

# Network Slicing using SDN and NFV - A 5G Use Case

*Thesis submitted in partial fulfillment of requirements*

*For the degree of*

**Master of Computer Application**

of

Computer Science and Engineering Department

of

Jadavpur University

by

**Md Juel Sekh**

**Regn. No. - 154229 of 2020-2021**

**Exam Roll No. - MCA2360036**

*under the supervision of*

**Dr. Mridul Sankar Barik**

Assistant Professor

Department of Computer Science and Engineering

JADAVPUR UNIVERSITY

Kolkata, West Bengal, India

2023

## Certificate from the Supervisor

This is to certify that the work embodied in this thesis entitled "**Network Slicing using SDN and NFV - A 5G Use Case**" has been satisfactorily completed by **Md Juel Sekh** (Registration Number 154229 of 2020 – 21; Class Roll No. 002010503021; Examination Roll No. *MCA2360036*). It is a bona-fide piece of work carried out under my supervision and guidance at Jadavpur University, Kolkata for partial fulfilment of the requirements for the awarding of the **Master of Computer Application** degree of the Department of Computer Science and Engineering, Faculty of Engineering and Technology, Jadavpur University, during the academic year 2022 – 23.

---

**Dr. Mridul Sankar Barik**

Assistant Professor,  
Department of Computer Science and Engineering,  
Jadavpur University  
(Supervisor)

Forwarded By:

---

**Prof. Nandini Mukherjee**

Head,  
Department of Computer Science and Engineering,  
Jadavpur University

---

**Prof. Ardhendu Ghoshal**

Dean,  
Faculty of Engineering & Technology,  
Jadavpur University

Department of Computer Science and Engineering  
Faculty of Engineering And Technology  
Jadavpur University, Kolkata - 700 032

## Certificate of Approval

This is to certify that the thesis entitled "**Network Slicing using SDN and NFV - A 5G Use Case**" is a bona-fide record of work carried out by **Md Juel Sekh** (Registration Number 154229 of 2020 – 21; Class Roll No. 002010503021; Examination Roll No. *MC*A2360036) in partial fulfilment of the requirements for the award of the degree of **Master of Computer Application** in the **Department of Computer Science and Engineering, Jadavpur University**, during the period of January 2023 to May 2023. It is understood that by this approval, the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein but approve the thesis only for the purpose of which it has been submitted.

**Examiners:**

---

(Signature of The Examiner)

---

(Signature of The Supervisor)

Department of Computer Science and Engineering  
Faculty of Engineering And Technology  
Jadavpur University, Kolkata - 700 032

## **Declaration of Originality and Compliance of Academic Ethics**

I hereby declare that the thesis entitled "**Network Slicing using SDN and NFV - A 5G Use Case**" contains literature survey and original research work by the undersigned candidate, as a part of his degree of **Master of Computer Application** in the **Department of Computer Science and Engineering, Jadavpur University**. All information have been obtained and presented in accordance with academic rules and ethical conduct.

I also declare that, as required by these rules and conduct, I have fully cited and referenced all materials and results that are not original to this work.

**Name:** Md Juel Sekh

**Examination Roll No.:** MCA2360036

**Registration No.:** 154229 of 2020 – 21

**Thesis Title:** Network Slicing using SDN and NFV - A 5G Use Case

**Signature of the Candidate:**

## ACKNOWLEDGEMENT

I am pleased to express my gratitude and regards towards my Project Guide **Dr. Mridul Sankar Barik**, Assistant Professor, Department of Computer Science and Engineering, Jadavpur University, without whose valuable guidance, inspiration and attention towards me, pursuing my project would have been impossible.

Last but not the least, I express my regards towards my friends and family for bearing with me and for being a source of constant motivation during the entire term of the work.

---

**Md Juel Sekh**

MCA Final Year

Exam Roll No. - MCA2360036

Regn. No. - 154229 of 2020 – 21

Department of Computer Science and Engineering,  
Jadavpur University.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.2	Research Statement . . . . .	2
1.3	Contribution of the Thesis . . . . .	2
1.4	Outline of the Thesis . . . . .	2
<b>2</b>	<b>5G Network</b>	<b>4</b>
2.1	Introduction . . . . .	4
2.2	5G Services . . . . .	4
2.3	5G Architecture . . . . .	4
<b>3</b>	<b>Network Slicing</b>	<b>7</b>
3.1	Introduction . . . . .	7
3.2	Network Slicing . . . . .	7
3.3	Benefits of Network Slicing . . . . .	8
3.4	Network Slicing Example . . . . .	8
3.5	Network Slicing in 5G . . . . .	9
3.6	Network Slice Security . . . . .	9
<b>4</b>	<b>Software Defined Networking</b>	<b>11</b>
4.1	Introduction . . . . .	11
4.2	Software-Defined Networking (SDN) . . . . .	11
4.3	SDN Architecture . . . . .	12
4.4	SDN Controller . . . . .	13
4.5	OpenFlow Protocol . . . . .	14
4.6	Open vSwitch . . . . .	14
<b>5</b>	<b>5G and SDN</b>	<b>16</b>
5.1	Introduction . . . . .	16
5.2	Use of SDN in 5G Network Slicing . . . . .	16
5.2.1	How We Manage Control Plane . . . . .	16
5.2.2	Managing Signal Processing and Slicing . . . . .	18
5.3	Network Function Virtualization (NFV) in 5G . . . . .	18
5.4	Implementing NFVI through VNFs in 5G Core Network Architecture . . . . .	19

<b>6</b>	<b>Experiments</b>	<b>20</b>
6.1	Ryu SDN Controller . . . . .	20
6.2	Mininet Emulator . . . . .	21
6.3	How to Use Ryu Controller in Mininet Topology . . . . .	21
6.4	RYU Framework Overview . . . . .	23
6.4.1	Components . . . . .	24
6.4.2	Events: . . . . .	24
6.4.3	Inserting/Adding a New Flow Using Program . . . . .	24
6.5	QoS (Quality of Service) in the Ryu controller . . . . .	25
6.6	QoS REST APIs Overview . . . . .	26
6.6.1	Get Status of Queue . . . . .	26
6.6.2	Get a Queue Configurations . . . . .	26
6.6.3	Set a Queue to the Switches . . . . .	26
6.6.4	Delete Queue . . . . .	27
6.6.5	Get Rules of QoS . . . . .	27
6.6.6	Set a QoS Rules . . . . .	27
6.6.7	Delete a QoS Rules . . . . .	28
6.6.8	Set a Meter Entry . . . . .	29
6.6.9	Delete a Meter Entry . . . . .	29
6.7	Experiment System Architecture . . . . .	29
6.7.1	Slice Architecture . . . . .	29
6.7.2	Network Topology Architecture . . . . .	30
6.8	Experiment Implementation and Setup . . . . .	31
6.8.1	Network Topology Implementation . . . . .	31
6.8.2	Network Topology Setup . . . . .	34
6.8.3	Ryu Controller Implementation . . . . .	35
6.8.4	Ryu Controller Setup . . . . .	38
6.8.5	Setting up the Servers . . . . .	46
6.8.6	Setting up the Clients . . . . .	46
6.9	Experiment Results . . . . .	46
6.9.1	Results of Before Setting up the Environment of Slicing . . . . .	46
6.9.2	Final Results of Experiments . . . . .	49
<b>7</b>	<b>Conclusion and Future Work</b>	<b>53</b>
7.1	Conclusion . . . . .	53
7.2	Future Work . . . . .	53

# List of Figures

2.1	5G architecture (Source-[1]) . . . . .	5
3.1	Network slicing example (Source-[12]) . . . . .	9
4.1	SDN architecture . . . . .	13
5.1	Signal Processing of the 5g Network Slicing (Source-[12]) . . . . .	17
5.2	Schematic diagram of implementing NFVI through VNFs in 5G core network architecture (Source-[12]) . . . . .	19
6.1	Starting the Ryu controller . . . . .	22
6.2	Starting the Mininet with linear topology and Ryu controller . . . . .	22
6.3	Before ping dump-flow of all switch . . . . .	23
6.4	Ping h2 from h1 . . . . .	23
6.5	After ping dump-flow of all switch . . . . .	23
6.6	Network Slice Flow . . . . .	30
6.7	Network Topology . . . . .	31
6.8	Setup network topology . . . . .	35
6.9	Setup Ryu controller . . . . .	39
6.10	Before adding QoS in controller flow table of nb(SW_1) switch . . . . .	40
6.11	Before adding QoS in controller flow table of rb1(SW_2) switch . . . . .	40
6.12	Before adding QoS in controller flow table of rb2(SW_3) switch . . . . .	41
6.13	Before adding QoS in controller flow table of a1(SW_4) switch . . . . .	41
6.14	Before adding QoS in controller flow table of a2(SW_5) switch . . . . .	42
6.15	Adding Queues and QoS rules in controller c0 . . . . .	43
6.16	After adding QoS in controller flow table of nb(SW_1) switch . . . . .	44
6.17	After adding QoS in controller flow table of rb1(SW_2) switch . . . . .	44
6.18	After adding QoS in controller flow table of rb2(SW_3) switch . . . . .	45
6.19	After adding QoS in controller flow table of a1(SW_4) switch . . . . .	45
6.20	After adding QoS in controller flow table of a2(SW_5) switch . . . . .	46
6.21	Before adding the Queues and QoS result of communication between ue1(server) and b(client) . . . . .	47
6.22	Before adding the Queues and QoS result of communication between ue3(server) and b(client) . . . . .	48
6.23	Before adding the Queues and QoS result of communication between ue2(server) and c(client) . . . . .	49



6.24	After adding the Queues and QoS result of communication between ue1(server) and b(client) . . . . .	50
6.25	After adding the Queues and QoS result of communication between ue3(server) and b(client) . . . . .	51
6.26	After adding the Queues and QoS result of communication between ue2(server) and c(client) . . . . .	52

### **Abstract**

In the context of the rapidly evolving 5G landscape, network slicing has emerged as a crucial architecture component. This paper focuses on the implementation of network slicing using Mininet as a network technology platform, Ryu controller for managing network functions, and OpenFlow switches for creating slices. The OpenFlow Queue command is utilized to prioritize and allocate bandwidth for each service within the slices.

The experimental results confirm the feasibility of network slicing using open-source technologies. The application of queues enables effective bandwidth assignment, ensuring quality of service (QoS) guarantees for each slice.

This paper highlights the potential of utilizing open-source solutions to achieve network slicing and QoS in 5G environments. By leveraging central virtualization technology and OpenFlow switches, the study contributes to the ongoing development of efficient and flexible network architectures for the diverse needs of 5G applications and services.

# Chapter 1

## Introduction

### 1.1 Background

5G mobile networks has been designed to carry a plethora of mobile devices and provide faster network connection speed. It is projected that the mobile data traffic in future 5G networks will experience an explosive growth.

The most challenging aspect of 5G networks is that, it will support a wide variety of use cases. Some applications, such as ultra-high definition (UHD) video and augmented reality need high-speed, high-capacity communications. While others such as the mission-critical Internet of Things (IoT) and autonomous vehicles require ultralow latency, ultra-reliable services.

Traditional mobile communication networks employ a flat approach where they provide services to mobile devices, regardless of the communication requirements of vertical services. But, this design principle can't offer differentiated services. Hence, there is a need to explore new techniques to address the challenges associated with supporting vertical industries in 5G networks.

Software-defined networking (SDN) and network functions virtualization (NFV) have been envisaged to be the key enabling technologies to build 5G systems. The key feature in SDN is that it decouples network control functions from data forwarding functions. Network control functions can run as applications independently in the logically centralized controllers. NFV decouples a network functions from dedicated and expensive hardware platforms to general-purpose commodity off the shelf hardware. With NFV, network operators can implement a variety of virtual network functions (VNFs) over the standard commodity servers. Mobile Edge Computing (MEC) is one of the emerging technologies in 5G - it is expected to serve low-latency communication that's one of the use cases in future 5G. MEC tries to move computing, storage, and networking resources from remote public clouds to the edge of the network. Benefit of this approach is that mobile clients can request virtual resources within the access network and experience low end-to-end delay.

The network slicing technique has been put forward as a solution to address the diversified service requirements in 5G. Network slicing involves provisioning of logical networks with a set of isolated virtual resources on the shared physical infrastructure. These logical networks are designated to provide different services to fulfill users' varying communication requirements. Network slicing provides a network-as-a-service (NaaS) model. It flexibly allocates and reallocates resources according to dynamic demands, such that it can customize network slices for diverse 5G communication use cases. Slice-based 5G has the following significant advantages when compared with

traditional networks:

- Network slicing can provide better performance with customized logical networks.
- A network slice can be flexibly scaled up or down as service requirements change.
- The reliability and security of each slice is enhanced by isolating the network resources of one slice from the other.
- A network slice is customized according to service requirements, and hence there is a scope to optimize the allocation and use of physical network resources.

## 1.2 Research Statement

The objective of this research is to investigate the implementation and effectiveness of Software-Defined Networking (SDN) and Network Function Virtualization (NFV) in the context of 5G networks, specifically focusing on network slicing and Quality of Service (QoS) provisioning.

The study aims to explore how SDN and NFV technologies can be utilized to dynamically create and manage network slices in a 5G environment. By leveraging the virtualization capabilities of NFV and the centralized control of SDN, the research seeks to demonstrate the feasibility of creating isolated network slices tailored to specific service requirements and user demands.

Furthermore, the research aims to evaluate the impact of network slicing on QoS provisioning in 5G networks. By implementing QoS mechanisms using OpenFlow switches and queue commands, the study intends to assess the ability of SDN-enabled network slices to effectively allocate bandwidth, prioritize traffic, and ensure consistent QoS levels for different services and applications.

## 1.3 Contribution of the Thesis

The thesis contributes to the field of 5G networks by implementing network slicing using Mininet, Ryu controller, and OpenFlow switches. It focuses on ensuring Quality of Service (QoS) by allocating bandwidth and prioritizing traffic through SDN and NFV technologies. The research validates the proposed architecture through extensive experiments, demonstrating improved network performance and resource utilization. By utilizing open-source solutions, the thesis emphasizes the practicality and affordability of implementing SDN, NFV, and network slicing in 5G networks. The findings provide insights for future network architectures, guiding the development of efficient infrastructures that meet the diverse requirements of 5G services and applications. Overall, the thesis advances the understanding and practical implementation of network slicing and QoS provisioning in 5G networks, contributing to the progress of the field.

## 1.4 Outline of the Thesis

This thesis is divided into seven chapters.

Chapter 1: Introduction

In this chapter, we provide a background on the topic of the thesis. We discuss the emergence of

5G networks and their significance in the telecommunications industry. We also state the research problem that this thesis aims to address and outline the contributions of this thesis.

#### Chapter 2: 5G Network

In this chapter, we introduce the concept of 5G networks. We discuss the key features and characteristics of 5G technology, as well as its potential applications and services.

#### Chapter 3: Network Slicing

This chapter focuses on network slicing, a fundamental concept in 5G networks. We provide an introduction to network slicing and explain its benefits in terms of enabling efficient resource allocation, service customization, and isolation among different network slices.

#### Chapter 4: Software Defined Networking

Here, we delve into the realm of Software-Defined Networking (SDN). We explain the concept of SDN and its architecture, highlighting the separation of control plane and data plane, as well as the role of the SDN controller and the OpenFlow protocol.

#### Chapter 5: 5G and SDN

In this chapter, we explore the integration of SDN in 5G networks. We discuss the use of SDN in network slicing, including the management of the control plane and signal processing. We also examine the role of Network Function Virtualization (NFV) in 5G core network architecture.

#### Chapter 6: Experiments

This chapter presents the experiments conducted to evaluate the effectiveness of integrating SDN and network slicing in 5G networks. We provide an overview of the Ryu SDN controller and the Mininet emulator used for experimentation. We explain the implementation and setup of the experiment system architecture, including the network topology and the configuration of QoS in the Ryu controller.

#### Chapter 7: Conclusion and Future Work

In the final chapter, we conclude the thesis by summarizing the main findings and contributions. We discuss the implications of our research and its potential impact on the telecommunications industry. Additionally, we outline potential areas for future work and research in the field of 5G networks and SDN integration.

## Chapter 2

# 5G Network

### 2.1 Introduction

The fifth generation of mobile network technology or 5G is the latest version of cellular technology. 5G can deliver faster data transfer speeds, lower latency, greater network capacity, and more reliable connectivity than previous generation mobile network technology.

5G is the system defined by 3GPP (Third Generation Partnership Project) from Release 15, functionally frozen in June 2018 and fully specified by September 2019. 3GPP also defines all the protocols and network interfaces that enable the entire mobile system[1].

### 2.2 5G Services

- **Enhanced Mobile Broadband (eMBB):** eMBB can provide higher data transfer speed of up to 20 Gbps. Today's day to day life we are consuming high quality videos, playing high graphics online game etc. which need high bandwidth. eMBB service can solve this problem.
- **Critical Communications (CC) and Ultra Reliable and Low Latency Communications (URLLC):** In some scenarios we need extremely high reliability. In remote control medical surgery 99.999% reliability expected. URLLC service can solve this problem
- **Massive Internet of Things (mIoT):** Several scenarios require the 5G system to support large number of devices with low data rate.

### 2.3 5G Architecture

Components of the 5G architecture provided by 3GPP and their some functionality:

- User Equipment (UE) or simply User Device is composed of a Mobile Station (MS) and a USIM that connects to the 5G network to access various services and applications.

- The main entity of the Radio Access Network (NG-RAN) or (R)AN is the gNB, which refers to the radio transmitter that is responsible for providing wireless connectivity between the user equipment (UE) and the core network.

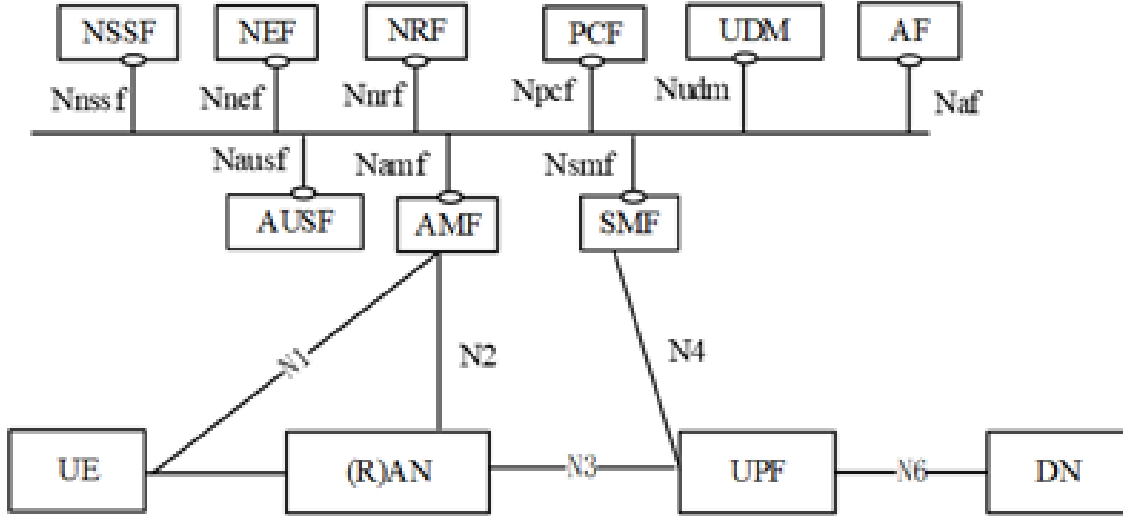


Figure 2.1: 5G architecture (Source-[1])

- User Plane Function (UPF) handling the user data as it flows between the user equipment (UE) and the external data networks.
- The Data Network (DN) which is the external network connector that connects the 5G core network (5GC) with external networks.
- Authentication Server Function (AUSF) is a network function that performs authentication and security-related functions for user equipment (UE).
- Access and Mobility management Function (AMF) that accesses the UE and the (R)AN and plays a critical role in managing and controlling the access and mobility of user equipment (UE).
- The Session Management Function (SMF) is responsible for managing calls, sessions, and contacts the UPF accordingly.
- The Application Function (AF) is responsible for managing and controlling the application-level services and interactions within the 5G core network (5GC).
- Unified Data Management (UDM) is a network function that is responsible for managing and storing user-related data, such as subscriber profiles, authentication credentials, and subscription information and enabling various services.

- Policy Control Function (PCF) is a network function that is responsible for managing policies related to Quality of Service (QoS), charging, and traffic management for user sessions in the 5G core network (5GC).
- Network Repository Function (NRF) is a network function that serves as a central repository for storing and managing information about network functions available in the 5G core network (5GC).
- Network Exposure Function (NEF) is a network function that enables third-party applications or services to securely access and interact with the 5G core network (5GC).
- Network Slice Selection Function (NSSF) is a network function that is responsible for selecting the appropriate and optimal network slice for a specific user or service based on policy-based decision-making.



## Chapter 3

# Network Slicing

### 3.1 Introduction

Different customer require different service but in all previous generation network till 4G network, service provider provide same type of service like same bandwidth, latency, etc. for every customer. Using network slicing concept in 5G network, service provider can deliver different service for different customer.

### 3.2 Network Slicing

Network slicing is dividing a physical network into multiple virtual networks or slices[12]. Each virtual network or slice is fulfill specific requirements of different applications or services and is isolated from other slice to ensure privacy, security and performance. Network slicing concept came many years ago but in previous generation networks there is no scope to implement it. One of the key benefits of network slicing is its ability to enable the deployment of diverse and specialized services on a single physical infrastructure. For example, a network operator can create different slices to support applications with varying requirements, such as a low-latency slice for autonomous vehicles, a high-bandwidth slice for video streaming, and a low-cost slice for Internet of Things (IoT) devices. Network slicing can also enable new business models, as network operators can offer slices to third-party service providers who can then offer their own services on top of the slices.

Network slicing involves of defining an isolated subset of the available virtual resources (computation, networking, storage) together with a set of rules for identifying the traffic that will run on those. A network slice consists of a set of virtual resources and the traffic flows associated with it. A network slice can be defined by slicing the available resources in the forms of:

- Bandwidth: Each slice should have its own fraction of bandwidth on a link.
- Topology: Each slice should have its own view of network nodes (switches, routers) and the connectivity between them.
- Device CPU: Each slice should be assigned proper computational resources.
- Storage: Each slice might have varying levels of storage capacity.

- **Storage:** Each slice might have varying levels of storage capacity.
- **Traffic:** A specific portion of the traffic to one (or more) virtual networks should be associated with a slice in order to be cleanly isolated from the remaining underlying network.

An effective procedure to build and manage network slices is to leverage the principles of NFV and SDN, described in the corresponding chapters of this book. In brief, the combination of the abstraction possible through SDN with the freedom in deployment of functionalities deriving from NFV allow the proper level of control on the network, computation, and storage resources to build and manage slices.

### 3.3 Benefits of Network Slicing

Network slicing offers several benefits to network operators and their customers. Some of the key benefits are:

- **Customized Services:** Network slicing enables network operators to offer customized services to their customers. They can create virtual networks that are tailored to meet specific requirements such as low latency, high bandwidth, and low power consumption. This allows network operators to offer differentiated services to their customers, which can help them to attract and retain customers.
- **Improved Network Performance:** Network slicing enables network operators to optimize their network performance. They can allocate network resources dynamically based on the requirements of each virtual network. This ensures that each virtual network gets the resources it needs to operate efficiently, which can improve the overall network performance.
- **Efficient Resource Utilization:** Network slicing enables network operators to maximize the utilization of their network resources. They can allocate resources dynamically based on the requirements of each virtual network. This can help to reduce network congestion and ensure that resources are used efficiently.
- **Reduced Costs:** Network slicing can help to reduce network costs. Network operators can create virtual networks that are tailored to meet specific requirements. This can help to reduce the cost of network infrastructure and maintenance.

### 3.4 Network Slicing Example

Figure 3.1 [12] depicts a network topology that deliver three different service through network slicing. It illustrates that these slices are isolated from each other, and the deployment of service slices can be based on demand, allowing for more flexibility beyond the three slices shown in the figure. First slice fulfill eMBB service, second slice fulfill mMTC/mIoT service and third slice fulfill uRLLC service. To achieve proper packet forwarding for each service, an SDN (Software-Defined Networking) controller is utilized. The SDN controller ensures that packets belonging to each service are directed to the appropriate router for slice deployment. This enables efficient management and routing of network traffic, ensuring that each service receives the necessary resources and maintains its isolation within the network.

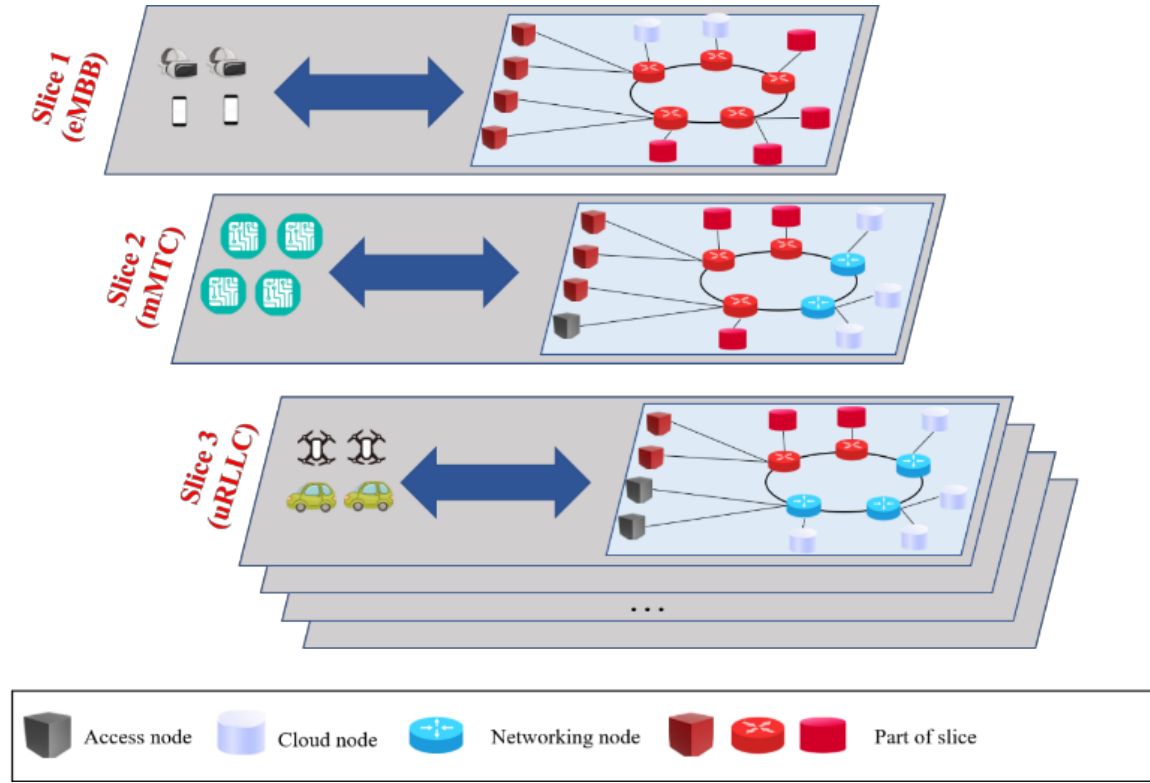


Figure 3.1: Network slicing example (Source-[12])

### 3.5 Network Slicing in 5G

Network slicing is a term commonly seen in the context of 5G[14]. A network slice is a logical network serving a certain business or customer need and consists of the necessary functions from the service-based architecture configured together. For example, one network slice can be set up to support mobile broadband applications with full mobility support, similar to what is provided by LTE, and another slice can be set up to support a specific non-mobile latency-critical industry-automation application. These slices will all run on the same underlying physical core and radio networks, but, from the end-user application perspective, they appear as independent networks.

Control-plane/user-plane split is emphasized in the 5G core network architecture, including independent scaling of the capacity of the two. For example, if more control plane capacity is needed, it is straightforward to add it without affecting the user plane of the network.

### 3.6 Network Slice Security

Network slice security is a critical aspect that needs to be addressed in the context of 5G networks[15]. As network slices comprise dedicated and/or shared resources, ensuring the security and isolation of each slice becomes essential. Network operators have the flexibility to deploy different types of

network slices to cater to the diverse requirements of various verticals or bundle multiple slices to meet the needs of business customers with multiple demands.

In the case of deploying a single network slice type for multiple verticals, it is crucial to implement robust security measures that can effectively isolate and protect the data and communications within each slice. This involves employing strong authentication mechanisms, access control policies, and encryption techniques to safeguard the confidentiality, integrity, and availability of the data transmitted across the network.

Furthermore, in scenarios where a single device or user requires multiple network slices with different requirements, such as a vehicle needing both a high bandwidth slice for infotainment and an ultra-reliable slice for telemetry and assisted driving, additional security considerations come into play. These include ensuring proper segregation and isolation between the different slices to prevent interference and unauthorized access. Access control policies should be in place to allow only authorized entities to access specific slices, and encryption should be applied to protect the data exchanged within each slice.

To enhance network slice security, continuous monitoring and threat detection mechanisms should be implemented to identify any suspicious activities or potential breaches. Intrusion detection and prevention systems can be deployed to detect and respond to security incidents promptly. Additionally, regular security audits and vulnerability assessments should be conducted to identify and address any potential weaknesses or vulnerabilities in the network slice infrastructure.

Standardization efforts play a crucial role in ensuring consistent and effective security across different network slice deployments. Developing standardized security frameworks, protocols, and interfaces will facilitate interoperability and enable seamless integration of security measures within network slices.

In summary, network slice security is vital for protecting the confidentiality, integrity, and availability of data and services in 5G networks. By implementing robust security measures, isolation mechanisms, and continuous monitoring, network operators can ensure that each network slice operates securely and meets the specific requirements of the verticals or customers it serves.

## Chapter 4

# Software Defined Networking

### 4.1 Introduction

Managing traditional computer networks can be a challenging task, and the limitations of traditional network architecture and protocols can further add to the complexity. The lack of a fundamental abstraction in network protocols means that network administrators have to deal with a multitude of protocols that are not necessarily designed to work together seamlessly.

Furthermore, the vendor dependence of network devices and the variations in configuration interfaces across devices from the same vendor can also make network management a cumbersome task. This not only increases operational costs but also raises the potential for inconsistencies and errors in network configurations.

Networking technology has been developed to connect hosts over long distances and support various network link speeds and topologies. As market needs have evolved, many high-performing, reliable, and secure networking protocols have been developed and improved over time. However, these protocols have typically been designed in isolation from one another, which has resulted in networks becoming increasingly complex.

One of the primary issues with this approach is that each protocol is designed to solve specific problems and does not provide a fundamental abstraction for networking. This has made it difficult to manage and scale networks, particularly as the number of networking devices in a network grows. Configuring hundreds or thousands of devices to implement a network-wide policy is a time-consuming and error-prone process, which can lead to inconsistencies and network disruptions.

### 4.2 Software-Defined Networking (SDN)

Software-Defined Networking (SDN) [13] is a relatively new concept in the field of network engineering, and it is changing the way networks are designed and managed. One of the key characteristics of SDN is the separation of the data plane and control plane of network devices. The data plane is responsible for forwarding network traffic, while the control plane decides how to handle that traffic.

With SDN, the control plane is consolidated, allowing a single software program to act as a central control software. This software, known as the SDN controller, can then control the data

planes of multiple network devices. This approach provides greater flexibility and control, allowing network administrators to manage the entire network from a central location. There are many popular SDN controllers available, each with its own unique set of features and capabilities.

The main benefit of separating the control and data planes in SDN is abstraction. Similar to how an operating system abstracts the underlying hardware for application developers, SDN allows for the abstraction of the underlying network hardware for networking application developers. This is achieved by separating the control and data forwarding functions of networking devices, making it possible to build a common interface through which a central controller can control the forwarding behavior of multiple switches from different vendors. In traditional networking devices, the control and data planes are tightly coupled, making it difficult for developers to create applications without having to consider the specifics of the different devices. By separating the planes, SDN allows for greater flexibility and ease of development.

To communicate with networking devices in an SDN environment, a secure connection using SSH and APIs is used. One commonly used API is the OpenFlow protocol, which outlines the standards and requirements for communication between an SDN controller and the network device. These network devices are called OpenFlow switches, which can act as switches, routers, NATs, or other networking devices depending on the instructions given by the controller. OpenFlow switches contain one or more flow-tables that store flow rules or entries, which are used to match network traffic and perform actions such as forwarding, dropping, or flooding based on the defined rules.

SDN has become a significant technology in the networking industry as it allows for designing and managing networks in non-traditional ways. Many commercial switches now support OpenFlow, which is a common and well-known API for communication between an SDN controller and the agent network device. The flow entries on OpenFlow switches store information that helps match network traffic and perform actions on the matching traffic. This has allowed network application developers to build load balancers, dynamic access control, and many other useful applications and network features.

SDN has been evolving over a long period and has transformed from the old telephone networks where there was a separation of the control plane and the data plane to simplify the network and manage it better. The SDN controllers give programmers the ability to innovate, which was not possible in the earlier closed networks of telephone services. Today, there are SDN industry consortia like the Open Networking Foundation and the Open Daylight initiative, with many top information-technology companies being a part of them.

## 4.3 SDN Architecture

The SDN model follows a 3-tier architecture, similar to the architecture of a typical operating system. This analogy helps in understanding the SDN model more easily. A desktop operating system can be divided into three layers: the operating system, the hardware, and the applications. The operating system acts as an interface between the applications and the hardware, managing access and providing core services. The hardware layer includes the CPU, memory, storage, and network interfaces. Above the operating system are the applications, which can be developed, added, or removed as needed.

Similarly, in the SDN model, the network is divided into three layers. The middle layer is called the network operating system (NOS), which is also known as the SDN controller. The NOS manages the network nodes and provides a programmable interface for network applications. The lower layer is made up of forwarding devices, which receive and take action on packets, such as dropping or

modifying packet headers. The forwarding devices are controlled by the NOS. The upper layer is comprised of network applications that serve various purposes, such as traffic engineering or security.

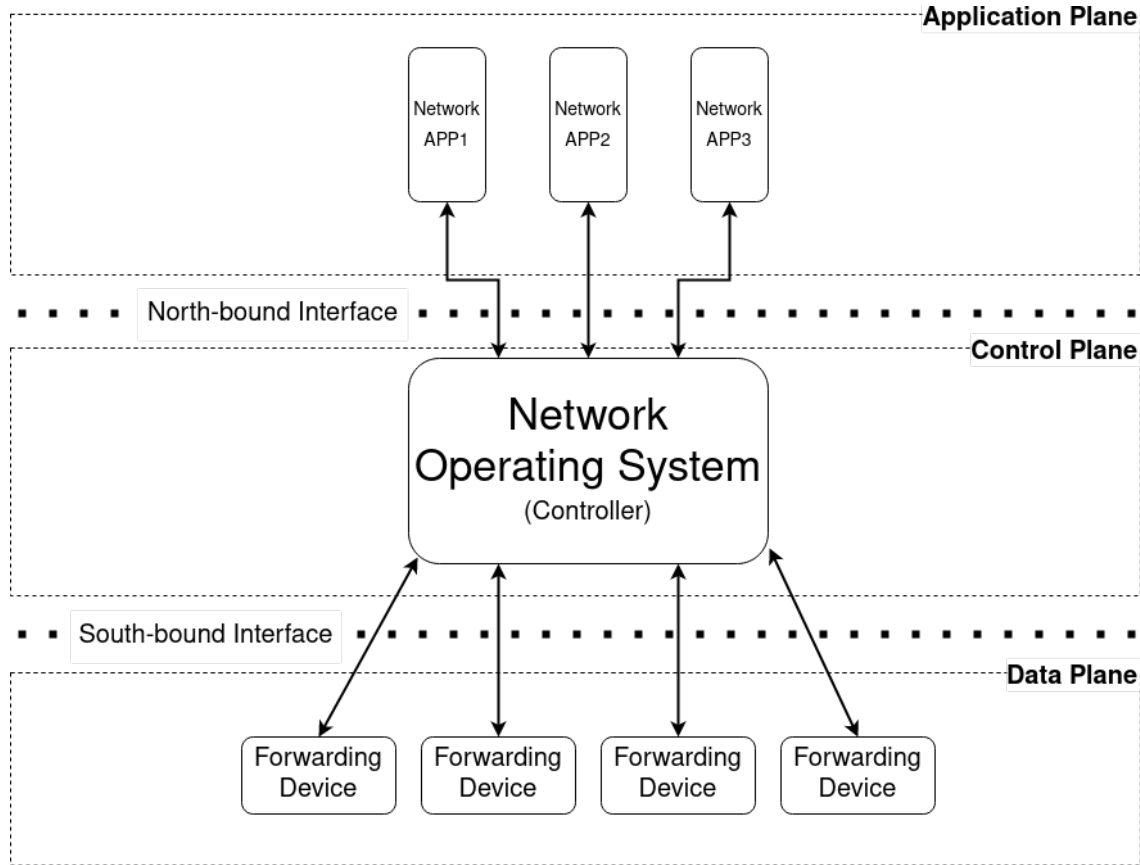


Figure 4.1: SDN architecture

The separation of the planes and open interfacing in the SDN model allows for the development of creative network applications. There are several popular SDN controllers with extensive APIs, such as NOX, POX, and Floodlight. The most commonly used protocol for the south-bound interface is OpenFlow, which provides communication standards between the controller and the forwarding devices. The OpenFlow protocol has a strong community and continues to evolve over time.

## 4.4 SDN Controller

While there are many SDN Controllers like POX, Project Floodlight, Open Network Operating System (ONOS) and OpenDaylight, the work presented in this report focuses on the Ryu controller.

## 4.5 OpenFlow Protocol

The OpenFlow protocol [6] is a key component of Software-Defined Networking (SDN) that enables communication between the SDN controller and network switches. It provides a standardized way for the controller to manage and control the forwarding behavior of network devices.

OpenFlow operates over a secure channel using the Transport Layer Security (TLS) protocol. It utilizes TCP ports 6633 or 6653 for communication between the controller and the supporting switches. By establishing this secure channel, the controller can send instructions to modify the flow table in the switches, which determines how network traffic is forwarded.

Since its initial development, OpenFlow has undergone several revisions and improvements. In 2011, the Open Networking Foundation (ONF) took over its management, aiming to promote and adopt SDN through open standards development. The protocol has evolved over time, with the latest version being 1.5.1. However, it's worth noting that network hardware typically supports up to version 1.3 of the OpenFlow protocol.

The continuous evolution of OpenFlow reflects the ongoing efforts to enhance SDN capabilities and ensure compatibility between the controller and network devices. By utilizing OpenFlow, SDN architectures can achieve centralized control, dynamic network provisioning, and flexible traffic management, ultimately enabling more efficient and programmable networks.

## 4.6 Open vSwitch

Open vSwitch (OvS) [5] is an open-source software implementation of a virtual switch. It was developed as a result of the OpenFlow protocol and the initial specification of a virtual Switch daemon (vswitchd) in 2008. OvS provides a flexible and scalable solution for network virtualization and software-defined networking (SDN) deployments.

One of the key features of Open vSwitch is its ability to operate over the OpenFlow protocol. OpenFlow is a standardized protocol that enables communication between the control plane and the data plane in SDN architectures. By leveraging OpenFlow, OvS allows network administrators to centrally manage and control the behavior of virtual switches in their virtualized environments.

Open vSwitch is particularly well-suited for virtualized environments where the use of physical switches is not necessary or desired. It can be seamlessly integrated into hypervisors like KVM, Xen, and VMware, providing networking capabilities for virtual machines and containers. OvS supports various tunneling protocols, such as VXLAN and GRE, allowing for the creation of virtual networks that span across physical hosts.

The flexibility of Open vSwitch extends beyond virtualization. It can also be deployed in traditional networking scenarios, enabling organizations to introduce SDN principles into their existing infrastructure. OvS supports features like VLAN tagging, access control lists (ACLs), Quality of Service (QoS) policies, and load balancing, empowering administrators to configure and manage their networks with granular control.

Being an open-source project, Open vSwitch benefits from a vibrant community of developers and users who contribute to its ongoing development and improvement. It is widely adopted in both academic and industrial settings and has become a cornerstone technology in the realm of software-defined networking.

Overall, Open vSwitch provides a powerful and flexible solution for network virtualization and SDN deployments, enabling efficient and centralized management of virtual switches in various environments, from data centers to cloud computing platforms. Its support for the OpenFlow



protocol and extensive feature set make it a valuable tool for network administrators seeking to optimize their network infrastructure and embrace the benefits of software-defined networking.

# Chapter 5

## 5G and SDN

### 5.1 Introduction

Main characteristic of SDN is the separation of the data plane and control plane of network devices. And NFV, or Network Functions Virtualization, refers to the process of virtualizing various network functions, such as firewalls, TCP optimizers, VPN, and DPI, and running them on standard commodity hardware devices. In previous generation network there is no scope to implement it but 5G network architecture gives us scope to using SDN & NFV we can implement the 5G network.

### 5.2 Use of SDN in 5G Network Slicing

In 5G network slicing we have manage the Control plane, Signal processing and Slice management.

#### 5.2.1 How We Manage Control Plane

To implement the network slicing architecture effectively, it is necessary to separate the control plane from the data plane. In practice, this can be achieved by utilizing the open-source SDN (Software Defined Networking) controller called Ryu and the OpenFlow protocol.

We leverages Ryu controller and OpenFlow to establish the control plane interface. Through Ryu controller, the resource management of network slice instances (NSIs) is realized, enabling efficient allocation and utilization of resources. The communication between components is facilitated through Application Programming Interfaces (APIs).

To ensure isolation between network slices, the system logically separates each slice using a combination of NFV (Network Functions Virtualization) and SDN technologies. Each slice is provided with the necessary network functions it requires, allowing for customization and tailored services.

In simpler terms, the implementation of network slicing architecture involves separating the control and data planes. This is achieved by using the Ryu controller and OpenFlow protocol to manage network resources and enable communication. Network slices are isolated from each other by combining NFV and SDN, ensuring that each slice has access to the specific network functions it needs for optimal operation.

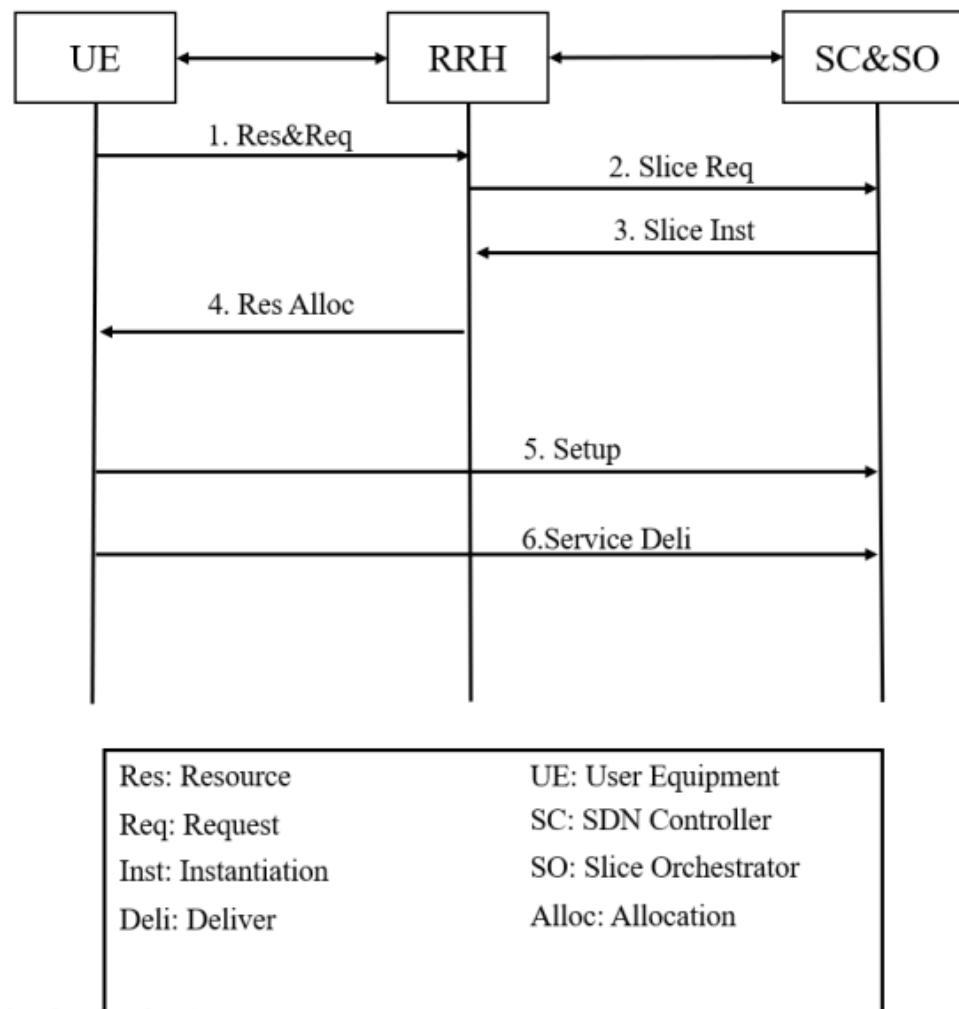


Figure 5.1: Signal Processing of the 5g Network Slicing (Source-[12])

### 5.2.2 Managing Signal Processing and Slicing

Figure 5.1 [12] depicts the signal processing involved in the 5G network slicing implementation mechanism. The process begins with the User Equipment (UE) sending its resource requirements to the Remote Radio Head (RRH). The RRH then forwards these requirements to the SDN Controller and Slice Orchestrator (SC&SO). The SC&SO is responsible for allocating the necessary resources to the UE and establishing a service-based slice dedicated to that UE.

In different service scenarios within the 5G context, there are distinct process sequences. In the case of enhanced Mobile Broadband (eMBB) services and Massive Internet of Things (mIoT) services, the service requirements are initially transmitted to the SC&SO through the API. Subsequently, the SC&SO sets the required resources for slicing based on the available network and spatial resources of the system.

However, in the case of Ultra-Reliable Low Latency Communications (URLLC) services, the required resources are directly created through the SC without involving the network functions managed by the SO. This allows the URLLC service to be delivered more rapidly, as it bypasses certain steps involved in the resource allocation process.

In summary, the 5G network slicing implementation mechanism involves the UE communicating its resource requirements, which are then passed to the RRH, SDN Controller, and Slice Orchestrator. Depending on the service scenario, the SC&SO either sets the required resources for slicing or the resources are directly created through the SC for faster delivery in URLLC services.

## 5.3 Network Function Virtualization (NFV) in 5G

NFV, which stands for Network Function Virtualization, [16] refers to the virtualization of network functions such as firewalls, TCP optimizers, NAT64, VPN, and DPI, on standard hardware devices. In NFV, Virtual Network Functions (VNFs) are instantiated on commodity hardware, breaking away from the traditional approach of relying on specialized software and hardware offered by vendors.

The key advantage of NFV is that it allows for easy deployment and dynamic allocation of Network Functions (NFs). It also enables efficient allocation of network resources to VNFs through dynamic scaling, which facilitates Service Function Chaining (SFC). By utilizing software-based NFV solutions, certain NFs can be moved to run on shared infrastructure, such as general-purpose servers, within Service Providers (SPs). This means that adding, removing, or updating functions for customers becomes more manageable, as changes can be made at the ISP level instead of at individual customer premises.

For Service Providers, NFV offers flexibility to scale services up or down to meet changing customer demands. It helps reduce capital and operational expenditure through cost-effective and agile network infrastructures, while also decreasing the deployment time for new network services in the market. In the context of future 5G networks, NFV plays a crucial role in optimizing resource provisioning to end-users, ensuring high Quality of Service (QoS), and guaranteeing the performance of VNF operations, including low latency and failure rates. Furthermore, NFV ensures the compatibility of VNFs with non-VNFs.

To achieve these benefits, NFV introduces three significant differences in how network services are provisioned compared to traditional practices [11]:

- **Decoupling of software from hardware platform:** In NFV, software and hardware entities are not integrated, allowing them to progress independently from each other.

- **Greater flexibility for network functions deployment:** Since software and hardware are detached, they can perform different functions at different times. This enables operators to deploy innovative services using the same hardware platform.
- **Dynamic network operation and service provisioning:** Network operators can introduce tailored services based on customer demands by dynamically scaling the performance of NFV.

## 5.4 Implementing NFVI through VNFs in 5G Core Network Architecture

In the 5G core network architecture, Figure 5.2 [12] illustrates the implementation of virtual network functions (VNFs) within the Network Function Virtualization Infrastructure (NFVI). The key components involved are as follows:

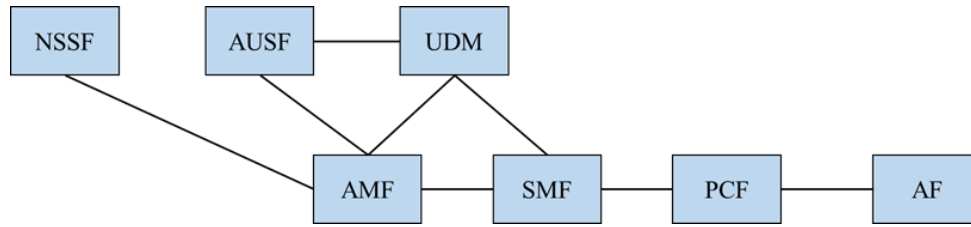


Figure 5.2: Schematic diagram of implementing NFVI through VNFs in 5G core network architecture (Source-[12])

- **Network Slice Selection Function (NSSF):** This function handles slice management, allowing for the creation and management of multiple network slices over the infrastructure.
- **Access and Mobility Management Function (AMF):** The AMF is responsible for session authentication, ensuring secure access and mobility management within the network.
- **Session Management Function (SMF):** The SMF works in conjunction with the AMF to manage sessions and ensure their smooth operation.
- **Authentication Server Function (AUSF):** The AUSF handles the authentication process in collaboration with the Unified Data Management (UDM) database, verifying user credentials and granting access to authorized users.
- **Policy and Charge Function (PCF):** The PCF is responsible for defining and enforcing policies related to network usage and charging, ensuring appropriate service provisioning and billing.
- **Application Function (AF):** The AF represents the functions required for a specific service or application within the network architecture.

Together, these components form the infrastructure for implementing VNFs in the 5G core network, enabling efficient management and delivery of services across network slices while ensuring authentication, policy enforcement, and application-specific functionality.

## Chapter 6

# Experiments

### 6.1 Ryu SDN Controller

Ryu [8] is a Japanese word meaning dragon. It is an SDN Framework developed at Nippon Telegraph and Telephone Corporation (NTT) Laboratories, Software Innovation Centre (SIC) in Japan and first released in 2012.

Ryu framework is a collection of tools and libraries for on the fly creation and management of software defined networks and is compatible with various OpenFlow versions 1.0, 1.2, 1.3, 1.4, 1.5 and Nicira extensions. Ryu is a component-based software defined networking (SDN) framework implemented entirely in Python. Another major advantage of Ryu is that it supports multiple southbound protocols, such as OpenFlow, Network Configuration Protocol (NETCONF), OpenFlow Management and Configuration Protocol (OF-Config), and others.

Installing Ryu:

```
sudo apt install python3-pip
sudo pip3 install ryu
```

Ryu gets installed in the folder `/usr/local/lib/python3.8/dist-packages/ryu`. Current version of Ryu is 4.34. In Ubuntu 22.04 LTS, while trying to run `ryu-manager` you may get the following error:

```
ImportError: cannot import name 'ALREADY_HANDLED' from
'eventlet.wsgi' (/home/msb/.local/lib/python3.10/site-packages/eventlet/wsgi.py)
```

This error might be due to version of installed `eventlet` module. Installing older version of `eventlet` module will solve the problem.

```
sudo pip3 show eventlet
sudo pip3 uninstall eventlet
sudo pip3 install eventlet==0.30.2
```

The home directory for the ryu application is:

```
/usr/local/lib/python3.8/dist-packages/ryu
```

The ryu application executables are located in:

```
/usr/local/bin
```

## 6.2 Mininet Emulator

Mininet [2] [3] is a network emulator that enables creation of instant virtual network. It creates a realistic virtual network of virtual hosts, switches, controllers, and links, on a single machine (VM, cloud or native) with various command line options. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and Software-Defined Networking.

Installing Mininet:

```
sudo apt install git
sudo apt-get install mininet
```

Once installed, you can check by running the following commands:

```
mn --version
sudo mn --switch ovsbr --test pingall
sudo mn --switch ovs --controller ref
    --topo tree,depth=2,fanout=8 --test pingall
```

If you wish to go through the Mininet walkthrough, you will want to install additional software. The following commands do that:

```
git clone https://github.com/mininet/mininet
mininet/util/install.sh -fw
```

## 6.3 How to Use Ryu Controller in Mininet Topology

First start the Ryu controller python file(Here I run the simple\_switch\_13 which support OpenFlow version 1.3) [7]:

```
ryu-manager ryu.app.simple_switch_13
```

It gives the result like Figure 6.1:

```
mdjs@juel:~$ ryu-manager ryu.app.simple_switch_13
loading app ryu.app.simple_switch_13
loading app ryu.controller.ofp_handler
instantiating app ryu.app.simple_switch_13 of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
█
```

Figure 6.1: Starting the Ryu controller

This Ryu controller listening in port 6653 of localhost (127.0.0.1). But if we want to listen to manual port then the command is:

```
ryu-manager --ofp-tcp-listen-port 6634 ryu.app.simple_switch_13
```

After that we need to start the mininet network topology [3] with that remote Ryu controller which actually run in 127.0.0.1:6653. Here I start a linear network topology with 4 switch and 4 host:

```
sudo mn --controller=remote,ip=127.0.0.1:6653 --switch=ovsk,protocols=
OpenFlow13 --topo=linear,4
```

It gives the result like Figure 6.2:

```
mdjs@juel:~$ sudo mn --controller=remote,ip=127.0.0.1:6653 --switch=ovsk,protocols=OpenFlow13 --topo=linear,4
[sudo] password for mdjs:
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (h4, s4) (s2, s1) (s3, s2) (s4, s3)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:
mininet> █
```

Figure 6.2: Starting the Mininet with linear topology and Ryu controller

Showing the dump-flow of switch:

```
sudo ovs-ofctl -O OpenFlow13 dump-flows <switch_name>
```

Before ping any host dump-flow of all switch:



```
mdjs@juel:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s1
cookie=0x0, duration=7.467s, table=0, n_packets=72, n_bytes=9313, priority=0 actions=CONTROLLER:65535
mdjs@juel:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s2
cookie=0x0, duration=9.252s, table=0, n_packets=75, n_bytes=9922, priority=0 actions=CONTROLLER:65535
mdjs@juel:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s3
cookie=0x0, duration=10.867s, table=0, n_packets=75, n_bytes=9922, priority=0 actions=CONTROLLER:65535
mdjs@juel:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s4
cookie=0x0, duration=12.291s, table=0, n_packets=76, n_bytes=9992, priority=0 actions=CONTROLLER:65535
mdjs@juel:~$ █
```

Figure 6.3: Before ping dump-flow of all switch

Ping h2 host from h1 host:

```
mininet> h1 ping -c 4 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=8.73 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.230 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.081 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.084 ms

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3051ms
rtt min/avg/max/mdev = 0.081/2.280/8.727/3.722 ms
mininet> █
```

Figure 6.4: Ping h2 from h1

After ping h2 from h1 dump-flow of all switch:

```
mdjs@juel:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s1
cookie=0x0, duration=41.578s, table=0, n_packets=5, n_bytes=434, priority=1,in_port="s1-eth2",dl_src=92:e4:21:44:ec:0e,dl_dst=52:64:04:90:ee:b4
4 actions=output:"s1-eth1"
cookie=0x0, duration=41.577s, table=0, n_packets=4, n_bytes=336, priority=1,in_port="s1-eth1",dl_src=52:64:04:90:ee:b4,dl_dst=92:e4:21:44:ec:0e
4 actions=output:"s1-eth2"
mdjs@juel:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s2
cookie=0x0, duration=271.001s, table=0, n_packets=128, n_bytes=15599, priority=0 actions=CONTROLLER:65535
mdjs@juel:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s3
cookie=0x0, duration=43.546s, table=0, n_packets=5, n_bytes=434, priority=1,in_port="s2-eth1",dl_src=92:e4:21:44:ec:0e,dl_dst=52:64:04:90:ee:b4
4 actions=output:"s2-eth2"
cookie=0x0, duration=43.542s, table=0, n_packets=4, n_bytes=336, priority=1,in_port="s2-eth2",dl_src=52:64:04:90:ee:b4,dl_dst=92:e4:21:44:ec:0e
4 actions=output:"s2-eth1"
mdjs@juel:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s4
cookie=0x0, duration=272.967s, table=0, n_packets=128, n_bytes=15599, priority=0 actions=CONTROLLER:65535
mdjs@juel:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s3
cookie=0x0, duration=274.664s, table=0, n_packets=126, n_bytes=15459, priority=0 actions=CONTROLLER:65535
mdjs@juel:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s4
cookie=0x0, duration=276.436s, table=0, n_packets=126, n_bytes=15459, priority=0 actions=CONTROLLER:65535
mdjs@juel:~$ █
```

Figure 6.5: After ping dump-flow of all switch

## 6.4 RYU Framework Overview

Ryu [8] applications are just Python scripts so we can save the file as (.py) on any location.

### 6.4.1 Components

#### **ryu-manager :**

- Main executable

#### **ryu.base.app\_manager:**

- The central management of Ryu applications Load Ryu applications
- Provide contexts to Ryu applications

#### **ryu.ofproto:**

- OpenFlow wire protocol encoder and decoder.

#### **ryu.controller:**

- openflow controller implementation
- generates openflow events

#### **ryu.packet:**

- all packet parsing libraries

### 6.4.2 Events:

RYU Application works based on Events. RYU Controller emits the events for the Openflow Messages received. This can be handled by the RYU Applications.

Example Events:

```
ofp_event.EventOFPSwitchFeatures ofp_event.EventOFPPacketIn ofp_event
.EventOFPPFlowStatsReply
```

### 6.4.3 Inserting/Adding a New Flow Using Program

Flow table consist of Match, Action, Counters so we need to first assign these things

#### A. Create a Match

This match is with no match field. It means matching all packets.

```
match = parser.OFPMatch()
```

This matches with in\_port, eth\_dst and eth\_src field.

```
match = parser.OFPMatch(in_port=in_port ,
                        eth_dst=dst ,
                        eth_src=src)
```

#### B. Create a Actions

Below action is send it to CONTROLLER(Reserved port).

```
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                   ofproto.OFPCML_NO_BUFFER)]
```

Below action, send it to port number 1.

```
out_port = 1
actions = [parser.OFPActionOutput(out_port)]
```

### C. Create a Instruction List with Actions

```
inst = [parser.OFPInstructionActions(
                                             ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
```

### D. Send OFP Flow Modification Message for Creating a New Flow

Most of the parameter are default. table\_id, timeouts, cookies etc.

```
mod = parser.OFPFlowMod(datapath=datapath,
                        buffer_id=buffer_id,
                        priority=priority,
                        match=match,
                        instructions=inst)
```

In Below example, explicitly specify tableid, timeouts.

```
mod = parser.OFPFlowMod(datapath=datapath,
                        table_id=10,
                        buffer_id=buffer_id,
                        idle_timeout=10,
                        hard_timeout=30,
                        priority=priority,
                        match=match,
                        instructions=inst)
```

## 6.5 QoS (Quality of Service) in the Ryu controller

QoS (Quality of Service) [7][9][10] in the Ryu controller refers to the ability to prioritize and manage network traffic based on different service requirements.

In the context of Ryu, QoS functionality can be implemented using various mechanisms, such as OpenFlow and network flow rules. Here's a general outline of how QoS can be achieved in Ryu:

1. **Traffic Classification:** First, network traffic needs to be classified into different classes or categories based on specific criteria, such as source/destination IP addresses, port numbers, protocols, or other header fields. This classification is typically done using OpenFlow match fields.

2. **Define QoS Policies:** Once the traffic is classified, QoS policies are defined to determine the treatment of each traffic class. QoS policies specify how the traffic should be prioritized, limited, or otherwise handled. For example, you can assign different levels of priority or allocate bandwidth limits to different traffic classes.
3. **Implement Flow Rules:** Ryu uses OpenFlow protocol to communicate with the network devices. To enforce the defined QoS policies, appropriate flow rules are installed in the switches controlled by Ryu. These flow rules match the classified traffic and specify the corresponding actions to be taken, such as modifying packet headers, applying rate limiting, or marking packets with specific QoS markings.
4. **QoS Actions:** Ryu provides various QoS actions that can be applied to the traffic. These actions can include setting DSCP (Differentiated Services Code Point) values, modifying packet header fields, prioritizing packets in the transmission queue, or dropping packets exceeding bandwidth limits.
5. **Monitoring and Measurement:** To ensure that the QoS policies are being effectively enforced, monitoring and measurement mechanisms can be implemented. Ryu can collect statistics about the network traffic, such as throughput, latency, or packet loss, allowing administrators to assess the performance and make adjustments if needed.

It's important to note that Ryu itself doesn't provide built-in QoS algorithms or policies. Instead, it offers a framework and APIs for developers to implement their own QoS functionality based on their specific requirements.

## 6.6 QoS REST APIs Overview

### 6.6.1 Get Status of Queue

```
GET /qos/queue/status/{switch-id}
```

### 6.6.2 Get a Queue Configurations

```
GET /qos/queue/{switch-id}
```

### 6.6.3 Set a Queue to the Switches

```
POST /qos/queue/{switch-id}
```

#### \* Request Body Format

```
{
  "port_name": "<name of port>",
  "type": "<linux-htb or linux-other>",
  "max_rate": "<int>",
  "queues": [{"max_rate": "<int>", "min_rate": "<int>"}, ...]
}
```

Note: This operation override previous configurations.  
 Note: Queue configurations are available for OpenvSwitch.  
 Note: port\_name is optional argument.  
       If does not pass the port\_name argument,  
       all ports are target for configuration.

#### 6.6.4 Delete Queue

```
DELETE /qos/queue/{switch-id}
```

Note: This operation delete relation of qos record from  
       qos colum in Port table. Therefore ,  
       QoS records and Queue records will remain.

#### 6.6.5 Get Rules of QoS

**\* For no vlan**

```
GET /qos/rules/{switch-id}
```

**\* For specific vlan group**

```
GET /qos/rules/{switch-id}/{vlan-id}
```

#### 6.6.6 Set a QoS Rules

QoS rules will do the processing pipeline, which entries are register the first table (by default table id 0) and process will apply and go to next table.

**\* For no vlan**

```
POST /qos/{switch-id}
```

**\* For specific vlan group**

```
POST /qos/{switch-id}/{vlan-id}
```

**\* Request body format**

```
{
  "priority": "<value>",
  "match": {
    "<field1>": "<value1>",
    "<field2>": "<value2>",
    ...
  },
  "actions": {
    "<action1>": "<value1>",
    "<action2>": "<value2>",
    ...
  }
}
```

Description

\* priority field

```
<value>  
"0 to 65533"
```

Note: When "priority" has not been set up,  
"priority: 1" is set to "priority".

```
* match field  
  <field> : <value>  
  "in_port" : "<int>"  
  "dl_src" : "<xx:xx:xx:xx:xx:xx>"  
  "dl_dst" : "<xx:xx:xx:xx:xx:xx>"  
  "dl_type" : "<ARP or IPv4 or IPv6>"  
  "nw_src" : "<A.B.C.D/M>"  
  "nw_dst" : "<A.B.C.D/M>"  
  "ipv6_src" : "<xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx/M>"  
  "ipv6_dst" : "<xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx/M>"  
  "nw_proto" : "<TCP or UDP or ICMP or ICMPv6>"  
  "tp_src" : "<int>"  
  "tp_dst" : "<int>"  
  "ip_dscp" : "<int>"  
  
* actions field  
  <field> : <value>  
  "mark": <dscp-value>  
  sets the IPv4 ToS/DSCP field to tos.  
  "meter": <meter-id>  
  apply meter entry  
  "queue": <queue-id>  
  register queue specified by queue-id
```

Note: When "actions" has not been set up,  
"queue: 0" is set to "actions".

### 6.6.7 Delete a QoS Rules

#### \* For no vlan

```
DELETE /qos/rule/{switch-id}
```

#### \* For specific vlan group

```
DELETE /qos/{switch-id}/{vlan-id}
```

#### \* Request body format

```
{"<field>": "<value>"}
```

```
<field> : <value>
"qos_id" : "<int>" or "all"
```

### 6.6.8 Set a Meter Entry

```
POST /qos/meter/{switch-id}
```

#### \* Request body format

```
{
  "meter_id": <int>,
  "bands": [{
    "action": "<DROP or DSCP.REMARK>",
    "flag": "<KBPS or PKTPS or BURST or STATS>",
    "burst_size": <int>,
    "rate": <int>,
    "prec_level": <int> } , ... ] }
}
```

### 6.6.9 Delete a Meter Entry

```
DELETE /qos/meter/{switch-id}
```

#### \* Request body format

```
{
  "<field>": "<value>"
}

<field> : <value>
"meter_id" : "<int>"
}
```

## 6.7 Experiment System Architecture

### 6.7.1 Slice Architecture

In this project we develop a network system which act like a 5G network system. Where we use SDN and NFV concept to made different slice for different service requirement. We implement three service and their configuration are

1. **Slice-1:** max-bandwidth is 5Mbps
2. **Slice-2:** max-bandwidth is 15Mbps
3. **Slice-3:** max-bandwidth is 25Mbps

In Figure 6.6 showing three network slice flow. When client-b communicate to server-ue1 that time max-bandwidth is 5Mbps and when communicate to server-ue3 that time max-bandwidth is 15Mbps and when client-c communicate to server-ue2 that time max-bandwidth is 25Mbps.

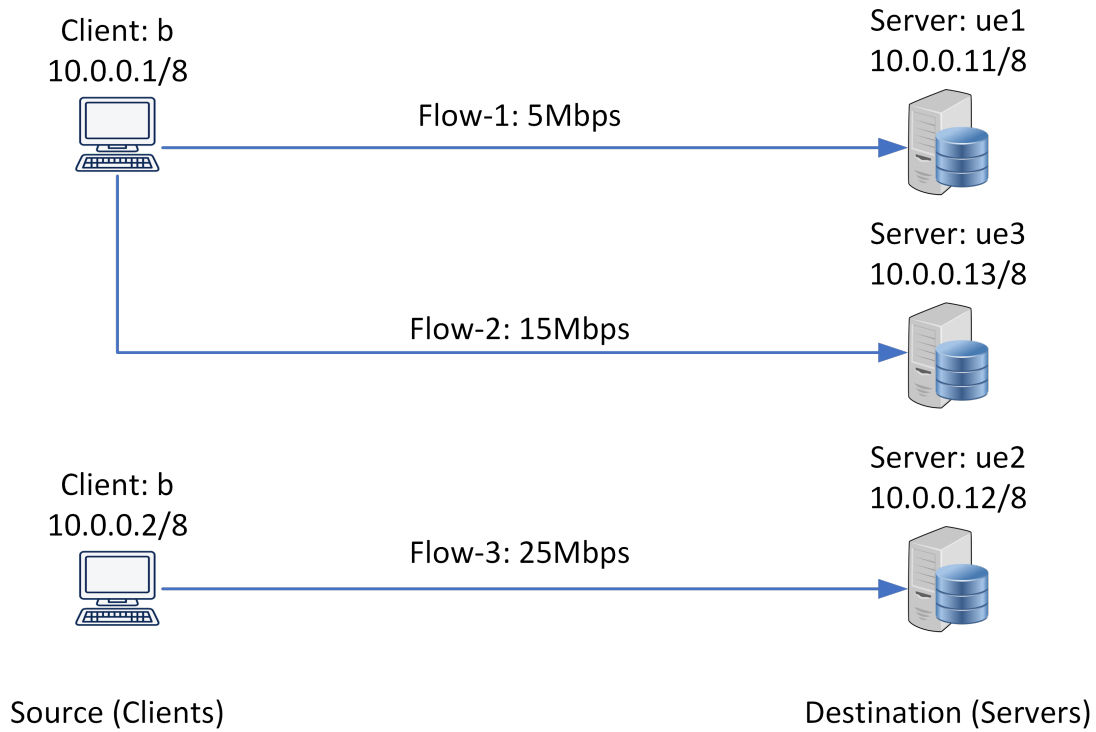


Figure 6.6: Network Slice Flow

### 6.7.2 Network Topology Architecture

In Figure 6.7 showing the experimental network topology. In topology we add 5 hosts (2 clients and 3 servers), 5 OpenFlow switch and 1 controller. Two clients are b (ip=10.0.0.1/8) and c (ip=10.0.0.2/8), three servers are ue1 (ip=10.0.0.11/8), ue2 (ip=10.0.0.12/8) and ue3 (ip=10.0.0.13/8), five OpenFlow switches are nb (dpid=1), rb1 (dpid=2), rb2 (dpid=3), a1 (dpid=4) and a2 (dpid=5), one Ryu controller is c0 (ip=127.0.0.1, port=6633).

**Links of the Network Topology are:**

```

b-eth0<->nb-eth1
c-eth0<->nb-eth2
rb1-eth1<->nb-eth3
rb2-eth1<->nb-eth4
a1-eth1<->rb1-eth2
a2-eth1<->rb2-eth2
ue1-eth0<->a1-eth2
ue2-eth0<->a1-eth3
ue3-eth0<->a2-eth2

```



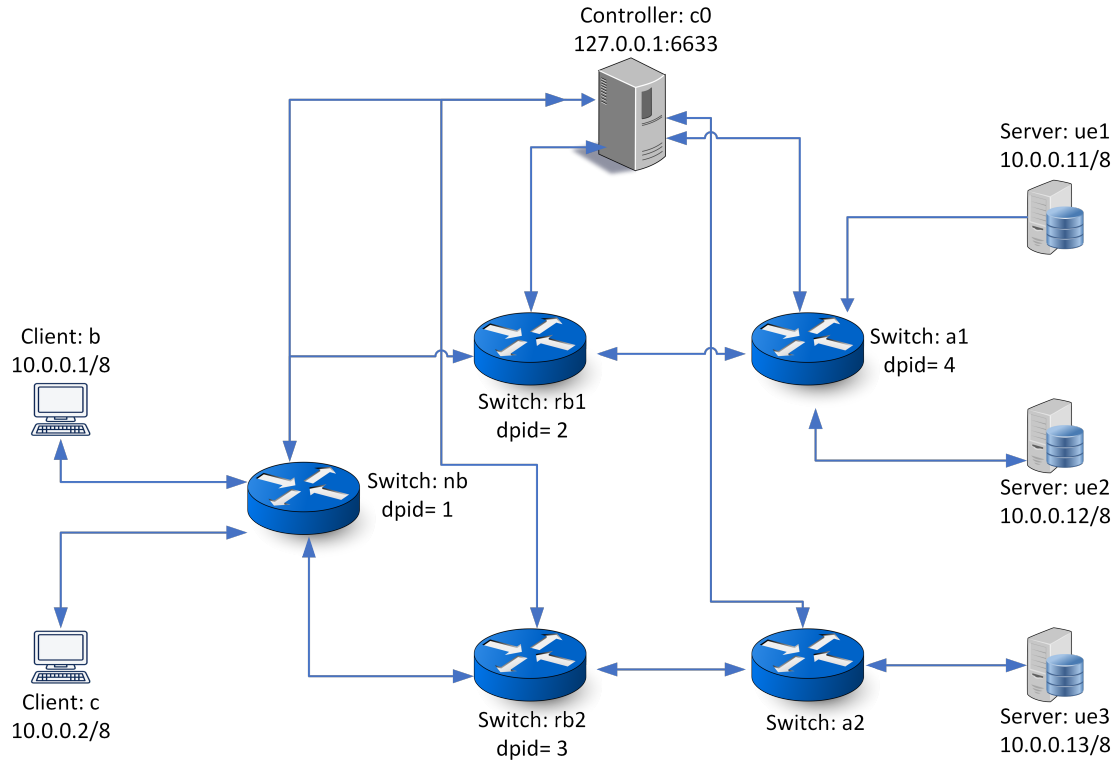


Figure 6.7: Network Topology

## 6.8 Experiment Implementation and Setup

This section describe how we implement the network topology using Mininet python API [4], the controller using Ryu controller API and slice/flow using QoS of Ryu controller REST API.

### 6.8.1 Network Topology Implementation

To create the network topology [4] of Figure 6.7, we write the code in python language and the codes are:

**Code of run\_mininet\_topology.py file:**

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

from mininet.node import RemoteController
from mininet.net import Mininet
from mininet.log import setLogLevel
from mininet.cli import CLI
import topologies.topologies
```

```

if __name__ == '__main__':
    # Tell mininet to print useful information
    setLogLevel('info')
    topo = topologies.topologies.ExperimentTopo()
    net = Mininet(topo, controller=RemoteController('c0',
                                                    ip='127.0.0.1',
                                                    port=6633))

    net.start()
    CLI(net)
    net.stop()

```

### Code Explanation of run\_mininet\_topology.py:

Let's go through the code to understand its functionality:

1. The script begins with importing necessary modules from the Mininet library, including RemoteController, Mininet, setLogLevel, CLI, and topologies.topologies.
2. RemoteController is used to specify a remote controller for the Mininet network. In this case, the Ryu controller is set to 'c0', with the IP address '127.0.0.1' (localhost) and port 6633.
3. The setLogLevel function is called to set the log level of Mininet to 'info', which means it will print useful information during network setup and operation.
4. The line `topo = topologies.topologies.ExperimentTopo()` creates an instance of a custom topology called 'ExperimentTopo'.
5. The Mininet class is instantiated with the created topology `topo` and the specified RemoteController. This sets up the Mininet network with the given topology and controller configuration.
6. `net.start()` is called to start the network. This initiates the network emulation and connects the Mininet switches to the specified controller.
7. `CLI(net)` is used to start the Mininet command-line interface (CLI) where you can interact with the network. The script execution will pause at this point, and you will have access to the Mininet CLI to execute commands and test the network.
8. Finally, `net.stop()` is called to stop the network and clean up any resources used by Mininet.

### Code of topologies.py file:

```

from mininet.topo import Topo

class ExperimentTopo(Topo):

    def build(self, *args, **params):
        # Add hosts, switches and links
        b = self.addHost('b', ip='10.0.0.1/8')

```

```

c = self.addHost('c', ip='10.0.0.2/8')
nb = self.addSwitch('nb',
                    datapath='osvk',
                    protocols='OpenFlow13',
                    dpid='1')

self.addLink(b, nb)
self.addLink(c, nb)

rb1 = self.addSwitch('rb1',
                    datapath='osvk',
                    protocols='OpenFlow13',
                    dpid='2')
rb2 = self.addSwitch('rb2',
                    datapath='osvk',
                    protocols='OpenFlow13',
                    dpid='3')

self.addLink(rb1, nb)
self.addLink(rb2, nb)

a1 = self.addSwitch('a1',
                    datapath='osvk',
                    protocols='OpenFlow13',
                    dpid='4')

self.addLink(a1, rb1)
ue1 = self.addHost('ue1', ip='10.0.0.11/8')
ue2 = self.addHost('ue2', ip='10.0.0.12/8')
self.addLink(ue1, a1)
self.addLink(ue2, a1)

a2 = self.addSwitch('a2',
                    datapath='osvk',
                    protocols='OpenFlow13',
                    dpid='5')

self.addLink(a2, rb2)
ue3 = self.addHost('ue3', ip='10.0.0.13/8')
self.addLink(ue3, a2)

topos = {'experiment_topo': (lambda: ExperimentTopo())}

```

### Code Explanation of topologies.py:

Let's go through the code to understand its functionality:

1. The code starts by importing the Topo class from the mininet.topo module. This class is used to create custom network topologies in Mininet.
2. The ExperimentTopo class is defined, inheriting from the Topo class. This allows us to create

a custom topology by implementing the build method.

3. Inside the build method, the network topology is defined by adding hosts, switches, and links. The topology consists of several components connected to each other.
4. The code adds two hosts b and c using the addHost method. The IP addresses of the hosts are specified as '10.0.0.1/8' and '10.0.0.2/8', respectively.
5. Next, a switch named nb is added using the addSwitch method. The switch is configured with the datapath type 'osvk', supports OpenFlow version 1.3 (protocols='OpenFlow13'), and has a datapath ID (DPID) set to '1'.
6. Links are added to connect the hosts (b and c) to the switch nb using the addLink method.
7. Another two switches rb1 and rb2 are added in a similar manner. They are also configured with the datapath type 'osvk', support OpenFlow version 1.3, and have unique DPIDs.
8. Links are added to connect rb1 and rb2 to the switch nb.
9. Another switch a1 is added, connected to rb1. It has similar configuration settings.
10. Two hosts named ue1 and ue2 are added, along with links to connect them to the switch a1.
11. Similarly, another switch a2 is added and connected to rb2.
12. A host named ue3 is added and linked to a2.
13. Finally, a dictionary named topos is defined. It maps the string key 'experiment\_topo' to a lambda function that creates an instance of the ExperimentTopo class when called.

## 6.8.2 Network Topology Setup

Run the file run\_mininet\_topology.py for setup the network topology.

```
sudo python3 run_mininet_topology.py
```

It gives the result like Figure 6.8

```

test@juel:~/Experiment/mininet$ sudo python3 run_mininet_topology.py
[sudo] password for test:
Unable to contact the remote controller at 127.0.0.1:6633
*** Creating network
*** Adding controller
*** Adding hosts:
b c ue1 ue2 ue3
*** Adding switches:
a1 a2 nb rb1 rb2
*** Adding links:
(a1, rb1) (a2, rb2) (b, nb) (c, nb) (rb1, nb) (rb2, nb) (ue1, a1) (ue2, a1) (ue3, a2)
*** Configuring hosts
b c ue1 ue2 ue3
*** Starting controller
c0
*** Starting 5 switches
a1 a2 nb rb1 rb2 ...
*** Starting CLI:
mininet> █

```

Figure 6.8: Setup network topology

After that run the file a1.sh or below command in terminal of a1 switch for set to listen on port 6632 to access OVSDB.

```
ovs-vsctl set-manager tcp:6632
```

### 6.8.3 Ryu Controller Implementation

We modify simple\_switch\_13.py used in “Switching Hub”. rest\_qos.py suppose to be processed on Flow Table pipeline processing, modify simple\_switch\_13.py to register flow entry into table id:1 and save this as qos\_simple\_switch\_13.py. For this we run the command in terminal:

```
sed '/OFPPFlowMod(/,/s/)/, table_id=1)/'
ryu/ryu/app/simple_switch_13.py > qos_simple_switch_13.py
```

**Code of run\_controller.sh file:**

```
#!/bin/bash

ryu-manager --observe-links flowmanager/flowmanager.py
ryu.app.rest_qos qos_simple_switch_13.py ryu.app.rest_conf_switch
```

We also write the bash script for controller to setting up the ovsdb\_addr, queues and flow.

**Code of c0.sh file:**

```
#!/bin/bash
```

```

# set ovsdb_addr in order to access OVSDB.

curl -X PUT -d '"tcp:127.0.0.1:6632"' http://localhost:8080/v1.0/conf
/switches/0000000000000001/ovsdb_addr

curl -X PUT -d '"tcp:127.0.0.1:6632"' http://localhost:8080/v1.0/conf
/switches/0000000000000002/ovsdb_addr

curl -X PUT -d '"tcp:127.0.0.1:6632"' http://localhost:8080/v1.0/conf
/switches/0000000000000003/ovsdb_addr

curl -X PUT -d '"tcp:127.0.0.1:6632"' http://localhost:8080/v1.0/conf
/switches/0000000000000004/ovsdb_addr

curl -X PUT -d '"tcp:127.0.0.1:6632"' http://localhost:8080/v1.0/conf
/switches/0000000000000005/ovsdb_addr

# execute setting of Queue.

curl -X POST -d '{"type": "linux-htb", "max_rate": "30000000",
"queues": [{"max_rate": "5000000"}, {"max_rate": "15000000"},
{"max_rate": "25000000"}]}' http://localhost:8080/qos
/queue/0000000000000001

curl -X POST -d '{"type": "linux-htb", "max_rate": "30000000",
"queues": [{"max_rate": "5000000"}, {"max_rate": "15000000"},
{"max_rate": "25000000"}]}' http://localhost:8080/qos
/queue/0000000000000002

curl -X POST -d '{"type": "linux-htb", "max_rate": "30000000",
"queues": [{"max_rate": "5000000"}, {"max_rate": "15000000"},
{"max_rate": "25000000"}]}' http://localhost:8080/qos
/queue/0000000000000003

curl -X POST -d '{"type": "linux-htb", "max_rate": "30000000",
"queues": [{"max_rate": "5000000"}, {"max_rate": "15000000"},
{"max_rate": "25000000"}]}' http://localhost:8080/qos
/queue/0000000000000004

curl -X POST -d '{"type": "linux-htb", "max_rate": "30000000",
"queues": [{"max_rate": "5000000"}, {"max_rate": "15000000"},
{"max_rate": "25000000"}]}' http://localhost:8080/qos
/queue/0000000000000005

# Install the following flow entry to the switch.

```

```

curl -X POST -d '{"match": {"nw_dst": "10.0.0.11", "nw_proto": "UDP",
"tp_dst": "5001"}, "actions":{"queue": "0"}}' http://localhost:8080
/qos/rules/0000000000000001

curl -X POST -d '{"match": {"nw_dst": "10.0.0.13", "nw_proto": "UDP",
"tp_dst": "5003"}, "actions":{"queue": "1"}}' http://localhost:8080
/qos/rules/0000000000000001

curl -X POST -d '{"match": {"nw_dst": "10.0.0.12", "nw_proto": "UDP",
"tp_dst": "5002"}, "actions":{"queue": "2"}}' http://localhost:8080
/qos/rules/0000000000000001

curl -X POST -d '{"match": {"nw_dst": "10.0.0.11", "nw_proto": "UDP",
"tp_dst": "5001"}, "actions":{"queue": "0"}}' http://localhost:8080
/qos/rules/0000000000000002

curl -X POST -d '{"match": {"nw_dst": "10.0.0.13", "nw_proto": "UDP",
"tp_dst": "5003"}, "actions":{"queue": "1"}}' http://localhost:8080
/qos/rules/0000000000000002

curl -X POST -d '{"match": {"nw_dst": "10.0.0.12", "nw_proto": "UDP",
"tp_dst": "5002"}, "actions":{"queue": "2"}}' http://localhost:8080
/qos/rules/0000000000000002

curl -X POST -d '{"match": {"nw_dst": "10.0.0.11", "nw_proto": "UDP",
"tp_dst": "5001"}, "actions":{"queue": "0"}}' http://localhost:8080
/qos/rules/0000000000000003

curl -X POST -d '{"match": {"nw_dst": "10.0.0.13", "nw_proto": "UDP",
"tp_dst": "5003"}, "actions":{"queue": "1"}}' http://localhost:8080
/qos/rules/0000000000000003

curl -X POST -d '{"match": {"nw_dst": "10.0.0.12", "nw_proto": "UDP",
"tp_dst": "5002"}, "actions":{"queue": "2"}}' http://localhost:8080
/qos/rules/0000000000000003

curl -X POST -d '{"match": {"nw_dst": "10.0.0.11", "nw_proto": "UDP",
"tp_dst": "5001"}, "actions":{"queue": "0"}}' http://localhost:8080
/qos/rules/0000000000000004

curl -X POST -d '{"match": {"nw_dst": "10.0.0.13", "nw_proto": "UDP",
"tp_dst": "5003"}, "actions":{"queue": "1"}}' http://localhost:8080
/qos/rules/0000000000000004

curl -X POST -d '{"match": {"nw_dst": "10.0.0.12", "nw_proto": "UDP",
"tp_dst": "5002"}, "actions":{"queue": "2"}}' http://localhost:8080

```

```

/qos/rules/0000000000000004

curl -X POST -d '{"match": {"nw_dst": "10.0.0.11", "nw_proto": "UDP",
"tp_dst": "5001"}, "actions": {"queue": "0"}}' http://localhost:8080
/qos/rules/0000000000000005

curl -X POST -d '{"match": {"nw_dst": "10.0.0.13", "nw_proto": "UDP",
"tp_dst": "5003"}, "actions": {"queue": "1"}}' http://localhost:8080
/qos/rules/0000000000000005

curl -X POST -d '{"match": {"nw_dst": "10.0.0.12", "nw_proto": "UDP",
"tp_dst": "5002"}, "actions": {"queue": "2"}}' http://localhost:8080
/qos/rules/0000000000000005

```

#### Code Explanation of c0.sh:

Let's go through the code to understand its functionality:

1. The first set of curl commands sets the ovsdb\_addr for each switch. It sends HTTP PUT requests to the Ryu controller's REST API at "http://localhost:8080/v1.0/conf/switches/<switch-dpid>/ovsdb\_addr" endpoint. The ovsdb\_addr is set to "tcp:127.0.0.1:6632", indicating that the OVSDB server is running on the localhost IP address and port 6632. The DPIDs used in the requests are 0000000000000001, 0000000000000002, 0000000000000003, 0000000000000004, and 0000000000000005.
2. The next set of curl commands configures the queues for QoS. It sends HTTP POST requests to the Ryu controller's REST API at "http://localhost:8080/qos/queue/<switch-dpid>" endpoint. The requests specify the queue settings in JSON format. Each switch is configured with a "linux-htb" type of queue, a maximum rate of "30000000" (30 Mbps), and three queues with different maximum rates: "5000000", "15000000", and "25000000".
3. After configuring the queues, the script proceeds to install flow entries for QoS. It uses curl commands to send HTTP POST requests to the Ryu controller's REST API at "http://localhost:8080/qos/rules/<switch-dpid>" endpoint. Each request specifies a flow entry match criteria and corresponding actions in JSON format. The match criteria include the destination IP address (nw\_dst), network protocol (nw\_proto), and transport protocol destination port (tp\_dst). The actions indicate which queue to apply for matching packets. The flow entries are installed for each switch and differentiate traffic based on the destination IP address and UDP transport protocol destination port.

### 6.8.4 Ryu Controller Setup

Run the run\_controller.sh file for setup the Ryu controller.

```
bash run_controller.sh
```

It gives the result like Figure 6.9



```

test@juel:~/Experiment/controller$ bash run_controller.sh
loading app flowmanager/flowmanager.py
You are using Python v3.8.5.final.0
loading app ryu.app.rest_qos
loading app qos_simple_switch_13.py
loading app ryu.app.rest_conf_switch
loading app ryu.controller.ofp_handler
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
creating context wsgi
instantiating app None of DPSet
creating context dpset
instantiating app None of ConfSwitchSet
creating context conf_switch
instantiating app flowmanager/flowmanager.py of FlowManager
Created flowmanager
instantiating app ryu.app.rest_qos of RestQoSAPI
instantiating app qos_simple_switch_13.py of SimpleSwitch13
instantiating app ryu.app.rest_conf_switch of ConfSwitchAPI
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu.topology.switches of Switches
(6585) wsgi starting up on http://0.0.0.0:8080
[QoS][INFO] dpid=000000000000000003: Join qos switch.
[QoS][INFO] dpid=000000000000000004: Join qos switch.
[QoS][INFO] dpid=000000000000000005: Join qos switch.
[QoS][INFO] dpid=000000000000000002: Join qos switch.
[QoS][INFO] dpid=000000000000000001: Join qos switch.

```

Figure 6.9: Setup Ryu controller

Before adding QoS if we check the flow table of all switches then it look like below Figures. We check that flow table in Flow Manger UI tool.

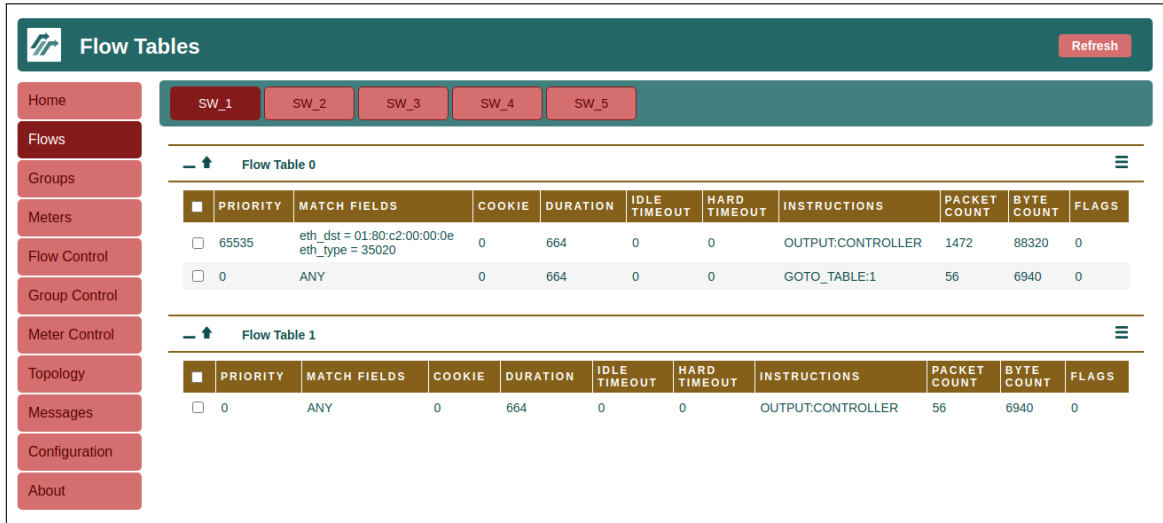


Figure 6.10: Before adding QoS in controller flow table of nb(SW\_1) switch

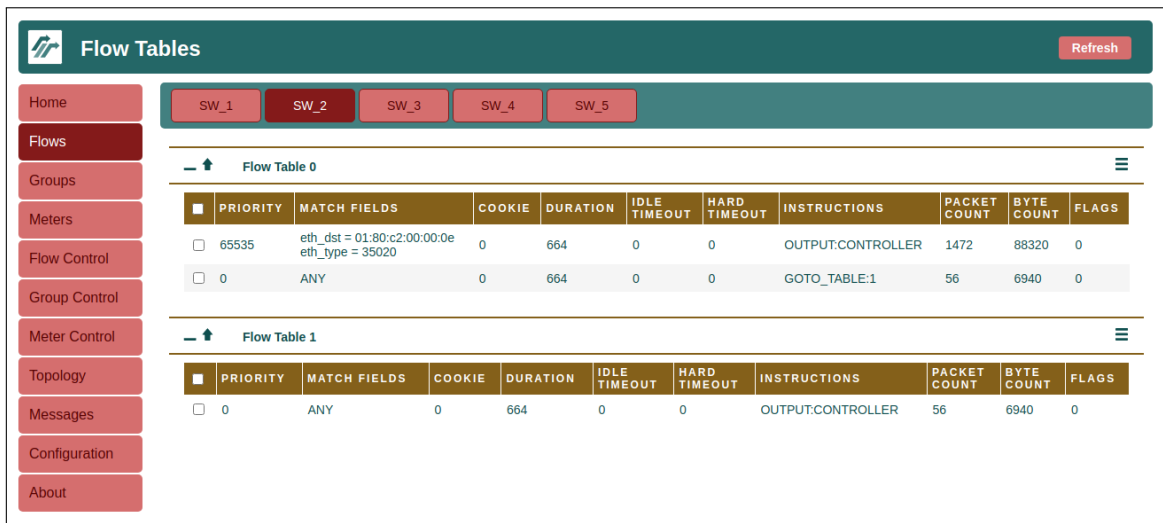


Figure 6.11: Before adding QoS in controller flow table of rb1(SW\_2) switch

Flow Tables <span>Refresh</span>																																											
<div> <div>Home</div> <div>Flows</div> <div>Groups</div> <div>Meters</div> <div>Flow Control</div> <div>Group Control</div> <div>Meter Control</div> <div>Topology</div> <div>Messages</div> <div>Configuration</div> <div>About</div> </div> <div> <div>SW_1</div> <div>SW_2</div> <div>SW_3</div> <div>SW_4</div> <div>SW_5</div> </div>																																											
<div> <div>Flow Table 0</div> <div></div> </div> <table> <tr> <th></th><th>PRIORITY</th><th>MATCH FIELDS</th><th>COOKIE</th><th>DURATION</th><th>IDLE TIMEOUT</th><th>HARD TIMEOUT</th><th>INSTRUCTIONS</th><th>PACKET COUNT</th><th>BYTE COUNT</th><th>FLAGS</th></tr> <tr> <td><input type="checkbox"/></td><td>65535</td><td>eth_dst = 01:80:c2:00:00:0e eth_type = 35020</td><td>0</td><td>664</td><td>0</td><td>0</td><td>OUTPUT:CONTROLLER</td><td>1472</td><td>88320</td><td>0</td></tr> <tr> <td><input type="checkbox"/></td><td>0</td><td>ANY</td><td>0</td><td>664</td><td>0</td><td>0</td><td>GOTO_TABLE:1</td><td>56</td><td>6940</td><td>0</td></tr> </table>												PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS	<input type="checkbox"/>	65535	eth_dst = 01:80:c2:00:00:0e eth_type = 35020	0	664	0	0	OUTPUT:CONTROLLER	1472	88320	0	<input type="checkbox"/>	0	ANY	0	664	0	0	GOTO_TABLE:1	56	6940	0
	PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS																																	
<input type="checkbox"/>	65535	eth_dst = 01:80:c2:00:00:0e eth_type = 35020	0	664	0	0	OUTPUT:CONTROLLER	1472	88320	0																																	
<input type="checkbox"/>	0	ANY	0	664	0	0	GOTO_TABLE:1	56	6940	0																																	
<div> <div>Flow Table 1</div> <div></div> </div> <table> <tr> <th></th><th>PRIORITY</th><th>MATCH FIELDS</th><th>COOKIE</th><th>DURATION</th><th>IDLE TIMEOUT</th><th>HARD TIMEOUT</th><th>INSTRUCTIONS</th><th>PACKET COUNT</th><th>BYTE COUNT</th><th>FLAGS</th></tr> <tr> <td><input type="checkbox"/></td><td>0</td><td>ANY</td><td>0</td><td>664</td><td>0</td><td>0</td><td>OUTPUT:CONTROLLER</td><td>56</td><td>6940</td><td>0</td></tr> </table>												PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS	<input type="checkbox"/>	0	ANY	0	664	0	0	OUTPUT:CONTROLLER	56	6940	0											
	PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS																																	
<input type="checkbox"/>	0	ANY	0	664	0	0	OUTPUT:CONTROLLER	56	6940	0																																	

Figure 6.12: Before adding QoS in controller flow table of rb2(SW\_3) switch

Flow Tables <span>Refresh</span>																																											
<div> <div>Home</div> <div>Flows</div> <div>Groups</div> <div>Meters</div> <div>Flow Control</div> <div>Group Control</div> <div>Meter Control</div> <div>Topology</div> <div>Messages</div> <div>Configuration</div> <div>About</div> </div> <div> <div>SW_1</div> <div>SW_2</div> <div>SW_3</div> <div>SW_4</div> <div>SW_5</div> </div>																																											
<div> <div>Flow Table 0</div> <div></div> </div> <table> <tr> <th></th><th>PRIORITY</th><th>MATCH FIELDS</th><th>COOKIE</th><th>DURATION</th><th>IDLE TIMEOUT</th><th>HARD TIMEOUT</th><th>INSTRUCTIONS</th><th>PACKET COUNT</th><th>BYTE COUNT</th><th>FLAGS</th></tr> <tr> <td><input type="checkbox"/></td><td>65535</td><td>eth_dst = 01:80:c2:00:00:0e eth_type = 35020</td><td>0</td><td>664</td><td>0</td><td>0</td><td>OUTPUT:CONTROLLER</td><td>736</td><td>44160</td><td>0</td></tr> <tr> <td><input type="checkbox"/></td><td>0</td><td>ANY</td><td>0</td><td>664</td><td>0</td><td>0</td><td>GOTO_TABLE:1</td><td>56</td><td>6940</td><td>0</td></tr> </table>												PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS	<input type="checkbox"/>	65535	eth_dst = 01:80:c2:00:00:0e eth_type = 35020	0	664	0	0	OUTPUT:CONTROLLER	736	44160	0	<input type="checkbox"/>	0	ANY	0	664	0	0	GOTO_TABLE:1	56	6940	0
	PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS																																	
<input type="checkbox"/>	65535	eth_dst = 01:80:c2:00:00:0e eth_type = 35020	0	664	0	0	OUTPUT:CONTROLLER	736	44160	0																																	
<input type="checkbox"/>	0	ANY	0	664	0	0	GOTO_TABLE:1	56	6940	0																																	
<div> <div>Flow Table 1</div> <div></div> </div> <table> <tr> <th></th><th>PRIORITY</th><th>MATCH FIELDS</th><th>COOKIE</th><th>DURATION</th><th>IDLE TIMEOUT</th><th>HARD TIMEOUT</th><th>INSTRUCTIONS</th><th>PACKET COUNT</th><th>BYTE COUNT</th><th>FLAGS</th></tr> <tr> <td><input type="checkbox"/></td><td>0</td><td>ANY</td><td>0</td><td>664</td><td>0</td><td>0</td><td>OUTPUT:CONTROLLER</td><td>56</td><td>6940</td><td>0</td></tr> </table>												PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS	<input type="checkbox"/>	0	ANY	0	664	0	0	OUTPUT:CONTROLLER	56	6940	0											
	PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS																																	
<input type="checkbox"/>	0	ANY	0	664	0	0	OUTPUT:CONTROLLER	56	6940	0																																	

Figure 6.13: Before adding QoS in controller flow table of a1(SW\_4) switch

Flow Tables <span>Refresh</span>																																																																	
<div> <div>Home</div> <div>Flows</div> <div>Groups</div> <div>Meters</div> <div>Flow Control</div> <div>Group Control</div> <div>Meter Control</div> <div>Topology</div> <div>Messages</div> <div>Configuration</div> <div>About</div> </div> <div> <div>SW_1</div> <div>SW_2</div> <div>SW_3</div> <div>SW_4</div> <div>SW_5</div> </div>																																																																	
<div> <div>Flow Table 0</div> <table> <tr> <th></th><th>PRIORITY</th><th>MATCH FIELDS</th><th>COOKIE</th><th>DURATION</th><th>IDLE TIMEOUT</th><th>HARD TIMEOUT</th><th>INSTRUCTIONS</th><th>PACKET COUNT</th><th>BYTE COUNT</th><th>FLAGS</th></tr> <tr> <td><input type="checkbox"/></td><td>65535</td><td>eth_dst = 01:80:c2:00:00:0e eth_type = 35020</td><td>0</td><td>664</td><td>0</td><td>0</td><td>OUTPUT:CONTROLLER</td><td>736</td><td>44160</td><td>0</td></tr> <tr> <td><input type="checkbox"/></td><td>0</td><td>ANY</td><td>0</td><td>664</td><td>0</td><td>0</td><td>GOTO_TABLE:1</td><td>56</td><td>6940</td><td>0</td></tr> </table> </div> <div> <div>Flow Table 1</div> <table> <tr> <th></th><th>PRIORITY</th><th>MATCH FIELDS</th><th>COOKIE</th><th>DURATION</th><th>IDLE TIMEOUT</th><th>HARD TIMEOUT</th><th>INSTRUCTIONS</th><th>PACKET COUNT</th><th>BYTE COUNT</th><th>FLAGS</th></tr> <tr> <td><input type="checkbox"/></td><td>0</td><td>ANY</td><td>0</td><td>664</td><td>0</td><td>0</td><td>OUTPUT:CONTROLLER</td><td>56</td><td>6940</td><td>0</td></tr> </table> </div>												PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS	<input type="checkbox"/>	65535	eth_dst = 01:80:c2:00:00:0e eth_type = 35020	0	664	0	0	OUTPUT:CONTROLLER	736	44160	0	<input type="checkbox"/>	0	ANY	0	664	0	0	GOTO_TABLE:1	56	6940	0		PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS	<input type="checkbox"/>	0	ANY	0	664	0	0	OUTPUT:CONTROLLER	56	6940	0
	PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS																																																							
<input type="checkbox"/>	65535	eth_dst = 01:80:c2:00:00:0e eth_type = 35020	0	664	0	0	OUTPUT:CONTROLLER	736	44160	0																																																							
<input type="checkbox"/>	0	ANY	0	664	0	0	GOTO_TABLE:1	56	6940	0																																																							
	PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS																																																							
<input type="checkbox"/>	0	ANY	0	664	0	0	OUTPUT:CONTROLLER	56	6940	0																																																							

Figure 6.14: Before adding QoS in controller flow table of a2(SW\_5) switch

For adding the Queues and QoS rules we run the c0.sh file terminal of controller c0.

```
bash c0.sh
```

It gives the result like Figure 6.15

```
"Node: c0" (root)
root@juel:/home/test/Experiment/mininet/config# bash c0.sh
[{"switch_id": "0000000000000001", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "5000000"}}, "1": {"config": {"max-rate": "15000000"}}, "2": {"config": {"max-rate": "25000000"}}}}][{"switch_id": "0000000000000002", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "5000000"}}, "1": {"config": {"max-rate": "15000000"}}, "2": {"config": {"max-rate": "25000000"}}}}][{"switch_id": "0000000000000003", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "5000000"}}, "1": {"config": {"max-rate": "15000000"}}, "2": {"config": {"max-rate": "25000000"}}}}][{"switch_id": "0000000000000004", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "5000000"}}, "1": {"config": {"max-rate": "15000000"}}, "2": {"config": {"max-rate": "25000000"}}}}][{"switch_id": "0000000000000005", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "5000000"}}, "1": {"config": {"max-rate": "15000000"}}, "2": {"config": {"max-rate": "25000000"}}}}][{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "QoS added. : qos_id=1"}]}][{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "QoS added. : qos_id=2"}]}][{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "QoS added. : qos_id=3"}]}][{"switch_id": "0000000000000002", "command_result": [{"result": "success", "details": "QoS added. : qos_id=1"}]}][{"switch_id": "0000000000000002", "command_result": [{"result": "success", "details": "QoS added. : qos_id=2"}]}][{"switch_id": "0000000000000002", "command_result": [{"result": "success", "details": "QoS added. : qos_id=3"}]}][{"switch_id": "0000000000000003", "command_result": [{"result": "success", "details": "QoS added. : qos_id=1"}]}][{"switch_id": "0000000000000003", "command_result": [{"result": "success", "details": "QoS added. : qos_id=2"}]}][{"switch_id": "0000000000000003", "command_result": [{"result": "success", "details": "QoS added. : qos_id=3"}]}][{"switch_id": "0000000000000004", "command_result": [{"result": "success", "details": "QoS added. : qos_id=1"}]}][{"switch_id": "0000000000000004", "command_result": [{"result": "success", "details": "QoS added. : qos_id=2"}]}][{"switch_id": "0000000000000004", "command_result": [{"result": "success", "details": "QoS added. : qos_id=3"}]}][{"switch_id": "0000000000000005", "command_result": [{"result": "success", "details": "QoS added. : qos_id=1"}]}][{"switch_id": "0000000000000005", "command_result": [{"result": "success", "details": "QoS added. : qos_id=2"}]}][{"switch_id": "0000000000000005", "command_result": [{"result": "success", "details": "QoS added. : qos_id=3"}]}]root@juel:/home/test/Experiment/mininet/config# █
```

Figure 6.15: Adding Queues and QoS rules in controller c0

After adding QoS if we check the flow table of all switches then it look like below Figures. We check that flow table in Flow Manger UI tool.

Flow Tables <span>Refresh</span>										
<div> <div>Home</div> <div>Flows</div> <div>Groups</div> <div>Meters</div> <div>Flow Control</div> <div>Group Control</div> <div>Meter Control</div> <div>Topology</div> <div>Messages</div> <div>Configuration</div> <div>About</div> </div> <div> <div>SW_1</div> <div>SW_2</div> <div>SW_3</div> <div>SW_4</div> <div>SW_5</div> </div>										
<div> <div>Flow Table 0</div> <div></div> </div>										
	PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS
<input type="checkbox"/>	65535	eth_dst = 01:80:c2:00:00:0e eth_type = 35020	0	1594	0	0	OUTPUT:CONTROLLER	3520	211200	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.11 ip_proto = 17 udp_dst = 5001	1	163	0	0	SET_QUEUE:0 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.13 ip_proto = 17 udp_dst = 5003	2	163	0	0	SET_QUEUE:1 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.12 ip_proto = 17 udp_dst = 5002	3	163	0	0	SET_QUEUE:2 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	0	ANY	0	1594	0	0	GOTO_TABLE:1	107060	161743566	0

Figure 6.16: After adding QoS in controller flow table of nb(SW\_1) switch

Flow Tables <span>Refresh</span>										
<div> <div>Home</div> <div>Flows</div> <div>Groups</div> <div>Meters</div> <div>Flow Control</div> <div>Group Control</div> <div>Meter Control</div> <div>Topology</div> <div>Messages</div> <div>Configuration</div> <div>About</div> </div> <div> <div>SW_1</div> <div>SW_2</div> <div>SW_3</div> <div>SW_4</div> <div>SW_5</div> </div>										
<div> <div>Flow Table 0</div> <div></div> </div>										
	PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS
<input type="checkbox"/>	65535	eth_dst = 01:80:c2:00:00:0e eth_type = 35020	0	1595	0	0	OUTPUT:CONTROLLER	3521	211260	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.11 ip_proto = 17 udp_dst = 5001	1	163	0	0	SET_QUEUE:0 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.13 ip_proto = 17 udp_dst = 5003	2	163	0	0	SET_QUEUE:1 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.12 ip_proto = 17 udp_dst = 5002	3	163	0	0	SET_QUEUE:2 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	0	ANY	0	1595	0	0	GOTO_TABLE:1	71383	107810232	0

Figure 6.17: After adding QoS in controller flow table of rb1(SW\_2) switch

Flow Tables <span>Refresh</span>										
<div> <div>Home</div> <div>Flows</div> <div>Groups</div> <div>Meters</div> <div>Flow Control</div> <div>Group Control</div> <div>Meter Control</div> <div>Topology</div> <div>Messages</div> <div>Configuration</div> <div>About</div> </div> <div> <div>SW_1</div> <div>SW_2</div> <div>SW_3</div> <div>SW_4</div> <div>SW_5</div> </div>										
<div> <div>Flow Table 0</div> <div></div> </div>										
	PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS
<input type="checkbox"/>	65535	eth_dst = 01:80:c2:00:00:0e eth_type = 35020	0	1595	0	0	OUTPUT:CONTROLLER	3522	211320	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.11 ip_proto = 17 udp_dst = 5001	1	163	0	0	SET_QUEUE:0 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.13 ip_proto = 17 udp_dst = 5003	2	163	0	0	SET_QUEUE:1 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.12 ip_proto = 17 udp_dst = 5002	3	163	0	0	SET_QUEUE:2 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	0	ANY	0	1595	0	0	GOTO_TABLE:1	35749	53941914	0

Figure 6.18: After adding QoS in controller flow table of rb2(SW\_3) switch

Flow Tables <span>Refresh</span>										
<div> <div>Home</div> <div>Flows</div> <div>Groups</div> <div>Meters</div> <div>Flow Control</div> <div>Group Control</div> <div>Meter Control</div> <div>Topology</div> <div>Messages</div> <div>Configuration</div> <div>About</div> </div> <div> <div>SW_1</div> <div>SW_2</div> <div>SW_3</div> <div>SW_4</div> <div>SW_5</div> </div>										
<div> <div>Flow Table 0</div> <div></div> </div>										
	PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS
<input type="checkbox"/>	65535	eth_dst = 01:80:c2:00:00:0e eth_type = 35020	0	1595	0	0	OUTPUT:CONTROLLER	1761	105660	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.11 ip_proto = 17 udp_dst = 5001	1	163	0	0	SET_QUEUE:0 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.13 ip_proto = 17 udp_dst = 5003	2	163	0	0	SET_QUEUE:1 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.12 ip_proto = 17 udp_dst = 5002	3	163	0	0	SET_QUEUE:2 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	0	ANY	0	1595	0	0	GOTO_TABLE:1	71383	107810232	0

Figure 6.19: After adding QoS in controller flow table of a1(SW\_4) switch

Flow Tables <span>Refresh</span>										
<div> <div>Home</div> <div>Flows</div> <div>Groups</div> <div>Meters</div> <div>Flow Control</div> <div>Group Control</div> <div>Meter Control</div> <div>Topology</div> <div>Messages</div> <div>Configuration</div> <div>About</div> </div> <div> <div>SW_1</div> <div>SW_2</div> <div>SW_3</div> <div>SW_4</div> <div>SW_5</div> </div>										
<div> <div>Flow Table 0</div> <div></div> </div>										
	PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS
<input type="checkbox"/>	65535	eth_dst = 01:80:c2:00:00:0e eth_type = 35020	0	1595	0	0	OUTPUT:CONTROLLER	1760	105600	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.11 ip_proto = 17 udp_dst = 5001	1	163	0	0	SET_QUEUE:0 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.13 ip_proto = 17 udp_dst = 5003	2	163	0	0	SET_QUEUE:1 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	1	eth_type = 2048 ipv4_dst = 10.0.0.12 ip_proto = 17 udp_dst = 5002	3	163	0	0	SET_QUEUE:2 GOTO_TABLE:1	0	0	0
<input type="checkbox"/>	0	ANY	0	1595	0	0	GOTO_TABLE:1	35749	53941914	0

Figure 6.20: After adding QoS in controller flow table of a2(SW\_5) switch

### 6.8.5 Setting up the Servers

For setting up the servers we run the ue1.sh, ue2.sh, ue3.sh files or below command on appropriate server terminal.

```
iperf -s -u -i 1 -p <port-number>
```

### 6.8.6 Setting up the Clients

For setting up the clients we run the b1.sh, b2.sh, c.sh files or below command on appropriate client terminal.

```
iperf -c <server-ip> -p <server-port> -u -b <bandwidth-speed>
```

## 6.9 Experiment Results

### 6.9.1 Results of Before Setting up the Environment of Slicing

First in host b we run the client with 10Mbps bandwidth and it gives the approximately 10Mbps bandwidth speed. Result Shown in Figure 6.21



The image shows two terminal windows. The top window, titled "Node: ue1", displays the output of a server script. It shows the server listening on UDP port 5001 and receiving 1470 byte datagrams. A table of results shows 10 intervals of 1.0 second each, with a total transfer of 12.5 MBytes and a bandwidth of 10.5 Mbits/sec. The bottom window, titled "Node: b", displays the output of a client script. It shows the client connecting to 10.0.0.11, UDP port 5001, and sending 1470 byte datagrams. A table of results shows 10 intervals of 1.0 second each, with a total transfer of 12.5 MBytes and a bandwidth of 10.5 Mbits/sec. The client also reports that it sent 8917 datagrams.

```

"Node: ue1"
root@juel:/home/test/Experiment/mininet/config# bash ue1.sh
-----
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.11 port 5001 connected with 10.0.0.1 port 60411
[ ID] Interval      Transfer    Bandwidth    Jitter    Lost/Total Datagrams
[ 5] 0.0- 1.0 sec  1.25 MBytes 10.5 Mbits/sec 0.012 ms   0/ 892 (0%)
[ 5] 1.0- 2.0 sec  1.25 MBytes 10.5 Mbits/sec 0.014 ms   0/ 892 (0%)
[ 5] 2.0- 3.0 sec  1.25 MBytes 10.5 Mbits/sec 0.006 ms   0/ 892 (0%)
[ 5] 3.0- 4.0 sec  1.25 MBytes 10.5 Mbits/sec 0.011 ms   0/ 892 (0%)
[ 5] 4.0- 5.0 sec  1.25 MBytes 10.5 Mbits/sec 0.013 ms   0/ 891 (0%)
[ 5] 5.0- 6.0 sec  1.25 MBytes 10.5 Mbits/sec 0.015 ms   0/ 892 (0%)
[ 5] 6.0- 7.0 sec  1.25 MBytes 10.5 Mbits/sec 0.012 ms   0/ 891 (0%)
[ 5] 7.0- 8.0 sec  1.25 MBytes 10.5 Mbits/sec 0.009 ms   0/ 892 (0%)
[ 5] 8.0- 9.0 sec  1.25 MBytes 10.5 Mbits/sec 0.013 ms   0/ 892 (0%)
[ 5] 0.0-10.0 sec 12.5 MBytes 10.5 Mbits/sec 0.012 ms   0/ 8917 (0%)
[

"Node: b"
root@juel:/home/test/Experiment/mininet/config# bash b1.sh
-----
Client connecting to 10.0.0.11, UDP port 5001
Sending 1470 byte datagrams, IPG target: 1121.52 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.1 port 60411 connected with 10.0.0.11 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 5] 0.0-10.0 sec 12.5 MBytes 10.5 Mbits/sec
[ 5] Sent 8917 datagrams
[ 5] Server Report:
[ 5] 0.0-10.0 sec 12.5 MBytes 10.5 Mbits/sec 0.012 ms   0/ 8917 (0%)
root@juel:/home/test/Experiment/mininet/config#

```

Figure 6.21: Before adding the Queues and QoS result of communication between ue1(server) and b(client)

Second in host b we run the client with 20Mbps bandwidth and it gives the approximately 20Mbps bandwidth speed. Result Shown in Figure 6.22

```

"Node: ue3"
root@juel:/home/test/Experiment/mininet/config# bash ue3.sh
-----
Server listening on UDP port 5003
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.13 port 5003 connected with 10.0.0.1 port 38419
[ ID] Interval      Transfer    Bandwidth    Jitter    Lost/Total Datagrams
[ 5] 0.0- 1.0 sec  2.50 MBytes 21.0 Mbits/sec 0.004 ms  0/ 1786 (0%)
[ 5] 1.0- 2.0 sec  2.50 MBytes 21.0 Mbits/sec 0.003 ms  0/ 1783 (0%)
[ 5] 2.0- 3.0 sec  2.50 MBytes 21.0 Mbits/sec 0.002 ms  0/ 1784 (0%)
[ 5] 3.0- 4.0 sec  2.50 MBytes 21.0 Mbits/sec 0.000 ms  0/ 1783 (0%)
[ 5] 4.0- 5.0 sec  2.50 MBytes 21.0 Mbits/sec 0.001 ms  0/ 1783 (0%)
[ 5] 5.0- 6.0 sec  2.50 MBytes 21.0 Mbits/sec 0.001 ms  0/ 1784 (0%)
[ 5] 6.0- 7.0 sec  2.50 MBytes 21.0 Mbits/sec 0.001 ms  0/ 1783 (0%)
[ 5] 7.0- 8.0 sec  2.50 MBytes 21.0 Mbits/sec 0.001 ms  0/ 1783 (0%)
[ 5] 8.0- 9.0 sec  2.50 MBytes 21.0 Mbits/sec 0.001 ms  0/ 1784 (0%)
[ 5] 0.0-10.0 sec 25.0 MBytes 21.0 Mbits/sec 0.001 ms  0/17833 (0%)

"Node: b"
root@juel:/home/test/Experiment/mininet/config# bash b2.sh
-----
Client connecting to 10.0.0.13, UDP port 5003
Sending 1470 byte datagrams, IPG target: 560.76 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.1 port 38419 connected with 10.0.0.13 port 5003
[ ID] Interval      Transfer    Bandwidth
[ 5] 0.0-10.0 sec 25.0 MBytes 21.0 Mbits/sec
[ 5] Sent 17833 datagrams
[ 5] Server Report:
[ 5] 0.0-10.0 sec 25.0 MBytes 21.0 Mbits/sec 0.000 ms  0/17833 (0%)
root@juel:/home/test/Experiment/mininet/config#

```

Figure 6.22: Before adding the Queues and QoS result of communication between ue3(server) and b(client)

Third in host c we run the client with 30Mbps bandwidth and it gives the approximately 30Mbps bandwidth speed. Result Shown in Figure 6.23

The image shows two terminal windows. The top window, titled "Node: ue2", displays the output of a server script. It shows the server listening on UDP port 5002, receiving 1470 byte datagrams, and having a UDP buffer size of 208 KByte. A table of performance metrics follows, showing a consistent bandwidth of 31.5 Mbits/sec and a transfer of 37.5 MBytes over a 10-second interval. The bottom window, titled "Node: c", shows the output of a client script. It displays the client connecting to 10.0.0.12 on UDP port 5002, sending 1470 byte datagrams with an IPG target of 373.84 us, and having a UDP buffer size of 208 KByte. It also shows a table of performance metrics, including a bandwidth of 31.5 Mbits/sec and a transfer of 37.5 MBytes over a 10-second interval.

```

"Node: ue2"
root@juel:/home/test/Experiment/mininet/config# bash ue2.sh
-----
Server listening on UDP port 5002
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.12 port 5002 connected with 10.0.0.2 port 45446
[ ID] Interval      Transfer    Bandwidth    Jitter    Lost/Total  Datagrams
[ 5] 0.0- 1.0 sec   3.75 MBytes 31.5 Mbits/sec 0.001 ms   0/ 2679 (0%)
[ 5] 1.0- 2.0 sec   3.75 MBytes 31.5 Mbits/sec 0.001 ms   0/ 2675 (0%)
[ 5] 2.0- 3.0 sec   3.75 MBytes 31.5 Mbits/sec 0.004 ms   0/ 2675 (0%)
[ 5] 3.0- 4.0 sec   3.75 MBytes 31.5 Mbits/sec 0.001 ms   0/ 2675 (0%)
[ 5] 4.0- 5.0 sec   3.75 MBytes 31.5 Mbits/sec 0.000 ms   0/ 2675 (0%)
[ 5] 5.0- 6.0 sec   3.75 MBytes 31.5 Mbits/sec 0.001 ms   0/ 2675 (0%)
[ 5] 6.0- 7.0 sec   3.75 MBytes 31.5 Mbits/sec 0.001 ms   0/ 2675 (0%)
[ 5] 7.0- 8.0 sec   3.75 MBytes 31.5 Mbits/sec 0.001 ms   0/ 2675 (0%)
[ 5] 8.0- 9.0 sec   3.75 MBytes 31.5 Mbits/sec 0.000 ms   0/ 2675 (0%)
[ 5] 0.0-10.0 sec  37.5 MBytes 31.5 Mbits/sec 0.003 ms   0/26750 (0%)
^

"Node: c"
root@juel:/home/test/Experiment/mininet/config# bash c.sh
-----
Client connecting to 10.0.0.12, UDP port 5002
Sending 1470 byte datagrams, IPG target: 373.84 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.2 port 45446 connected with 10.0.0.12 port 5002
[ ID] Interval      Transfer    Bandwidth
[ 5] 0.0-10.0 sec  37.5 MBytes 31.5 Mbits/sec
[ 5] Sent 26750 datagrams
[ 5] Server Report:
[ 5] 0.0-10.0 sec  37.5 MBytes 31.5 Mbits/sec 0.002 ms   0/26750 (0%)
root@juel:/home/test/Experiment/mininet/config#

```

Figure 6.23: Before adding the Queues and QoS result of communication between ue2(server) and c(client)

## 6.9.2 Final Results of Experiments

First in host b we run the client with 10Mbps bandwidth and it gives the approximately 5Mbps bandwidth speed due to we set that slice/flow with 5Mbps max-rate of bandwidth. Result Shown

in Figure 6.24

The figure shows two terminal windows. The top window, titled "Node: ue1", shows a server listening on UDP port 5001 and receiving 1470 byte datagrams. It then shows a table of communication results for 10 intervals, with a total transfer of 5.90 MBytes and a bandwidth of 4.86 Mbits/sec. The bottom window, titled "Node: b", shows a client connecting to 10.0.0.11 on UDP port 5001 and sending 1470 byte datagrams. It then shows a table of communication results for 10 intervals, with a total transfer of 5.90 MBytes and a bandwidth of 4.91 Mbits/sec.

```
root@juel:/home/test/Experiment/mininet/config# bash ue1.sh
-----
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.11 port 5001 connected with 10.0.0.1 port 55206
[ ID] Interval      Transfer    Bandwidth    Jitter    Lost/Total  Datagrams
[ 5] 0.0- 1.0 sec   596 KBytes  4.88 Mbits/sec  7.887 ms    0/ 415 (0%)
[ 5] 1.0- 2.0 sec   591 KBytes  4.85 Mbits/sec  2.946 ms    0/ 412 (0%)
[ 5] 2.0- 3.0 sec   593 KBytes  4.86 Mbits/sec  3.450 ms    0/ 413 (0%)
[ 5] 3.0- 4.0 sec   593 KBytes  4.86 Mbits/sec  4.372 ms    0/ 413 (0%)
[ 5] 4.0- 5.0 sec   594 KBytes  4.87 Mbits/sec  5.932 ms    0/ 414 (0%)
[ 5] 5.0- 6.0 sec   593 KBytes  4.86 Mbits/sec  9.121 ms    0/ 413 (0%)
[ 5] 6.0- 7.0 sec   593 KBytes  4.86 Mbits/sec  3.041 ms    0/ 413 (0%)
[ 5] 7.0- 8.0 sec   591 KBytes  4.85 Mbits/sec  3.712 ms    0/ 412 (0%)
[ 5] 8.0- 9.0 sec   593 KBytes  4.86 Mbits/sec  4.792 ms    0/ 413 (0%)
[ 5] 9.0-10.0 sec   591 KBytes  4.85 Mbits/sec  5.495 ms    0/ 412 (0%)
[ 5] 0.0-10.2 sec  5.90 MBytes  4.86 Mbits/sec  9.560 ms    0/ 4211 (0%)
[ ]

root@juel:/home/test/Experiment/mininet/config# bash b1.sh
-----
Client connecting to 10.0.0.11, UDP port 5001
Sending 1470 byte datagrams, IPG target: 1121.52 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.1 port 55206 connected with 10.0.0.11 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 5] 0.0-10.1 sec  5.90 MBytes  4.91 Mbits/sec
[ 5] Sent 4211 datagrams
[ 5] Server Report:
[ 5] 0.0-10.2 sec  5.90 MBytes  4.86 Mbits/sec  9.560 ms    0/ 4211 (0%)
root@juel:/home/test/Experiment/mininet/config# [ ]
```

Figure 6.24: After adding the Queues and QoS result of communication between ue1(server) and b(client)

Second in host b we run the client with 20Mbps bandwidth and it gives the approximately 15Mbps bandwidth speed due to we set that slice/flow with 15Mbps max-rate of bandwidth. Result

Shown in Figure 6.25

```

"Node: ue3"
root@juel:/home/test/Experiment/mininet/config# bash ue3.sh
-----
Server listening on UDP port 5003
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.13 port 5003 connected with 10.0.0.1 port 49151
[ ID] Interval      Transfer    Bandwidth    Jitter    Lost/Total Datagrams
[ 5] 0.0- 1.0 sec  1.71 MBytes 14.4 Mbits/sec 0.993 ms   0/ 1222 (0%)
[ 5] 1.0- 2.0 sec  1.73 MBytes 14.5 Mbits/sec 2.293 ms   0/ 1237 (0%)
[ 5] 2.0- 3.0 sec  1.73 MBytes 14.5 Mbits/sec 1.379 ms   0/ 1237 (0%)
[ 5] 3.0- 4.0 sec  1.74 MBytes 14.6 Mbits/sec 1.009 ms   0/ 1238 (0%)
[ 5] 4.0- 5.0 sec  1.73 MBytes 14.5 Mbits/sec 2.502 ms   0/ 1237 (0%)
[ 5] 5.0- 6.0 sec  1.73 MBytes 14.5 Mbits/sec 1.502 ms   0/ 1236 (0%)
[ 5] 6.0- 7.0 sec  1.74 MBytes 14.6 Mbits/sec 1.048 ms   0/ 1238 (0%)
[ 5] 7.0- 8.0 sec  1.73 MBytes 14.5 Mbits/sec 2.915 ms   0/ 1236 (0%)
[ 5] 8.0- 9.0 sec  1.74 MBytes 14.6 Mbits/sec 1.537 ms   0/ 1238 (0%)
[ 5] 9.0-10.0 sec 1.73 MBytes 14.5 Mbits/sec 1.084 ms   0/ 1237 (0%)
[ 5] 0.0-10.0 sec 17.4 MBytes 14.5 Mbits/sec 3.177 ms   0/12416 (0%)
[ ]

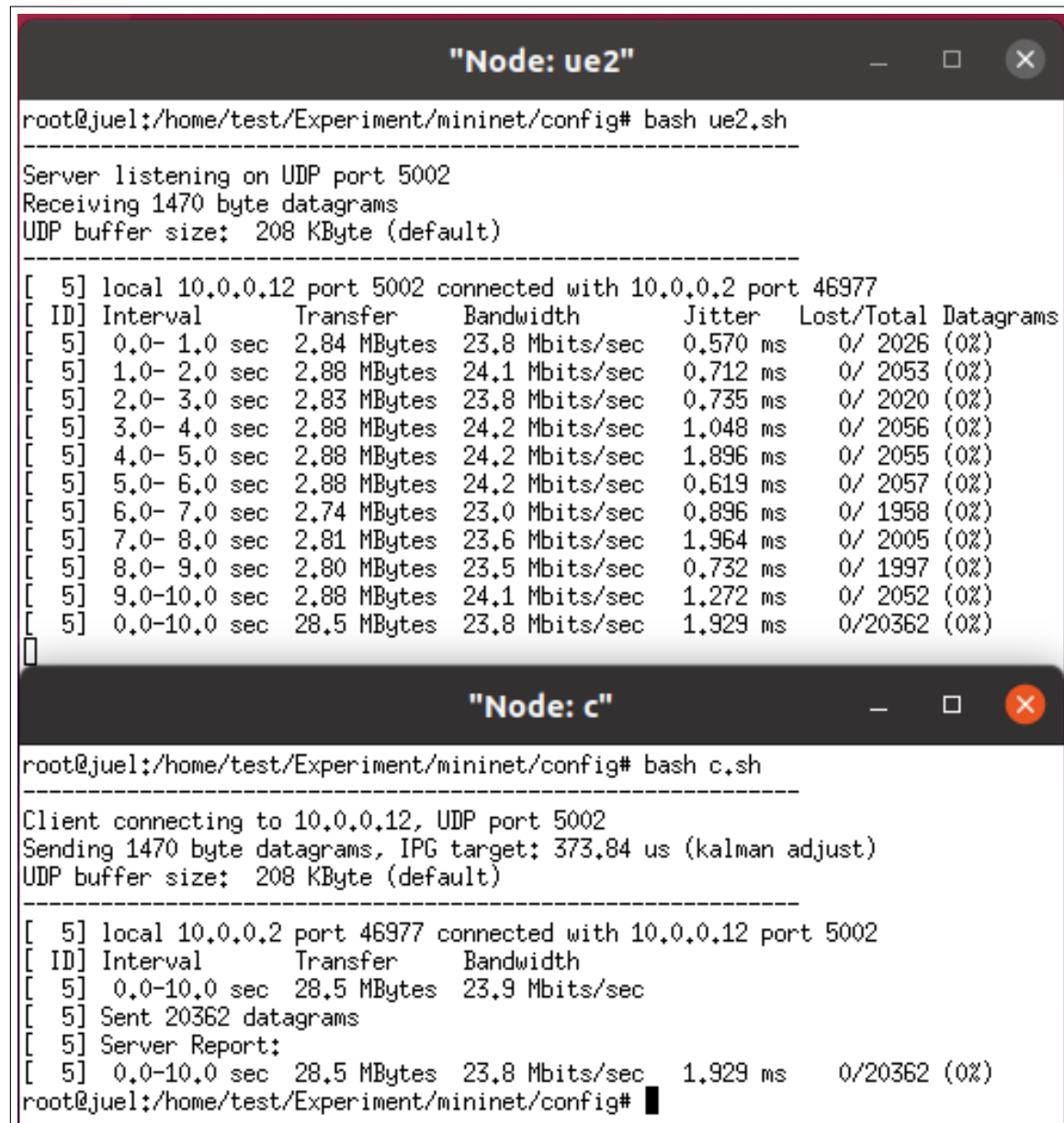
"Node: b"
root@juel:/home/test/Experiment/mininet/config# bash b2.sh
-----
Client connecting to 10.0.0.13, UDP port 5003
Sending 1470 byte datagrams, IPG target: 560.76 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.1 port 49151 connected with 10.0.0.13 port 5003
[ ID] Interval      Transfer    Bandwidth
[ 5] 0.0-10.0 sec  17.4 MBytes 14.6 Mbits/sec
[ 5] Sent 12416 datagrams
[ 5] Server Report:
[ 5] 0.0-10.0 sec  17.4 MBytes 14.5 Mbits/sec 3.177 ms   0/12416 (0%)
root@juel:/home/test/Experiment/mininet/config# [ ]

```

Figure 6.25: After adding the Queues and QoS result of communication between ue3(server) and b(client)

Third in host c we run the client with 30Mbps bandwidth and it gives the approximately 25Mbps bandwidth speed due to we set that slice/flow with 25Mbps max-rate of bandwidth. Result Shown

in Figure 6.26



The figure displays two terminal windows. The top window, titled '"Node: ue2"', shows the server's perspective. It starts with the command `bash ue2.sh`, followed by status messages: 'Server listening on UDP port 5002', 'Receiving 1470 byte datagrams', and 'UDP buffer size: 208 KByte (default)'. A table of performance metrics follows, showing data received over 10-second intervals. The bottom window, titled '"Node: c"', shows the client's perspective. It starts with `bash c.sh`, followed by status messages: 'Client connecting to 10.0.0.12, UDP port 5002', 'Sending 1470 byte datagrams, IPG target: 373.84 us (kalman adjust)', and 'UDP buffer size: 208 KByte (default)'. It then shows a summary of data sent and a 'Server Report' table that mirrors the server's data.

```
root@juel:/home/test/Experiment/mininet/config# bash ue2.sh
-----
Server listening on UDP port 5002
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.12 port 5002 connected with 10.0.0.2 port 46977
[ ID] Interval      Transfer    Bandwidth    Jitter    Lost/Total Datagrams
[ 5] 0.0- 1.0 sec    2.84 MBytes 23.8 Mbits/sec 0.570 ms    0/ 2026 (0%)
[ 5] 1.0- 2.0 sec    2.88 MBytes 24.1 Mbits/sec 0.712 ms    0/ 2053 (0%)
[ 5] 2.0- 3.0 sec    2.83 MBytes 23.8 Mbits/sec 0.735 ms    0/ 2020 (0%)
[ 5] 3.0- 4.0 sec    2.88 MBytes 24.2 Mbits/sec 1.048 ms    0/ 2056 (0%)
[ 5] 4.0- 5.0 sec    2.88 MBytes 24.2 Mbits/sec 1.896 ms    0/ 2055 (0%)
[ 5] 5.0- 6.0 sec    2.88 MBytes 24.2 Mbits/sec 0.619 ms    0/ 2057 (0%)
[ 5] 6.0- 7.0 sec    2.74 MBytes 23.0 Mbits/sec 0.896 ms    0/ 1958 (0%)
[ 5] 7.0- 8.0 sec    2.81 MBytes 23.6 Mbits/sec 1.964 ms    0/ 2005 (0%)
[ 5] 8.0- 9.0 sec    2.80 MBytes 23.5 Mbits/sec 0.732 ms    0/ 1997 (0%)
[ 5] 9.0-10.0 sec    2.88 MBytes 24.1 Mbits/sec 1.272 ms    0/ 2052 (0%)
[ 5] 0.0-10.0 sec    28.5 MBytes 23.8 Mbits/sec 1.929 ms    0/20362 (0%)
[ ]

root@juel:/home/test/Experiment/mininet/config# bash c.sh
-----
Client connecting to 10.0.0.12, UDP port 5002
Sending 1470 byte datagrams, IPG target: 373.84 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.2 port 46977 connected with 10.0.0.12 port 5002
[ ID] Interval      Transfer    Bandwidth
[ 5] 0.0-10.0 sec    28.5 MBytes 23.9 Mbits/sec
[ 5] Sent 20362 datagrams
[ 5] Server Report:
[ 5] 0.0-10.0 sec    28.5 MBytes 23.8 Mbits/sec 1.929 ms    0/20362 (0%)
root@juel:/home/test/Experiment/mininet/config#
```

Figure 6.26: After adding the Queues and QoS result of communication between ue2(server) and c(client)

## Chapter 7

# Conclusion and Future Work

### 7.1 Conclusion

In conclusion, this thesis has made some contributions towards the implementation of network slicing and quality of service (QoS) provisioning in 5G networks. Through the utilization of Mininet as a network topology platform, Ryu controller for network function management, and OpenFlow switches for slicing, the thesis successfully demonstrated the feasibility and practicality of network slicing in a virtualized environment. By leveraging software-defined networking (SDN) and network function virtualization (NFV) technologies, the thesis achieved efficient resource allocation and management, enabling the creation and management of multiple network slices tailored to different service requirements.

The experimental results presented in this thesis showcased the effectiveness of the implemented architecture in guaranteeing QoS for each network slice. By applying OpenFlow Queue commands, the thesis effectively scheduled and prioritized traffic flows, ensuring that each slice received the appropriate bandwidth allocation and QoS parameters. This capability is crucial in supporting diverse services and applications within the 5G ecosystem, where different slices may have varying QoS requirements.

The findings of this research hold some implications for the advancement of 5G networks. The successful implementation of network slicing and QoS provisioning contributes to the body of knowledge surrounding the practical realization of 5G network architectures. Moreover, the research highlights the importance of SDN and NFV in enabling flexible and efficient network slicing, opening up new opportunities for service providers to deliver customized services and improve the overall user experience.

### 7.2 Future Work

In terms of future work, there are several areas that can be explored to improve the implementation of network slicing and quality of service (QoS) in 5G networks. Firstly, scalability and resource optimization need to be addressed, ensuring efficient allocation of resources while accommodating diverse QoS requirements. This involves developing algorithms and strategies for dynamic resource allocation and load balancing. Secondly, dynamic slice management is crucial, allowing network

slices to adapt to changing conditions and user demands. Real-time monitoring and adjustment mechanisms can be developed for optimal slice performance. Thirdly, security and privacy measures must be enhanced to protect network slices from unauthorized access and attacks. Robust security mechanisms, encryption techniques, and access control mechanisms are needed. Fourthly, the integration of network slicing with edge computing can boost network performance. Research can focus on optimizing resource management, slice orchestration, and data routing at the network edge. Lastly, standardization and interoperability efforts should be strengthened for widespread adoption. Developing standardized interfaces, protocols, and APIs will ensure seamless interoperability between different network slicing implementations. By addressing these areas, researchers can advance network slicing, improve 5G networks, and enable innovative applications and services.



# Bibliography

- [1] 5g system overview. <https://www.3gpp.org/technologies/5g-system-overview>. Accessed: 2023-05-20.
- [2] Mininet. <http://mininet.org/>. Accessed: 2023-05-25.
- [3] Mininet: How does it work. <https://github.com/mininet/mininet#how-does-it-work>. Accessed: 2023-05-25.
- [4] Mininet python api: Reference manual. <http://mininet.org/api/hierarchy.html>. Accessed: 2023-05-25.
- [5] Open vswitch documentation. <https://docs.openvswitch.org/en/latest/>. Accessed: 2023-05-20.
- [6] Openflow. <https://www.opennetworking.org/sdn-resources/openflow>. Accessed: 2023-05-20.
- [7] Qos in ryu controller documentation. [https://osrg.github.io/ryu-book/en/html/rest\\_qos.html](https://osrg.github.io/ryu-book/en/html/rest_qos.html). Accessed: 2023-05-28.
- [8] Ryu controller documentation. <https://ryu.readthedocs.io/en/latest/>. Accessed: 2023-05-20.
- [9] A. R. Alkhafaji and F. S. Al-Turaihi. Multi-layer network slicing and resource allocation scheme for traffic-aware qos ensured sdn/nfv-5g network. In *2021 1st Babylon International Conference on Information Technology and Science (BICITS)*, pages 327–331, 2021.
- [10] A. R. Alkhafaji and F. S. S. Al-Turaihi. Traffic-aware qos guaranteed sdn/nfv-5g network with multi-layer network slicing and resource allocation. In *2022 8th International Conference on Contemporary Information Technology and Mathematics (ICCITM)*, pages 273–277, 2022.
- [11] A. A. Barakabitze, A. Ahmad, R. Mijumbi, and A. Hines. 5g network slicing using sdn and nvf: A survey of taxonomy, architectures and future challenges. *Computer Networks*, 167:106984, 2020.
- [12] S. Chen, C.-N. Lee, and M.-F. Lee. Realization of 5g network slicing using open source softwares. In *Proceedings of the 2020 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pages 1549–1556, Taiwan, 2020. Asia-Pacific Signal and Information Processing Association (APSIPA).

- [13] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.
- [14] X. Li, M. Samaka, H. A. Chan, D. Bhamare, L. Gupta, C. Guo, and R. Jain. Network slicing for 5g: Challenges and opportunities. *IEEE Internet Computing*, 21(5):20–27, 2017.
- [15] R. F. Olimid and G. Nencioni. 5g network slicing: A security overview. *IEEE Access*, 8:99999–100009, 2020.
- [16] C. Tipantuña and P. Yanchapaxi. Network functions virtualization: An overview and open-source projects. In *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6, 2017.