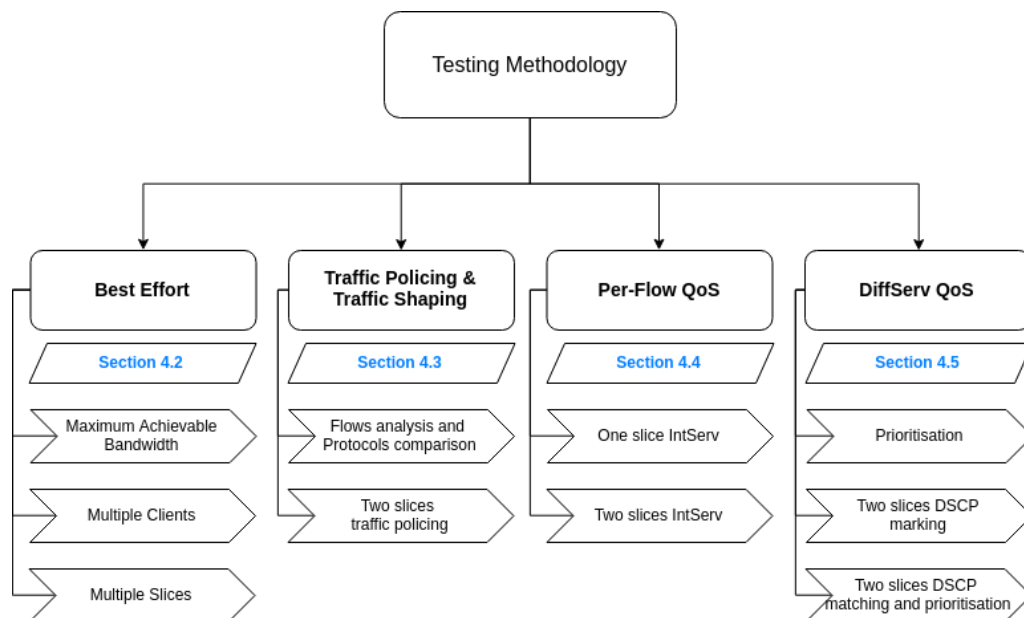


Figure 4.1: Performed tests overview.

4.2 Best Effort

It is important to understand the natural characteristics of the default SDN4IoT implementation. Hence, it is pertinent to test its functionalities without taking advantage of QoS Manager, for now. This section main purpose is to find what are the implemented solution limitations, showing potential drawbacks that some processing environments and network characteristics may provoke.

4.2.1 Maximum achievable bandwidth

One interesting result would be to find what is the maximum bit rate the current version of SDN4IoT can handle. This test was performed on two scenarios, TCP and UDP. Only one Iperf3 client was used in each case, which gradually increased the connection requested bandwidth. The test result is depicted in Figure 4.2.

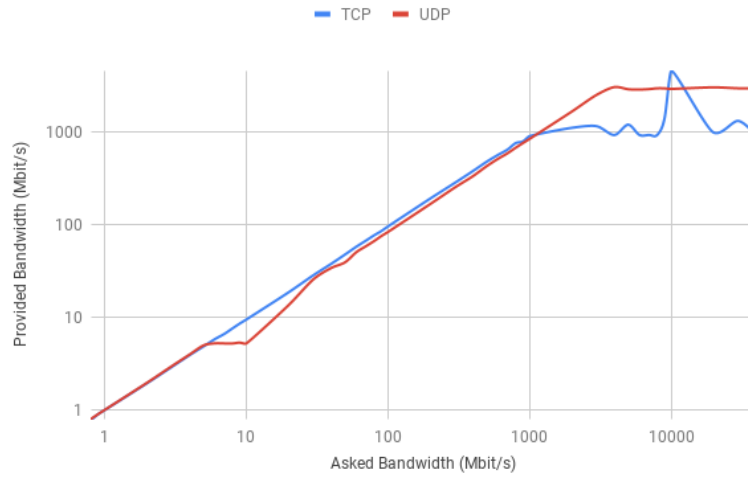


Figure 4.2: Maximum SDN4IoT achievable bit rate when utilising TCP and UDP protocols.

As it can be seen by the above chart, despite one noticeable peak of 4.6 Gbit/s, TCP stabilised roughly at 1 Gbit/s, while UDP found balance at approximately 3 Gbit/s. The observed chart reports that TCP has a stabilised best effort bit rate smaller than UDP. Since UDP has a nonexistent Acknowledge Packet (ACK) process, a continuous packet stream is permitted to exist and consequently it is able to achieve higher bit rates. On the other hand, TCP offers error correction and for that reason is not considered to be a real-time protocol. However, the proposed solution addresses two complex scenarios where several end devices are allowed to exist and that may serve a wide range of applications. Hence, it should be inspected how both protocols behave when traffic provided by different devices is simultaneously flowing through SDN4IoT.

4.2.2 Multiple clients

The solution is intended to deal with several client connections. Because of this, it is important to analyse the resultant impact of growing devices in the gateway networking environment. In order to do so, the same protocols comparison was made where each client was imposing a bit rate of 1 Mbit/s. It should be mentioned that only one slice was considered and the tests below were not performed simultaneously. Figure 4.3 shows the total bit rate of TCP and UDP protocols resulting from a growing client number.

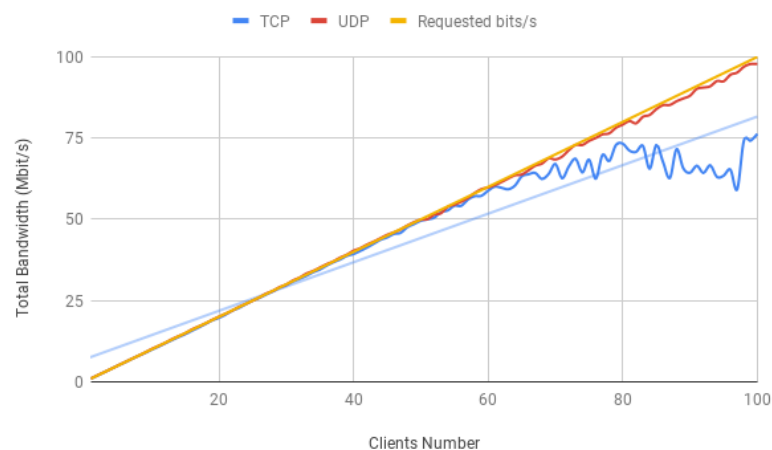


Figure 4.3: Total bit rate of TCP and UDP protocols resultant from a growing client number.

Figure 4.4 shows the clients average bit rate in the considered test.

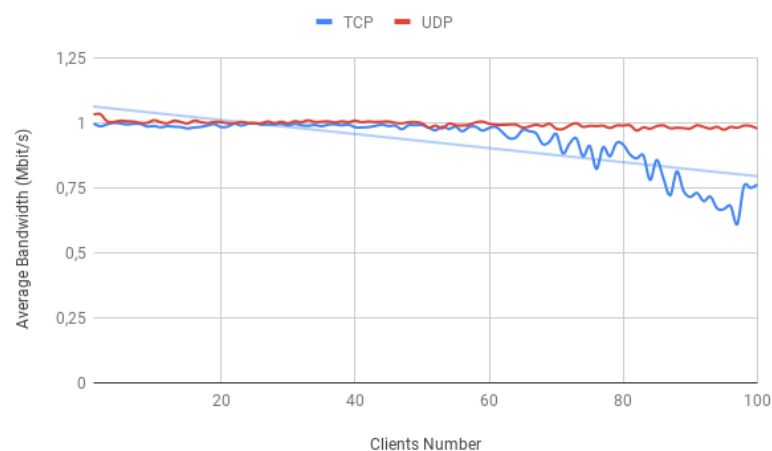


Figure 4.4: Average clients bit rate of TCP and UDP protocols resultant from a growing client number.

TCP clients suffer and cannot keep up with the requested bandwidth, which at roughly 70 clients it begins to deteriorate. Understanding the performance of the system resources is important and might reveal some limitations of an SDN implementation. Because usually software performance is directly limited by hardware, the Iperf3 CPU utilisation reports should be taken into account.

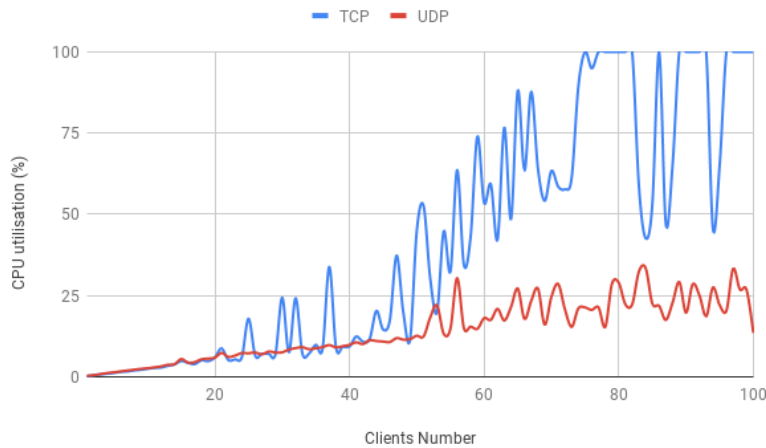


Figure 4.5: CPU utilisation when gradually increasing clients in TCP and UDP protocols.

Despite of not being a surprise, it is clear that the CPU usage is damaging the TCP scenario performance. As was anticipated, the bit rate degradation starts when the machine enters full processing power, causing the total clients bandwidth not to go further than 75 Mbit/s. Natural intuition would blame the protocol characteristics, which contrarily to UDP is connection oriented and focuses on reliability.

It is relevant to mention that Iperf3 enables the usage of a zero-copy flag which was not considered in this test. Since copying data is a relatively simple process, assigning this task to the CPU may mean a wasteful choice. Hence, this feature enables simpler system components to perform the data copying process while the CPU is in parallel dealing with others. This would allow a more efficient resources usage and consequently a considerable improved CPU performance [62]. It is clear that this feature usage would greatly improve the TCP scenario previous results, but that does not explain the noticeable difference between both protocols.

After a close inspection on both cases, it was noticed a considerable difference in their packet lengths. The used Iperf3 version sets UDP default send size to 8000 bytes, while TCP was being put at 2962 bytes. However, Wireshark [63] reported that the packets size in the communication process of UDP are not necessarily the same as the one that is set up in the application. Hence, due to the overall network Maximum Transmission Unit (MTU) value, which was 1500 bytes, datagrams were being segmented in the IP layer in portions of roughly the same size. This phenomenon results in the SDN environment being forced to deal with approximately two times bigger TCP packets than the UDP ones.

Now, TCP is a bytestream protocol, and consequently does not preserve the size of send operations from the Iperf3 application. However, clients are able to get more expected results by defining the TCP MSS (Maximum Segment Size) but that is not guaranteed either. Despite of that, and on a try-and-error basis, a relation was found when setting up the MSS value of 750 bytes when clients were sending 1 Megabit/s, which Wireshark reported as TCP packets size of 1542

bytes and is considered to be close enough of the UDP scenario. Figure 4.6 shows the clients total bandwidth on the new test conditions.

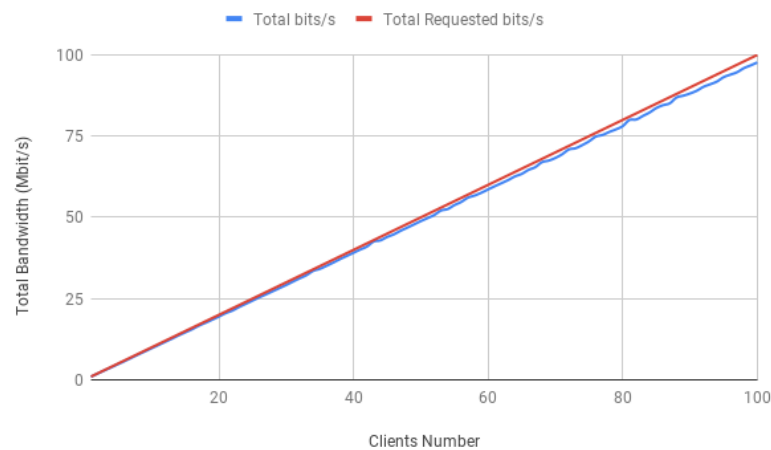


Figure 4.6: Clients total bandwidth on the Iperf3 non default TCP sized packets test.

For complementing purposes, Figure 4.6 depicts the clients average bandwidth.

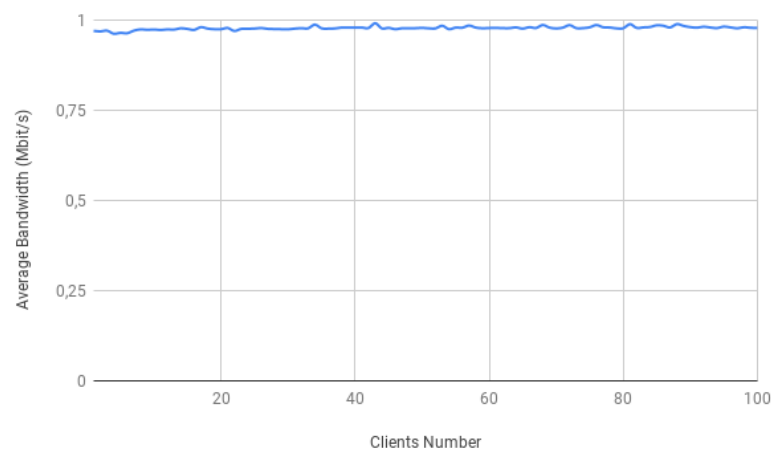


Figure 4.7: Clients average bandwidth on the Iperf3 non default TCP sized packets test.

It is safe to say that the difference in packet lengths was influencing the results. The TCP clients bit rate is now in compliance with the requested value, resembling the UDP related results that were achieved in Figure 4.3. Similarly, this test CPU utilisation should be in conformity with Figure 4.5 UDP values. The test CPU utilisation related values can be analysed in Figure 4.8.

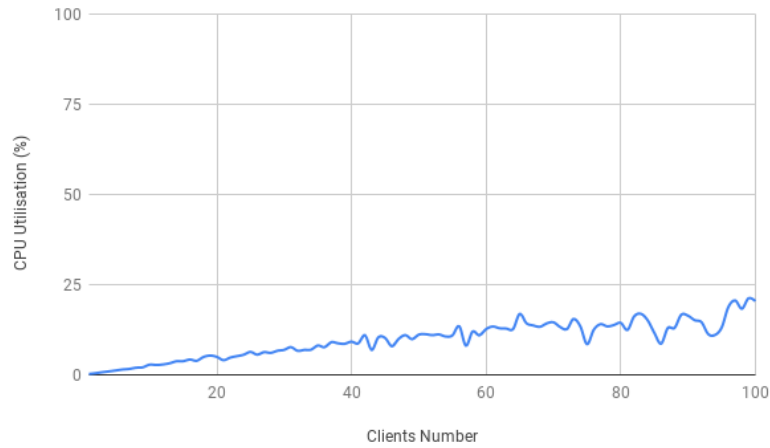


Figure 4.8: CPU utilisation when gradually increasing clients in TCP non default sized packet.

4.2.3 Multiple Slices

Since SDN4IoT addresses multiple use cases, it is relevant to test the good functioning of two simultaneously running slices. This testing scenario addresses both network segments with 50 clients each, which are instructed to inject 1 Mbit/s per interface with Iperf3 default packet sizes in the gateway. To better illustrate and differentiate each slice, this test also considers the previously mentioned protocols where Slice A and Slice B are using TCP and UDP, respectively.

Figure 4.9 shows a comparison of both Slices achieved bandwidth.



Figure 4.9: Comparison of two Slices total bandwidth.

In conformity, Figure 4.10 illustrates each slice clients average bandwidth.

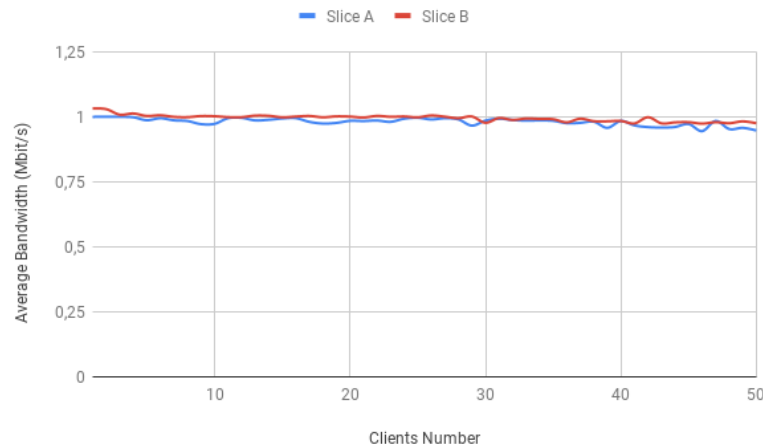


Figure 4.10: Clients average bandwidth on the two Slices test.

As expected, the implemented solution is able to handle several end devices, even if their destination slices happen to be different. Because this work proposed solution only addresses two use cases, tests will consequently be limited to two slices only. Despite of the commitment to perform exhaustive tests, it is crucial to understand that the main objective is to find the QoS implementation that would better fit the Industry 4.0 and Digital Farming environments.

4.3 Traffic Shaping and Traffic Dropping

This section was designed to go through the two forms of QoS that OVS can provide: Egress Traffic Shaping and Ingress Traffic Dropping. Hence, it is important to mention that besides the default flow entries applied by Ryu Controller, no flows entries are applied by QoS Manager.

As previously explained, policing can only be implemented on interface "eth1" of VM-Application (Figure 3.10), mainly because it represents the only interface which deals with traffic flows entering the system from the outside. On the other hand, each created slice port represents the way out of the whole OVS system, what enables their usage for outgoing traffic shaping. Because of that, traffic policing is inevitably applied to all the gateway created slices, affecting them in a single-level fashion. Contrarily, shaping is applied to each slice independently, where rules are set and are only inherent to one single virtualised network segment.

4.3.1 Flows analysis and Protocols comparison

Due to the characteristics of each QoS implementation, similar generated flows are expected to produce different results even when the same maximum bit rate is applied. For instance, if it is chosen to generate a flow of 20 Mbit/s, with a limit of 10 Mbit/s for each appliance, potential dissimilarities will occur.

4.3.1.1 Traffic Policing flows

Figure 4.11 depicts those differences and also considers the utilisation of TCP and UDP protocols when flows are being limited by traffic policing.

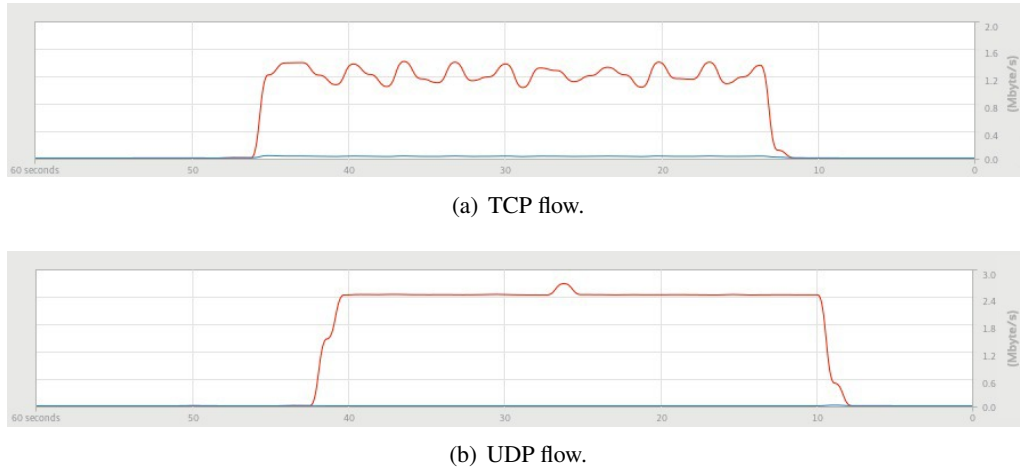


Figure 4.11: Traffic policing TCP and UDP flows comparison.

It is easy to perceive that flows have different characteristics. The first thing to notice is Figure 4.11(a) flow irregularity. In order to understand it, it must be remembered that subsection 3.3.3.1 mentions that possible settings for the policing implementation are rate (which sets a maximum bit rate) and burst (which allows the algorithm to be more forgiving). Additionally, it should be noticed that this test was performed with the default policing burst value (0), which actually provokes OVS to set it at 1 Mbit. Moreover, the same subsection refers that this type of OVS QoS takes advantage of a bucket approach which might be crucial for the good functioning of the TCP protocol that due to its characteristics reacts aggressively to dropped packets [64].

Hence, the chart clearly shows that the flow mean value was approximately 1.2 Mbyte/s, which actually translates in 9.6 Mbit/s and can be considered close enough to the defined policing bit rate limit. Furthermore, the flow variation can be explained by the functioning of the token bucket, which is constantly being refilled and drained out when reaches full capacity. In order to support this statement, it is noticeable that the peak-to-peak value of the sinusoidally shaped flow irregularity is slightly bigger than 0.1 Mbyte/s (0.8 Mbit/s) and can be related with the size of the OVS defined burst (token) size (1Mbit).

On the other side, Figure 4.11(b) reports a client-sided UDP flow of nearly 20 Mbit/s. This may induce to misunderstandings regarding traffic dropping effectiveness on this protocol flows. However, due to the lack of error checking the client is simply not realising that packets are being dropped and data is being lost.

4.3.1.2 Traffic Shaping flows

The same test was made on a traffic shaping limited slice. In order to guarantee testing homogeneity, both maximum bandwidth limit and client requested bit rate had the same values as previously defined. Furthermore, because OpenFlow only had default entries, all flows were using an OVS QoS implementation which only contained a default queue. The results are shown in Figure 4.12.

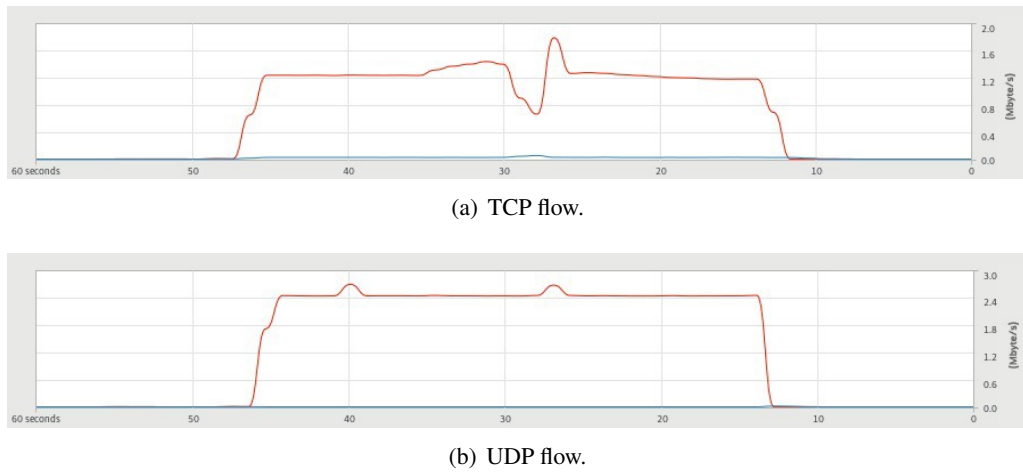


Figure 4.12: Traffic shaping TCP and UDP flows comparison.

Once again, it is clear that dissimilarities in both flows do exist. Figure 4.12(a) undoubtedly depicts the longest flow of the ones considered so far, what is caused by the protocol re-transmissions. Furthermore, due to the properties of traffic shaping, which tries to assure that traffic dropping does not occur, excessive packets are being queued in order to be processed later on. This mainly happens when the buffer reaches full capacity, causing the queued packets to be only processed when it happens to be available for doing so. However, despite of traffic shaping characteristics, there are situations that the buffer is being flooded in such a manner it has not enough bandwidth to place packets in its queue, what consequently leads to their loss [51].

Additionally, Figure 4.12(b) shows some effects of traffic shaping, which prevents the discarding of a certain amount of packets. Nevertheless, this flow is very similar with the one of Figure 4.11(b) which can be explained by both cases being taking advantage of UDP. As previously referred, this protocol does not attempt to perform error recovery.

Hence, in order to assure the good functioning of both rate limiting options, this scenario server perspective charts should be analysed. Those can be found in Figure 4.13.



Figure 4.13: Server perspective UDP flows on traffic policing and traffic shaping.

It is obvious that, on the server perspective, both flows are respecting the limits that are imposed by the QoS implementations. Despite an irregular traffic policing flow, which can be explained by its limiting characteristics (referred in subsection 3.3.3.1), it is possible to acknowledge both solutions have the expected behaviour.

4.3.2 Two slices traffic policing

Policing can only be implemented in a single-level approach, what results in necessarily affect all slices when the solution is taking advantage of it. Hence, this test is intended to illustrate that. In light of this, clients number was gradually increased in two simultaneously running slices whose combined bit rate was limited by 50 Mbit/s. All clients were attempting to poke the server with 1 Mbit/s, using TCP. Additionally, the defined policing burst value will be set as 10 Mbit/s. The rate limiting efficiency is depicted in Figure 4.14.



Figure 4.14: Two slices policing implementation total bandwidth.

The slices aggregated bandwidth is observable in Figure 4.15.

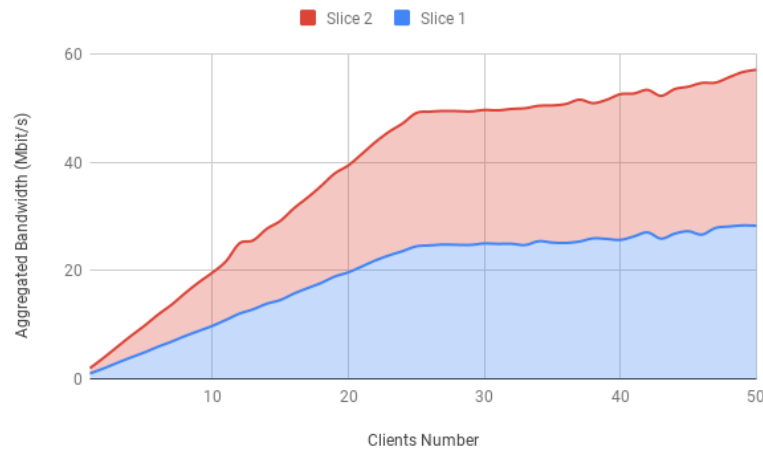


Figure 4.15: Two slices policing implementation aggregated bandwidth.

As it can be seen, both slices are being affected when their bit rates reach 25 Mbit/s. This can be explained by their aggregated bandwidth of 50 Mbit/s at this time, which is the rate limit that this Traffic Policing implementation imposed. OVS documentation states that defining a burst size to be a sizeable fraction of the policing maximum limit rate enables the flow to more easily achieve the full rate. However, if the defined burst size is set to be a large value, the client will actually notice an average rate slightly higher than the specified maximum one [58]. Hence, because this test defined burst value is around 20% of the maximum rate limit, the slices aggregated bandwidth is moderately overcoming it.

OVS documentation claims traffic shaping to be more accurate and effective than policing, explaining it by the difference in implementation complexities [58]. Since this kind of QoS can only be applied within each slice, which is running several applications, it would never behave similarly to the previous case test. Hence, rules need to be applied in a multi-level fashion and are slice independent, what results in different queues managing each one of them. Because of that, it is possible to set distinct bit rate limits for each slice and even services within them. This is demonstrated in next section.

4.4 Per-Flow QoS testing

The document already went through network bandwidth reservation rules in the last section. Additionally, it is already known that QoS policing actuates at an interface level, dropping excess traffic that crosses into it and not taking advantage of QoS queues for doing so. On the other hand, traffic shaping is applied on a OVS port, what allows for multiple queues creation and flow control on its OVS switch side. For that reason, from now on, only output traffic shaping is considered.

As mentioned in section 3.3.3.2, this enables the use of a set of IntServ rules to be applied, which in our solution context supports Classification and Queueing.

4.4.1 One slice IntServ

The following test case addresses IntServ classification within a slice for previously known services. Hence, two Queues exist with different defined maximum bit rates. By taking advantage of OpenFlow, multiple flow entries are added to each slice dedicated switch, defining the queue each flow should use based on its source IP address. After that, Queueing forwards packets according to those flow entries that have been granted.

In light of this, IntServ classification resided in determining whether the last octet of the source IP address would be an even decimal number or not. Hence, even numbers would be redirected to Queue A. It should be noticed that the Queue A related flow entry has bigger priority than the default one, hence, the packet is matched against the table and only the highest priority flow entry that matches it is selected [65]. Because flow entries did not specified rules for the remaining IP addresses, odd numbers would simply use the switch QoS default flow, which consequently redirects their packets to the default queue.

Detailed information can be inspected in Table 4.1.

Table 4.1: One Slice IntServ implementation details.

Queue ID (Name)	Defined maximum bandwidth	OpenFlow Match: Destination IP last octet	OpenFlow Priority
0 (Default)	60 Mbit/s	Odd	1
1 (A)	30 Mbit/s	Even	2

For complementing purposes, the destination application container was running 100 Iperf3 servers, which would be connected by one client each. All clients were using the TCP protocol, injecting 1 Mbit/s in the slice network environment. Results of each queue total achieved bandwidth are shown in Figure 4.16.

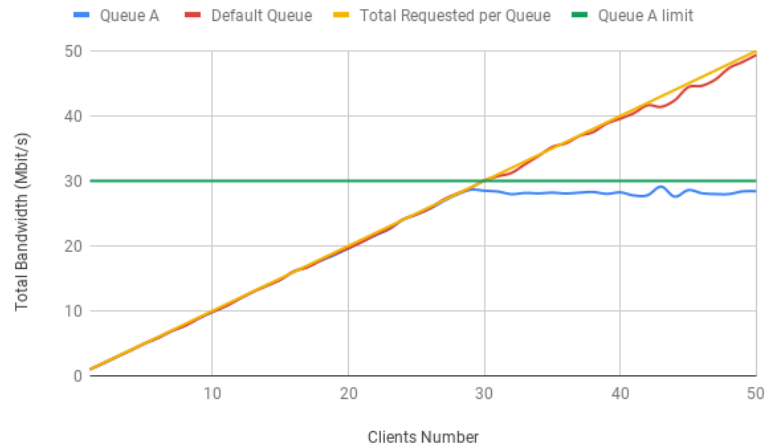


Figure 4.16: One slice IntServ implementation queues total bandwidth.

Additionally, the queues average bit rate can be found in Figure 4.17.

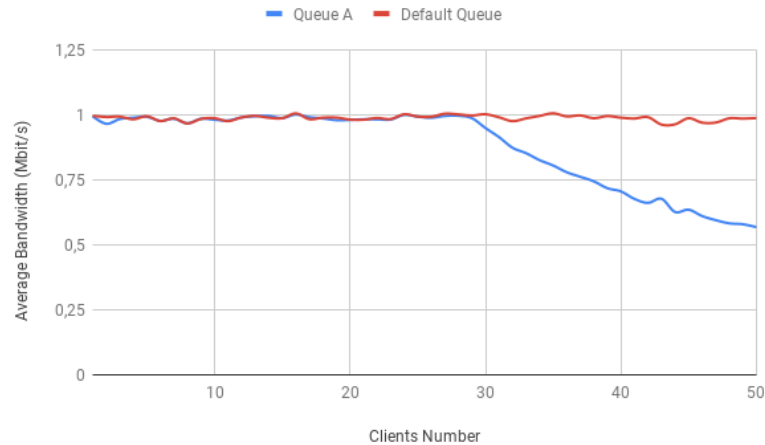


Figure 4.17: Queues average bit rate on the one slice IntServ test.

4.4.2 Two slices IntServ

As already mentioned, since this Master Thesis addresses two use cases, it is important to guarantee the good functioning of the implemented scenarios, always having two network slices in mind. Hence, a test was made with this characteristics which additional scope was not to take advantage of the default queue. Because of this, all clients required to possess an IntServ classification (flow entry) that would forward them based on their destination IP port. This resulted in a massive addition of information in the switch OpenFlow table. The details of the implemented OVS queues can be analysed on Table 4.2.

Table 4.2: Two Slices IntServ implementation details.

Slice ID	Queue ID (Name)	Defined maximum bandwidth	Openflow Match: Destination Port
1	1 (A)	20 Mbit/s	Odd
1	2 (B)	10 Mbit/s	Even
2	1 (A)	15 Mbit/s	Odd
2	2 (B)	5 Mbit/s	Even

Clients were behaving in the same way of the previous test, using TCP and injecting 1 Mbit/s each. With that in mind, Slice 1 is expected not to go beyond 30 Mbit/s, while Slice 2 should stabilise around 20 Mbit/s. Each slice aggregated bandwidth chart that resulted from this test is shown in Figure 4.18 and Figure 4.19, respectively.

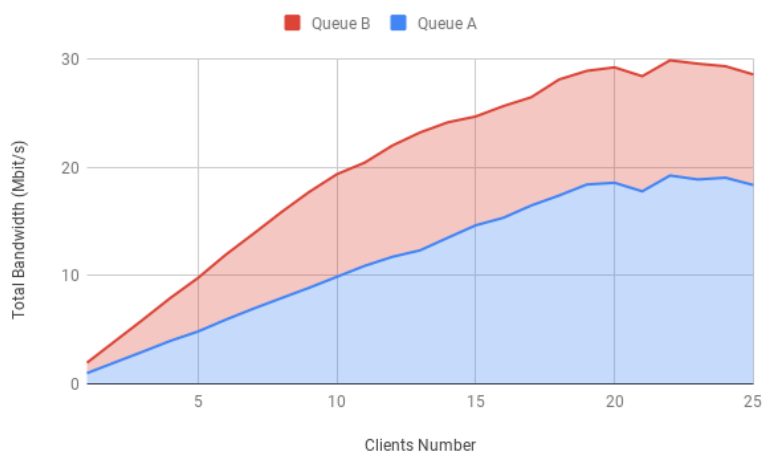


Figure 4.18: Two slices IntServ - Slice 1 aggregated bandwidth.

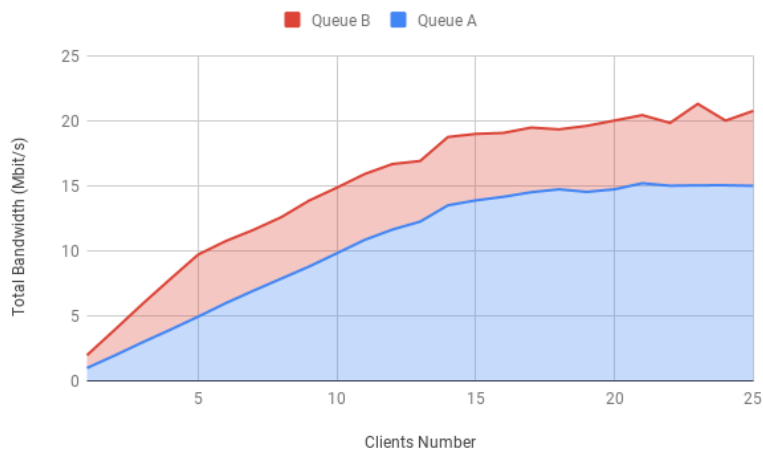


Figure 4.19: Two slices IntServ - Slice 2 aggregated bandwidth.

Additionally, the comparison of both slices achieved bandwidth is depicted in Figure 4.20.



Figure 4.20: Two slices IntServ - Total achieved bandwidth comparison.

It can be noticed that the sum of the theoretical limits both Queues imposed on the respective slices were being respected. However, the slices aggregated bandwidth charts do not clearly show whether each Queue was having a stable behaviour. This is clarified on both Figures 4.21 and 4.22.

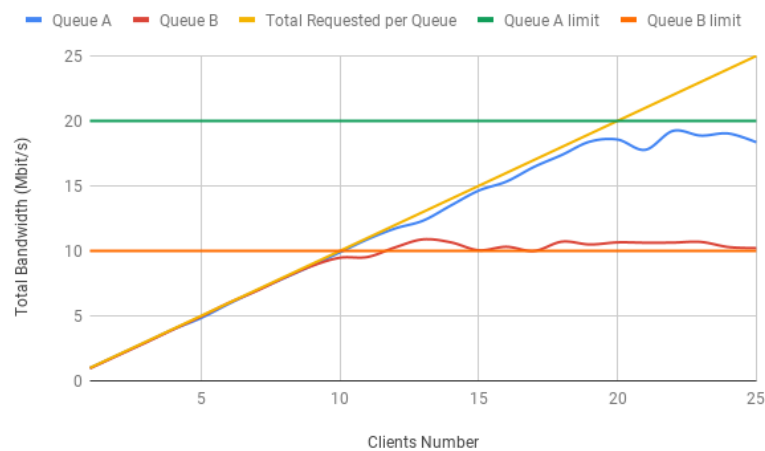


Figure 4.21: Two slices IntServ - Slice 1 Queues bandwidth.

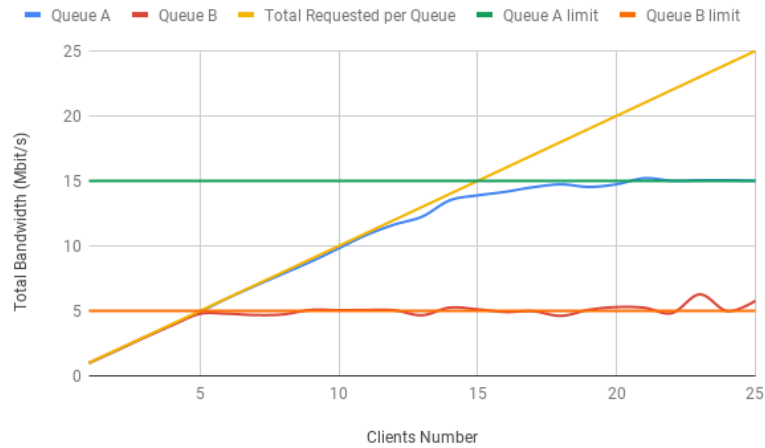


Figure 4.22: Two slices IntServ - Slice 2 Queues bandwidth.

Figure 4.23 comprehensively illustrates this test each queue average bit rate.

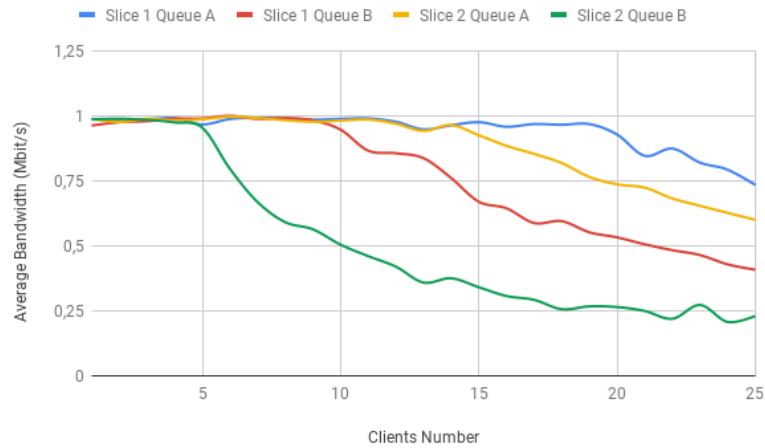


Figure 4.23: Two slices IntServ - Average clients bandwidth per Queue.

SDN4IoT's IntServ capabilities are being well implemented and having the expected behaviour. However, for doing so, each slice switch is being overloaded with flow entries in order to correctly classify and schedule their respective traffic. Hence, Per-Flow processing and associated state can be the cause of high resource consumption on the virtualised network nodes. In light of this, it is possible to say that despite being a good model to deal with flows in a small network context, IntServ does not represent a scalable solution.

4.5 DiffServ QoS Testing

Contrarily to IntServ, Diffserv seeks to use simple methods in order to categorise traffic into distinct classes. It is already known that due to original implementation issues, the solution North-bound core OVS switch is not able to apply the tool QoS rules, what forces them to be implemented in each slice switch ports instead and creating the so called DiffServ domains (subsection 3.3.1). However, OpenFlow related parameters can still be applied in "br-int", what allows QoS Manager to take advantage of it in order to mark DSCP values. Moreover, since IP headers contain the DSCP bit pattern, which is positioned in the ToS octet, it is possible to define OpenFlow rules that mark and redirect traffic flows based on their packets DSCP value. Concluding, the services differentiation also helps OVS to define traffic prioritisation within a given slice, which causes important queues to receive all the excess bandwidth that it can use before a less prioritised queue receives any.

It should be noticed that QoS Manager does not perform prearranged rules on what DSCP values should be given priority to. Instead, it merely provides an OVS and OpenFlow level API to allow classification and differentiate services within each slice domain. Hence, in case of prioritisation requirements, the relation between them and DSCP values have to be set by the gateway administrator.

This testing section is designed to go through the DiffServ possible applications in the SDN4IoT context. Furthermore, all tests respected the commonly utilised DSCP values list, which is described on RFC 2475 [66].

4.5.1 Prioritisation

Since the DiffServ implementation took advantage of traffic shaping, two levels of HTB queues were required in order to perform this test. The first level queue specified the slice maximum allowed bit rate, which inevitably affected all its devices. On the other hand, level two queues permitted to perform services differentiation regarding their bit rate and traffic prioritisation, based on packets DSCP values. Therefore, the core queue firstly limits the slice global traffic before redirecting it to the levels below, where it is handled based on its assigned queue priority. In order to do so, one Docker container was running on VM-Application with two distinct interface, where two Iperf3 servers were bound and waiting to be connected by clients. Additionally, the core switch performed the DSCP marking based on the packet source IP address. The implemented queues are depicted in Figure 4.24.

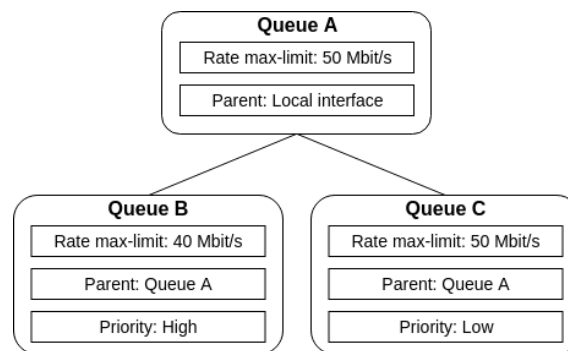


Figure 4.24: Prioritisation test queues implementation.

One hundred clients were gradually being inserted in the test context, adding one unit to each Queue per iteration. Because of this, despite of the aggregated flow of 100 Mbit/s in the last iteration, Queue A should be able to force a total bit rate of its maximum specified in both slices combined bandwidth. In addition, the higher priority in Queue B should represent a traffic bit rate close to the lower bounds of its max-limit, while Queue C, despite of a bigger valued limit, will only handle the remaining portion of the slice total. Results are shown in Figures 4.25 and 4.26.

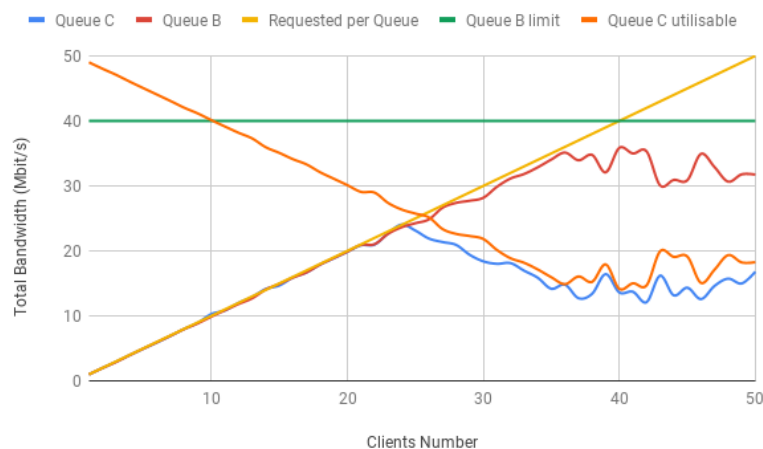


Figure 4.25: Prioritisation test queues limit and total bandwidth.

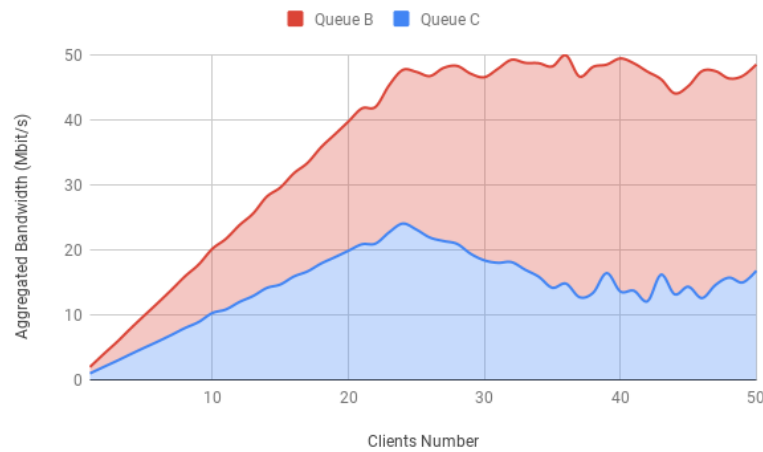


Figure 4.26: Prioritisation test distribution of queues bandwidth.

Keeping the above figures in mind, charts comprehensibly depict Queue A related traffic being prioritised over Queue B. Additionally, the prioritised queue is successfully implementing its defined upper bounds, while the other one is limited by the slice total left over bit rate.

4.5.2 Two slices DSCP marking

Due to the natural characteristics of DiffServ, its implementation can also comprehend some functionalities that IntServ can provide. This is because this model fits networks of any size. While the previously mentioned QoS model inspects and processes each flow in the source-destination path, DiffServ only analyses the pre-defined traffic classes on each domain node. Hence, this model is able to deal with flows differentiation if it is chosen to previously classify them based on some matching parameters. In light of this, this test is meant to show the DiffServ perspective of the one mentioned on subsection 4.4.2, trying to demonstrate that the same functionalities were being implemented. It should be mentioned that this test also took advantage of two levels of HTB queues. However, the root queue was built in a way that would never affect the below levels and because of that, is not referenced in this implementation. The defined Queue rules are shown on Table 4.3.

Table 4.3: Two Slices IntServ implementation details.

Slice ID	Queue ID (Name)	Defined maximum bandwidth	Match: DSCP value (Decimal)
1	1 (A)	15 Mbit/s	10
1	2 (B)	5 Mbit/s	18
2	1 (A)	20 Mbit/s	10
2	2 (B)	10 Mbit/s	18

Hence, the core switch classified traffic, applying a DSCP value based on a match of the packets transport layer protocol. Traffic that took advantage of TCP was marked with 10, while

in traffic that was using UDP was applied the value of 18 instead. Additionally, each slice switch had QoS OVS rules, specifying the queue rules to where traffic should be redirected based on OpenFlow entries that would match the known DSCP values. This means that after the core switch, the following path OVS elements did not have to deal with flows anymore, simply inspecting the packets ToS octet in order to apply QoS. Both slices aggregated bandwidth results are shown in Figures 4.27 and 4.28.

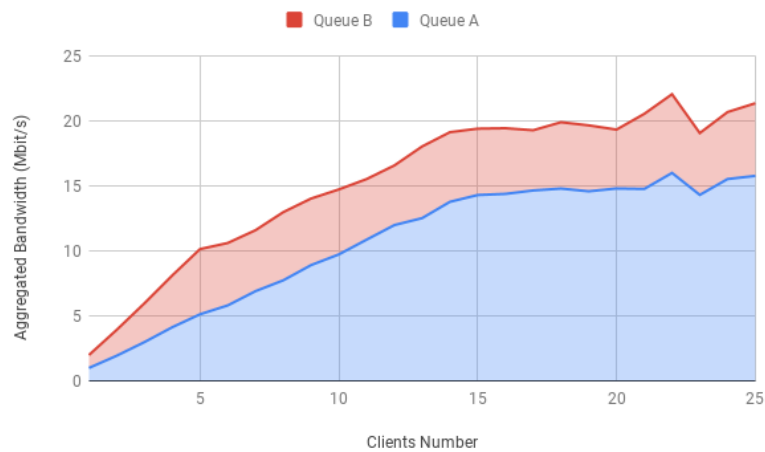


Figure 4.27: Two slices DSCP marking - Slice 1 aggregated bandwidth.

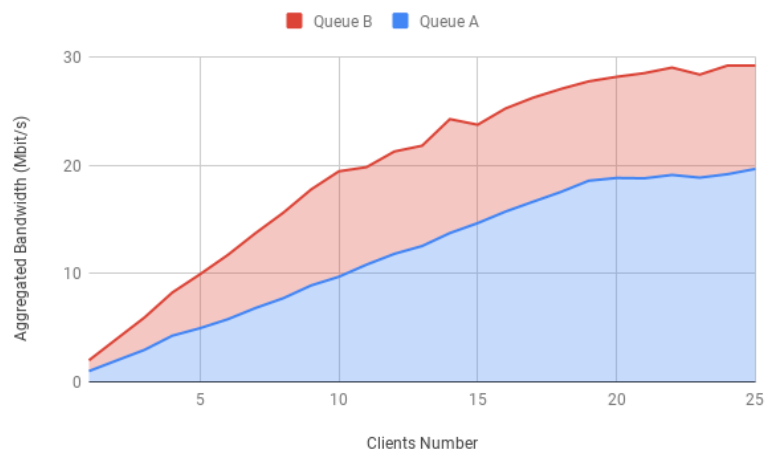


Figure 4.28: Two slices DSCP marking - Slice 2 aggregated bandwidth.

The total achieved bandwidth in each slice conditions is illustrated in Figure 4.29.



Figure 4.29: Two slices DSCP marking - Total achieved bandwidth in each slice.

Similarly to the test on subsection 4.4.2, the above charts prove the good functioning of each slice limiting capabilities. However, it does not clearly shows DSCP based differentiation which allegedly should respect each matching Queue restrictions. This is depicted in Figure 4.30 (Slice 1) and Figure 4.31 (Slice 2).

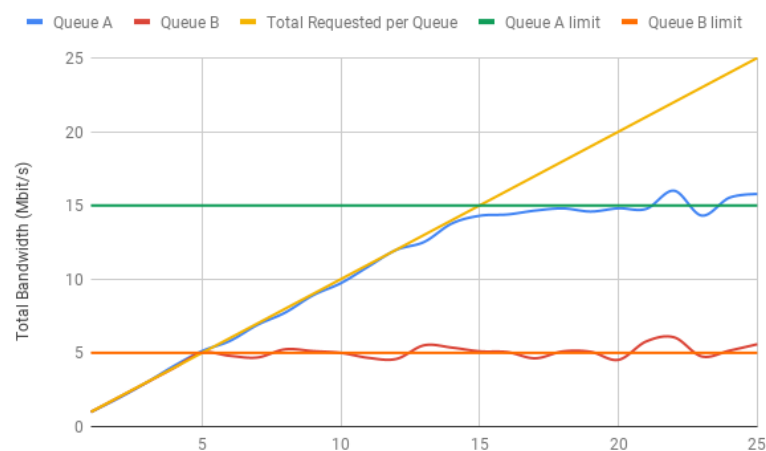


Figure 4.30: Two slices DSCP marking - Slice 1 Queues bandwidth.

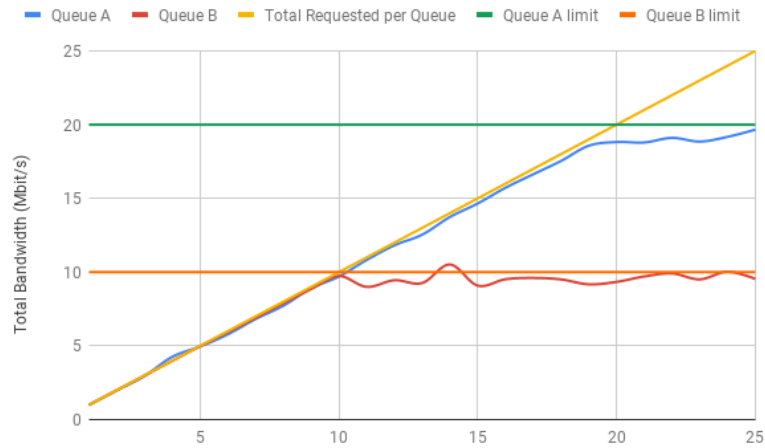


Figure 4.31: Two slices DSCP marking - Slice 2 Queues bandwidth.

The resulting average client bit rate values are available in Figure 4.32.

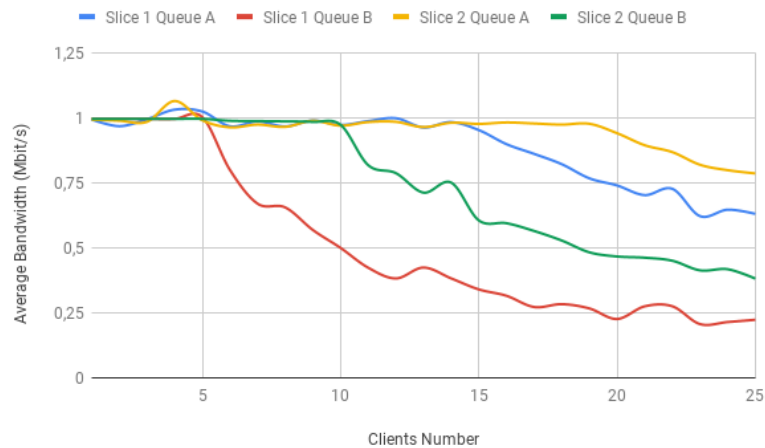


Figure 4.32: Two slices DSCP marking - Average clients bandwidth per Queue.

As can be noticed, charts demonstrate that traffic differentiation was well implemented. Each queue is forcing its assigned packets to obey the bit rate is being imposed by its max-limit value. This proves that the previously mentioned implementation, where similar results were achieved, can also be executed taking advantage of DSCP marking instead.

4.5.3 Two slices DSCP matching with prioritisation

As previously mentioned, DiffServ also deals with prioritisation. Because of that, this test scope was to obtain a mixed implementation of DSCP matching and prioritisation tests. Hence, OpenFlow similarly defined rules based on previously known DSCP values. However, traffic

with an unknown ToS octet would be redirected to the default queue, which was less prioritised relatively to others. It should be noticed that for diversity purposes, this test did not take advantage of the core root in order to perform DSCP marking. Instead, the packet ToS byte was established on the client side. Additionally, it is important to refer that despite this action, if it was chosen to still apply rules on the core switch, the latter would prevail. This guarantees a good QoS Manager network control, even if the client would try to provide himself an higher level of QoS.

As a consequence, six level two OVS queues were defined, which were running under their respective core queues in each slice port. Slice 1 queues hierarchy and applied rules can be seen in Figure 4.33.

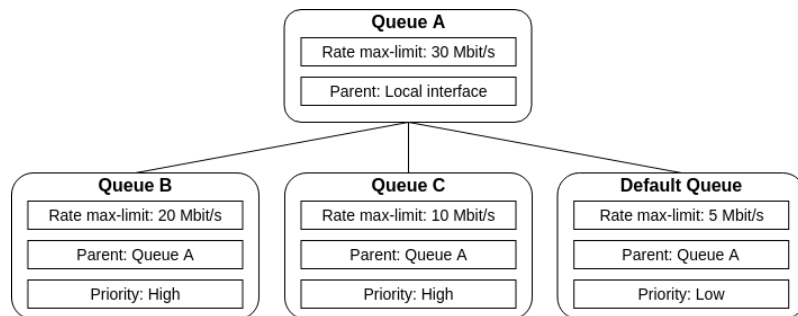


Figure 4.33: Two slices DSCP matching with prioritisation - Slice 1 queues implementation.

As expected, both slices were running simultaneously. Hence, the Slice 2 queues implementation details should also be detailed. Those are in Figure 4.34.

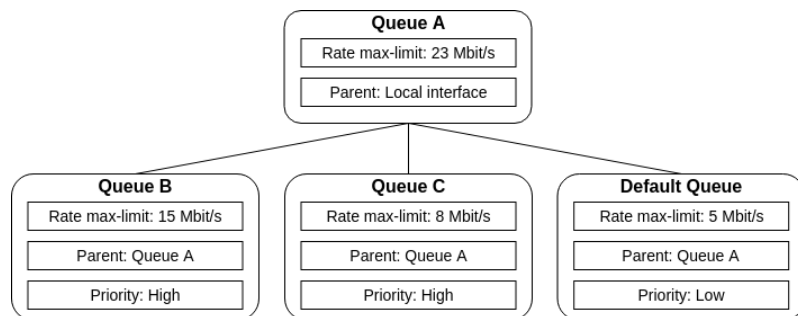


Figure 4.34: Two slices DSCP matching with prioritisation - Slice 2 queues implementation.

The test was designed in a way where 30 Iperf3 clients were using TCP and introducing 1 Mbit/s each in a single Queue. It is clear that in this way, the core Queues will be rapidly flooded and would have to prioritise data at the same time the packets ToS byte is inspected. Additionally, it should be noticed that in both cases the higher priority queues limit perfectly match the core queue upper bound bit rate, what at a given point should result in unknown DSCP valued packets to have no chance for their transmission. In a similar manner of the previous tests, each slice total bit rate are illustrated in Figure 4.35.



Figure 4.35: Two slices DSCP matching with prioritisation - Both slices total bandwidth.

Both slices total bandwidth is limited to its respective HTB core queue defined maximum bit rate value. In order to show the DSCP packets matching and prioritisation correct functioning all level two queues bit rate should also be inspected. This is shown in Figures 4.36 and 4.37.

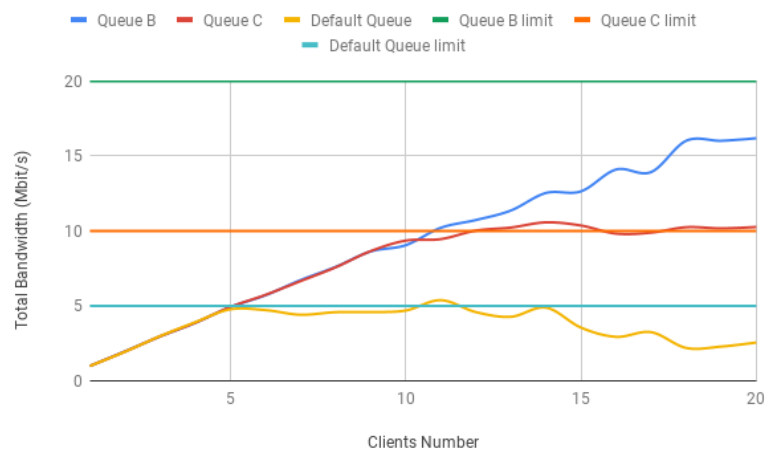


Figure 4.36: Two slices DSCP matching with prioritisation - Slice 1 total bandwidth.

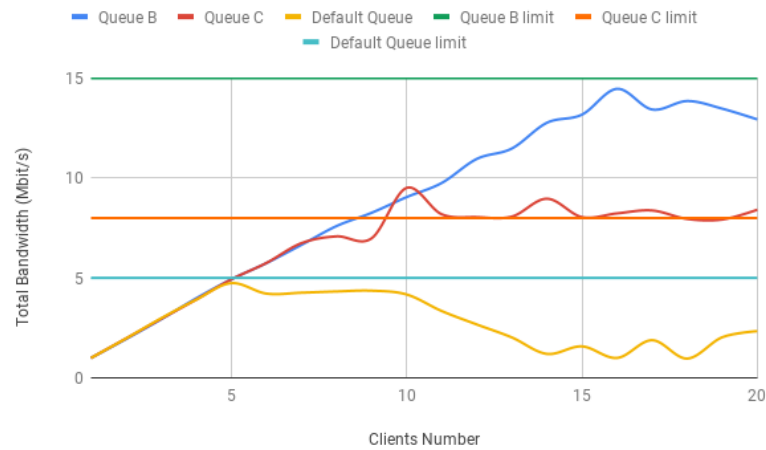


Figure 4.37: Two slices DSCP matching with prioritisation - Slice 2 total bandwidth.

It is clear that each queue bit rate is being respected and particularly each slice B and C queues are being prioritised. Furthermore, similarly to section 4.5.1, the default queue is limited to the residual bandwidth that prioritised queues choose not to utilise. This proves DiffServ to be a scalable solution that allows OpenFlow to match differentiated services packets, prioritising them based on each OVS switch QoS defined values.

Chapter 5

Conclusions and Future Work

After the performance results of the proposed work, this chapter starts by presenting concluding remarks considering the obtained results, taking the chance to analyse the addressed QoS scenarios and propose the ideal model attending each use case characteristics. Additionally, the chapter addresses new objectives towards future developments that should be taken into account concerning SDN4IoT.

5.1 Conclusions

IoT is considered to be an emerging world where millions of devices are connected every-day. Additionally, that enormous quantity of devices is destined to perform different tasks and can actually fetch data for a variety of purposes. This technology generality was proved to be compensated by SDN, that through different applications was able to handle devices differences. Similarly, due to their natural characteristics, Digital Farming and Industry 4.0 represent complex use cases where several types of sensors simultaneously coexist. It was concluded that in compliance with an ideal scenario, those "things" require different resources allocation in order to take the most of the implemented virtualised network. Hence, that process should be implemented having scalability in mind.

SDN has proved to be an extremely adaptable technology that comes to facilitate several problems resulting from the use of traditional networks. *SOFTWAY4IoT* is a great example of those capabilities and it was built in a way that allows for modules reinvention and insertion. Because of that, it was possible to renovate SDWI and provide connection to several end devices via the proposed wireless technologies without taking advantage of SDR. Furthermore, the software-defined gateway even provided flexibility on how the received data would be handled, allowing this action to be done adopting either a fog computing or cloud computing approach. Hence, applications are allowed to choose the option that will better attend the needs of each device.

A new module was then developed and inserted in the *SOFTWAY4IoT* context. QoS Manager resides on the solution Northbound Interface, which is the API that allows applications to orchestrate the data plane to perform several tasks. This module respected the principles under

which the gateway was originally built. Because of that, QoS Manager was developed in a way that enables it to be independent from the other gateway components, allowing it to function in any SOFTWAY4IoT version. Its main purpose is to provide different connectivity environments that distinct devices in the same slice may demand. Hence, due to the core tools that the majority of SDN related tools take advantage of, this module is able to implement the main QoS models. OVS and OpenFlow are able to join forces in order to perform per-flow and differentiated services related tasks.

Regarding the performed tests, the implemented solution has proved to well behave on several QoS models implementations. Best-effort test reported that when utilising the Iperf3 tool default settings, the maximum stabilised gateway bandwidth can be different if it is chosen to use different transport layer protocols. TCP reported a bandwidth around 1 Gbit/s while UDP stabilised on 3 Gbit/s. Furthermore, the OVS bit rate limitations options were analysed, taking the chance to perform a detailed analysis on the resulting flows characteristics. The IntServ model reserves resources based on flows. This is suitable for real-time scenarios on end-to-end data paths, but mainly due to the high resources consumption, this model might not be appropriate for situations where an elevated number of devices is present in the network. It is important to acknowledge the generality of the addressed use cases. Due to the Industry world nature, distinct types of sensors and consequently services will be running on smart factories of different Industry sectors (i.e. textile industry and food industry). Similarly, Digital Farming is dependant of the considered farming type. A small example would be that mushroom cultivation has not the same requirements as viticulture. Hence, the approach of trying to define a QoS implementation is dependant of the use case application that is being considered. However, it might be interesting to specify what QoS Model would be suitable to use. In light of that, DiffServ is the model that better fits both use cases characteristics. Because Industry 4.0 and Digital Farming rely on IoT, scalability is a key point to have in mind when structuring their networks. As previously stated, there are complexities inherent to the heterogeneity of devices that both scenarios might require, being necessary to take a look on their roles as well. Hence, the DSCP traffic classification allied with the possibility of prioritisation, allows for a services differentiation level that is not possible to obtain in the IntServ model.

5.2 Future Work

Because the used SOFTWAY4IoT is still in the pilot version, several limitations were found throughout its implementation and utilisation. It is clear that the proposed gateway takes advantage of emerging technologies that are being improved everyday. However, some further work could improve the inter-module communication and consequently the robustness of the whole system.

It is necessary to acknowledge some limitations regarding the renovated SDWI module too. It is understandable that the gateway aims to solve problems regarding the multiple wireless technologies devices accessing it. However, when applied to a considered use case production

scenario, the implemented module might become a bottleneck. This happens mainly due to restrictions related with each technology physical interfaces, whose available bit rate might not be enough when considering the high number of connected devices in a real scenario.

Despite the good functioning of QoS Manager in the current *SOFTWAY4IoT* version, there are still things to be done that would allow it to support more functionalities. As previously mentioned, mainly due to the Northbound API architecture, the different rate limiting approaches can only be applied either at a single (policing) or multi-level (shaping). This occurs because the current solution chooses to use patches in order to connect to each slice created OVS switch. Hence, a relevant choice would be to do it by a newly created either logical or physical interface, allowing for the previously mentioned problem mitigation. This architecture change would enable an even greater flexibility regarding QoS appliances. A small example would be the allowance of slices prioritisation.

Another thing to consider is the possibility of using OpenFlow meter tables in the QoS implementation. Contrasting to OVS queues, which redirect packets at a either a minimum or maximum specified bit rate, meters enable to perform a given action based on live packets bandwidth inspection.

In order to conclude, as it could be seen in the testing section, the SDN performance can be affected by the host machine CPU utilisation. It is a known issue that software implementations may suffer with the hardware that they use to run on top. However, it might be interesting to analyse how different QoS implementations and stress scenarios can affect a SDN environment, taking the chance to compare the results on a variety of machines.

Appendix A

Source Code

A.1 Tests Container Dockerfile

```
FROM debian

# Install app dependencies
RUN apt-get update && apt-get install -y \
    python3 \
    python3-pip \
    iperf3

RUN pip3 install iperf3
RUN apt-get update -y && apt-get upgrade -y

# Bundle app source
COPY iperf.py /src/iperf.py

# Change working directory
WORKDIR /src

# Exposes Iperf3 server port numbers
EXPOSE 5900 5901 5902 5903 5904 5905 5906 5907 5908 5909 \
    5910 5911 5912 5913 5914 5915 5916 5917 5918 5919 \
    5920 5921 5922 5923 5924 5925 5926 5927 5928 5929 \
    5930 5931 5932 5933 5934 5935 5936 5937 5938 5939 \
    5940 5941 5942 5943 5944 5945 5946 5947 5948 5949

# Run Iperf3 servers
CMD python3 -u iperf.py
```

A.2 Code to create 50 Iperf3 servers (Running on container)

```
#!/usr/bin/env python3

#Necessary imports
import iperf3
import time
from multiprocessing import Process

#Iperf3 python3 module related code
def start_server(server_name , port):
    try:

        server = iperf3.Server()
        server.bind_address = '10.54.0.1'
        server.port = port
        server.verbose = False

        print("Starting_" + server_name + "_on_ip_" +
              str(server.bind_address) + "_and_port_"
              + str(server.port) + "...")

        while True:
            result = server.run()
            if hasattr(result , 'remote_host'):
                print(str "[" + time.strftime ("%m/%d/%Y, %H:%M:%S") + "]"
                      + server_name + "_got_connection_from_" + result.remote_host))

    except KeyboardInterrupt:
        print("Stopping_" + server_name + "...")
    finally:
        print("Stopped_" + server_name + "...")

#Start necessary Iperf3 servers
try:
    for i in range(0,51):
        p = Process(target = start_server ,
                    args = ("Server-" + str(i), 5900 + i,))

        p.start()
```

```
except KeyboardInterrupt:
    print("Stoping_servers ...")
except:
    print("Error:_unable_to_start_servers")
```

A.3 Function that starts an Iperf3 client and handles the test result (Host)

```
def iperf_run(interface_name , interface_ip , port ,
queue , num_elements , container):

    #General settings of the iperf3 client
    client = iperf3.Client()
    client.duration = 30
    client.protocol = 'tcp'
    client.server_hostname = '10.54.0.1'
    client.bind_address = interface_ip
    client.port = port
    client.num_streams = 1
    client.bandwidth = 1000000

    #Added option on the Iperf3 python3 module pull request
    client.get_server_output = True

    result = client.run()

    #Handle the test result
    object = json.loads(result)
    if(object.get('error')):
        print(object['error'])
    else:
        handle_json(object , interface_name ,
num_elements , container , queue)
```

Bibliography

- [1] Jr. Cox J. H. et al. “Advancing software-defined networks: a survey”. In: *IEEE Access* 5 (2017), pp. 25487–526. ISSN: 2169-3536. DOI: [10.1109/ACCESS.2017.2762291](https://doi.org/10.1109/ACCESS.2017.2762291). URL: <http://dx.doi.org/10.1109/ACCESS.2017.2762291>.
- [2] N. A. Jagadeesan and B. Krishnamachari. “Software-Defined Networking Paradigms in Wireless Networks: A Survey”. In: *ACM Computing Surveys* 47.2 (2015), 27 (11 pp.) ISSN: 0360-0300. DOI: [10.1145/2655690](https://doi.org/10.1145/2655690). URL: <http://dx.doi.org/10.1145/2655690>.
- [3] Francisco Fraga et al. “SDN4IoT: Software-Defined Network for Internet of Things”. In: *3rd Doctoral Congress of Engineering* (2019).
- [4] B. Trevizan de Oliveira, R. C. A. Alves, and C. Borges Margi. “Software-defined wireless sensor networks and internet of things standardization synergism”. In: *2015 IEEE Conference on Standards for Communications and Networking (CSCN)*, 28-30 Oct. 2015. 2015 IEEE Conference on Standards for Communications and Networking (CSCN). IEEE, pp. 60–5. DOI: [10.1109/CSCN.2015.7390421](https://doi.org/10.1109/CSCN.2015.7390421). URL: <http://dx.doi.org/10.1109/CSCN.2015.7390421>.
- [5] M. B. Yassein et al. “Combined software-defined network (SDN) and Internet of Things (IoT)”. In: *2017 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*, 21-23 Nov. 2017. 2017 International Conference on Electrical and Computing Technologies and Applications (ICECTA). IEEE, 6 pp. DOI: [10.1109/ICECTA.2017.8252003](https://doi.org/10.1109/ICECTA.2017.8252003). URL: <http://dx.doi.org/10.1109/ICECTA.2017.8252003>.
- [6] B. Hammi et al. “IoT technologies for smart cities”. In: *IET Networks* 7.1 (2018), pp. 1–13. ISSN: 2047-4954. DOI: [10.1049/iet-net.2017.0163](https://doi.org/10.1049/iet-net.2017.0163).
- [7] D. Cemernek, H. Gursch, and R. Kern. “Big data as a promoter of industry 4.0: lessons of the semiconductor industry”. In: *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*, 24-26 July 2017. 2017 IEEE 15th International Conference on Industrial Informatics (INDIN). Proceedings. IEEE, pp. 239–44. DOI: [10.1109/INDIN.2017.8104778](https://doi.org/10.1109/INDIN.2017.8104778). URL: <http://dx.doi.org/10.1109/INDIN.2017.8104778>.
- [8] Michael R. Collins. *What is Digital Farming - Really?* <https://www.precisionag.com/systems-management/what-is-digital-farming-really/>. Web Page. Visited on November 2018.

- [9] S. Singh and R. K. Jha. “A Survey on Software Defined Networking: Architecture for Next Generation Network”. In: *Journal of Network and Systems Management* 25.2 (2017), pp. 321–74. ISSN: 1064-7570. DOI: [10.1007/s10922-016-9393-9](https://doi.org/10.1007/s10922-016-9393-9). URL: <http://dx.doi.org/10.1007/s10922-016-9393-9>.
- [10] *Docker*. <https://docs.docker.com/>. Visited on February 2019.
- [11] *Open vSwitch (OVS)*. <https://docs.openvswitch.org/en/latest/>. Visited on February 2019.
- [12] *Open Virtual Networking With Docker*. <http://docs.openvswitch.org/en/latest/howto/docker/>. Visited on June 2019.
- [13] *OpenFlow Switch Specification v1.3*. <https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf>.
- [14] *Network slicing could match network capabilities to a variety of 5G use cases*. <http://inside5g.com/wp-content/uploads/2015/01/Screen-Shot-2015-01-26-at-14.39.48-e1422284898330.jpg>. Visited on February 2019.
- [15] Ericsson. *Network Slicing - Learn how Network Slicing can be a piece of cake*. <https://www.ericsson.com/en/digital-services/trending/network-slicing>. Web Page. Visited on February 2019.
- [16] J. Velez et al. “IEEE 1451-1-6: Providing common network services over MQTT”. In: *2018 IEEE Sensors Applications Symposium (SAS)*. Mar. 2018, pp. 1–6. DOI: [10.1109/SAS.2018.8336750](https://doi.org/10.1109/SAS.2018.8336750).
- [17] *Getting to know MQTT*. <https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/>. Visited on April 2019.
- [18] *MQTT*. <http://mqtt.org/faq>. Visited on April 2019.
- [19] Klever Cardoso, Antonio Oliveira, and Sand Correa. “SOFTWAY4IoT: SOFTware-defined gateWAY and fog computing for IoT (Internet of Things)”. In: *RNP 2* (2018).
- [20] *Flask v1.0*. <http://flask.pocoo.org/docs/1.0/>.
- [21] *PostgreSQL: The World’s Most Advanced Open Source Relational Database v11.2*. <https://www.postgresql.org/files/documentation/pdf/11/postgresql-11-A4.pdf>.
- [22] *Python*. <https://www.python.org/doc/>. Visited on February 2019.
- [23] *HTML5*. <https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5>. Visited on February 2019.
- [24] *JavaScript.com is a resource for the JavaScript community*. <https://www.javascript.com/about>. Visited on February 2019.
- [25] *AngularJS*. <https://docs.angularjs.org/guide>. Visited on February 2019.
- [26] *Ryu SDN Framework*. <https://osrg.github.io/ryu/>. Visited on February 2019.

- [27] *GNU Radio - The Free And Open Source Radio Ecosystem*. <https://www.gnuradio.org/docs/>. Visited on February 2019.
- [28] *C++ Language*. <http://www.cplusplus.com/info/description/>. Visited on February 2019.
- [29] *IoT Wireless Technology: Enabling Radio Frequencies*. <https://bridgera.com/wireless-technology-for-iot/>. Visited on February 2019.
- [30] M. O. Farooq and D. Pesch. “Analyzing LoRa: A use case perspective”. In: *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. Feb. 2018, pp. 355–360.
- [31] A. F. Rochim and R. F. Sari. “Performance comparison of IEEE 802.11n and IEEE 802.11ac”. In: *2016 International Conference on Computer, Control, Informatics and its Applications (IC3INA)*. Oct. 2016, pp. 54–59. DOI: [10.1109/IC3INA.2016.7863023](https://doi.org/10.1109/IC3INA.2016.7863023).
- [32] *Digital Farming: what does it really mean?* https://www.cema-agri.org/images/publications/position-papers/CEMA_Digital_Farming_-_Agriculture_4.0__13_02_2017_0.pdf. Visited on February 2019.
- [33] *What is Industry 4.0? Here Is A Super Easy Explanation For Anyone*. <https://www.forbes.com/sites/bernardmarr/2018/09/02/what-is-industry-4-0-heres-a-super-easy-explanation-for-anyone/>. Visited on February 2019.
- [34] James McCabe. *Network Analysis, Architecture, and Design*. The Morgan Kaufmann Series in Networking. Morgan Kaufmann, 2003.
- [35] M. Karakus and A. Duresi. “Quality of Service (QoS) in Software Defined Networking (SDN): A survey”. English. In: *Journal of Network and Computer Applications* 80 (2017/02/15), pp. 200–18. ISSN: 1084-8045. URL: <http://dx.doi.org/10.1016/j.jnca.2016.12.019>.
- [36] H. Ghalwash and Chun-Hsi Huang. “A QoS framework for SDN-based networks”. English. In: Los Alamitos, CA, USA, 2018//, pp. 98–105. URL: <http://dx.doi.org/10.1109/CIC.2018.00024>.
- [37] *Xen Project: Open Source Hypervisor*. https://wiki.xenproject.org/wiki/Main_Page. Visited on February 2019.
- [38] *Docker Networking Overview*. <https://docs.docker.com/network/>. Visited on June 2019.
- [39] *Arduino Uno Rev3*. <https://store.arduino.cc/arduino-uno-rev3>. Visited on June 2019.
- [40] *LoRa Shield for Arduino*. <http://www.dragino.com/products/module/item/102-lora-shield.html>. Visited on June 2019.
- [41] *RH-RF95 - LoRa RadioHead library*. <https://github.com/kenbiba/RH-RF95>. Visited on June 2019.

- [42] A. R. Chandan and V. D. Khairnar. “Bluetooth Low Energy (BLE) Crackdown Using IoT”. In: *2018 International Conference on Inventive Research in Computing Applications (ICIRCA)*. July 2018, pp. 1436–1441. DOI: [10.1109/ICIRCA.2018.8597189](https://doi.org/10.1109/ICIRCA.2018.8597189).
- [43] *GATT Specifications* | Bluetooth Technology Website. <https://www.bluetooth.com/specifications/gatt/>. Visited on June 2019.
- [44] *GATT* | Introduction to BLE | Adafruit Learning System. <https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt>. Visited on June 2019.
- [45] *Bluetooth GATT SDK for Python*. <https://github.com/getsenic/gatt-python>. Visited on June 2019.
- [46] Dong-Hoon Kim, Eun-Kyu Lee, and Jibum Kim. “Experiencing LoRa Network Establishment on a Smart Energy Campus Testbed”. In: *Sustainability* 11 (Mar. 2019), p. 1917. DOI: [10.3390/su11071917](https://doi.org/10.3390/su11071917).
- [47] *Ryu Rest QoS*. https://osrg.github.io/ryu-book/en/html/rest_qos.html. Visited on June 2019.
- [48] *Quality of Service (QoS) - Open vSwitch Documentation*. <http://docs.openvswitch.org/en/latest/faq/qos/>. Visited on June 2019.
- [49] M. I. Alipio et al. “Demonstration of Quality of Service mechanism in an OpenFlow testbed”. In: *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*. Oct. 2016, pp. 443–447. DOI: [10.1109/IMCEC.2016.7867251](https://doi.org/10.1109/IMCEC.2016.7867251).
- [50] *Configuration - Open vSwitch Documentation*. <http://docs.openvswitch.org/en/latest/faq/configuration/>. Visited on June 2019.
- [51] Twala B. Adedayo A.O. “QoS functionality in software defined network”. In: *Journal of Network and Systems Management* 25.2 (2017), pp. 321–74. ISSN: 1064-7570. DOI: [10.1007/s10922-016-9393-9](https://doi.org/10.1007/s10922-016-9393-9). URL: <http://dx.doi.org/10.1007/s10922-016-9393-9>.
- [52] Campus Network Design Fundamentals. *QoS Tools*. <https://flylib.com/books/en/3.293.1.56/1/>. Web Page. Visited on June 2019.
- [53] G.F. Anastasi et al. “QoS Guarantees for Network Bandwidth in Private Clouds”. English. In: *Procedia Computer Science* 97 (2016//), pp. 4–13. ISSN: 1877-0509. URL: <http://dx.doi.org/10.1016/j.procs.2016.08.275>.
- [54] *tc(8) - Linux man page*. <https://linux.die.net/man/8/tc>. June 2019.
- [55] *Actions - Open vSwitch Manual*. <http://www.openvswitch.org/support/dist-docs/ovs-actions.7.txt>. Visited on June 2019.

- [56] R. Durner, A. Blenk, and W. Kellerer. “Performance study of dynamic QoS management for OpenFlow-enabled SDN switches”. In: *2015 IEEE 23rd International Symposium on Quality of Service (IWQoS)*. June 2015, pp. 177–182. DOI: [10.1109/IWQoS.2015.7404730](https://doi.org/10.1109/IWQoS.2015.7404730).
- [57] W. Hsu et al. “The implementation of a QoS/QoE mapping and adjusting application in software-defined networks”. In: *2016 2nd International Conference on Intelligent Green Building and Smart Grid (IGBSG)*. June 2016, pp. 1–4. DOI: [10.1109/IGBSG.2016.7539431](https://doi.org/10.1109/IGBSG.2016.7539431).
- [58] *Quality of Service (QoS) Rate Limiting - Open vSwitch Documentation*. <http://docs.openvswitch.org/en/latest/howto/qos/>. Visited on June 2019.
- [59] *Mininet: Rapid Prototyping for Software Defined Networks*. <https://github.com/mininet/mininet>. Visited on February 2019.
- [60] *iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool*. <https://github.com/esnet/iperf>. Visited on June 2019.
- [61] *iperf3-python Module*. <https://github.com/thiezn/iperf3-python>. Visited on June 2019.
- [62] *Iperf3 Manual Page - ESnet*. <https://software.es.net/iperf/invoking.html>. Visited on June 2019.
- [63] *Wireshark | Go Deep*. <https://www.wireshark.org/>. Visited on June 2019.
- [64] *Open vSwitch database schema*. <http://www.openvswitch.org//ovs-vswitchd.conf.db.5.pdf2>. Visited on June 2019.
- [65] *OpenFlow Switch Specification - Open Networking Foundation*. <https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf>. Visited on June 2019.
- [66] *RFC 2475 - An Architecture for Differentiated Services*. <https://tools.ietf.org/html/rfc2475>.