

**Software Defined Networking**

**End-to-end Quality-of-Service in Software  
Defined Networking**

by

**Pradeep Jha, B.Tech.**

**Dissertation**

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

**Master of Science in Computer Science**

**University of Dublin, Trinity College**

September 2017

## **Declaration**

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

---

Pradeep Jha

August 28, 2017

## **Permission to Lend and/or Copy**

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

---

Pradeep Jha

August 28, 2017

# Acknowledgments

First, I would like to take the opportunity to thank my family who always believed in me and helped me become the person I am. Their values of hard work, honesty and humility has helped me in every aspect of my life.

I would like to convey my gratitude to my supervisor Dr. Marco Ruffini for his guidance and assistance in completing this dissertation. I would also like to thank Mr. Frank Slyne for always listening to the issues and providing suggestions for the troubles I had.

Thank you to all my friends for their continued motivation and support throughout the course. This has been an extraordinary year for me, and I am grateful to Dr. Stefan Weber for his guidance throughout the year.

PRADEEP JHA

*University of Dublin, Trinity College*

*September 2017*

# **Software Defined Networking**

## **End-to-end Quality-of-Service in Software Defined Networking**

Pradeep Jha, M.Sc.

University of Dublin, Trinity College, 2017

Supervisor: Marco Ruffini

Quality of Service (QoS) deals with providing end-to-end guarantees to the users. These guarantees may include parameters like bandwidth and latency guarantees, packet loss, jitter, congestion control, etc. There are many ways in which such assurances can be obtained. On the control plane, a network operating system may exploit various services like prioritized scheduling, resource reservation, queue management, routing, etc. A NOS can monitor and implement group policies and management in the network which all contribute the overall QoS. ONOS is a production ready controller which is fine-tuned for performance. Most SDN controllers usually provide support for some of these features but do not provide an implementation of end-to-end guarantees.

This project aims at exploiting the implementation of ONOS controller to implement QoS mechanism to provide user guarantees. We write an application on the application plane of the SDN architecture running an ONOS controller, OpenFlow protocol, Open

vSwitch and a simulated network on Mininet. Exploit ONOS primitives for resource allocation and queue management to provide QoS guarantees. The idea is to evaluate the extent to which these QoS services can be guaranteed on ONOS controller. The project identifies the gaps in the current state of affairs for QoS in SDN. The dissertation performs a number of experiments to implement different possible QoS mechanisms on ONOS and identifies the abilities and shortcoming of the SDN controller. These deficiencies need to be addressed to help SDN move beyond the boundary of merely a research interest, and to be accepted in the production level day-to-day internet.

# Contents

<b>Acknowledgments</b>	<b>iv</b>
<b>Abstract</b>	<b>v</b>
<b>List of Tables</b>	<b>x</b>
<b>List of Figures</b>	<b>xi</b>
<b>Chapter 1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	2
1.2 Motivation . . . . .	4
1.3 Research Question . . . . .	5
1.4 Dissertation Structure . . . . .	6
<b>Chapter 2 State of the Art</b>	<b>7</b>
2.1 Software Defined Networking - SDN . . . . .	8
2.2 Software Switches for the Data Plane . . . . .	11
2.3 SDN Controllers for the Control Plane . . . . .	14
2.4 OpenFlow Protocol for Communication . . . . .	18
2.5 Quality-of-Service in SDN . . . . .	22
2.5.1 IntServ . . . . .	22
2.5.2 DiffServ . . . . .	24
2.5.3 Related Research in QoS . . . . .	25

2.5.4	Summary . . . . .	41
2.6	This Project . . . . .	42
<b>Chapter 3 Design</b>		<b>43</b>
3.1	QoS in OpenFlow . . . . .	43
3.1.1	Queues . . . . .	44
3.1.2	Linux Traffic Control . . . . .	46
3.1.3	Meter Tables . . . . .	49
3.2	QoS in Software Switches . . . . .	50
3.3	ONOS: Architecture and Subsystems . . . . .	51
3.3.1	System Components . . . . .	51
3.3.2	ONOS Intent Framework . . . . .	53
3.3.3	Flow Rule Subsystem . . . . .	54
3.4	Mininet Network Emulator . . . . .	55
3.5	Measurement Tools . . . . .	56
3.6	Summary and Experiment Design . . . . .	57
<b>Chapter 4 Implementation</b>		<b>60</b>
4.1	Experiments . . . . .	60
4.2	Experiment 1 . . . . .	61
4.2.1	Statement . . . . .	61
4.2.2	Implementation . . . . .	61
4.2.3	Result . . . . .	64
4.2.4	Observations . . . . .	65
4.3	Experiment 2 . . . . .	66
4.3.1	Statement . . . . .	66
4.3.2	Implementation . . . . .	66
4.3.3	Result . . . . .	66
4.3.4	Observations . . . . .	67

4.4	Experiment 3 . . . . .	68
4.4.1	Statement . . . . .	68
4.4.2	Implementation . . . . .	68
4.4.3	Result . . . . .	72
4.4.4	Observations . . . . .	72
4.5	Experiment 4 . . . . .	73
4.5.1	Statement . . . . .	73
4.5.2	Implementation . . . . .	73
4.5.3	Results . . . . .	76
4.5.4	Observations . . . . .	76
4.6	Experiment 5 . . . . .	78
4.6.1	Statement . . . . .	78
4.6.2	Implementation . . . . .	79
4.6.3	Results . . . . .	80
4.6.4	Observation . . . . .	80
<b>Chapter 5</b>	<b>Discussion</b>	<b>81</b>
<b>Chapter 6</b>	<b>Conclusion</b>	<b>86</b>
6.1	Contributions . . . . .	87
6.2	Future Work . . . . .	88
<b>Appendix A</b>	<b>Abbreviations</b>	<b>89</b>
<b>Bibliography</b>		<b>91</b>

# List of Tables

2.1	Popular SDN Controllers . . . . .	16
2.2	OpenFlow Versions and Enhancements . . . . .	20
2.3	Components of an entry in the flow table [27] . . . . .	21
2.4	Components of meter entry in meter table [27] . . . . .	21
3.1	QoS Support in Popular Software Switches . . . . .	50
3.2	Software Requirement for the experiments . . . . .	59
4.1	QoS Settings on Switch S1 . . . . .	74
4.2	QoS Settings on Switch S2 and S3 . . . . .	75
4.3	Meter Entry in Meter Table 1. . . . .	75
5.1	Summary of Achieved QoS . . . . .	84

# List of Figures

2.1	Software Defined Networking Framework [6] . . . . .	9
2.2	Software Defined Networking Architecture [7] . . . . .	10
2.3	Open vSwitch Interfaces [12] . . . . .	12
2.4	OpenFlow Protocol . . . . .	19
2.5	RSVP Protocol . . . . .	23
2.6	Categories of QoS efforts . . . . .	26
2.7	Relevant Research in QoS in SDN . . . . .	26
2.8	The area of SDN and QoS that this project aims at. . . . .	42
3.1	OVSDB Table Relationships . . . . .	45
3.2	Sample HTB class hierarchy . . . . .	47
3.3	General ONOS architecture. [50] . . . . .	51
3.4	ONOS Intent Framework Compilation and Flow Installation. [50] . . . . .	53
4.1	Setup for Experiment 1. . . . .	62
4.2	Building a Queue . . . . .	62
4.3	Building a QoS . . . . .	63
4.4	Applying QoS Settings . . . . .	63
4.5	Removing QoS Settings . . . . .	63
4.6	Queues with ONOS. . . . .	67
4.7	Configuring a network environment with Lagopus . . . . .	69
4.8	Network for Experiment 4. . . . .	74

4.9	AF11 Excess vs Best Effort Traffic . . . . .	77
4.10	AF11 Excess Traffic vs Best Effort vs AF11 Non-Excess Traffic . . . . .	77
4.11	AF11 Excess AF11 Excess Traffic . . . . .	78

# Chapter 1

## Introduction

SDN presents a simple data plane architecture consisting of simple packet forwarding elements with flow tables. These elements match the incoming flows in the flow table and apply the specified action on the matching flows. The control plane lies above the data plane, and this is where all the routing decisions are made. Once a decision is reached for a “family” of flow, it is updated to all the flow tables, thus saving time for subsequent flows. In other words, the networking decisions are now taken in software rather than hardware.

The control plane usually runs a Network Operating System (NOS) like Ryu or ONOS. The OS makes the decision for the data plane elements connected to the controller. Thus, it is agile, centrally managed and easily programmable. This makes the deployment of new protocols and services faster and cheaper as no hardware replacement is required. The application plane sits above the control plane and runs on the network operating system like any other applications for desktop computers would run on a desktop operating system. This means that developers now have the power not only to write useful applications for the internet but also to mold the network to suit the applications need.

The SDN controllers communicate with the application layer by North Bound In-

terfaces (NBI) and to the data plane by South Bound Interfaces (SBI) like OpenFlow. All the above are open source implementations, and it makes SDN vendor independent. Current internet architecture has several drawbacks which make it unsuitable for modern applications like cloud computing and big data processing. SDN provides the scalability and flexibility required to serve the changing traffic patterns.

Quality-of-Service (QoS) is defined by Cisco as “the capability of a network to provide better service to selected network traffic over various technologies with the primary goal to provide priority including dedicated bandwidth, controlled jitter and latency, and improved loss characteristics.” QoS deals with providing end-to-end guarantees to the users. There are many ways in which such assurances can be obtained. One can use one or any combinations of these technologies to implement QoS. On the control plane, a network operating system may exploit various services like prioritized scheduling, resource reservation, queue management, routing, etc. A NOS can monitor and implement group policies and administration in the network which all contribute the overall QoS.

This chapter provides a brief background of Software Defined Networking. The origin, progress, and challenges faced by SDN and how QoS fits into this new networking paradigm. Next, it discusses the objectives and approach of this dissertation. The chapter concludes by outlining the structure of the dissertation and provides a quick synopsis of the chapters to follow.

## 1.1 Background

Quality of Service on the internet has been an area of continuous research and improvement for decades. The traditional internet, which was not initially designed with QoS in mind, was later supplemented by many techniques to achieve the desired performance tuning. These techniques allowed the Internet Service Providers (ISP) to fine tune the

internet as required. However, the traditional internet is facing new challenges with every new emerging technology. The increasing number of devices, the growing volume and velocity of traffic, big data and cloud computing are some of the problems that the traditional internet is finding hard to cater to. Software Defined Networking (SDN) provides a solution to these challenges by making the internet flexible and programmable.

As more and more vendors are accepting SDN as the new networking paradigm, the demands on SDN is changing with time. SDN, just like the traditional internet was not designed with quality of service in mind. The primary solution that SDN provided over the traditional internet was that of flexibility and programmability. This means that, for SDN to be deployed on the world wide internet, it needs to support fine grained QoS, equivalent to what is possible in the traditional internet, if not better.

While QoS in SDN is still an area of research, it would not be wrong to believe that achieving them on SDN would be far easier than it was on the traditional internet considering it is programmable nature. One of the biggest advantages of SDN is its vendor agnostic and open source nature which has led to rapid acceptance and involvement of research communities world wide, both in academia as well as the industry. Which, in turn, has led to rapid development and improvement in SDN in very little time. This has led to the birth of a plethora of independent projects working in different areas of SDN. These projects range from various software switches for the data plane, different protocols for south bound and north bound interaction and most notably, different approaches to the implementation of the network controller. Some of the notable network controller projects worth mentioning are Ryu, POX, ONOS, OpenDayLight, FloodLight, etc. All these controllers have a different design approach towards SDN, a different target audience and understandably, different priorities and features. Projects like Ryu and POX are mainly targeted towards the research community while others like OpenDayLight and ONOS is aiming to create a production ready controller for SDN.

## 1.2 Motivation

SDN, being a flexible and programmable network, provides an effective solution to many problems faced by the current internet architecture. It makes it easier to test and deploy new protocols, it has increased the rate of innovation in the networks, and it decreases the barrier for a new technology to be deployed at large scale. These properties have led to some major players like Google<sup>1</sup> and Microsoft<sup>2</sup> to switch to SDN for their cloud and data center operations. This has led to an interest in the community of vendors and ISP's. Vendors like Cisco, Ericsson, Fujitsu, Nokia and ISP's like T-mobile, Verizon and NTT are actively involved with the Open Networking Foundation (ONF) for research and development of SDN [1]. For SDN to replace the traditional architecture in the world wide internet and for SDN to expand beyond the boundary of research to everyday life of the end-users of internet, the ISP's need the ability to fine tune almost every aspect of the network as per their requirement.

The present architecture and hardware network devices have undergone an immense amount of research and support for performance and fine grained quality of service. The software counterparts are still under research to provide the same level of performance as the current hardware devices. SDN Controllers, software switches, and various north bound and south bound protocols are continuously developing and providing better performance benchmarks and new QoS features.

SDN is dubbed by Nick McKeown of Stanford University, the creator of the OpenFlow protocol, as the next revolution in technology. The growth of SDN is expected to rival that of the World Wide Web in the coming years [2]. All of this together makes Quality of Service in Software Defined Networks a fascinating area to study and explore.

---

<sup>1</sup>Google Keynote at Open Networking Summit; <https://goo.gl/2YhsCr>

<sup>2</sup>Microsoft TechNet on SDN in Azure; <https://goo.gl/ZRNytJ>

### 1.3 Research Question

The Open Network Operating System (ONOS) by Open Networking Lab (ON.Lab) is one of the leading SDN controller available. The Open vSwitch (OVS) software switch and the OpenFlow (OF) south bound protocol are considered the de facto technologies to be used in SDN. All these different parts of the SDN puzzle, the controller, the software switch and the communication protocol between them, needs to fit perfectly into each other to achieve an industrial level quality. These projects are independently maintained and have different set of priorities and aspirations.

ONOS claims to be one of the best “production ready” SDN controller available[3] . It has a huge community of over 50 partners, and numerous open-source collaborates. It has a solid architecture and is maturing very quickly. ONOS provides a strong case for taking SDN to the next step of its evolution, i.e., to be deployed in the general internet by the ISP’s and data center operators. If ONOS has to become the driver of SDN to real world networks, it has to support fine grained, production-level Quality of Service features. Based on the above discussion, the following research question has been posed:

**“Can ONOS be used to provide end-to-end Quality-of-Service in real world networks?”**

This can be further sub-divided into the following questions:

1. Can we provide end-to-end quality of service with ONOS?
2. Can we prioritize guaranteed traffic over best-effort traffic and control the rate using OpenFlow Queues?
3. Can we provide class-of-traffic based QoS using Differentiated Service and exploit OF1.3 Meter Tables to enhance the end-to-end bandwidth guarantee model?
4. To what extent does ONOS supports the QoS features of the OF specifications?
5. How does ONOS compare to its alternatives in QoS capabilities?

## 1.4 Dissertation Structure

- Chapter 1 provides an introduction to the project. It presents a brief background and motivation behind the project and then presents the research questions addressed in this dissertation.
- Chapter 2 presents the State of the Art of Software Defined Networking and Quality of Service. This chapter briefly introduces the technologies and areas involved in this project and then reviews the relevant literature.
- Chapter 3 explains the architecture and internal details of the various components of SDN framework that would be used in this project. This chapter sets up the required technical background for the experiments to follow.
- Chapter 4 presents the various experiments conducted in this dissertation. It presents the statement of the experiment followed by its implementation, results, and observations.
- Chapter 5 provides an overview and discussion of the observations gathered throughout the project. This chapter places the research and findings into the overall big picture of SDN.
- Chapter 6 concludes the dissertation, talking about the challenges faced and contributions made in this project. It ends with a discussion on the future work in this direction.

# Chapter 2

## State of the Art

The proposed dissertation is broadly about Quality of Service in Software Defined Networking, which consists of a variety of technologies working in tandem with each other. For example network virtualization, the network operating system, various communication protocols between different components of the network, etc. This chapter discusses the various elements that together constitute this project and discusses the previous and ongoing research in these broad areas. Section 2.1 discusses the notion of Software Defined Networking in general and how is it different from the traditional networking architecture. We examine the state-of-the-art of SDN, its benefit over traditional networks and the challenges faced by it. Section 2.2 introduces the reader to the concept of software switches which constitute the data plane of SDN. It then presents the ongoing advancement and various alternatives available for the switches. The section also briefly talks about network virtualization. Section 2.3 introduces the readers to the concept of SDN Controller and explores the different approaches towards creating the most efficient controller. Section 2.4 addresses the communication protocol between the control plane and the data plane. There are many protocols available for this communication, but the OpenFlow protocol is the leading protocol used for this purpose. We will briefly examine the evolution of this protocol, and various QoS capabilities of OpenFlow will be discussed. We will then move on to discussing Quality of Service in SDN in section 2.5. We will explore in some

detail the various approaches taken in the research community towards achieving QoS in SDN. We will discuss, in detail, the previous and ongoing research that is relevant to this dissertation. This discussion will establish the need and inspiration for the project. The next section 2.6 will briefly describe where this project stands regarding the state of the art in the research of Quality of Service in Software Defined Networking.

## 2.1 Software Defined Networking - SDN

Software Defined Networking, as described earlier, decouples the current internet architecture into separate planes. B. Raghavan *et al.*[4] is one of the early adopters to propose the decoupling of the data plane from the control plane. The primary motivation for SDN was that many new research and proposals never saw the light of day because it required a massive replacement and up-gradation of the hardware infrastructure. It proposes a top-down approach instead of the bottom-up approach of the traditional internet because no global agreement is required anymore. The control is logically centralized at a layer above the data plane.

Let us understand the concept of SDN in a little more detail and throw some light on how is it different from the traditional internet. In the current architecture decisions about routing a packet is taken per-hop at every router in the path. If someone plans to implement a new protocol for routing, each router needs to be reprogrammed or replaced with a new supporting hardware. SDN proposes a decoupling of the data plane forwarding elements from the control plane. The data plane devices will be simple forwarding elements which forward a packet or flow as per the rules defined by the controller. The controller, on the other hand, acts as the brain of the network and is implemented in software. This makes the entire network programmable and centrally managed. A number of Network Function Virtualization (NFV) techniques can be integrated with SDN, and it has given rise to interesting use cases in the networking domain.

The controllers will use the South Bound APIs like OpenFlow to relay information to the

switches and routers in the data plane below. The North Bound APIs will be used to communicate with the applications implementing the “rules” and “policies.” This makes it easier for network administrators to shape traffic and for researchers to experiment with new protocols. The Open Networking Foundation provides the following framework for SDN:

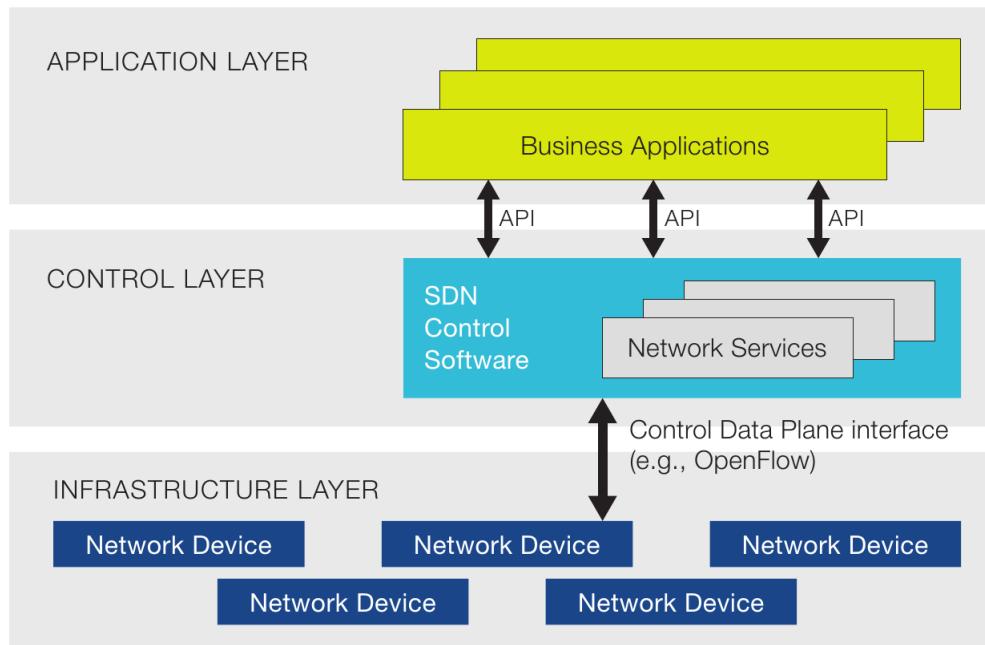


Figure 2.1: Software Defined Networking Framework [6]

The architectural difference between the traditional internet and SDN is very well presented in the following diagram by [7]. It represents clearly how the control is (logically) centrally managed, and the data plane is simplified into simple forwarding elements. The programmable switches of the data plane can either be implemented in hardware or software as long as they support the OpenFlow protocol for communication and configuration with the controller.

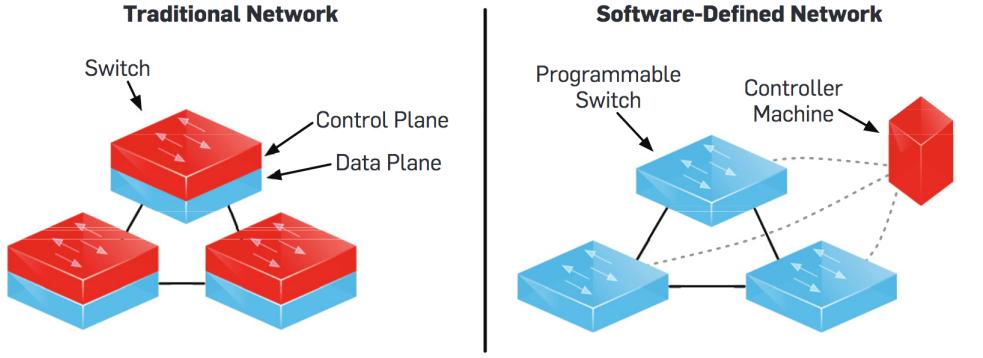


Figure 2.2: Software Defined Networking Architecture [7]

R. Masoudi *et al.*[5] presents a very comprehensive survey of SDN and related technologies. It provides a categorical review of the research covering all the different components of SDN. It points out how researchers have exploited parallelism in hardware, Linux's networking prowess, traffic engineering techniques, etc. and have used various simulators, debuggers and testbed to build upon the initial idea of SDN and provide future direction for it. It talks about the different issues faced by the current state of the control plane and presents various works in the area categorically. It reviews works around scalability, consistency, load-balancing, fault tolerance, security and latency before providing a brief overview of different controller implementations. It also provides a brief review of the simulators, debuggers, and testbeds for SDN research. It presents an interesting perspective about Software Defined Environments (SDE) and how SDN was the missing piece of the puzzle. SDN, along with software defined storage, compute and management makes on-demand automated provisioning and orchestration of IT infrastructures possible, thus reducing cost (CAPEX and OPEX) and the time for new ideas and products to become a reality. While SDN is just a principle, it needs the various components of the architecture to work together. The simple forwarding elements, the software controllers, the communication protocols etc. complete the architecture of SDN and all these components are discussed briefly in the following sections.

## 2.2 Software Switches for the Data Plane

As mentioned earlier, the data plane of SDN is made up of simple forwarding devices. These can be either implemented in hardware or software. Hardware switches have been tried and tested and proven to provide good performance. OpenFlow compatible hardware switches can be used to implement real life networks with SDN. These switches usually use various virtualization techniques and can run either on the user or the kernel level of the host system. The real benefit of software switch is that they provide a platform for rapid testing within the research community. Moreover, their predominantly open source nature has increased the rate of innovation in networking. Many industry-leading partners and vendors have jumped into soft-switch (an abbreviation for software switches) wagon. Some of the notable ones being VMWare, Cisco, Big Switch Networks and NTT. Various research groups have also come up with their open source projects for soft-switches like the CPqD OfSoftSwitch13[9] switch. For a long time, the software switches have struggled to provide as good a performance as their hardware counterparts. However, there has been a significant improvement in these statistics recently with the better use of the underlying hardware. Intel has released the Data Plane Development Kit (DPDK)[10] which exploits parallelism and other techniques to provide fast packet processing. The Lagopus Switch[11] by NTT uses a DPDK based data plane and provides higher performance when running in the DPDK mode. It can also be run in the default raw socket mode.

**Open vSwitch**[8] is one of the most popular software switches and is backed by the Linux Foundation. It is considered by many as the de facto software switch for SDN. OVS was created by Nicira which was later acquired by VMWare. It was targeted to meet the requirements of the open source community and uses Linux based hypervisors for virtualization. It is feature rich and supports all the major protocols and technologies like VLAN tagging, NetFlow, Port Mirroring, etc. OVS uses the OpenFlow protocol,

which we will discuss in a subsequent chapter, for communication with the controller and uses the Open vSwitch Database (OVSDB)[12] for configuration management. Figure 2.3 represents the schematic setup of OVS and its interfaces.

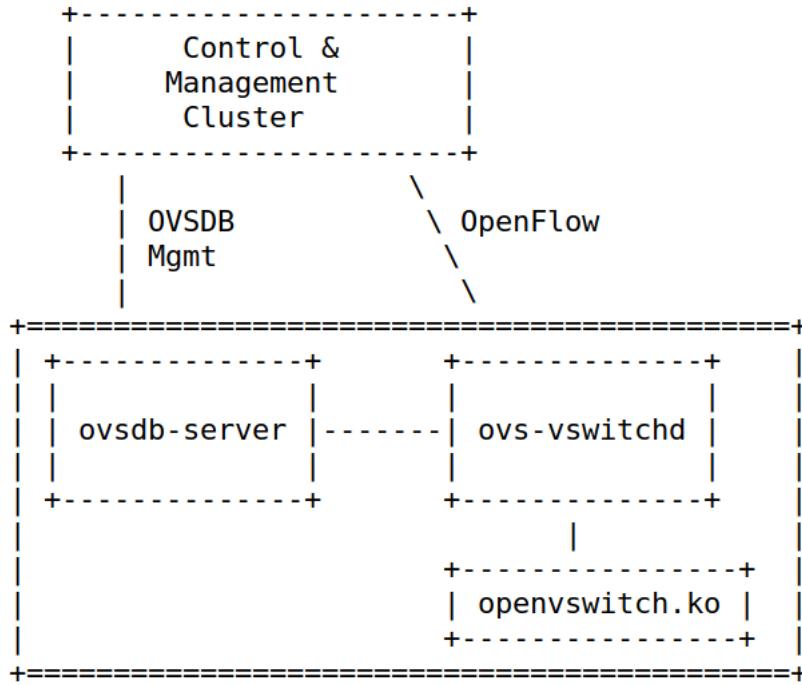


Figure 2.3: Open vSwitch Interfaces [12]

While OVSDB is being used extensively by the community for configuring the OVS, it is not the standard (per say) protocol for OF switch management. The ONF has released OF-CONFIG[13] which is based on the well known standard NETCONF for configuration management of OF switches. T. Cejka *et al.*[18] talks about replacing OVSDB with OF-CONFIG and possible benefits of having a protocol standard. It uses JSON-RPC messages to communicate between the server and the client. The paper and ONF claims that it is the more secure protocol of the two as it uses SSH by default and recommends using TLS for communication. The paper suggests two alternative approaches to introduce OF-CONFIG in OVS. One in which the NETCONF server sits alongside the OVSDB

server and other in which the NETCONF server is the standalone server for configuration management. The authors expect the standalone NETCONF version to be more secure and to have faster development and deployment times, considering it's independent nature.

The **CPqD OfSoftSwitch13**[9] is another switch that is widely used in the research community. It is an experimental switch forked from the Ericsson TrafficLab 1.1 SoftSwitch implementation with changes in the forwarding plane to support OF1.3. It comes packaged with the following tools to control and manage the data plane:

- OfDatapath: The switch implementation.
- OfLib: A library for converting to/from OF1.3 wire formats.
- DPCTL: Console tool to configure the switch.
- OfProtocol: A secure communication channel with the controller.

All this makes it a complete alternative to the OVS. However, the authors of the switch state the following, "*Despite the fact the switch is still popular for adventurers trying to implement own changes to OpenFlow, support now is on a best-effort basis. Currently, there are lots of complaints about performance degradation, broken features, and installation problems.*"[9]. The switch still has one of the best support for OF1.3 features among the available soft-switches, specifically the optional features like meter tables, etc., which makes it an attractive candidate to get a hands-on with.

The implementation of switches in software has made it easier for researchers to try and test new protocols and techniques. However, testing on a handful of switches may not necessarily provide enough insights to how is it going to behave in real networks with many devices and links. **Mininet**[16] is a network emulator that uses Linux's network virtualization techniques like network namespaces to create a realistic virtual network on a single machine. N. McKeown *et al.*[17] provides the details of mininet's implementation.

The paper demonstrates with experiments that mininet can be used to emulate large topologies. This is because instead of using full virtualization it shares the kernel, device drivers and other resources and reduces the overhead from 70 MB/host to 1 MB/host. They also demonstrate that bandwidth of around 2-3 Gbps with one switch and more than 10 Gbps with 100 switches are attainable. Simple Python codes can be written on a familiar CLI to create and control different nodes. It also provides some basic GUI-based tools to interact with individual nodes directly. Mininet uses CPU multiplexing for queue control. For this, it does not implement a novel scheduler but uses the Linux scheduler on which it is running. The Linux scheduler, however, does not provide any guarantees on packet scheduling. Mininet supports use cases in prototyping, optimization, tutorials, demos and regression testing and solves problems that would have been very difficult to solve otherwise. Mininet with SDN and OpenFlow provides a rapid solution for testing new ideas in realistic settings.

## 2.3 SDN Controllers for the Control Plane

The control plane of an SDN constitutes the brain of the network and all the decision regarding routing and policies etc. are taken at the control plane of an SDN. It is the middle man between the applications connected to the northbound interface and the network devices connected to the southbound interface. Control plane consists of 2 main components; applications and the network operating system a.k.a the controller. The design of the controller is a critical task and must be done efficiently. There may be more than one controller in a network, and they must coordinate to create a consistent network state. This is usually done by assigning one controller as the master controller and others as backup. Even in the presence of multiple controllers, the control should logically appear to be centralized. The NBI and the SBI are the two important abstractions provided by the controllers. The applications sitting above the controller need to bother about the internal architecture and functioning of the layers below it and the devices below the con-

troller on the SBI need not bother about the layers above it. This facilitates independent research and development of each layer. The controller design and architecture is central to the performance of the network on the whole and there has been numerous research for in this area. R. Masoudi *et al.*[5] categorizes the area of concerns for controller design into following categories:

- **State Consistency:** A real life network would be too large to be handled by a single controller. It will also be too big a risk as it would be a single point of failure. Thus, usually, a network has multiple controllers managing a group of network devices. There is redundancy, i.e., a switch might be connected to multiple controllers in master and backup mode for fault tolerance. These multiple controllers must maintain a consistent state of the network among themselves, or it may lead to anomalous conflicting decisions by different controllers.
- **Scalability:** Scalability of the SDN controllers become a critical problem with the increase in the popularity of SDN. One way to achieve scalability is to design the controller with a distributed architecture. ONOS is one such distributed network operating system and hence has been the most widely accepted production, ready controller. Server replication, load-balancing policies, east-west communication with other replicas and controllers, distributed network graph, etc. are some of the techniques used by the such distributed controllers.
- **Flexibility and Modularity:** There has also been an interest in the research regarding the flexibility and modularity in the design of the SDN controllers. The static mapping between the switches and controller often leads to uneven load balancing, especially in the case of network churn. Efforts are being made to make the relationship more dynamic. Modularity ensures that the faults or failures of one module don't interfere with the working of the rest of the modules. It also ensures faster development and maintenance.

- **High Availability and Low Latency:** Tolerating failure and recovering from link and switch faults are some the most important consideration of controller design. A distributed system must ensure high availability even in the presence of faults. The Controller Placement Problem[19] is a well-known problem which has been explored by many researchers. It addresses the problem of latency and where to place the controller in the network for the best possible result.
- **Security and Dependability:** The SDN architecture transfers the power of the network to the application developers. These applications that run on the controllers manipulate the network behavior. They can be put to misuse or various other types of attacks, compromising the entire network. There has been ongoing research on how to mitigate these risks and areas such as role-based authorization, conflict evaluation, resource consumption, thread based isolation, etc. are being explored.

Since the inception of SDN, there has been increasing interest in the research and industrial community towards creating an SDN controller that can be deployed in real world networks. The following table 2.1 presents a list of the most notables controllers, the language it is implemented in and the developers.

Table 2.1: Popular SDN Controllers

Controller	Implementation	Developers
NOX [20]	C++	Nicira
POX [21]	Python	Nicira
Ryu [22]	Python	NTT
OpenDayLight [23]	Java	The Linux Foundation
FloodLight [24]	Java	Big Switch Networks
ONOS [25]	Java	Open Networking Lab

L. Stancu *et al.*[26] presents a comparative analysis of several SDN Controllers. This paper chooses some of the most widely used controllers: POX, Ryu, ONOS, and OpenDaylight (ODL) and performs experiments to compare their performances under different test conditions. The authors use Mininet simulation of a tree topology with a fan-out of 4. 16 hosts were created with each switch from the lowest level having two hosts. In the 1st phase the Open vSwitch acted like a mere hub with flooding, and in the next step, it worked like an L2 learning switch in which the controller used MAC addresses for forwarding flows. Two experiments were conducted:

- `ping` between host `h1` and `h16` with 10 ICMP packets to determine average RTT in hub mode and then how fast the RTT decreases in switch mode.
- `iperf` performance test to test to-and-fro bandwidth in hub mode and in switch mode.

Mininet topology was recreated from scratch after every experiment to keep the flow tables empty. The results were presented, and ONOS and ODL (written in Java) performed better than POX and Ryu (written in Python). Considering that POX and Ryu are meant for rapid development in research community while ONOS is a production ready controller, these are expected results. Considering that C++ and Java provide better performance over Python (good coding practices considered), the authors expected the performance of ONOS to be better than POX and Ryu. But this might not be the only factor affecting performance. The fact that ONOS is a distributed controller and that, it is production ready, means that the developers may have implemented special performance improvement techniques in it.

However, ONOS is indeed one of the best production ready controller available, and thus we choose the ONOS controller to test its QoS capabilities for this project. P.

Berde *et al.*[15] presents the experience and insights of building the ONOS controller from the ground up. It helps us understand the design decisions and rationale behind them. They first identify that a good distributed network operating system should provide high throughput as would be processing a huge volume of requests per second. It should have a massive global network size and high availability which is expected of any distributed system. They created two different versions of the ONOS prototypes and evaluated their performance. While the first prototype aimed primarily at achieving fault tolerance, consistency, and a distributed registry, the second prototype implemented performance improvements by creating better abstractions, more efficient data model, atomic notifications and reducing the number of expensive data store operation. The project was then released to the open source community and has undergone massive changes over time. But, the core principles remains the same. Since Open Networking Lab (ON.Lab) along with its partner AT&T and NTT released the source code, the ONOS controller has grown from strength to strength. It now consists of over 50 partners and collaborators including industry leaders like Cisco, Ericsson, Google, Fujitsu, Intel and Huawei to name a few and numerous other collaborators from academia and research groups. All this makes it a perfect candidate to take SDN from a research interest to reality in the real world network. The current state of the controller will be discussed in a later chapter on design.

## 2.4 OpenFlow Protocol for Communication

OpenFlow is the most widely used South Bound Interface for controllers in SDN implementations. It is the proposed standard communication protocol between the controller and the data plane by the Open Networking Foundation (ONF). When a new flow arrives at the data plane whose corresponding entry is not found in the flow table, the element forwards the flow to the controller using OpenFlow protocol. The controller then decides the route for the flow and makes an entry in the flow table. The controller, also, from

time to time, inject QoS, maintenance and management flow into the data plane. All the communication between the controller and the data plane takes place using the OpenFlow protocol. Since each flow can be manipulated separately, it provides a solid ground for testing experimental traffic in real networks without interfering with the production traffic. Figure 2.4 represents the schematic position of the OpenFlow protocol between the control and the data plane.

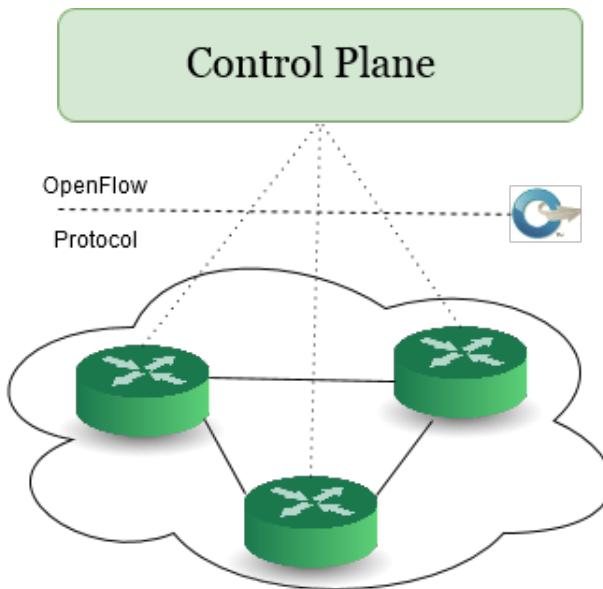


Figure 2.4: OpenFlow Protocol

N. McKeown *et al.*[14] talks about how most of the new ideas from the networking research community goes untested. That how network infrastructure has stagnated due to the reluctance of the network operators to experiment with production data. This paper provides the first strong case for the need of flexibility in the network and proposes the first version of the OpenFlow protocol and OpenFlow compatible switches. The paper argued that this would enable innovation in campus networks across the world. The paper was published in 2008 and looking back at last few years; the claim turned out to be true. IP based routing has many problems especially regarding the increasing number of devices and their address exhaustion. Network research community has been trying to move away

from IPv4 routing from a long time. OpenFlow routing does not need the addresses of the flow to a format if it is unique. It can route packets based on several criteria including IP, MAC, VLAN and other addressing and thus provide a strong replacement for IP-based routing. The OpenFlow protocol has undergone rapid enhancements since then releasing numerous new versions. The table below sums up some of the significant improvements of OpenFlow:

Table 2.2: OpenFlow Versions and Enhancements

<b>OpenFlow Version</b>	<b>Support Description</b>
OF v1.1.x	MPLS, VLAN, Multipath, Group Tables, Multiflow Tables and Queues
OF v1.2.x	In-Match, Packet-In, Extensible Headers, Flexible flow match and IPV6
OF v1.3.x	Tunneling, Meter Tables and Additional fields in the flow tables (Priority, Timeouts, Cookie, flags)
OF v1.4.x	Optical Port Management, Synchronized Tables, Eviction and Flow Monitoring
OF v1.5.x	Egress Tables, Flow Entry Statistics, Packet Type Aware Pipeline TCP Flag Matching

## Flow Tables

Flow tables are the data structures that reside on OpenFlow switches and facilitates the communication among the hosts. The switches refer to the flow table to match an incoming flow and then apply the specified action for the given flow type. Table 2.3 represents the components of a flow entry in the flow table of an OpenFlow 1.5.1

compatible switch. These entries are installed and maintained by the controller.

Table 2.3: Components of an entry in the flow table [27]

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags
--------------	----------	----------	--------------	----------	--------	-------

When a new flow arrives at the switch, the switch matches it with the available match fields. If it matches any of the match fields available in the flow table, the corresponding instructions are applied to the flow according to the defined priority and corresponding counters are incremented. If the new flow doesn't match any existing match field, the request is sent to the controller and controller then installs a new entry in the table.

## Meter Tables

The OpenFlow protocol standard introduced meter tables in OF1.3. A meter table defines per-flow meters which enable us to implement simple QoS operations like rate limiting. Meters can be used along with per-port queues to implement complex QoS mechanisms like DiffServ[29]. Meters and its functioning are central to this project and will be discussed in detail in a later chapter on design. However, a simplistic view of the main components of a meter table is shown in Table 2.4 below.

Table 2.4: Components of meter entry in meter table [27]

Meter Identifier	Meter Bands	Counters
------------------	-------------	----------

OpenFlow and its features would be discussed in more details throughout the following chapters of this dissertation. Particularly the features related to QoS like rate limiting

and policing using the meter tables and queues. The next section discusses QoS and its related research which would make the discussion on OF's QoS capabilities more relevant.

## 2.5 Quality-of-Service in SDN

Quality of Service in SDN is an area of ongoing research and has been garnering increasing interest in the research community. This section will present and summarize some of the interesting research works in the area of QoS in SDN.

The Internet Engineering Task Force (IETF) categories QoS into 2 major architecture, i.e. Integrated Service (IntServ)[28] and Differentiated Service (DiffServ)[29]. Rate guarantees can be classified into Hard QoS and SoftQoS. Hard QoS guarantees are rigid that reserves a portion of the bandwidth to be used only by a specific flow. It has stringent policies on the admission of the flow. If the required bandwidth is not available, the flow is rejected at ingress itself. Soft QoS, on the other hand, is more flexible in implementation but it does not provide very strong guarantees. Let us discuss the two in details.

### 2.5.1 IntServ

End-To-End QoS by using resource reservation and admission control as the key building blocks. IntServ uses the Resource Reservation Protocol (RSVP) to notify/request every network device along the path of a flow to reserve the particular amount of resources for the flow. If every device along the way can reserve the necessary bandwidth, only then the transmission can begin. The RSVP protocol has two important messages; the **path** message and the **resv** message. The **path** message is sent by the sender with the information about the previous hop IP address, traffic specifications, and the QoS requirements. The **resv** message is the reply to the path message from the receiver to the sender in the reverse direction of the **path** message. It sets the resource reservation for the flow along the route. Figure 2.5 demonstrates this setup mechanism. The RSVP

protocol then uses the **pathTear** and **resvTear** messages to tear down the connection and free the resources once the devices have done transmitting.

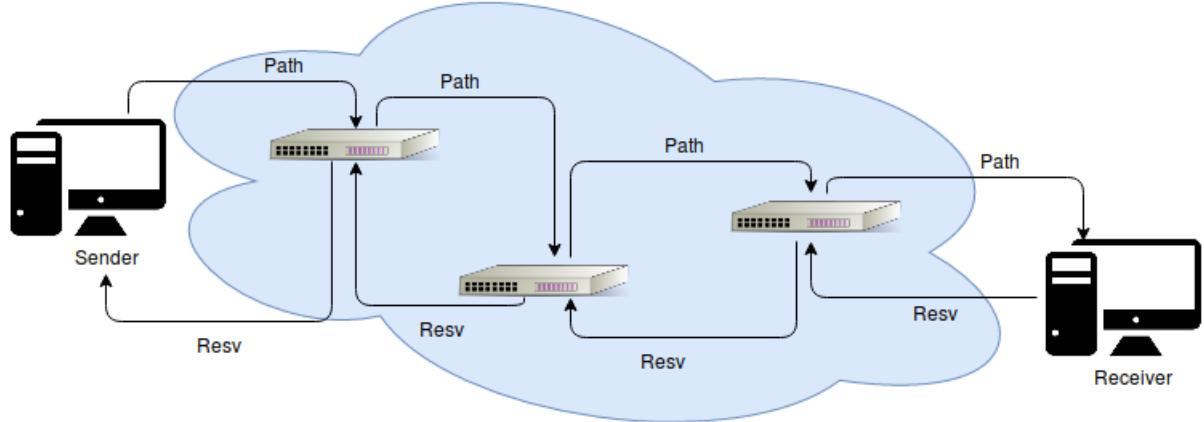


Figure 2.5: RSVP Protocol

While IntServ is an effective way to provide a Hard QoS guarantee, it never took off as an all-in-one solution for the QoS requirements of the internet. This was primarily because of the following disadvantages of IntServ:

- All the routers along the traffic must support IntServ.
- Every router along the path needs to store many states which become very expensive as the network scales.
- The periodic refresh of creating new QoS and tearing down the unused ones requires a huge number of messages to be transmitted into the network.

For these reasons, IntServ was never deployed on the internet. It is simply not scalable. However, it has been used widely in local and smaller enterprise networks. The primary advantage of IntServ is that it provides a hard guarantee and can be used to provide service class differentiation. A critical and intolerant application can be guaranteed the required bandwidth while the rest of the traffic will compete in best effort mode.

## 2.5.2 DiffServ

Scalability was the biggest issue impeding the growth of IntServ as a QoS mechanism. DiffServ was introduced to address that problem. While IntServ treated the traffic end-to-end, DiffServ applied policies on every hop. This is known as Per-Hop Behavior (PHB). Unlike IntServ which worked on a per-flow basis, DiffServ works on aggregated flows. Packets can be classified using Differentiated Services Code Points (DSCP) bits contained in the packet header. All the packets with the same header receive the same treatment, irrespective of the flows they are part of. Unlike IntServ, DiffServ thus becomes easier to implement and maintain without much overhead of message passing and global agreement.

Once the packets have been classified, corresponding PHB's can be applied to those packets. PHB refers to the packet scheduling, queuing, policing, or shaping behavior of a node on any given packet belonging to a DSCP aggregated packets. There are four standard PHB's that can be applied using DiffServ.

- **Default PHB:** Default PHB simply means that a packet marked with DSCP value of 00000 will be treated as a traditional best-effort traffic at every DS-compliant node. Also if a packet has no DSCP marking it will be mapped to the default PHB.
- **Class Selector PHB [30]:** This PHB is defined to maintain the backward compatibility with any IP precedence scheme that may have been applied to a packet.
- **Assured Forwarding PHB [31]:** Assured Forwarding is similar to the controlled load service of IntServ. It assures delivery as long as the traffic does not exceed a particular rate.
- **Expedited Forwarding PHB [32]:** Packets marked for expedited forwarding experience less delay, less jitter and low loss. These packets are given priority queuing over other traffic. Flows using EF compete with each other.

DiffServ is more scalable than IntServ and thus is used in the real internet. It can be used to give priority to one class of flows to other. However, it cannot provide end-to-end QoS guarantees. While IntServ and DiffServ are the general QoS ideas on the internet, all of it is not yet available on SDN infrastructure. Let us now dive into the QoS capabilities of present day SDN and look at some of the related research.

### 2.5.3 Related Research in QoS

Quality of Service in SDN and OpenFlow, in particular, is said to be limited as so far it has only realized two features namely queues and meter tables. There are no defined standards like the IntServ and DiffServ in SDN. One one hand, it adds to the flexibility of the architecture and gives the network administrators freedom to implement their own QoS algorithms. On the other hand, it makes it difficult to implement QoS. This lack of QoS models has inspired many researchers to propose models for achieving QoS in SDN. These models have explored a wide array of areas which can help improve performance and QoS in SDN. These areas range from monitoring to resource reservation to different routing algorithm. This section will discuss some of these relevant research for proposing QoS models for SDN. Karakus *et al.* [33] provides a comprehensive picture of QoS related literature in SDN-OpenFlow networks. It first provides an overview of the SDN framework and OpenFlow protocol. It describes briefly the metrics considered for QoS in networks like bandwidth, jitter, delay, and loss. The paper then presents the QoS features that have been implemented in the OpenFlow protocol over the years. It talks about the integration of MPLS labels, traffic classes, OF-CONFIG, counters, meter actions and monitors into the OpenFlow protocol over various releases. The paper claims that despite a large volume of work, QoS has not been achieved entirely and attributes it to the manual configuration of these systems. In conclusion, it states that the emerging applications on the internet have raised a greater demand for QoS guarantees and that SDNs features make it a valid candidate for providing such guarantees.

The QoS efforts in SDN can be categorized into seven categories as represented in figure 2.6:

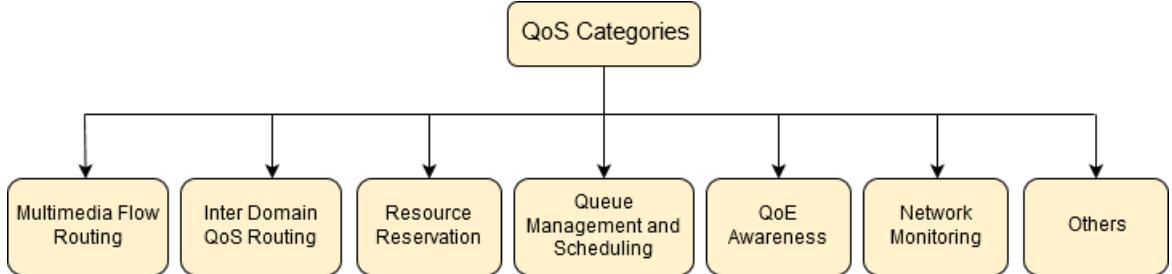


Figure 2.6: Categories of QoS efforts

We choose relevant research in every category of QoS to understand the bigger picture of the entire QoS effort undergoing in the community. Figure 2.7 summarizes and categorizes the various research that we are going to discuss in the section.



Figure 2.7: Relevant Research in QoS in SDN

Let us discuss the relevant research in different categories.

## **Monitoring:**

W. Kim *et al.* [34] presents the first ever OpenFlow monitoring system for ONOS controller called OFMon. The paper first describes how SDN had a bottleneck problem with the centralized controller and how ONOS came along with a distributed controller. It then describes why a monitoring system is needed in an SDN controller and then proposes OFMon. The OFMon architecture comprises of 4 main components operating at different layers of the ONOS architecture. A database running at the provider layer keeps a record of collective monitoring of OpenFlow messages. An OFMonitor component monitors OpenFlow and records all the messages into the database as well as prints them into ONOS logs for the network administrator to access directly. OFMon provides two modes of access to the network administrator. A CLI-based application and a web-based GUI application, both of which run on the application layer of the ONOS architecture. These applications access the monitoring results from the database created earlier. It uses several inbuilt interfaces and overrides the methods to add additional OFMon functionality into ONOS. The paper first provides an explicit detail of all the classes and interfaces in the ONOS Device Subsystem and then builds their implementation on top of it. It adds additional classes like `OFMonitor`, `OFMonCommand`, `OFMonUIComponent` and a database store with `<DPID, Results>` tuples. All these classes extend and implement ONOS builtin classes and interfaces.

The paper then presents experiments for performance evaluation of ONOS with and without OFMon and find out that OFMon does add the monitoring overhead on ONOS and that the overhead increases with increase in the number of switches. The experimental results show that OFMon causes a monitoring overhead of maximum 4% of CPU usage and maximum 6% of memory usage. This overhead might be useful for use cases where monitoring is important irrespective of performance, but in cases where performance is important, this can be a huge turn-off.

## **Inter-AS Routing:**

C. Xu *et al.* [35] claims to present a novel approach for serving QoS by Dynamic Resource Allocation in SDN. The authors present a framework for implementing QoS by classifying flows into different classes and then implementing priority queuing to allocate required resources to appropriate flows. For their model to work, the controller must have a good knowledge of the network utilization state including load, delay, and jitter. They create a separate thread to periodically monitor the network utilization. They also implement a proxy to reduce frequent communication between the switches and the controller that may lead to extra traffic. The authors delete the original flow table entries to increase efficiency and create different levels of flow. These levels are QoS Level-1, QoS Level-2, and Best-Effort Flow. This categorization can be done by many trivial ways. They first choose the best path for a request by solving it as a constrained shortest path (CSP) algorithm. They choose packet loss and delay to be the target variables they are applicable to wide variety of applications. The paper then goes on to describe the algorithm in detail and solves the equations to come up with the shortest path for requests. The authors then implement a priority queuing technique. Flows are divided into different levels and are assigned to particular queues based on their priority, as long as the link of the given queue can satisfy the guarantee requested by the application. They then test their mechanisms on a simulated network on Mininet and also on a physical network with real hardware switches and controllers.

They flush the network with a good amount of best-effort traffic after it has released the level-1 and two traffic and study the fluctuation of the packet loss, throughput and delay variation. And then when they turn on the QoS mechanism at 40 seconds, the phenomenon of disturbance and fluctuation disappears. This proves the effectiveness of their QoS mechanism. The solving of the problem as a constrained shortest path algorithm is a good solution, but it also comes with a big question of choosing the right variable. They

choose delay variation and throughput for their solution, which is very specific. This can be, however, suited to anyones need by changing the variable as per the requirement.

S. Muqaddas *et al.* [36] presents an experimental evaluation of the bandwidth expenses of the inter-controller traffic. Early implementations of SDN used the concept of a centralized controller that would control the entire network connected to the controller. It was later found out that it formed a central point of failure and suffered the problems of communication overheads. The research then shifted towards a physically distributed but logically centralized controller. This ensured reliability and scalability but added the overhead of control traffic between the controllers to maintain consistency.

The paper explains the importance of timely updating topology information to avoid undesirable results in routing like loops etc. The reactivity of the controllers also depends on the delay in inter-controller communications. This is often neglected in the literature. The paper then describes the various consistency concepts and explains in details the Consistency, Availability, and Partition (CAP) theorem. The authors then describe two type of consistency models implemented in ONOS. First, the Eventual Consistency Model which is implemented for topology discovery in ONOS and second, the Strong Consistency Model which is implemented for mastership store in ONOS. ONOS uses 2 shared data stores to maintain coordination among the various controllers. They are the Mastership Store and the Network Topology Store. These are implemented consistently using the Anti-Entropy Protocol for topology discovery and RAFT Protocol for maintaining a shared log.

The author uses 2 main scenarios for their study. One with 2 controllers and one with 3 controllers connected in a cluster of controllers. They used Ubuntu 14.10 server with ONOS 1.2 installed inside a Linux Containers (LXC). Network topology was emulated using Mininet 2.1.0. The experiments were conducted using different topologies. The

change in the bandwidth utilization was studied with adding additional nodes to one controller at a time. Results show that in both the cases of the addition of isolated nodes as well the addition of linear nodes to controller A, the traffic from A to B increased linearly. This is due to the hard consistency protocol RAFT for mastership. In a 3-controller scenario, similar traits were found, but, the bandwidth utilization was less than that in a 2-controller model, and this was due to the softer Anti-Entropy consistency guarantee that used a random selection of controller to update topology information. Considering that this is one of the very few research that talks about inter-controller traffic and its effect on the overall responsiveness of the network, it is a very valuable addition to by literature review. The inter-controller traffic and its understanding will play a huge role in delivery QoS in SDN networks when it is deployed in large scale in the real world.

M. Karakus *et al.*[37] paper proposes an architecture for Inter-AS QoS routing in SDN. The paper proposes a hierarchical framework of the network for QoS delivery and to the best of their knowledge is the only such work at the time. Scalability in SDN is a problem that has hindered the complete evolution of SDN. A number of factors affect the scalability of the control plane. The placement of the controllers in a network, distributed or centralized controller and various optimization techniques make a difference to the scalability of an SDN. The global internet is mostly divided into different autonomous systems (AS) and communication between them is the key to the scalability issue.

The authors then present the architecture of their proposal after an extensive literature review. The architecture consists of 2 levels. The first level is the Network-Level (bottom-level) which consist of independent domains and ISPs that are controlled by a single independent controller. The second level is the Broker-Level (upper-level) which consist of a super controller that acts as a supervisor to the local controllers. Every local AS domain controller advertise their reachable addresses to the broker, so the broker has the global knowledge of all the source and destinations in the network. The broker level

on receiving a request from the bottom levels will ask the source and the destination AS controllers to advertise all possible paths to the source and destinations along with the QoS metrics like bandwidth, jitter, etc. The requesting AS will also advertise the required QoS for the given request. At this stage, the broker has the global knowledge of all the possible routes in the network. And thus, the broker will assign the route that is best suited for the given request and its QoS parameters.

The authors then compare the results of a hierarchical structure of controller to a non-hierarchical structure of controllers to check if this proposal increases the scalability of the inter-AS QoS routing in an SDN network. The results show that the number of messages served by any controller in the network is reduced by 50% by deploying the hierarchical structure of controllers. However, having a broker to administer all the controller in various networks makes the broker a central point of failure which is undesirable. Also, the performance of the entire network might be affected if the broker acts as a bottleneck. There might also be an added latency to the responsiveness of the controller due to the introduction of the broker. However, these are the issues that the SDN community has been facing as a whole.

### **Multimedia Flow Routing:**

H.E. Eglimez *et al.* [38] proposes a novel QoS management design called OpenQoS to provide end-to-end QoS delivery for multimedia flows. The paper explains the architecture and modules of the OpenQoS controller. The main interfaces of the controller design are the controller-forwarder interface, controller-controller interface, and the controller service interface. The controller performs the functions of topology management, route management, flow management, route calculation, call admission and traffic policing. The call admission function admits or denies a requested QoS if it cannot be satisfied. The controller recognizes the multimedia flows by observing one or the other headers. Traffic

class header of MPLS, Type of Service header of IPv4 and Traffic Class header of IPv6 all can be used to classify a multimedia flow.

The paper then moves on to explain the various algorithms that it uses to calculate QoS routes. An up-to-date global network information is essential for making the calculation. The paper uses the Lagrangian Relaxation Based Aggregated Cost (LARAC) algorithm to efficiently calculate the cost of the routes. The route management function updates the QoS parameters, topology management function detects the topology and the route calculation function runs the LARAC algorithm. The authors use the 70% utilization of the network to be congestion. The authors test the controller on a physical network with VLC running on the client, and the server side and they send the same data with and without OpenQoS routing. The results show considerable improvement in delivery when OpenQoS is used. They then discuss some of the future direction of work in the area like video streaming with MDC, Load Balancing in CDNs and Cross Layer Design on the internet. In conclusion, OpenQoS is a novel approach to providing an end-to-end guarantee for multimedia delivery without affecting the other flows and can inspire future works in this direction.

H.E. Eglimez *et al.* [39] is a paper that builds on the previous one. It presents an algorithm-driven approach to providing QoS in adaptive video streaming in SDN using OpenFlow protocol. It uses different scalable encodings for multimedia video and streams them through different level-1, and level-2 QoS flows.

The authors describe three different kinds of flows for their system. QoS Level-1 are the flows that are dynamically routed with the highest priority. QoS Level-2 will be given priority only after level-1 traffic is fixed. The rest of the traffic follows the best effort delivery routing without dynamic routing. They encode the video using MPEG-4 SVC (scalable) encoding. This helps them to divide the video into a base layer which is

sent with the highest priority as level-1 flows to facilitate continuous video playback, and the layer-2 enhancement layer is sent either via best effort delivery in 1 st approach and level-2 priority dynamic rerouting in the 2nd approach. The authors solve the problem of dynamic routing as a constrained shortest path problem. For the CSP they choose the cost metrics as the delay variation as it is best suited for video streaming applications. They solve the problem using Dijkstra and LARAC algorithms and find out that LARAC is better suited for real-time dynamic routing as there is a trade off between optimality and runtime of the algorithm. These can be controlled by relaxing some of the controls of the algorithm. The results show that there is a significant improvement in the quality of the streaming video by using the adaptive dynamic routing in OpenFlow networks. This solution also causes a negligible disturbance in the best effort traffic as it does not utilize resource reservation mechanism but only dynamic rerouting. Both the approaches used in the paper have their trade offs and depending on the requirement of the situation either approach can be used.

## **Resource Reservation and Queue Management**

K. Govindarajan *et al.*[40] presents the implementation of a QoS system for SDN-based cloud infrastructure called QoS Controller (Q-Ctrl). It is a system that can execute in an SDN equipped infrastructure in different environments like virtual overlay networks, physical networks and simulated networks on Mininet. The paper first talks about SDN and OpenFlow and how these technologies have made it possible to control the traffic flowing through the network. The authors then summarize some of the related works and assert how their work is different because Q-Ctrl is the first QoS solution that can control different environments like direct, controller and simulation. Q-Ctrl creates network slices and assigns different traffic applications into dynamically created slices to satisfy QoS requirements. The paper presents the Q-Ctrl architecture which consists of two components. A QoS Manager, which receives the QoS application requests and schedules

the request to suitable network links. Moreover, the Network Topology monitoring that interacts with the Topology Manager and updates the Topology Information repository. The architecture uses Floodlight Controller which can provide services such as QoS, Firewall, and Load-Balancing, etc. The proposed Q-Ctrl architecture operates in 2 modes. The Direct Mode, in which the QoS controller directly talks to the OVS switch using OVS connector by injecting QoS flows in the switches ovs-ofctl libraries. Moreover, the Controller Mode, in which the QoS controller talks to the SDN controller using SDN connector. Injection is done using Curl and Rest API. The authors then explain the QoS Life Cycle which consists of the Queue Creation, QoS Flow Addition, QoS Flow Modification, QoS Flow Deletion followed by Queue Deletion. The paper then presents experiments on video streaming showing the effects of various QoS flow injection in online video streaming application.

The paper, however, does not talk about the effect of such QoS flow injection on the rest of the network. It also does not address the possibility of 2 competing QoS demands made at the same time. Since Q-Ctrl makes slices of the network to provide end-to-end guarantees, the network cannot be infinitely divided into slices while catering to the required QoS. This poses a question on the scalability of the system, especially when it is to be deployed on cloud infrastructure where thousands of users could be demanding the same content at the same time with same QoS guarantees. It mentions testing Q-Ctrl with increasing number of hosts as future work, which a step in the right direction. Also, it mentions the exploration of integrating Network Prediction Engine for Q-Ctrl which would be an interesting research to read.

Ishimori *et al.* [41] talks about manipulating the multiple packet schedulers in the Linux kernel to improve the QoS on SDN networks. There have been numerous attempts at bandwidth guarantees in SDN, but they are all done on the well-known FIFO scheduling of the Linux kernel. The paper proposes QoSFlow, a technique to implement different types

of schedulers on top of the default Linux kernel and associate them with a different type of flows. They highlight the fact that QoS is not only dependent on the traffic shaping capabilities of the network but also the ability to send the packets in correct order. QoS-Flow implements three packet schedulers: Hierarchical Token Bucket (HTB), Randomly Early Detection (RED) and Stochastic Fairness Queuing (SFQ). HTB performs traffic shaping by configuring a minimum and maximum bandwidth capacity. SFQ performs a round-robin treatment of flows so that every flow gets an equal chance at the transmission channel. It uses the Netlink socket channel to connect to the kernel. Netlink message is the type of message that Linux kernel accepts for network resource management. It then uses the inbuilt data structures like `OFPT_FLOW_MOD`, `skb-priority`, `SO_PRIORITY`, etc. to implement traffic shaping, packet schedulers and queuing. The paper then describes the difference between classful and classless queuing discipline. HTB is classful, and SFQ and RED are classless disciplines. They also implement versions of FIFO called Packet FIFO (PFIFO) and Byte FIFO (BFIFO).

The authors perform five different types of test with QoSFlow. These are measuring the Response Time, Switch Capacity, the impact of the Number of Queues, Bandwidth Isolation and QoE Evaluation. They present the results in every case. Primary of them being that the implementation has a constant overhead of 7% on memory usage and that there is no relationship between the number of queues and the CPU usage. Bandwidth isolation is possible as TCP and UDP did not interfere with each other. A different implementation of network operating systems may have a different built-in scheduler, and this solution may or may not have the same effect on every scheduler. It is still a very feasible and generic solution because it uses and manipulates the default Linux kernels schedulers, which is the base for almost all NOS present to date.

R. Durner *et al.* [42] presents a performance study of dynamic queue management in SDN. Dynamic queue management is widely studied by researchers as a mean to imple-

ment QoS in OpenFlow networks. Applications have been shown to benefit from a frequent change in network resource allocations; this is called Dynamic QoS Management. There has been some previous work regarding the analysis of performance, but the closest to this work uses the Mininet emulator for the study. The paper claims to be the first work that studies the performance of Dynamic QoS Management on real hardware and software switches.

The paper measures the impact on TCP flows when multiple queues are changed at runtime. It measures two fundamental QoS concepts called priority queuing and bandwidth guarantee. The authors use 3 different switches for their assessment. 2 real hardware switches, the NEC PF5250 and PICA8 P3290 and a software Open vSwitch. The paper describes in detail the various queuing disciplines like Classless and Classful queuing with examples like FIFO, Stochastic Fairness Queuing, and Hierarchical Token Bucket. The paper then describes its experimental setup. It uses 2 hosts, one TCP source and another TCP sink connected to a common SDN switch. One additional host runs the SDN Floodlight controller. `iperf` was used to generate traffic and data flow was recorded using `tcpdump`. The RTT of the network was  $< 1ms$ . Two TCP flows were routed via a bottleneck. The three different switches mentioned above have a different level of support for QoS mechanisms, and these differences are visible in the results of the performance analysis. The results examined Priority Queuing and Bandwidth Guarantees. A major observation was the presence of duplicate packets due to queue switching. The OVS switch with SFQ performed as one would expect it to behave while the other 2 hardware switches had varied results.

This shows the importance of considering the variety of switches in creating a QoS solution. Priority inversion also caused major differences in results regarding PQ. These guarantees, however, does not thwart the TCP connection at any time. While there have been a number of works in priority queuing and dynamic resource allocation, this is the

first work that analyzed the performance of such systems. This is also the only work that recognized the variety of switches to be a factor in performance. The paper shows with experimental results that the same systems and algorithms perform differently on different switches. This surely is a novel contribution to the literature.

### **Quality of Experience:**

Since the inception of SDN, there have been numerous attempts at ensuring QoS to the end users. The demand for QoS has also increased multi fold in modern times with huge file sharing, HD video streaming, and gaming applications becoming ubiquitous. One of the major technique introduced to achieve this goal is dynamic resource management. SDN provides means for applications and network to be aware of each other and resources can be allocated dynamically to the resources on demand. T. Zinner *et al.* [43] summarized various resource management mechanisms and then evaluates the impact of such mechanisms on two competing applications and user perceived Quality-of-Experience.

Various methods of resource allocation involve per-flow meters with fixed rate limit, priority queuing of the flows, and redirection of traffic away from congestion. Priority queuing utilizes different queuing disciplines like minimal forwarding rate or weighted fair queuing etc. For their experiments, the authors use 2 hosts connected to an Internet Gateway running a software-defined Traffic Shaper via a Controller and a Wireless Access Gateway. The hosts use two competing services, a video streaming service like YouTube and a file downloading server. The single link between the internet gateway and the wireless access gateway acts as the bottleneck on the network. The traffic shaper controls the data uplink from internet gateway to the wireless gateway. It uses Priority Queuing approach (PRIO) and Weighted Fair Queuing (WFQ) for their analysis. The metrics measured are throughput, retransmission and duplicated packets on the transport layer and the buffered playtime of the videos for application layer performance evaluation.

The impact of the queue size on the number of duplicate packets is seen to be negligible for queue sizes greater than 50 packets. The average throughput remains constant for all queue sizes. Small queue sizes lead to packet loss and retransmission and may result in decreased throughput. WFQ was found to have faster and more accurate bandwidth allocation than PRIO. WFQ also performed better on the application layer buffered playtime.

The number of research in the area of QoS and QoE has increased significantly in the recent time. This paper provides a numerical benchmark for different techniques, and one can use these numbers to plan the QoS mechanism they want to deploy. The implementation and numbers are that of the then current OpenFlow version 1.3 and one can understand that with the evolution of OpenFlow over time, these numbers may not hold completely true. The traits, however, should remain the same.

### **Other QoS Techniques:**

T. Y. Cheng *et al.* [44] talks about the famous controller placement problem in SDN. While there have been numerous other works in this area, these works have mostly targeted optimization of some other performance metrics like minimizing the communication delay between controllers, load distribution among controllers, etc. This paper looks at the controller placement problem with the perspective of guaranteeing QoS. The authors state that response time of the controllers is an important QoS parameter and consists of 2 components: the round-trip delay between controller and the switch (the uplink and the downlink) and the service time of the controller to serve a request (sojourn time).

The paper then formulates the controller placement problem. They use a graph representation for the network with each switch being a candidate for controller placement. They take an upper bound on the response time of the controller and then try to calculate the minimum number of controllers required to achieve the guarantee. They finally define

their problem as, "Given a network topology  $G$ , we are asked to place the minimal number of controller subject to the response time constraint  $t \leq \delta$ ." They then solve the problem using three different algorithms: Incremental Greedy Algorithm, Primal-Dual-based Algorithm, and Network-Partition based Algorithm. They solve each of the algorithms and come up with the expression to calculate the minimum number of controllers needed for the guarantee. They then test the algorithms with many simulations. They use the three heuristic algorithms on network topologies from the Internet Topology Zoo. The number of switches in the topologies ranges from 4 to 74. NOX controllers are used for the simulation, and the algorithms are written in Python programming language.

The average performance metrics show the incremental greedy algorithm to be slightly better than the other two. The complexity or the run time of the algorithms shows the network partition based algorithm to cost the least and thus have least response time. The authors use three different algorithms to solve the same problem and find out on a simulation that one performs better than the other. This avers the fact that any theory or formulation is only hypothetical until tested. The paper does not mention why they choose these 3 algorithms or if there are other alternatives available for this problem. Of course, no paper is exhaustive in its area, and we are not sure if the algorithms performed the way they did, depending on the controller used, the complexity of the program written and the language chosen to write the program. However, the paper does provide a very novel approach to solve the controller placement problem and to use it to guarantee QoS in SDN.

M.S. Seddiki *et al.*[45] presents a novel approach for QoS management for home broadband access networks called FlowQoS. It uses the concepts of SDN and applies it to home broadband access points to forward traffic through the appropriate rate shapers. The paper describes the need for such a system, presents an architecture and implementation for FlowQoS. The architecture has 2 main components, the *flow classifier* that classify the

type of the flows based on certain keys and the SDN-based *rate shaper* which shapes the traffic using virtual links in the home gateway. It states that no previous work in the field of QoS has been deployed in the home network and that QoS functions are complex and it needs an abstraction to be used by a home user. In FlowQoS the user should simply specify the high-level application that should be given a prioritized treatment and the rest is done on-the-fly. QoS requirement for home user usually involves video streaming, VoIP, etc. FlowQoS does on-the-fly identification of the flows and installs rules in the flow table of the data plane to forward those flows with the user specified priorities for those applications.

The Flow Classifier uses a flow tuple to identify flows on the fly. These tuples can consist of the IP-address, destination address, protocol, source port, etc. The first classification it does is whether the flow is HTTP or HTTPS based on ports. It also uses the DNS responses including A or CNAME records to classify flows. It uses the `libprotoident` library to perform application layer identification. This classification can be done using the first 4 bytes sent, first 4 bytes received, first payload size sent, etc. As the QoS enforcement is lazy, it does not prohibit the performance of the network. The second component is the Rate Controller. It uses a layer of 2 OVS along with OpenWRT to emulate a virtual network because several features of the OpenFlow protocol that would enable achieving the goal is not supported by the then current version of OVS. Each link between the 2 OVS corresponds to a different application type like video, web or gaming. This is implemented using Linux's `tc` utility. Configuring `tc` on these the virtual links is very like **OF-CONFIG**. Raspberry Pi was used the controller hardware, and POX controller was using to achieve the QoS system. Results show that the system improves the performance of adaptive video streaming and VoIP in competing traffic. However, the system requires specialized hardware which is often a limiting factor in widespread acceptance of any system. The system is, of course, preliminary and could be improved and integrated better with the existing hardware. The initial results show good performance without

being prohibitive and augur well for future work on the system.

#### 2.5.4 Summary

This section presented the ongoing research in the field of QoS in SDN. It educated us about the various type of techniques that can be used to implement QoS systems in SDN, various challenges faced by the SDN community and suggested plausible solutions to some of the most pressing issues. It demonstrated the need for QoS in the new era of software defined networks. While most of these research provided one or other form of achieving QoS, it also demonstrated the lack of QoS in the present day infrastructure of SDN. SDN aims to address the problem of flexibility in the present day internet architecture and provides a software driven approach for new techniques and protocols to thrive in its ecosystem. The biggest advantage of SDN is that it is easier to adapt to it and move away from the existing "rigid" internet setup. But, for SDN to replace the current architecture of the real world network, it needs to provide very fine-grained control to the network administrators to control the quality of services along with several other enhancements.

To replace the current internet architecture, SDN has to come with solutions to a number of problems. Some of them have been addressed in the literature review. For example, scalability, high availability and fault tolerance on the control plane has been addressed with the distributed design of ONOS, matching the speed of the hardware switches with their software counterparts is being addressed with DPDK, etc. However, quality of service capabilities of all the different components of SDN will also play a major role in the widespread adoption of SDN in the real internet. ONOS is one of the leading production ready controllers for SDN and is (sort of) leading the charge of SDN's adoption by ISP's. It will be an interesting study to explore ONOS's QoS capabilities and to understand how ready is it to replace the age old, tried and tested internet infrastructure.

## 2.6 This Project

This project aims at exploring the extent to which the QoS capabilities are supported by ONOS. In particular, it will try to explore the rate limiting and resource reservation capabilities of ONOS with different SDN components. While OpenFlow itself has matured to version 1.5, various switches and controllers have varying levels of support for OpenFlow features. For a switch or device to be called compatible with a particular version of OpenFlow, the OpenFlow Specification generally announces some features to be mandatory. While many other features are left optional for the switch vendors to support or not.

The project will explore which pieces of the entire architecture work with each other. It will primarily explore the egress queues and meter tables defined in the OF1.3 specification. How many and to what extent does ONOS support these features and how production ready the QoS capabilities of ONOS are.

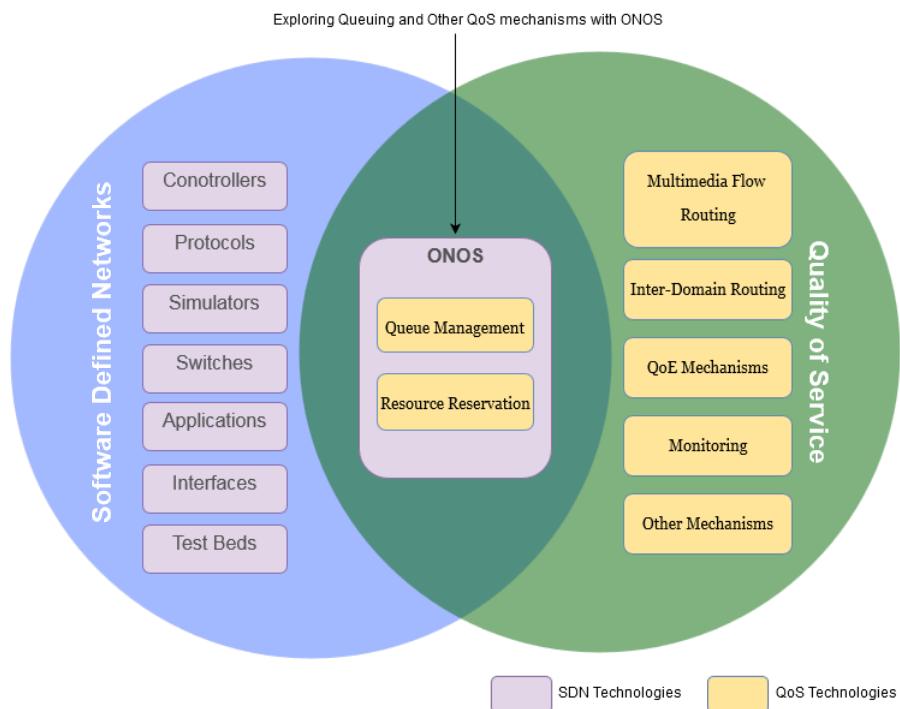


Figure 2.8: The area of SDN and QoS that this project aims at.

# Chapter 3

## Design

This chapter discusses the design and architecture of various technologies that will be used and examined in this dissertation. The discussion will start with QoS support and features in the OpenFlow protocol. In particular, we will discuss OpenFlow queues. It will then be extended to the Linux Kernel’s Traffic Control (TC) [46] mechanism which the Open vSwitch uses for implementing queues. The discussion will then move on to the OpenFlow meter tables which will also be explored in this dissertation. The chapter will then present an in-depth analysis of ONOS’s internal architecture and subsystems, as exploring ONOS is one of the primary objectives of this project. The chapter will then end with a discussion on how all these different technologies will be put together for the experiments of this thesis.

### 3.1 QoS in OpenFlow

OpenFlow has supported the notion of QoS since the beginning. However, the support has been limited. The earliest versions of OpenFlow OF1.0 - OF1.1 supported queues with *minimum rates*. OF1.2+ started supporting queues with *both minimum and maximum rates*. OpenFlow queues have broad support across the board. Most of the popular software switch implementations (e.g., OVS and CPqD OfSoftSwitch) and many hardware

vendors (e.g., HP 2920, IBM RackSwitch G8264 and Pica8 P-3290) support OpenFlow queues. OF1.3 introduced the concept of meter tables to achieve more fine grained QoS in OpenFlow networks. While queues control the egress rate of the traffic, meter tables can be used for rate-monitoring of the traffic prior to output. In other words, queues control the egress rate and meter tables can be used to control the ingress rate of traffic. This makes queues and meter tables complementary to each other. OpenFlow switches also have the ability to read and write the Type of Service (ToS) bits in the IP header. It is a field that can be used to match a packet in a flow entry. All these features collectively enable the network administrator to implement QoS in their networks.

### 3.1.1 Queues

As mentioned earlier, the OpenFlow protocol described a minimum rate limiting queue in OF1.0 and a minimum and maximum rate limiting queue in OF1.2. Although the queues are specified in the OpenFlow protocol, the OpenFlow protocol does not handle queue management on switches. The management of queues on the switches happens outside of the OF protocol, and the OF protocol itself can only query the queue statistics from the switch. The queue management task (creation, deletion, and alterations) are left to the switch configuration protocols like OF-CONFIG and OVSDB.

OVSDB uses many tables to manage the Open vSwitch. These tables include the flow tables, port tables, NetFlow tables and others. Similarly, it maintains tables for QoS and Queues. While most of the other tables are **root-set** tables of the OVSDB schema, i.e., the table and its entries are not automatically deleted if it cannot be reached. Thus the QoS and the queue tables exist and can be altered independently, whether or not they are referenced by a port. The relationship between some of the relevant tables is shown in figure 3.1. The port table is related to an interface table and a QoS table. The relation with the interface table is mandatory, meaning that each port has to be associated with

an interface. The relationship with QoS, however, is optional. A port may exist without a QoS setting attached to it. A port can have at most one QoS attached to it while a QoS table can have multiple queues assigned to it.

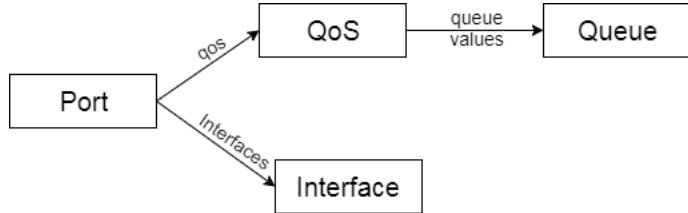


Figure 3.1: OVSDB Table Relationships

Once the QoS and queues have been setup on a switch, flows can be directed to a particular queue using the OpenFlow `set_queue` action. This action will forward the flows that match the matching criteria to the mentioned queue. If more than one flow goes through the switch at the same time, the aggregate rate of the flows will be controlled at the egress according to the defined `min_rate` and `max_rate` by the queue. Let us dive a little deeper on how the queues are implemented in the OpenFlow protocol and OVS.

The OpenFlow Specification [27] states the following properties about queues:

- **`min_rate`**: This property defines the guaranteed minimum data rate for a queue. If the `min_rate` property is set, the switch will prioritize this queue (and any flow forwarded via this queue) to achieve the mentioned minimum rate, at the cost of other flows rates. If there is more than one queue in one port, with total `min_rate` higher than the capacity of the link, the rates of all those queues are penalized. The capacity is shared proportionally based on each queues `min_rate`.
- **`max_rate`**: Maximum data rate allowed for this queue. If the actual rate of flows using this queue is more than the specified `max_rate`, the switch will delay packets or drop them to satisfy the `max_rate`.

While OpenFlow specifications mention these guidelines for the OpenFlow compatible switches, it is left to the switch implementation to realize these features. The Open vSwitch, which is based on Linux, uses the Linux Kernel's Traffic Control (TC) program to implement queues.

### 3.1.2 Linux Traffic Control

Linux Traffic Control (`tc`) is a Linux utility used to configure traffic control in the Linux Kernel. TC can be used to achieve the following in the Linux kernel:

- **Traffic Shaping:** It can be used to shape the transmission rate of the traffic going through a Linux server or any other devices. It can also smooth out any bursts of traffic for better network behavior.
- **Scheduling:** By scheduling the packets, it is possible to achieve better network behavior during bulk transfers. Reordering and scheduling of packets can also be called prioritizing, which is a widely accepted phenomenon in QoS.
- **Policing:** Several network policies can be implemented in TC. This policing occurs at ingress.
- **Dropping:** Traffic exceeding the defined bandwidth can also be dropped either at ingress or egress, based on usage.

TC uses three types of objects to achieve this: Queuing Discipline (Qdisc), classes and filters. Whenever the kernel needs to send any traffic to an interface, it enqueues the traffic into a qdisc. Which is then sent to the interface by the qdisc later. A simple qdisc is a simple FIFO queue. Classes and filters are used to implement more sophisticated queuing disciplines like the classful and the classless queuing disciplines. The OVS has two classful queuing disciplines that are used in OVS, they are Hierarchical Token Bucket (HTB) [47] and Hierarchical Fair Service Curve (HFSC) [48]. Both these systems are

Hierarchical Queuing Disciplines and allow bandwidth borrowing. The main difference between the two is that HFSC balances delay-sensitive traffic against general traffic. Since in our experiments we are only concerned about the bandwidth and not delay, HTB will be used in this thesis for queue management.

## Hierarchical Token Bucket (HTB)

In the hierarchical token bucket algorithm, tokens are generated at a fixed rate, then stored in a fixed capacity bucket. Packets can only be dequeued or sent to an output port if there is available token in the bucket. HTB is a queuing discipline that uses the concepts of multilevel token buckets to allow granular control over the outbound bandwidth on a given link. It is intended to be a replacement for Class Based Queuing (CBQ) which was a standard in older TC implementations. Within an HTB instance, multiple classes may exist. Figure 3.2 demonstrate a simple HTB hierarchy for solving the following problem: *"Two customers A and B are connected to the internet via the same connection. We need to allocate 40Kbps and 60 Kbps to A and B respectively. As bandwidth needs to be subdivided into 30Kbps for WWW and 10Kbps for other applications. Any unused bandwidth should be shared among the two customers."*

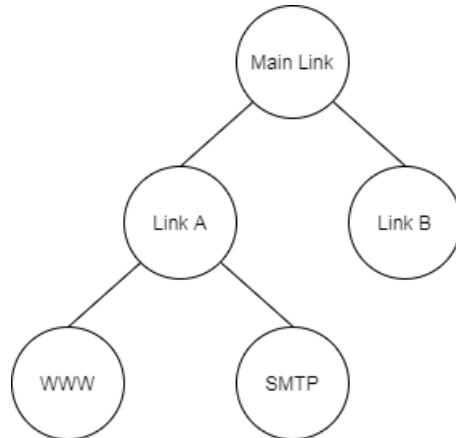


Figure 3.2: Sample HTB class hierarchy

In this example, 40Kbps is assigned to A. If As bandwidth usage for WWW is less than the allocated bandwidth, the unused bandwidth will be used for other traffic if demanded. The sum of As WWW and other traffic will not exceed 40Kbps. If A were to request less than 40 kbps in total then the excess would be given to B. In the OVS implementation of the OpenFlow Queue, a child class of a root can not have any children. So, there exist only two levels of hierarchy. HTB classes have several important properties.

- **rate:** Maximum guaranteed rate for this class and its children. It is equivalent to Committed Information Rate (CIR).
- **ceil\_rate:** the Maximum rate at which this class is allowed to send.
- **priority:** Defines the priority of the class. Classes with higher priority (priority 0 has the highest priority) are offered idle bandwidth first. This prioritization should not affect other classes guaranteed rate.

Queues are created using the ovs-vsctl command in OVS. This command creates an entry in OVSDB and then implements it in the switch using Linux TC. An example of creating QoS and queues in an OVS port is shown below:

```
~$ ovs-vsctl set port eth1 qos=@newqos -- --id=@newqos create qos
type=linux-htb other-config:max-rate=10000000 queues:1=@newqueue1
queues:2=@newqueue2 -- --id=@newqueue1 create queue
other-config:min-rate=4000000 -- --id=@newqueue2 create queue
other-config:min-rate=4000000
```

The example above creates a QoS and queues in port eth1. In OVSDB, it is manifested as new entries in QoS table and Queue table. OVSDB then puts a relation between entry eth1 in Port table and the newly created QoS entry. This relation indicates that eth1 should behave according to rules stated in this QoS. During this process, the switch invokes the TC application to create qdisc and classes in the background.

### 3.1.3 Meter Tables

Meter Table is a new feature that was introduced in the OpenFlow protocol in OF1.3. Unlike queues which are used to control the egress rate, meter tables are used to monitor the ingress rate of the flows. A meter table contains meter entries, defining *per-flow meters*. Meters are associated with flows rather than ports. Flow entries can specify meters in its instruction; the meter controls the aggregate rate of all the flow entries associated with it. Per-flow meters enables OpenFlow to implement different QoS techniques such as rate limiting. It can be combined with per-port queues to implement complex QoS frameworks like DiffServ. This exactly what we want to test on ONOS in this thesis. Let us understand how meter tables work.

A meter table entry consists of the following components as shown in table 2.4:

- **Meter Identifier:** It is a 32-bit unsigned integer uniquely identifying the meter.
- **Meter Band:** An unordered list of meter bands where each meter band specifies the rate of the band and contains the instruction to process the associated packets.
- **Counters:** It is a simple counter that is updated every time a packet is processed by a meter. It is mainly for statistical purposes.

There are two band types to define how a packet would be processed; these are *drop* and *dscp-remark*. Bands works on the traffic that exceeds the defined rate. The drop band drops the packets that exceed the rate specified in the band's rate. It can be used to define a rate limiting band. The DSCP remark band, on the other hand, is used to increase the drop precedence of the DSCP field in the IP header. It can be used to implement DiffServ. For example, if there are 100 Mbps flows with ToS value 64 via a meter with 80 Mbps rate, the ToS bits of the excess 20 Mbps traffic will be remarked to 32. The rest of the 80 Mbps traffic stays the same. This settles the traffic into a flow between the bands. The same table can use multiple meters, exclusively.

## 3.2 QoS in Software Switches

While the OpenFlow specification has progressed to version 1.5.1 as of August 2017, the most popular OpenFlow switches are yet to support all the latest features defined in the specification. This is down to the fact that the OpenFlow specification makes a list of features that are mandatory for a switch to implement to be called an OF compatible switch. However, the specification leaves some features as optional. These features are left to the decision of the switch maker to implement. This has led to an unclear state-of-the-art in the field of QoS support in OpenFlow switches.

Most of the commercially produced production-ready hardware switches support majority of QoS features. This is understandable as they are expected to be deployed in real world networks which require fine grained QoS. The software switches, on the other hand, are mostly open source implementations and are lagging behind in implementing all these features. The Ryu SDN Controller team regularly tests various popular switches for their OpenFlow compatibility and releases a certification. The list can be accessed at [49]. Hardware switches are beyond the scope of this project. We will discuss the capabilities of the three most popular software switches namely the Open vSwitch, the CPqD OfSoftSwitch, and the Lagopus Switch. The following table 3.1 summarizes the support available for various switches for OF QoS features.

Table 3.1: QoS Support in Popular Software Switches

Name	Queues	Meter Tables
Open vSwitch	YES	NO
Lagopus Switch	NO <sup>1</sup>	YES
CPqD OfSoftSwitch	YES	YES

---

<sup>1</sup>Partial untested implemented.

## 3.3 ONOS: Architecture and Subsystems

The purpose of this thesis is to explore ONOS' QoS capabilities. To work with ONOS, it is critical to understand its internal functioning and architecture. The way different modules of ONOS are arranged. The API's that it exposes for us to exploit, the way it implements the routing, forwarding, and other subsystems. This section explores ONOS controller architecture in detail and is widely based on the ONOS Wiki[50].

### 3.3.1 System Components

The ONOS project takes great pride in calling ONOS a highly *distributed and modular* SDN controller. Indeed most of the features of ONOS are arranged in a modular fashion which is largely independent of each other's internal implementation as long as they expose sufficient APIs for the depending modules to work upon. The system, in general, is divided into various services and subsystems. The high-level architecture, as most of the other controllers, is based on the general SDN architecture discussed in the previous sections. The figure 3.3 presents the general architecture of ONOS.

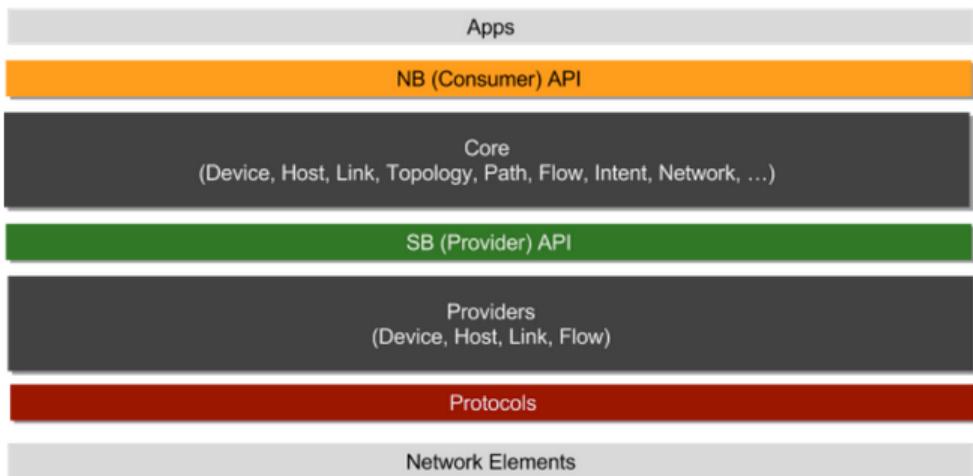


Figure 3.3: General ONOS architecture. [50]

The architecture suggests that the controller consists of the **core** elements which expose the **SBI** for the providers and the **NBI** for the applications on the control plane. The Providers use the SBI to communicate with the controllers and the protocols like OpenFlow to communicate with the network elements. During our communications with the people involved in the development of ONOS, we found out that this modularity not only helps keep the code organization clean, but it has also added benefits of simplicity for the developers. Most of the ONOS developers we talked to worked on their module without caring much about the internals of the other modules as long as the exposed APIs worked perfectly. ONOS defines a **service** as *a unit of functionality that is comprised of multiple components that create a vertical slice through the tiers as a software stack.* A **subsystem** *the collection of components making up the service.* The terms are often used interchangeably. ONOS defines the following primary services:

- *Device Subsystem* - Manages the inventory of infrastructure devices.
- *Link Subsystem* - Manages the inventory of infrastructure links.
- *Host Subsystem* - Manages the inventory of end-station hosts and their locations on the network.
- *Topology Subsystem* - Manages time-ordered snapshots of network graph views.
- *Path Service* - Computes paths between infrastructure devices or between end-station hosts using the most recent topology graph.
- *FlowRule Subsystem* - Manages inventory of the match/action flow rules installed on infrastructure devices.
- *Packet Subsystem* - Allows applications to listen for data packets received from network devices and to emit data packets out onto the network.

While there are a lot of features and services in ONOS, all are not relevant to this dissertation. The one that is the most important for this project is the Intent Framework.

### 3.3.2 ONOS Intent Framework

The intent subsystem allows applications to specify their network desires to the OS. This policy-based directive is called **intents**. The ONOS core accepts the intent request and translates them, using intent compilation, into installable intents. Installable intents can be actionable operations on the network elements. These actions can range from provisioning tunnel links to installing flow rules on switches to reserving optical lambdas and much more. The entire process of requesting and installing an intent can be represented as a state machine as shown in figure 3.4. Intents can be described regarding *network resources, constraints, criteria or instructions*. Intents are identified by both the unique `IntentId` and the `ApplicationId` of the application that submitted it.

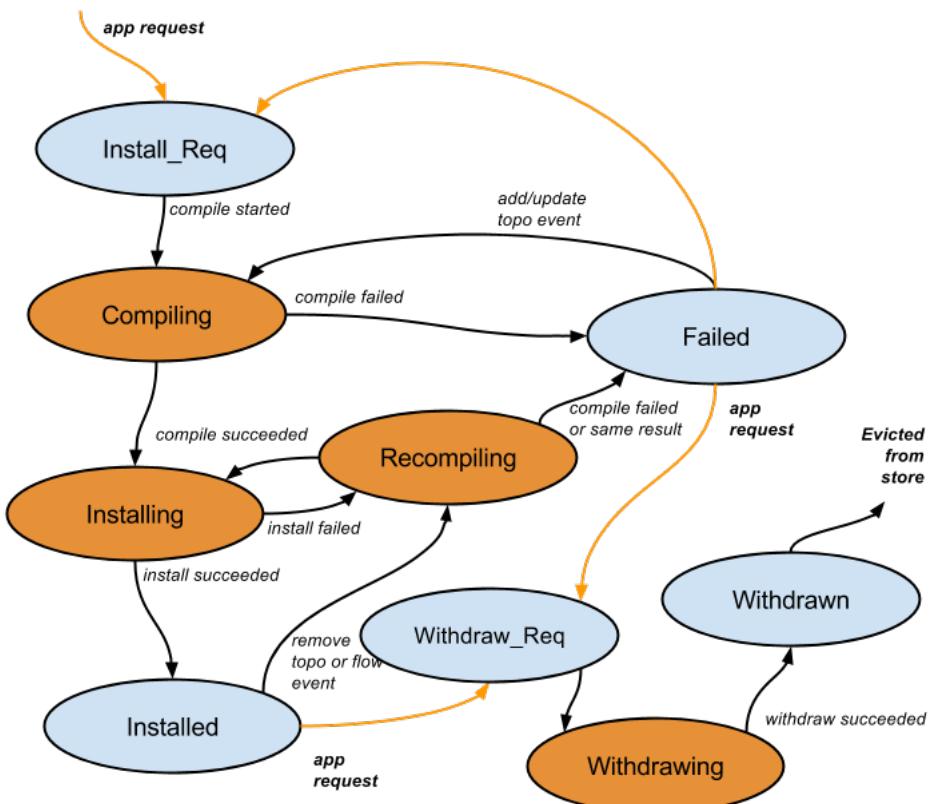


Figure 3.4: ONOS Intent Framework Compilation and Flow Installation. [50]

Any network action in an OpenFlow network is defined by the `FlowRule` installed on the network elements. Thus, understandably, the **intents** are also compiled into

`FlowRule`. Its a two-step process; the first step is the compilation of the requested intent into an installable intent by the `IntentCompiler` and the second step is the conversion of installable intents into `FlowRuleBatchOperation` by the `IntentInstaller`. The `IntentManager` coordinates the compilation and installation of `FlowRules` by managing the invocation of available `IntentCompilers` and `IntentInstallers`.

`FlowRuleBatchOperations` are handed off to the `FlowRuleService` to be written down as protocol-specific messages. In the case of OpenFlow, the `OpenFlowRuleProvider` is responsible for generating `OFFlowMod` messages from `FlowRules`, and sending it to the appropriate network switch. It relies on the OpenFlow subsystem for its message factories and `OpenFlowSwitch` references for this task. Intent forwarding relies on the `topology subsystem` to maintain the `intent subsystem` updated on the current network topology through the `Topology Listener`, and to perform the end-to-end path computation through the `Path Service`. When a networking event occurs and affects current paths, the Intent subsystem can react based on the information received on the `Topology Listener`, making decisions of either submit new Intents or withdraw the affected ones. Then, during the Intent compilation process, the `Path Service` is consulted for a best available path for the new end-to-end path. Another important subsystem in light of this project is the Flow Rule Subsystem which the Intent Framework uses to install flows into the network elements. Let us now take a brief look at this subsystem.

### 3.3.3 Flow Rule Subsystem

The `FlowRule Subsystem` manages the flow rules in the system and installs them on the devices in the network. It uses a distributed approach to flow table management. The master copy of the flow rules lies in the controller which is then pushed to the network devices. This reduces the additional workload of discovering information from the network every time a new request arrives. If a flow on a device is detected which is not consistent

with the master copy of the flow rules, those flows are removed from the devices. Flows in the ONOS system can exist in the following states:

- **PENDING\_ADD**: This state means that the flow rule subsystem has received the request from the application to install it but has not observed this on the device.
- **ADDED**: The PENDING\_ADD state transitions to the ADDED state when the subsystem observes the flow to be successfully installed on the device.
- **PENDING\_REMOVE**: When the flow rule subsystem receives a request to remove a flow but is yet to confirm if the flow has been removed from the device, the flows go in this state.
- **REMOVED**: When the subsystem receives confirmation from the device that the flow has been removed, the flow moves to the removed state. It will be removed from the store shortly.
- **FAILED**: the device indicated that the flow rule installation failed.

The flow rule provider is responsible for the collection of statistics on the installed flows. It reports these statistics to the `FlowRule Subsystem`. The `OpenFlowRuleProvider` uses the OpenFlow `FLOW_STATS_REQUEST` messages to query the stats and the device responds with `FLOW_STATS_REPLIES`. The flow rule subsystem then matches those flows to its copy in the store and update the statistics of those flows. During the update, if it detects flow that is missing on the network, it will reinstall it, and if it sees any extra flow that does not exist in the store, it will remove them.

## 3.4 Mininet Network Emulator

To evaluate ONOS' QoS, we need to test the implemented QoS policies on a network. Working on real world network or a network with many hardware switches is beyond the scope of this project. Mininet is a network emulator. As discussed earlier it provides

a virtual network on a computer and emulates real world network. It has been tested under many different scenarios and has proved to be a very efficient emulator [16]. It uses the Linux's network virtualization techniques like network namespace to emulate multiple hosts and can create complex networks in a single machine. It is compatible with most popular software switches and is regarded as the de facto network emulator. Mininet will be used with different switches to conduct our QoS experiments.

### 3.5 Measurement Tools

For this research, accuracy in the measurement is not a mandatory objective. Nevertheless, it should be precise enough to reveal general behavior and tendencies of the network traffic. Therefore, the following tools were selected for the measurement of the achieved bandwidth and analysis of traffic under different QoS settings: iPerf<sup>2</sup>, and Wireshark<sup>3</sup>.

#### iPerf

iPerf is a tool for live measurements on IP networks, in which its main purpose is to determine the maximum achievable throughput in the IP network. It can be used for other measurements like jitter, and packet losses. It is customizable to generate UDP and TCP packets of different MTUs and at a specified rate and interval of transmission.

This tool acquires its statistics based on the number of packets transmitted within the transmission time and interval. It comprises two elements, a client, and a server. The client generates the traffic to the server according to the packet specifications desired, and determines an average transmitted bandwidth; while the server receives the generated traffic, performs the statistics, and sends the results back to the client. iPerf works at the application level, which means that the throughput measured is not exclusive of the

---

<sup>2</sup>iPerf - The network bandwidth measurement tool. [online] Available at: <https://iperf.fr/>

<sup>3</sup>Wireshark protocol analyzer. [online] Available at: <https://www.wireshark.org/>

performance of the links, but also of the packet process through the TCP/IP model. Additionally, iPerf is executed at the user-space level of Linux OS, working at the top of the system architecture and using system calls to use the resources.

## Wireshark

Wireshark is a network packet analyzer that captures a packet in the network to display its TCP/IP layer information as detailed as possible, letting to examine its content for purposes like network troubleshooting, security examinations, protocol debugging and network protocol learning. This tool allows to capture live packets from a network interface (physical or virtual), displaying the packet data with detailed protocol information, and saving all packet captures for further studies and statistics. The captured data can be analyzed under different criteria, in which the information can be filtered by time stamps, TCP/UDP ports, protocols, TCP sessions, and more.

Since Wireshark is a tool that works at user-space, it uses the pcap library to allow Wireshark to capture packets at a lower level. This permits the capture of packets with a more precise timestamp since there is no additional delay caused by the internal communication process between the user-space and kernel-space levels. For this research, it is used the Wireshark dissector provided by a Mininet package, which enables the OpenFlow filter to capture OpenFlow messages and observe their message format in detail, including flow entries and group entries.

## 3.6 Summary and Experiment Design

SDN and QoS themselves are just concepts. A number of technologies, techniques, and applications have to work together to realize them. All the different parts of the puzzle discussed above in this section have to fit together to create an SDN network with QoS capabilities. All the different components discussed above are being developed in-

dependently without a strict common guideline, and thus they have different approaches towards SDN. They also have a different level of support and features for various SDN features. For example, ONOS supports features of OpenFlow version 1.5 while most of the software switches are stuck on OF1.2 or 1.3. Also, as all the features of the OF specifications are not mandatory, two devices supporting the same OF version can have a different set of features, and they may or may not work together. These are all the potential challenge in the design of the experiments to explore the QoS capabilities of ONOS.

While the finer details will change with different experiments, the overall design of the experiment will follow the general architecture of SDN as shown in figure 2.1. The control plane will run the modular and distributed controller, ONOS. The network will be emulated using mininet and one of the mentioned software switches, i.e., OVS, Lagopus or CPqD based on requirement and feasibility. The protocol for communication between the controller and dataplane will be OpenFlow. iPerf and Wireshark will be used on the application layer to test the effects of the QoS settings. For our experimental setup, we would be using one or two virtual machines running inside Oracle Virtual Box<sup>4</sup> depending on the requirement. The virtual machines will run the latest Ubuntu Desktop 17.04<sup>5</sup> with 4.10 Linux Kernel<sup>6</sup>. The general software requirements include general purpose tools and software for development like `git`, `curl`, `openjdk`. The complete list of software requirement is provided in table 3.2.

For our experiments, we would be creating different topologies and try to implement different QoS settings to the network elements. Moreover, then test the effects of the deployed settings. We will test the kinds and levels of QoS settings that can be deployed with ONOS. The details of individual experiments will be provided in depth in the next chapter on implementation.

---

<sup>4</sup>Oracle VM VirtualBox. [online] Available at: <https://www.virtualbox.org/>

<sup>5</sup>Ubuntu. [online] Available at: <https://www.ubuntu.com/>

<sup>6</sup>The Linux Kernel Archives. [online] Available at: <https://www.kernel.org/>

Table 3.2: Software Requirement for the experiments

Category	Software(s) Requirement/Used
Virtual Machine	Oracle Virtual Box
Operating System	Ubuntu 17.04 Desktop
Controller	ONOS, Ryu <sup>7</sup>
Network Emulator	Mininet
Software Switch	OVS, CPqD and Lagopus
SouthBound Communication	OpenFlow
Traffic Analyzer	Wireshark
Traffic Generator and Measurement	iPerf and iPerf3
<b>Additional Requirements:</b>	
ONOS	Git, Bash, Apache Maven, Buck, OpenJDK <sup>8</sup>
Mininet & Ryu	Python
CPQD OfSoftSwitch	GCC, G++, CMake and NetBee <sup>9</sup>
Java IDE	IntelliJ IDEA <sup>10</sup>

<sup>7</sup>Ryu Controller. [online] Available at: <https://osrg.github.io/ryu/>

<sup>8</sup>OpenJDK8. [online] Available at: <http://openjdk.java.net/>

<sup>9</sup>NetBee [online] Available at: <https://github.com/netgroup-polito/netbee>

<sup>10</sup>IntelliJ IDEA. [online] Available at: <https://www.jetbrains.com/idea/>

# Chapter 4

## Implementation

The previous chapters of the report have setup the context of the work presented in this dissertation, which is to test ONOS with different components of the SDN architecture for possible QoS implementations. To test ONOS' QoS capabilities we performed five different experiments where we tried to implement the available OpenFlow QoS features, i.e., per-port queues and per-flow meter tables.

This chapter describes the experiments, their implementation, results, and observations. This section will highlight the capabilities and gaps in QoS capabilities of ONOS. It will also introduce us to some practical problems in implementing QoS in SDN. These observations, in turn, will create the bigger picture for the current state-of-the-art of QoS and the direction in which the development of SDN is heading. The following sections describe the experiments in detail.

### 4.1 Experiments

The experiments section is arranged in the following manner. Each section first presents the rationale and the statement of the experiment, stating what we are trying to achieve. Then the implementation of that experiment is explained in details. The results of the

experiment are presented after that. Each section ends with a brief note on the critical observations of the experiment.

## 4.2 Experiment 1

An application developer on SDN should, ideally, not be worried about how the controller handles the components on the south side of the controller. The NOS provides northbound API's which the developer should use to program the underlying network as they wish. Most NOS, including ONOS, provides abstractions on the NBI to do so. We would like to test the ability of the application layer of ONOS to configure QoS queues on the network devices. Thus, an application running on the application layer of ONOS should be able to configure queues on the switches in the network.

### 4.2.1 Statement

*Write an application on the application layer of ONOS to configure per port queues on the network switches.*

### 4.2.2 Implementation

We decided to use a simple network of two hosts connected to a switch which is controlled by the ONOS controller. The network was emulated using mininet and Open vSwitch. The setup is shown in Figure 4.1. The ONOS CLI is based on the Apache Karaf which makes it extensible and one can add any new CLI application to the list of ONOS commands. We wrote a CLI application called `myqos` and installed it on ONOS.

ONOS uses the Java builder pattern which makes it easier and flexible to create new objects. It also exposes many builder classes for significant OS components. These classes can be used to build new queues and QoS as well. The APIs like `DeviceService`

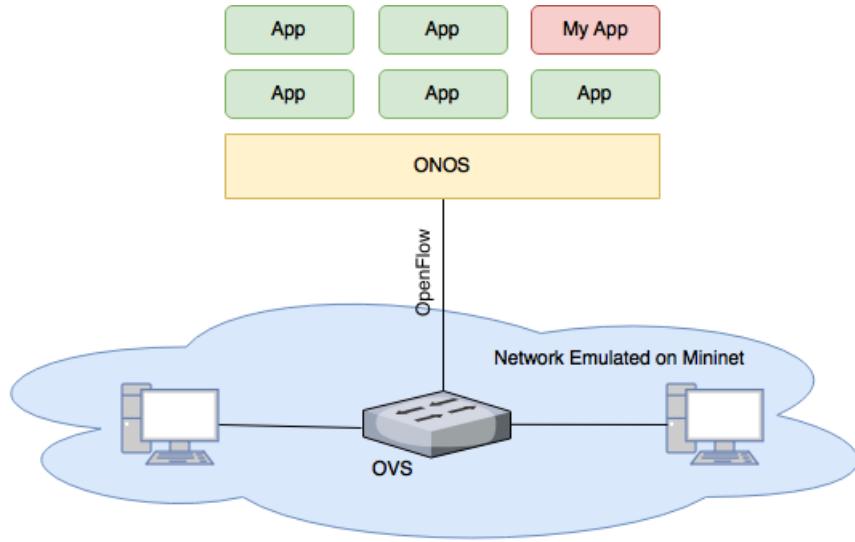


Figure 4.1: Setup for Experiment 1.

and the `PortDescription` allows discovery of the network elements to be configured. These can then be used with the northbound abstraction for device configuration. The two most important classes for this experiment are the `QueueConfigBehaviour` and the `QosConfigBehaviour` to configure QoS queues on the switch ports. The following code snippets show the usage of these APIs.

Figure 4.2 demonstrate how to create a new queue to be assigned to a QoS. As mentioned earlier, every QoS is attached to a port, and a queue is attached to a QoS. While each port can have only one QoS setting, each QoS can have multiple queues attached to it. Figure 4.3 shows how to build a QoS for the port.

```
QueueDescription queueDesc = DefaultQueueDescription.builder()
    .queueId(QueueId.queueId(name))
    .maxRate(Bandwidth.bps(Long.parseLong(rate)))
    .minRate(Bandwidth.bps(Long.valueOf(rate)))
    .burst(Long.valueOf(burst))
    .build();
```

Figure 4.2: Building a Queue

```

QosDescription qosDesc = DefaultQosDescription.builder()
    .qosId(QosId.qosId(name))
    .type(QosDescription.Type.HTB)
    .maxRate(Bandwidth.bps(Long.valueOf("100000")))
    .queues(queues)
    .build();

```

Figure 4.3: Building a QoS

The data structure “queues” that is passed to the queues argument in QoS description can either be a single queue or a map of multiple queues identified by their unique queue id; e.g., `Map<Long, QueueDescription> queues = new HashMap<>();`. The CLI command provides optional arguments to either add, delete or display the QoS settings on the switch. Based on the provided arguments, the corresponding actions can be taken. Figure 4.4 shows how to add the QoS settings to a particular port. Figure 4.5 demonstrates how to delete the QoS settings from the port.

```

if (type.equals("add")) {
    queueConfig.addQueue(queueDesc);
    qosConfig.addQoS(qosDesc);
    portConfig.applyQoS(portDesc, qosDesc);
}

```

Figure 4.4: Applying QoS Settings

```

} else if (type.equals("del")) {
    queueConfig.deleteQueue(queueDesc.queueId());
    qosConfig.deleteQoS(qosDesc.qosId());
    portConfig.removeQoS(portDesc.portNumber());
}

```

Figure 4.5: Removing QoS Settings

Additionally, the driver needs to be set to OVSDB explicitly. The driver can be configured using the command line tools of OVSDB. The command `sudo ovs-vsctl`

`set-manager tcp:127.0.0.1:6640` sets the OVS' manager to OVSDB on the port on which the switch is connected to ONOS in active listening mode. Also, some changes need to be made in the `ovsdb-driver.xml` file in ONOS to enable queues settings on the OVS. Moreover, the following ONOS apps need to be activated for OVSDB to work.

- `org.onosproject.openflow`,
- `org.onosproject.ovsdb` and
- `org.onosproject.drivers.ovsdb`.

For creating an app on ONOS, one has to follow the provided archetype. ONOS uses the maven build system. Once the code changes are complete, we need to run the command `mvn clean install` in the appropriate directory. The application archive file (.oar stands for **O**nos **A**Rchive) gets installed in the local maven repository when the build is complete. The application can then be installed into running ONOS instance by running the following command:

```
onos-app localhost install target/myqos-1.0-SNAPSHOT.oar
```

The application is now ready to use as a new command-line tool. Once the command was created and installed successfully on the ONOS cluster, we used the command to create, query and delete queues on OVS and observed the effects on the port.

#### 4.2.3 Result

The results of the experiment can be summarized in the following statements:

- The command was built, installed and run successfully. The modular design of ONOS made it easy to introduce a new feature to the ONOS CLI.
- The command did not affect the switches in the network. The command `ovs-vsctl list port` shows the Queue and QoS table to be empty.

#### 4.2.4 Observations

The time spent on debugging the application and learning why it did not work, gave us valuable insights into ONOS' state. While ONOS did provide APIs on the NBI to manipulate queues and QoS on the network devices, these APIs are simple wrappers that have been provided on the north side. The implementation of the wrappers, to interact with the southbound devices and protocols, is yet to be completed. The discussions with the ONOS developers confirmed the fact that the southbound implementation of these wrapper classes is a work in progress.

The OpenFlow protocol defines queues since version 1.1, but, the queues are not configured with the OpenFlow protocol. The OpenFlow protocol only defines the queues. It can point a flow to a queue and to query the statistics from a queue, but the queues are still configured outside of OpenFlow. Every OpenFlow switch usually provides a separate configuration protocol for precisely this reason. `ovs-vsctl` on the Open vSwitch, `dpctl` on the CPqD switch and the `Lagosh` command line on the Lagopus switch are all different approach to device management and are not interoperable. In an attempt to standardize the management protocol, the ONF introduced the `OF-CONFIG` protocol, but it has not seen widespread adoption yet. This means that the process of configuring Queues and QoS on switches remain a tiresome task, something that has to be done on individual switches manually. If a network consists of different types of devices, the problem becomes even bigger.

There are signs of positive steps being taken in this direction from the ONOS developer teams. These wrappers will soon be strengthened with the southbound implementation for device configuration. However, having different configuration protocols for various switch implementation makes the task challenging. A standardized protocol for all the OpenFlow switches will make the problem easier to address.

## 4.3 Experiment 2

The previous experiment explained that the queues are to be configured on the switches outside of the OpenFlow protocol and outside of the controller. The controller can, however, use the pre-configured queues on these devices. In this experiment, we try to test the abilities of ONOS to deal with queues on the switches. Capacity to direct the traffic to a queue and then observe the effect of such a configuration.

### 4.3.1 Statement

*Configure queues on the switches and test effectiveness with ONOS.*

### 4.3.2 Implementation

This is a simple test to see verify the efficiency of queues on ONOS with OVS switch. We setup two different queues with different rates on the OVS switch using the `ovs-vsctl` tool and then direct traffic on these queues using the `setQueue` option of the `add-host-intent` CLI command and test the effect using iPerf. We set one queue with the maximum rate of 5 Mbps and the other queue with a maximum rate of 3 Mbps. We then test the maximum achievable bandwidth between the hosts using iPerf. Figure 4.6 presents the results of the iPerf test.

### 4.3.3 Result

We see in Figure 4.6 that the queues are showing the desired effect on the network traffic. Queues work perfectly on OVS and ONOS. Different subsystems of ONOS responsible for querying the queues as well as setting the queues in flow rules work perfectly with each other.

```
root@peace:~# iperf -c 10.0.0.2
-----
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 85.3 KByte (default)
[ 13] local 10.0.0.1 port 59284 connected with 10.0.0.2 port 5001
[ ID] Interval Transfer Bandwidth
[ 13] 0.0-10.0 sec 957 MBytes 800 Mbits/sec
root@peace:~# iperf -s
```

Figure a: Performance at line rate of 1 Gbps.

```
root@peace:~# iperf -s
-----
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
[ 14] local 10.0.0.1 port 5001 connected with 10.0.0.2 port 33804
[ ID] Interval Transfer Bandwidth
[ 14] 0.0-13.4 sec 7.50 MBytes 4.71 Mbits/sec
[ 15] local 10.0.0.1 port 5001 connected with 10.0.0.2 port 33806
[ 15] 0.0-17.2 sec 5.88 MBytes 2.86 Mbits/sec
```

Figure b: Performance in queue set to 5 Mbps.

```
root@peace:~# iperf -s
-----
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
[ 14] local 10.0.0.1 port 5001 connected with 10.0.0.2 port 33804
[ ID] Interval Transfer Bandwidth
[ 14] 0.0-13.4 sec 7.50 MBytes 4.71 Mbits/sec
[ 15] local 10.0.0.1 port 5001 connected with 10.0.0.2 port 33806
[ 15] 0.0-17.2 sec 5.88 MBytes 2.86 Mbits/sec
```

Figure c: Performance in queue set to 3 Mbps.

Figure 4.6: Queues with ONOS.

#### 4.3.4 Observations

- Queues have been supported in the OpenFlow protocol since the beginning, i.e., version 1.1. This version of OpenFlow was released way back in 2011.
- It only fits to expect good support for this feature across the SDN stack. Both the ONOS controller as well as the OVS switch fully supports OpenFlow queues.
- The process of setting up the queues on the devices is still manual. It takes place outside of the OpenFlow protocol as well as outside of the ONOS controller. While this test was successful due to small test network size, it is not a feasible option when the network scales.
- There needs to be some procedure to configure queues from the control plane itself. As discussed in the previous experiment, steps are being taken in that direction by the ONOS team.
- Other controllers, Ryu for example, can configure queues on the devices using REST API calls. Ryu's abilities will be explored in a later experiment.

## 4.4 Experiment 3

Queues and Meter Tables are the only QoS features supported by OpenFlow as of now. While queues have been around for a very long time, OF1.3 introduced per flow meter table. Meter tables can be used along with per port queues to implement DiffServ in SDN. The purpose of this experiment is to test the ability of ONOS to implement DiffServ with meter tables and queues. A setting similar to the first experiment will be used where two hosts will be connected to one switch which will be controlled by the ONOS controller.

### 4.4.1 Statement

*Use OpenFlow Meter Tables and Queues together with DSCP markings to implement DiffServ on SDN.*

### 4.4.2 Implementation

The **Open vSwitch**, which considered the leading de facto OpenFlow software switch, does not support meter tables as of now. The current version of OVS, i.e., 2.7.1 does not have any implementation of meter tables. Neither the kernel-based nor user-space version of OVS supports meter tables. The support for meter table is being planned to be a part of the OVS version 2.8 release. Unfortunately, for this reason, the OVS switch could not be used to test meters with ONOS. While most of the OpenFlow compliant hardware switches support meter tables, it turned out that there is only two software switch implementation that supports the both the OpenFlow QoS features, i.e., Queues and Meter Tables. The two switches being the CPqD OfSoftSwitch13 and NTT's Lagopus switch. While ONOS does not provide any working mechanism to configure queues, it does provide an abstraction on the NBI to create and query meters on switches. Both the ONOS CLI as well as the REST API can be used to create meters on the switches. The details of meter table manipulation from ONOS will be discussed in detail in experiment 5. Let us first discuss the QoS capabilities of the available software switches in question.

As OVS comes with its dedicated configuration mechanism with OVSDB, **Lagopus**, comes with its CLI called **Lagosh**. The Lagopus switch can be run in two modes: the raw-socket mode and the DPDK mode. For this experiment, the performance of the switch is not a limiting factor. Therefore, we choose to run the switch in raw socket mode. We connected two hosts by using Linux Network Namespaces on a separate VM as shown in Figure 4.7. Once the network has been setup, the switch can be configured using Lagosh. Lagosh can be invoked on the Linux command line simply by invoking `sudo lagosh`.

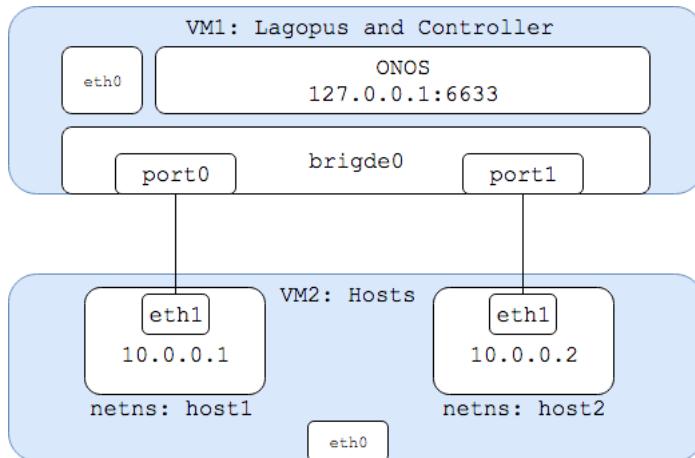


Figure 4.7: Configuring a network environment with Lagopus

Lagopus uses the “Lagopus Datastore”, where configuration and other data (ex: flows) used at runtime are stored. It uses a DSL syntax configuration file which is loaded to datastore at boot time. Different configuration objects like `ports`, `interfaces`, `controller`, `queues` etc can be added to the DSL file to configure the switch. A queue object can be set in the DSL file in the following format: `queue <queue-identifier> create <attribute value>`; for example:

```
queue queue01 create -type two-rate -id 1 -priority 50
```

It turned out during the implementation that the queue support for Lagopus is a work

in progress. While creating and querying the meter tables worked perfectly on the Lagopus switch, every time the switch was configured with a queue, it crashed. Upon more investigation and discussions with the developers at NTT, we found out that the implementation of queues on the Lagopus switch is not complete. While it has been included in the latest release, it is yet to be tested.

The **CPqD OfSoftSwitch** is the only remaining software switch implementation that supports both the features required for this experiment. The author of the switch mentions, as discussed earlier in section 2.2 that the switch is currently having issues regarding broken features and installation problem. The installation problem, as it turned out, was related to a missing dependency called Netbee. Netbee is an advanced library for packet processing which includes NetVM [51], NetPDL [52], and NetPFL<sup>1</sup>. Netbee is not a generic software on the internet, and it is created and hosted by the CPqD team themselves. It turned out during the project that all the links to download Netbee did not exist anymore. On raising the issue with the CPqD's developers, they addressed the issue and made it available.

The OfSoftSwitch13 switch works on the user level in the Linux environment. It can be used to create a network using mininet by using the argument `--switch=user` while starting a mininet network. The ONOS core installs a minimal number of flows, when it detects a network, to get the communication started. For this, it uses the different drivers for different switches, as detected by the system. The CPqD switch is correctly identified by the ONOS core, as can be seen in the image below. For the Stanford Reference OpenFlow switches like the CPqD switch, ONOS uses a driver called the `spring-open-cpqd`. Thus the ONOS core tries to install minimum flows on the device to get started.

---

<sup>1</sup>NetPFL is a high-level programming language that enables the manipulation of packets and header fields whose format is described through NetPDL.

```

onos> devices
id=of:0000000000000001, available=true, local-status=connected 2m33s ago, role=MASTER, type=SWITCH, mfr=Stanford University, Ericsson Research and CPqD Research, hw=OpenFlow 1.3 Reference Userspace Switch, sw=Aug 15 2017 11:08:20, serial=1, driver=spring-open-cpqd, channelId=127.0.0.1:39486, managementAddress=127.0.0.1, protocol=OF_13
onos>
onos>

```

These are the basic flows that are important for the network to start communicating. However, these flows never got installed on the switches. As described earlier in the section about the Intent Framework, the flow status goes into the PENDING\_ADD state if the flow exists in the ONOS system but the listener cannot confirm the existence of the flow on the switches. The flows installed by the system on the CPqD switch never exited the PENDING\_ADD state. Installing new host-to-host intents or point-to-point intents also remained in the pending state.

```

onos> flows
deviceId=of:0000000000000001, flowRuleCount=11
    id=7b0008db945b9, state=PENDING_ADD, bytes=0, packets=0, duration=0, liveType=UNKNOWN, priority=0, tableId=0, appId=org.onosproject.driver.SpringOpenTTP, payLoad=null, selector=[], treatment=DefaultTrafficTreatment{immediate=[OUTPUT:CONTROLLER], deferred=[], transition=None, meter=None, cleared=false, metadata=null}
    id=7b000f54354b, state=PENDING_ADD, bytes=0, packets=0, duration=0, liveType=UNKNOWN, priority=0, tableId=1, appId=org.onosproject.driver.SpringOpenTTP, payLoad=null, selector=[], treatment=DefaultTrafficTreatment{immediate=[], deferred=[], transition=TABLE:4, meter=None, cleared=false, metadata=null}
    id=7b0005a38058a, state=PENDING_ADD, bytes=0, packets=0, duration=0, liveType=UNKNOWN, priority=0, tableId=2, appId=org.onosproject.driver.SpringOpenTTP, payLoad=null, selector=[], treatment=DefaultTrafficTreatment{immediate[], deferred=[OUTPUT:CONTROLLER], transition=TABLE:5, meter=None, cleared=false, metadata=null}
    id=7b000f4b3b75c, state=PENDING_ADD, bytes=0, packets=0, duration=0, liveType=UNKNOWN, priority=0, tableId=3, appId=org.onosproject.driver.SpringOpenTTP, payLoad=null, selector=[], treatment=DefaultTrafficTreatment{immediate[], deferred=[OUTPUT:CONTROLLER], transition=TABLE:5, meter=None, cleared=false, metadata=null}
    id=7b000f5981dc1e, state=PENDING_ADD, bytes=0, packets=0, duration=0, liveType=UNKNOWN, priority=0, tableId=4, appId=org.onosproject.driver.SpringOpenTTP, payLoad=null, selector=[], treatment=DefaultTrafficTreatment{immediate[], deferred=[], transition=TABLE:5, meter=None, cleared=false, metadata=null}
    id=1000069e6cf51, state=PENDING_ADD, bytes=0, packets=0, duration=0, liveType=UNKNOWN, priority=40000, tableId=5, appId=org.onosproject.core, payLoad=null, selector=[ETH_TYPE:arp], treatment=DefaultTrafficTreatment{immediate=[VLAN_POP, OUTPUT:CONTROLLER], deferred=[], transition=None, meter=None, cleared=true, metadata=null}
    id=10000ef564c87, state=PENDING_ADD, bytes=0, packets=0, duration=0, liveType=UNKNOWN, priority=40000, tableId=5, appId=org.onosproject.core, payLoad=null, selector=[ETH_TYPE:lldp], treatment=DefaultTrafficTreatment{immediate=[VLAN_POP, OUTPUT:CONTROLLER], deferred=[], transition=None, meter=None, cleared=true, metadata=null}
    id=10000f7d9e4c, state=PENDING_ADD, bytes=0, packets=0, duration=0, liveType=UNKNOWN, priority=40000, tableId=5, appId=org.onosproject.core, payLoad=null, selector=[ETH_TYPE:bddp], treatment=DefaultTrafficTreatment{immediate=[VLAN_POP, OUTPUT:CONTROLLER], deferred=[], transition=None, meter=None, cleared=true, metadata=null}
    id=100002c78af51, state=PENDING_ADD, bytes=0, packets=0, duration=0, liveType=UNKNOWN, priority=5, tableId=5, appId=org.onosproject.core, payLoad=null, selector=[ETH_TYPE:arp], treatment=DefaultTrafficTreatment{immediate=[VLAN_POP, OUTPUT:CONTROLLER], deferred=[], transition=None, meter=None, cleared=true, metadata=null}
    id=100003dcaa79b, state=PENDING_ADD, bytes=0, packets=0, duration=0, liveType=UNKNOWN, priority=5, tableId=5, appId=org.onosproject.core, payLoad=null, selector=[ETH_TYPE:ipv4], treatment=DefaultTrafficTreatment{immediate=[VLAN_POP, OUTPUT:CONTROLLER], deferred=[], transition=None, meter=None, cleared=true, metadata=null}
    id=7b00003d70d9bb, state=PENDING_ADD, bytes=0, packets=0, duration=0, liveType=UNKNOWN, priority=0, tableId=5, appId=org.onosproject.driver.SpringOpenTTP, payLoad=null, selector=[], treatment=DefaultTrafficTreatment{immediate=[NOACTION], deferred=[], transition=None, meter=None, cleared=false, metadata=null}
onos>
onos> ■

```

Now, considering that the CPqD switch has been reported to be buggy in the past, this could have happened for many reasons. This could have happened due to a bug in the SpringOpenTTP implementation of ONOS, this could have been due to errors in the switch implementation, or this could be an issue of interoperability between the two systems. Either way, in our experiments, ONOS could not install flows on the CPqD switch, and thus the network never became usable.

#### **4.4.3 Result**

Queues with Meter Tables could not be tested on the ONOS platform. At least, not within the limitation of only using software switches. Many commercially available hardware switches support both the features. However, the only software switch that did support both the features did not work with ONOS.

#### **4.4.4 Observations**

Important observations from this experiment are listed below:

- Software switches, in general, have poor support for QoS features defined in the OpenFlow protocol.
- All the software switches are independent open-source projects with no common guideline. This raises the issues of interoperability.
- It is a challenge for controller implementations to support all the different types of switches. Different switch implementation requires different drivers. Writing and maintaining them is an overhead while the projects focus on introducing new features and improving the existing ones. This could be solved by having some standardization in the area.
- This remained untested in this experiment whether the issue was that of the interoperability or with the CPqD switch implementation. This can be addressed by testing the switch with some other controller. This issue is tested in Experiment 4 with the Ryu controller.
- Since the network never became operational, the state of support for meters in ONOS remained untested. This needs to be addressed by performing other *meters only* experiments. This issue is discussed in Experiment 5.

## 4.5 Experiment 4

The implementation of DiffServ with Queues and Meter Tables remained untested in the last experiment. To test the capability of the mentioned feature we decided to check the CPqD switch with the Ryu Controller. Ryu is another SDN controller which has a different approach to controller design than that of ONOS. Ryu is written entirely in Python and is supported by OpenStack and NTT. Ryu is more of a research-oriented controller design than a production ready controller. Given that it is written in Python, it has shown poor performance as compared to other controllers written in languages like C++ or Java [26]. Ryu is not as “production ready” as ONOS. However, performance and production readiness is not something that we are testing in this experiment. Ryu has one of the best support for OpenFlow features and is easy to install and get it running on Linux environment. Thus, we choose Ryu to test DiffServ on SDN with the CPqD switch as a proof of concept of what we were trying to achieve in Experiment 3.

### 4.5.1 Statement

*Use OpenFlow Meter Tables and Queues together with DSCP markings to implement DiffServ on Ryu as a Proof of Concept.*

### 4.5.2 Implementation

For this experiment, we use Ryu as a controller and emulate a network on mininet with the CPqD switch. We want to emulate a network with differentiated service, thus, let us consider a network where different hosts represent hosts on different (differentiated) domains. The network used is shown in Figure 4.8. Consider that hosts H2 and H3 are part of separate domains. Since metering is done at the ingress point, the switches S2 and S3 will implement metering for the domains of H2 and H3. Data generates from H1 and causes congestion on S1. Queues are created on S1, and traffic is sent to different queues depending on their DSCP requests. Metering is executed by the ingress router which is

usually located at the boundary. Any traffic that exceeds the bandwidth specified by the meter will be re-marked. Usually, remarked packets are dropped preferentially or treated as low priority class.

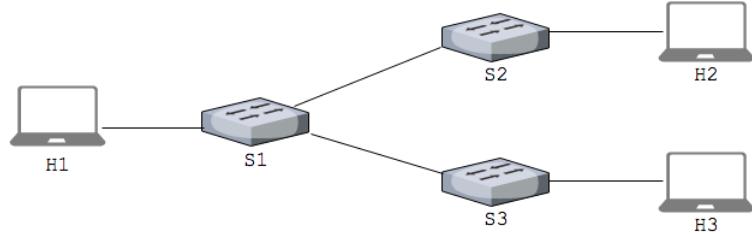


Figure 4.8: Network for Experiment 4.

In this experiment, we provide the bandwidth guarantee of 800Kbps to AF1 class. We are guaranteeing a rate of 400Kbps to traffic marked with AF11. AF11 traffic that exceeds the rate of 400Kbps will be considered as excess traffic and be remarked to AF12 class. The AF12 class will still have more preference over the best effort traffic. Three queues are created on S1 with different rate guarantees. These queues are identified by their queue id's 1, 2 and 3 and have a rate limit of 80, 120 and 800Kbps respectively. This is set by the python script starting the topology using the CPqD's dpctl tool, as shown in the code snippet below.

```

def run(net):
    s1 = net.getNodeByName('s1')
    s1.cmdPrint('dpctl unix:/tmp/s1 queue-mod 1 1 80')
    s1.cmdPrint('dpctl unix:/tmp/s1 queue-mod 1 2 120')
    s1.cmdPrint('dpctl unix:/tmp/s1 queue-mod 1 3 800')

```

Table 4.1: QoS Settings on Switch S1

Priority	DSCP	Queue ID	QoS ID
1	0 (BE)	1	1
1	12(AF12)	2	2
1	10(AF11)	3	3

Table 4.1 presents the QoS setting to be deployed on S1. These settings will send the traffic marked AF11 to queue 3 and AF12 to queue 2 and so on. Table 4.2 represents the QoS settings on S2 and S3 where the excess AF11 traffic is remarked to AF12.

Table 4.2: QoS Settings on Switch S2 and S3

Priority	DSCP	Meter ID	QoS ID
1	10(AF11)	1	1

Table 4.3 shows the meter table entry for remarking. Settings can be verified using simple REST commands like `curl GET`. We then test the installed configurations. We measure the bandwidth using iPerf. We start a server on h1 set to listen to the ports 5001, 5002 and 5003. The protocol used is UDP. Traffic is sent for each of the classes mentioned to h1 by h2 and h3.

Table 4.3: Meter Entry in Meter Table 1.

Meter ID	Flag	Bands
1	KBPS	rate 400000, type DSCP_REMARK, prec_level 1

We perform a number of tests with a different combination of competing traffics and observe the results. These tests include (i) AF11 Excess vs Best-Effort traffic (ii) AF11 excess traffic vs Best Effort vs AF11 non-excess traffic and (iii) AF11 excess vs AF11 excess traffic. The expected results should show that the AF11 traffic gets the highest preference on bandwidth and the best effort traffic gets the least. It will be interesting to see the AF11 excess traffic's behavior competing with the AF11 and best effort traffic.

The results of the tests are discussed in the following subsection.

### 4.5.3 Results

#### AF11 Excess vs. Best-Effort Traffic

The experiment's results are presented in Figure 4.9. The best effort traffic demanded a bandwidth of 800Kbps and the AF11 traffic require a bandwidth of 600Kbps while it is guaranteed at 400Kbps. We can see in the results that AF11 traffic is given preference over the best effort traffic, even if it demands an excess.

#### AF11 Excess Traffic vs. Best Effort vs. AF11 Non-Excess Traffic

The results of this experiment can be seen in Figure 4.10. Here, we had two AF11 traffic, one requesting 600Kbps and another 400Kbps. A competing best-effort traffic demanding 500Kbps was also generated in the network. We see that the AF11 non-excess traffic, demanding 400Kbps, has 0% packet drop. The AF11 excess traffic still receives preference over the best effort traffic and the best effort traffic only achieves a bandwidth of 210Kbps rather the demanded value of 500Kbps.

#### AF11 Excess AF11 Excess Traffic

In this test we make two AF11 excess traffic compete with each other. The result is shown in Figure 4.11. We can see that the excess traffic is dropped at nearly the same rate.

### 4.5.4 Observations

- Ryu is a research-oriented controller and thus is usually quick to implement new features. Ryu has excellent support for the available OpenFlow QoS features.
- Ryu is relatively difficult to learn and use as compared to ONOS, which is understandable considering the target users. ONOS has its priorities on performance, availability and other production environment issues.

```

Client connecting to 10.0.0.1, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size: 208 KByte (default)

[ 4] local 10.0.0.3 port 60324 connected with 10.0.0.1 port 5001
[ ID] Interval Transfer Bandwidth
[ 4] 0.0-10.0 sec 979 KBytes 800 Kbits/sec
[ 4] Sent 682 datagrams
[ 4] Server Report:
[ 4] 0.0-11.9 sec 639 KBytes 441 Kbits/sec 18.458 ms 229/ 682 (34%)

```

*Figure a: Best Effort Traffic*

```

Client connecting to 10.0.0.1, UDP port 5002
Sending 1470 byte datagrams
UDP buffer size: 208 KByte (default)

[ 4] local 10.0.0.2 port 53661 connected with 10.0.0.1 port 5002
[ ID] Interval Transfer Bandwidth
[ 4] 0.0-10.0 sec 735 KBytes 600 Kbits/sec
[ 4] Sent 512 datagrams
[ 4] Server Report:
[ 4] 0.0-10.0 sec 735 KBytes 600 Kbits/sec 7.497 ms 6/ 512 (1.2%)
[ 4] 0.0-10.0 sec 6 datagrams received out-of-order

```

*Figure b: AF11 Excess Traffic*

Figure 4.9: AF11 Excess vs Best Effort Traffic

```

Client connecting to 10.0.0.1, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size: 208 KByte (default)

[ 4] local 10.0.0.2 port 49358 connected with 10.0.0.1 port 5001
[ ID] Interval Transfer Bandwidth
[ 4] 0.0-10.0 sec 735 KBytes 600 Kbits/sec
[ 4] Sent 512 datagrams
[ 4] Server Report:
[ 4] 0.0-10.0 sec 666 KBytes 544 Kbits/sec 500.361 ms 48/ 512 (9.4%)
[ 4] 0.0-10.0 sec 192 datagrams received out-of-order

```

*Figure a: AF11 Excess*

```

Client connecting to 10.0.0.1, UDP port 5002
Sending 1470 byte datagrams
UDP buffer size: 208 KByte (default)

[ 4] local 10.0.0.3 port 42759 connected with 10.0.0.1 port 5002
[ ID] Interval Transfer Bandwidth
[ 4] 0.0-10.0 sec 613 KBytes 500 Kbits/sec
[ 4] Sent 427 datagrams
[ 4] WARNING: did not receive ack of last datagram after 10 tries.
[ 4] Server Report:
[ 4] 0.0-14.0 sec 359 KBytes 210 Kbits/sec 102.479 ms 177/ 427 (41%)

```

*Figure b: Best Effort*

```

Client connecting to 10.0.0.1, UDP port 5003
Sending 1470 byte datagrams
UDP buffer size: 208 KByte (default)

[ 4] local 10.0.0.3 port 35475 connected with 10.0.0.1 port 5003
[ ID] Interval Transfer Bandwidth
[ 4] 0.0-10.1 sec 491 KBytes 400 Kbits/sec
[ 4] Sent 342 datagrams
[ 4] Server Report:
[ 4] 0.0-10.5 sec 491 KBytes 384 Kbits/sec 15.422 ms 0/ 342 (0%)

```

*Figure c: AF11 Non-Excess*

Figure 4.10: AF11 Excess Traffic vs Best Effort vs AF11 Non-Excess Traffic

```

Client connecting to 10.0.0.1, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size: 208 KByte (default)

[ 4] local 10.0.0.3 port 50761 connected with 10.0.0.1 port 5001
[ ID] Interval Transfer Bandwidth
[ 4] 0.0-10.0 sec 735 KBytes 600 Kbits/sec
[ 4] Sent 512 datagrams
[ 4] Server Report:
[ 4] 0.0-11.0 sec 673 KBytes 501 Kbits/sec 964.490 ms 43/ 512 (8.4%)
[ 4] 0.0-11.0 sec 95 datagrams received out-of-order

```

Figure a: AF11 excess traffic 1.

```

Client connecting to 10.0.0.1, UDP port 5002
Sending 1470 byte datagrams
UDP buffer size: 208 KByte (default)

[ 4] local 10.0.0.2 port 53066 connected with 10.0.0.1 port 5002
[ ID] Interval Transfer Bandwidth
[ 4] 0.0-10.0 sec 735 KBytes 600 Kbits/sec
[ 4] Sent 512 datagrams
[ 4] Server Report:
[ 4] 0.0-10.6 sec 665 KBytes 515 Kbits/sec 897.126 ms 49/ 512 (9.6%)
[ 4] 0.0-10.6 sec 93 datagrams received out-of-order

```

Figure b: AF11 excess traffic 2.

Figure 4.11: AF11 Excess AF11 Excess Traffic

- While DiffServ was implemented with the given settings, the process was very manual, where every setting needed to be manually set on every network device. This is not suited for real-world network and ISP's.
- The CPqD switch, irrespective of the complaints towards it, works fine with Ryu. It is, at present, the only software switch to support both the OpenFlow QoS features.

## 4.6 Experiment 5

Since ONOS' capabilities to deal with OpenFlow Meter Tables could not be tested due to interoperability issues with the CPqD switch. We decided to test it with the Lagopus switch with a “meter only” experiment. This experiment explores meter table handling abilities of ONOS.

### 4.6.1 Statement

*Use OF meter tables on ONOS with the Lagopus switch.*

## 4.6.2 Implementation

As explained in experiment 3 before (see fig. 4.7), we used a single switch network to test the Lagopus switch. This is because mininet does not support Lagopus natively. We found out during the dissertation that mininet requires some changes to support Lagopus. Thankfully, the Lagopus developers, have made the changes in a forked version of mininet<sup>2</sup> and they made it available for us to use for our experiment. We created a simple network of two switches connected linearly and one host connected to each of the switches. ONOS ran at the control plane. ONOS detected the Lagopus switch and was able to install intents (flows) on the switch. The hosts were able to ping each other, and the communication was through.

Both the ONOS CLI and the REST API has support for manipulating meter tables on OF devices. The ONOS CLI has command like `meter-add`, `meter-remove` and `meters` to add, remove and list all the meters respectively. The ONOS REST API has support for GET, POST and DELETE operations for meters as shown in the screen-shot below.

meters : Query and program meter rules		Show/Hide   List Operations   Expand Operations
GET	/meters	Returns all meters of all devices
DELETE	/meters/{deviceId}/{meterId}	Removes the specified meter
GET	/meters/{deviceId}/{meterId}	Returns a meter by the meter id
GET	/meters/{deviceId}	Returns a collection of meters by the device id
POST	/meters/{deviceId}	Creates new meter rule

Using the REST API calls we were able to install meters on the Lagopus switch. Both the ONOS CLI and the REST calls were able to detect the meters successfully. An added sample meter is shown below. It has been installed by the REST API, and the screenshot presents the CLI view.

```
onos> meters
DefaultMeter{device=of:0000000000000001, id=1, appId=org.onosproject.rest, unit=KB_PER_SEC, isBurst=true, state=ADDED, bands=[DefaultBand{rate=400, burst-size=0, type=REMARK, drop-precedence=0}]}
onos>
```

<sup>2</sup>Mininet version with Lagopus support. [online] Available: <https://github.com/lagopus/mininet>

The next logical step is to point the respective flows to the meter. This is where things get complex. We know that the ONOS system is subdivided into different components. Which means that the REST, GUI, CLI as well the Intent Subsystem are separate components of the complete ONOS system. While some subsystems like the REST and CLI support querying and manipulating meter tables on OF devices, the Intent subsystem does not support directing the intents to meters yet. This would be something similar to the `setQueue` argument in the `add-host-intent` and related commands to install flows rules on the devices. In our experiments, we could not find a way to map flows to the meters. The flow subsystem supports the action argument called `meter`. These action arguments are the list of actions supported by the OpenFlow protocol itself. However, trying to install flows through the flow subsystem instead of the intent subsystem caused differences in the global view of rules for the system. Thus, these flows always stayed in pending state.

#### 4.6.3 Results

ONOS has limited or partial support for meter tables.

#### 4.6.4 Observation

- ONOS has partial support for meter tables. Since the different subsystems have different teams working on them, they also have a different level of support for features.
- A recent ONOS Technical Steering Meeting<sup>3</sup> this year has pointed out adding more support for meter tables in ONOS as their primary objective.
- While the current state of SDN infrastructure lacks support for QoS features, there are steps taken in the positive direction.

---

<sup>3</sup>ONOS Technical Steering Meeting. [online] Available: <https://youtu.be/-oB2gmSy7zk>

# Chapter 5

## Discussion

The previous chapter presented all the different experiments that we conducted on ONOS with different implementations of various SDN components. This chapter discusses the experiments and their observations in the context of the complete picture of SDN and QoS. We discuss SDN's place in the internet and QoS' spot in SDN. We summarize the gaps and challenges found in the present state of SDN and the direction in which it is headed.

The internet is facing numerous new challenges with every emerging technology and SDN provides solutions to some of those problems. SDN is deemed to be the next big wave of innovation and replace the current internet. While there has been wide acceptance of SDN as a possible candidate to pervade the internet, there are many deficiencies which SDN needs to address. Different applications on the internet demand very specific services and quality guarantees from the network. For example, VoIP, Online Gaming, HD video streaming and Web Browsing all have different requirements from the internet. By making the internet programmable and flexible SDN can cater to all these needs. Data Center operators can use SDN to implement network function virtualization which could reduce the CAPEX and OPEX to meet the customer demands. Cloud service providers can use it to interconnect their distributed data centers. Many enterprise and campuses

have implemented the Bring Your Own Device (BYOD) policies in their organization. This requires fine grained control over privileges that every device have on the network and SDN can be used to deploy granular policies in their networks.

While QoS techniques like the IntServ and the DiffServ has been around for a long time, we have hardly seen them implemented on the internet. This is either because ISP's consider them to be too complicated to implement on such a large scale or they do not see any profitable return on investment. Most ISP's rather choose to over provision themselves to meet the customers demands. Since SDN makes the network programmable and flexible, it makes it relatively easier and cheaper to implement new technologies. Therefore, it also gives us an opportunity to reintroduce QoS on the internet to prevent over provisioning and bring down the costs for network operators.

The SDN community recognizes this opportunity, and therefore we have seen many industry vendors and ISP's join hands with SDN projects. Most of these projects have some level of QoS features. The OpenFlow protocol supports QoS features like queues since version 1.1 and has been adding new features with every version. Projects like Ryu controller and Lagopus switch which is backed by NTT have good support for latest QoS features. Needless to say that many researchers have also been working on QoS frameworks for SDN, some of which are mentioned in Chapter 2.

Our experiments to test the QoS capabilities of the ONOS controller presented us with many interesting insights. The problem with the SDN framework is that, while there have been numerous projects working towards making SDN a success, these projects work independently of each other. No general guidelines are driving the progress of SDN. These projects have been created with different aims in mind and have different priorities. SDN is a framework rather than technology, and for SDN to become a success, different parts of the framework need to work together as one. We experienced during our experiments

that interoperability and varying level of support for OpenFlow features were the biggest obstacles in implementing the QoS settings. Some of these problems are listed below:

- Queues need to be configured outside of the OpenFlow protocol, and a separate protocol is required for it. Even with the different protocol, the controller should be able to configure queues on the switches, which was not the case with ONOS.
- The Ryu controller can use the OVSDB protocol to create and manipulate queues on the switches from the control plane using its REST API, but the process is extremely manual.
- The management protocol varies from switch to switch. This makes interoperability a problem. A real world network can consist of numerous types of devices and to configure them separately is not a feasible proposition.
- There are just two QoS features described in the OF specification, and those features are not fully supported by most of the popular software switch implementations.
- Some switches and controllers have interoperability issues. The CPqD switch did not work with ONOS and worked perfectly with the Ryu controller.
- A production ready controller like ONOS has many other issues to address. Like high-availability, distributed design and scalability and it is understandable that QoS is not their priority.
- The process of configuring QoS is very manual and not suited for large networks.
- The lack of support and documentation also had a significant contribution to the challenges of this project.

While there are many issues with the current SDN framework and its QoS abilities, there has been growth in the positive direction, for example:

- The Lagopus switch already has a partial implementation of queues, and we expect it to be tested and ready very soon.
- The developers of Open vSwitch have announced the release of meter support in OVS version 2.8 onwards.
- ONOS has taken steps in configuring QoS from the control plane by providing the wrapper classes. The south bound implementation of the classes is a work in progress and is already in the pipeline. Thus, they care about QoS as well.
- Meter support for the intent subsystem should be available the next release of ONOS. It has been put on priority in a recent technical steering meeting.

The work done in this project is by no means exhaustive and has its limitations. Given the scope of the project, the following Table 5.1 summarizes the state of support for QoS features tested in this work. N/A denotes that it could not be tested.

Table 5.1: Summary of Achieved QoS

<b>QoS Achieved</b>	<b>Ryu</b>	<b>ONOS</b>
Setting QoS from NBI	YES	NO
Queue + OVS	YES	YES
Queue + CPqD	YES	N/A
Queue + DSCP + CPqD	YES	N/A
Queue + DSCP + Meters + CPqD	YES	N/A
Meters + Lagopus	N/A	NO

While we only tested the ability to implement DiffServ in SDN, the QoS capabilities of SDN need not be limited to IntServ and DiffServ like the current internet. The pro-

grammability of SDN opens a whole new world of possibilities for QoS. We have seen some of the novel approaches towards QoS in chapter 2. While all those models addressed very particular issues and had their drawbacks, they collectively augur well for an overarching solution to QoS. Considering with it, the discussions of this research and the conversations with the developers of different SDN components during this thesis, the evolution of SDN looks to be on the right path, and we should be able to come up with comprehensive solution for QoS in SDN.

Having discussed the potential challenges and steps in the direction to solve them, it is still a long way to go for SDN to be ready for the public internet. The internet provided a solution to the human communication problem and saw widespread acceptance. With scaling, it found its challenges and solutions. Similarly, SDN provides a solution to many current challenges, and as it will gain acceptance and as it will scale, it will face new challenges of its own, and the evolution will continue. Only this time, the solution will be written in software, which would be so much easier and cheaper to implement and deploy on the scale of the world wide internet.

# Chapter 6

## Conclusion

This thesis was motivated by the rapid rise of SDN, claims of being the replacement of the current internet architecture, increasing industry participation in research and an enormous amount of research in the field of QoS in SDN. As we have discussed throughout the thesis that to achieve the mentioned goal of “global domination”, SDN has to provide fine grained QoS. QoS is the need of the hour, and attempts have been made by the researchers to come up with a comprehensive model for QoS.

ONOS is leading the charge of SDN to production environments and this project aimed at testing its QoS abilities and identifying the gaps in the present day SDN architecture. We found out in this thesis that every implementation of SDN components provides some level of QoS support. The OpenFlow Switch Specification is the primary driver of the features available in the SDN projects. While most of the controllers like ONOS and Ryu has support for the latest OF1.5 protocol, it is the existence of optional features in the OpenFlow specification that make the level of support asynchronous among the various components of the SDN framework. Lack of a standard south-bound device management and configuration protocol also contributes to the interoperability problem among the components of a network. Implementing QoS in SDN is easier and cheaper as compared to the traditional internet. If the gaps identified in this thesis are addressed, it would

be practically possible to implement complex QoS techniques like DiffServ in a network, which is deemed too complex to implement in the current internet. Additionally, the programmability and flexibility of SDN provides us with an opportunity to look beyond the available QoS techniques like IntServ and DiffServ and come up with a radical and comprehensive approach for QoS which would never be possible in today's internet.

We also found out that while there are gaps in the present level of QoS support, most of the projects have put QoS on priority and are working to support more QoS features with every new release. Controllers like Ryu already supports all the QoS features defined in the OpenFlow protocol. These are all positive signs for QoS' future. While QoS techniques never became a reality in the current internet even after 20 years of research, we may see new SDN implementation with a comprehensive, "never-seen-before" QoS techniques become a reality very soon. After all, the entire pitch for SDN and OpenFlow is standing on "an increased rate of innovation in networks".

## 6.1 Contributions

The contributions of this thesis can be listed briefly in the following statements.

- Examined the current state of QoS support in one of the leading production ready SDN controller, ONOS.
- Identified gaps in the QoS abilities of the current state of SDN framework.
- Compared two leading controllers ONOS and Ryu in terms of their QoS abilities, within the scope of this project.
- Identified the direction in which the present research and projects in Quality of Service in Software Defined Networking is headed.

## 6.2 Future Work

Most of the work in this project was aimed at exploring the current state and abilities of QoS rather than implementing or proposing a new QoS model. We have mentioned throughout the text that the process of configuring QoS is extremely manual and is not suitable for large networks. The automation of QoS configuration presents a wide area of research and work. In the present setting, the network administrators have to know and provision the required resources to satisfy the QoS requirements beforehand. Work can be done for system that automates this process.

A potential implementation of artificial intelligence running on SDN controller that can keep track of the available resources in the network, can take request from applications and configure corresponding QoS settings on the network devices to guarantee the demanded resources would be interesting to work. Although it would first need to address all the identified gaps first, it is not something that is unachievable if proper resources are put to that task. Moreover, many research reviewed in the thesis provided the solution to one category of the QoS implementation. A work that can exploit all those techniques and combine them into a QoS system that can cater to all the categories of QoS techniques would be an interesting solution for SDN. Standardization of the device configuration protocol (OF-CONFIG by ONF is a possible candidate) is something that would smoothen the process of implementing any QoS system in SDN and is something that should be implemented as quickly as possible. This could increase the rate of research in QoS in SDN even further. And, as future work for the entire SDN community, interoperability between different implementation of SDN components should be of prime importance because a real network would have a heterogeneous set of devices and implementing different QoS settings on different devices that do not cooperate with each other is something that would hinder the growth of SDN to great extent.

# Appendix A

## Abbreviations

Short Term	Expanded Term
API	Application Program Interface
CAPEX	Capital Expenditure
CDN	Content Distribution Network
CLI	Command Line Interface
DPDK	Data Plane Development Kit
DSCP	Differentiated Service Code Points
FIFO	First In First Out
GUI	Graphical User Interface
HTB	Hierarchical Token Bucket
HFSC	Hierarchical Fair Service Curve
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IP	Internet Protocol
ISP	Internet Service Provider
MPLS	Multi Protocol Label Switching
NBI	North Bound Interface

<b>Short Term</b>	<b>Expanded Term</b>
NFV	Network Function Virtualization
OF	OpenFlow Protocol
ON.Lab	Open Networking Lab
ONF	Open Networking Foundation
ONOS	Open Network Operating System
OPEX	Operating Expenditure
OVSDDB	Open vSwitch Database
OVS	Open vSwitch
PHB	Per Hop Behavior
QDISC	Queuing Discipline
QoS	Quality of Service
RSVP	Resource Reservation Protocol
RTT	Round Trip Time
SBI	South Bound Interface
SDN	Software Defined Networking
SSH	Secure Shell
TCP	Transmission Control Protocol
TOS	Type of Service
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network

# Bibliography

- [1] Our Members - Open Networking Foundation. (2017). [Opennetworking.org](https://www.opennetworking.org/about/onf-members). Retrieved 3 August 2017, from <https://www.opennetworking.org/about/onf-members>
- [2] Nick McKeown - Talks. (2017). [Yuba.stanford.edu](http://yuba.stanford.edu/~nickm/talks.html). Retrieved 3 August 2017, from <http://yuba.stanford.edu/~nickm/talks.html>
- [3] Open Network Operating System. (2017). ONOS. Retrieved 3 August 2017, from <http://onosproject.org/>
- [4] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker. “Software Defined Internet Architecture: Decoupling Architecture from Infrastructure, In Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets-XI), pp. 43-48, Redmond, WA, USA, Oct 2012.
- [5] R. Masoudi and A. Ghaffari, “Software Defined Networks: A Survey, ACM Journal of Network and Computer Applications, vol. 67, issue C, pp. 1-25, May 2016.
- [6] Software-Defined Networking (SDN) Definition - Open Networking Foundation. (2017). [Opennetworking.org](https://www.opennetworking.org/sdn-resources/sdn-definition). Retrieved 3 August 2017, from <https://www.opennetworking.org/sdn-resources/sdn-definition>
- [7] Martin Casado, Nate Foster, and Arjun Guha. “Abstractions for software-defined networks,” Communications of the ACM, vol. 57, pp. 86-95, Sep 2014.

- [8] Open vSwitch. (2017). [openvswitch.org](http://openvswitch.org/). Retrieved 3 August 2017, from <http://openvswitch.org/>
- [9] OpenFlow 1.3 Software Switch by CPqD. (2017). [cpqd.github.io](http://cpqd.github.io/ofsoftswitch13/). Retrieved 3 August 2017, from <http://cpqd.github.io/ofsoftswitch13/>
- [10] DPDK. (2017). [dpdk.org](http://dpdk.org). Retrieved 3 August 2017, from [http://dpdk.org/](http://dpdk.org)
- [11] Lagopus switch. (2017). [lagopus.org](http://www.lagopus.org). Retrieved 3 August 2017, from <http://www.lagopus.org/>
- [12] B. Pfaff, B. Davie, “The Open vSwitch Database Management Protocol”, RFC 7047 (Informational) Internet Engineering Task Force, Dec. 2013, [online] Available: <http://www.ietf.org/rfc/rfc7047.txt>.
- [13] “OF-CONFIG 1.2”, [online] Available: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1.2.pdf>.
- [14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: Enabling Innovation in Campus Networks, ACM SIGCOMM Computer Communication Review, vol. 38, Issue 2, pp. 69-74, Apr 2008.
- [15] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O’Connor, P. Radoslavov, W. Snow, and G. Parulkar, “ONOS: towards an open, distributed SDN OS, In Proceedings of the third Workshop on Hot topics in Software Defined Networking (HotSDN ’14), pp. 1-6, Chicago, Illinois, USA, Aug 2014.

- [16] Mininet. (2017). Mininet: An Instant Virtual Network on your Laptop (or other PC) - Mininet. [mininet.org](http://mininet.org/). Retrieved 4 August 2017, from <http://mininet.org/>
- [17] B. Lantz, B. Heller, and N. McKeown, “A Network in a Laptop: Rapid Prototyping for Software-Defined Networks, In Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX), Article 19, pp. 1-6, Monterey, California, USA, Oct 2010.
- [18] T. Cejka and R. Krejci, “Configuration of open vSwitch using OF-CONFIG,” In Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS 2016), pp. 883-888, Istanbul, Turkey, Apr 2016.
- [19] Brandon Heller, Rob Sherwood, and Nick McKeown. “The controller placement problem”. In Proceedings of the first workshop on Hot topics in software defined networks (HotSDN ’12), pp. 7-12, New York, NY, USA, Aug 2012.
- [20] Nox Controller. (2017). GitHub. Retrieved 4 August 2017, from <https://github.com/noxrepo/nox>
- [21] POX Controller. (2017). GitHub. Retrieved 4 August 2017, from <https://github.com/noxrepo/pox>
- [22] Ryu SDN Framework. (2017). Osrg.github.io. Retrieved 4 August 2017, from <https://osrg.github.io/ryu/>
- [23] OpenDaylight Controller. (2017). Retrieved 4 August 2017, from <https://www.opendaylight.org/>
- [24] Floodlight OpenFlow Controller. (2017). Project Floodlight. Retrieved 4 August 2017, from <http://www.projectfloodlight.org/floodlight/>
- [25] Open Network Operating System. (2017). ONOS. Retrieved 4 August 2017, from <http://onosproject.org/>

- [26] L. Stancu, S. Halunga, A. Vulpe, G. Suciu, O. Fratu and E. C. Popovici, “A comparison between several Software Defined Networking controllers,” In Proceedings of 2015 12th International Conference on Telecommunication in Modern Satellite, Cable and Broadcasting Services (TELSIKS), pp. 223-226, Ni, Serbia, Oct 2015.
- [27] “OpenFlow Switch Specification Version 1.5.1, 2015. Retrieved 4 August 2017, from <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf>
- [28] R. Braden, D. Clark, and S. Shenker. “Integrated Services in the Internet Architecture:an Overview”. RFC 1633 (Informational). Internet Engineering Task Force, 1994. Available: <http://www.ietf.org/rfc/rfc1633.txt>.
- [29] D. Black, P. Jones, “Differentiated Services (Diffserv) and Real-Time Communication”, RFC 7657 (Informational) Internet Engineering Task Force, Nov. 2015, [online] Available: <https://tools.ietf.org/html/rfc7657>
- [30] K. Nichols, S. Blake, F. Baker, D. Black, “Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers.” RFC: 2474 (Standards Track), Internet Engineering Task Force, Dec 1998, [online] Available: <https://www.ietf.org/rfc/rfc2474.txt>
- [31] J. Heinanen, F. Baker, W. Weiss, J. Wroclawski, “Assured Forwarding PHB Group.”, RFC 2597 (Standards Track), Internet Engineering Task Force, Jun 1999, [online] Available: <https://tools.ietf.org/html/rfc2597>
- [32] V. Jacobson, K. Nichols, K. Poduri, “An Expedited Forwarding PHB”, RFC 2598 (Standards Track), Internet Engineering Task Force, Jun 1999, [online] Available: <https://tools.ietf.org/html/rfc2598>

- [33] M. Karakus, A. Durresi, “Quality of Service (QoS) in Software Defined Networking (SDN): A Survey, Elsevier Journal of Network and Computer Applications, vol. 80, pp. 200-218, Dec 2016.
- [34] W. Kim, J. Li, J. W. K. Hong, and Y. J. Suh, “OFMon: Open Flow monitoring system in ONOS controllers,” In Proceedings of 2016 IEEE NetSoft Conference and Workshops (NetSoft), pp. 397-402, Seoul, South Korea, June 2016.
- [35] C. Xu, B. Chen, and H. Qian, “Quality of Service Guaranteed Resource Management Dynamically in Software Defined Network,” Journal of Communications, vol. 10, no. 11, pp. 843-850, Nov 2015.
- [36] S. Muqaddas, A. Bianco, P. Giaccone and G. Maier, “Inter-controller traffic in ONOS clusters for SDN networks,” In Proceedings of 2016 IEEE International Conference on Communications (ICC), pp. 2293-2298, Kuala Lumpur, Malaysia, May 2016.
- [37] M. Karakus and A. Durresi, “A Scalable Inter-AS QoS Routing Architecture in Software Defined Network (SDN),” In Proceedings of 2015 IEEE 29th International Conference on Advanced Information Networking and Applications, pp. 148-154, Gwangju, South Korea, Mar 2015.
- [38] H. E. Egilmez, S. T. Dane, K. T. Bagci and A. M. Tekalp, “OpenQoS: An Open Flow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks,” In Proceedings of the 2012 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference, pp. 1-8, Hollywood, CA, USA, Dec 2012.
- [39] H. E. Egilmez, S. Civanlar, and A. M. Tekalp, “An Optimization Framework for QoS-Enabled Adaptive Video Streaming Over Open Flow Networks,” In IEEE Transactions on Multimedia, vol. 15, no. 3, pp. 710-715, April 2013.

- [40] K. Govindarajan, K. C. Meng, H. Ong, W. M. Tat, S. Sivanand and L. S. Leong, “Realizing the Quality of Service (QoS) in Software-Defined Networking (SDN) based Cloud infrastructure,” In Proceedings of IEEE 2nd International Conference on Information and Communication Technology (ICoICT), pp. 505-510, Bandung, Indonesia, May 2014.
- [41] Ishimori, F. Farias, E. Cerqueira and A. Abelm, “Control of Multiple Packet Schedulers for Improving QoS on Open Flow/SDN Networking,” In Proceedings of Second European Workshop on Software Defined Networks, pp. 81-86, Berlin, Germany, Oct 2013.
- [42] R. Durner, A. Blenk, and W. Kellerer, “Performance study of dynamic QoS management for OpenFlow-enabled SDN switches,” In Proceeding of IEEE 23rd International Symposium on Quality of Service (IWQoS), pp. 177-182, Portland, OR, USA, June 2015.
- [43] T. Zinner, M. Jarschel, A. Blenk, F. Wamser and W. Kellerer, “Dynamic application-aware resource management using Software-Defined Networking: Implementation prospects and challenges,” In Proceeding of IEEE Network Operations and Management Symposium (NOMS), pp. 1-6, Krakw, Poland, May 2014.
- [44] T. Y. Cheng, M. Wang and X. Jia, “QoS-Guaranteed Controller Placement in SDN,” In Proceedings of 2015 IEEE Global Communications Conference (GLOBECOM), pp. 1-6, San Diego, CA, USA, Dec 2015.
- [45] M.S. Seddiki, M. Shahbaz, S. Donovan, S. Grover, M. Park, N. Feamster, and Y. Q. Song, “FlowQoS: QoS for the rest of us,” In Proceedings of the third workshop on Hot Topics in Software Defined Networking (HotSDN ’14), pp. 207-208, Chicago, Illinois, USA, Aug 2014.

- [46] linux.die.net. (2017). tc(8): show/change traffic control settings - Linux man page. [online] Available at: <https://linux.die.net/man/8/tc> [Accessed 11 Aug. 2017].
- [47] D. G. Balan and D. A. Potorac, “Linux HTB queuing discipline implementations,” IEEE First International Conference on Networked Digital Technologies, pp. 122-126, Ostrava, Czech Republic, Jul 2009.
- [48] I. Stoica, H. Zhang and T. S. E. Ng, “A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services,” in IEEE/ACM Transactions on Networking, vol. 8, no. 2, pp. 185-199, Apr 2000.
- [49] Ryu Certification. (2017). osrg.github.io. Retrieved 11 August 2017, from <http://osrg.github.io/ryu/certification.html>
- [50] wiki.onosproject.org. (2017). Wiki Home - ONOS - Wiki. [online] Available at: <https://wiki.onosproject.org/display/ONOS/Wiki+Home> [Accessed 12 Aug. 2017].
- [51] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. “NetVM: high performance and flexible networking using virtualization on commodity platforms.” In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI’14). USENIX Association, pp. 445-458, Berkeley, CA, USA, Apr 2014.
- [52] Fulvio Risso and Mario Baldi. “NetPDL: an extensible XML-based language for packet header description.” Computer Networks: The International Journal of Computer and Telecommunications Networking, Volume 50, Issue 5, pp. 688-706, New York, NY, USA, Apr 2006 .