

國立臺灣大學電機資訊學院電機工程學研究所

碩士論文

Graduate Institute of Electrical Engineering  
College of Electrical Engineering and Computer Science  
National Taiwan University  
Master Thesis



設計與實作基於軟體定義網路與服務品質考量之網路

位址轉換

Design and Implement an SDN-based QoS Aware  
Network Address Translation

鄭淙勻

Tsung-Yun Cheng

指導教授：廖婉君 博士

Advisor: Wanjiun Liao, Ph.D.

中華民國 106 年 7 月

July, 2017

國立臺灣大學碩士學位論文  
口試委員會審定書

設計與實作基於軟體定義網路與服務品質考量之網路  
位址轉換

Design and Implement An SDN-based QoS Aware  
Network Address Translation

本論文係鄭淙勻君（學號 R00921031）在國立臺灣大學電機工程  
學系完成之碩士學位論文，於民國 106 年 7 月 14 日承下列考試委員  
審查通過及口試及格，特此證明。

口試委員：

廖婉元

（簽名）

（指導教授）

郭耀輝

林宗男

陳俊良

周承亨

系主任

劉志文

（簽名）

# 誌謝



首先，我要感謝我的指導老師廖婉君教授，她在我人生最迷惘的時候指示我一條明路，並在我一年半的研究生涯之中不斷鼓勵與教導我，才能讓我完成碩士的學業。我也要感謝 IRL 實驗室的各位學長與學姊，感謝強大的安笛學長總是在研究上盡力的幫助我，甚至在口試之前幾乎是陪我熬了一個禮拜的夜來準備 DEMO。謝謝孫哥、江江學長都適時的給我鼓勵，也感謝江萱學姐從以前到現在的大力支持與幫忙，在此祝福各位學長姐能夠順利的畢業。另外也感謝實驗室的同學們一路以來的相伴，還有口試時來義務幫忙的學弟們，以及實驗室助理筑涵在各項大小瑣事上的幫忙與照料，辛苦你們了。

最後，我要感謝我的父母與我的妹妹，感謝你們在我學業遇到瓶頸時，不但沒有給我壓力，反倒給我更多的鼓勵。感謝我身邊最重要的人，是因為你的陪伴我才能走到今天。

我在碩士生涯的路上，走的比其他人來的久，往後的路必須走的更快，才能追上大家的腳步，也才能不負身邊眾人的期盼。謹以此論文獻給所有我在乎的人，謝謝你們！

# 摘要



網路位址轉換 (NAT) 是當今網路架構中最常見的中間元件之一。以網際網路服務供應商為例，他們通常會在其內部網路中採用 NAT444 架構來減輕 IPv4 網路位址耗盡問題。但是，由於網路位址轉換會將封包表頭資訊偽裝，並將由使用者端裝置發送的所有資料流混雜在一起，因此將導致網路營運者很難辨識出每條資料流，也很難對其網路有完整且全面的控制能力。不僅如此，此種技術也會產生許多其他問題，例如規模性、可靠性與打破網路終端對終端原則等等。

因此，我們利用軟體定義式網路 (SDN) 技術來實作網路位址轉換，以達到更細微的管控。藉由將傳統的網路位址轉換硬體替換為支援 OpenFlow 協議的交換器並統一接受中央控制器的控管，網路管理者將擁有網路的全局資訊，能做到以資料流為基礎的網路行為管理。在本論文中，我們設計且實作出以 SDN 為基礎的網路位址轉換，並將其實作在樹梅派 (Raspberry Pi) 上作為一個初始模型。此外，我們進一步利用 OpenFlow 協議支援的隊列模組，在以 SDN 為基礎的 NAT 架構網路下，做到以資料流為基礎的服務品質加強。我們以效用函數與傳輸速率來估測每筆資料流的使用者體驗，並將其規劃成一組最佳化問題，於考量公平性因素的同時計算出得到最佳使用者體驗的頻寬分配方式。實驗結果顯示我們的實作能夠以 SDN 實現 NAT 的功能，並做到以資料流為基礎的設定，進而達成服務品質加強。

關鍵字：網路位址轉換、NAT444, 軟體定義式網路、服務品質、樹梅派(Raspberry Pi)

# Abstract



Network address translation (NAT) is one of the most commonly used middle-boxes in the network architectures nowadays. Take the Internet Service Providers (ISP) as an instance, they usually adopt the NAT444 architecture in their internal network to mitigate the IPv4 exhaustion problem. However, since NAT middle-boxes will masquerade the packet header information and mix-up all network traffic flows originating from user devices, the network operators could hardly identify the origin of the data flows or have an overall and complete control of their internal network. Moreover, such a technology also raises a variety of issues such as scalability, reliability, and breaking the end-to-end principle.

Therefore, we utilize Software Defined Network (SDN) to implement the NAT function to achieve a more fine-grained control. By replacing the traditional NAT hardware with the OpenFlow switches and making them centrally controlled by the SDN controller, the network operators could have a global network view to manage the network behavior in a flow-based manner. In our work, we design and implement the SDN-based NAT architecture on a low-cost Raspberry Pi platform as a prototype. In addition, we exploit the queue module supported in the OpenFlow protocol to implement a flow-level QoS (Quality of Service) enforcement scheme in the SDN-based NAT. We use utility functions to measure the quality of experience of data flows with respect to the received data rate, and model the bandwidth allocation problem as an optimization problem to derive a solution with optimal utility scores while considering the fairness criterion. Experiment results show that our implementation could achieve the function of NAT and

could do flow-level configuration to perform the QoS enforcement.



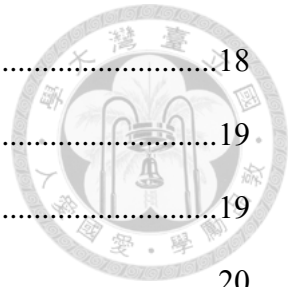
Keywords — Network address translation, NAT444, Software defined network, Quality of Service, Raspberry Pi

# Contents



誌謝	i
摘要	ii
Abstract	iii
Contents	v
List of Figures	vii
List of Tables	ix
<b>Chapter 1</b>	<b>Introduction.....1</b>
1.1	Network Address Translation.....1
1.2	NAT 444 Architecture .....2
1.3	Problems with Today’s NAT .....4
1.4	SDN-based QoS Aware NAT .....5
1.5	Thesis Organization .....6
<b>Chapter 2</b>	<b>Related Work.....7</b>
<b>Chapter 3</b>	<b>System Architecture.....9</b>
3.1	SDN Controller .....10
3.2	Southbound API .....11
3.2.1	OpenFlow Protocol .....11
3.2.2	OVSDB Protocol.....11
3.3	Raspberry Pi as an OpenFlow Switch.....12
3.4	SDN-based NAT Function .....13
<b>Chapter 4</b>	<b>QoS Enabled NAT.....18</b>

4.1	Design of QoS Enabled NAT .....	18
4.1.1	Flow Classification.....	19
4.1.2	Rate Controller .....	19
4.2	Bandwidth Allocation Problem.....	20
4.2.1	Basic Model .....	20
4.2.2	Utility Function .....	23
4.2.3	Utility Proportional Fairness .....	25
4.3	Heuristic Algorithm .....	26
4.3.1	Problem Analysis .....	27
4.3.2	Greedy Approach Algorithm .....	29
<b>Chapter 5</b>	<b>Evaluation.....</b>	<b>31</b>
5.1	Experiment Setup.....	31
5.2	Experiment Results .....	32
5.2.1	NAT Function.....	33
5.2.2	Synthetic Data .....	34
<b>Chapter 6</b>	<b>Conclusion and Future Work .....</b>	<b>40</b>
6.1	Conclusion .....	40
6.2	Future Work .....	40
<b>Reference</b>	<b>43</b>	





# List of Figures



Figure 1-1 NAT gateway router.....	1
Figure 1-2 NAT function example.....	2
Figure 1-3 NAT 444 architecture.....	3
Figure 3-1 SDN-based NAT444 architecture .....	9
Figure 3-2 Ryu controller architecture .....	10
Figure 3-3 OpenFlow switch configurations.....	13
Figure 3-4 Raspberry Pi.....	13
Figure 3-5 SDN-based NAT flow chart.....	14
Figure 3-6 SDN-based NAT system architecture .....	15
Figure 3-7 NAT mapping table.....	15
Figure 4-1 QoS Aware NAT architecture.....	18
Figure 4-2 Two-layer network architecture .....	20
Figure 4-3 Queues implementation in a switch.....	22
Figure 5-1 Experiment environment.....	31
Figure 5-2 Utility functions.....	32
Figure 5-3 Iperf terminal in the server.....	33
Figure 5-4 Iperf terminal in the client .....	33
Figure 5-5 The NAT mapping table in the controller .....	33
Figure 5-6 Synthetic data.....	34
Figure 5-7 Bandwidth configuration result of synthetic data.....	35
Figure 5-8 Comparison of transmission rate and assigned available bandwidth for	

different application types .....	35
Figure 5-9 Utility scores comparison of synthetic data I.....	35
Figure 5-10 Utility drops due to flow ending.....	38
Figure 5-11 Utility scores comparison of synthetic data II .....	39



# List of Tables



Table 3-1 Southbound API .....	12
Table 4-1 Notation table .....	27
Table 5-1 Parameters setting in synthetic data II.....	39



# Chapter 1

## Introduction

### 1.1 Network Address Translation

Network Address Translation (NAT), which shares an IP address among multiple subscribers, is one of the most widely deployed network functions in today's network. The basic mechanism of NAT is to map various IP/port combinations of flows from subscribers to a set of specially selected IP/port combinations. The NAT function is usually configured on the network edge router as a gateway allowing multiple devices in the inner private network to communicate with the Internet, as shown in Figure 1-1. Thus, these subscribers look like a single user from the Internet.

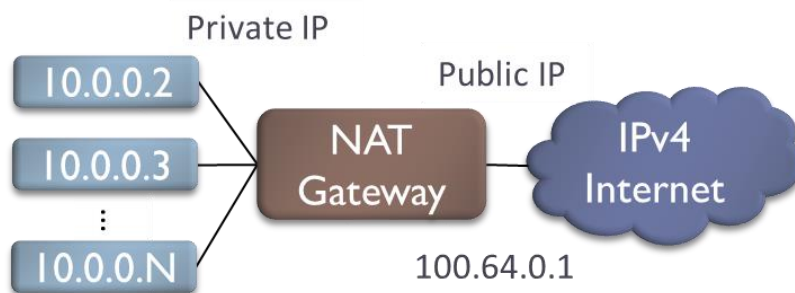


Figure 1-1 NAT gateway router

In Figure 1-1, several devices with private IP addresses in the local area network of the NAT gateway connect to the wide area network by using a public IP 100.64.0.1. Consider an example that a host with the private IP 10.0.0.2 and the port number 10000 try to establish a connection to a public server with the public IP 140.112.42.99 and the

port number 12345. When the new traffic flow initiated by the host 10.0.0.2 arrives at the NAT gateway, a new port mapping is generated, let us say the port number 20000, and stored in the gateway. For every packet in the same data flow, the NAT gateway will modify the corresponding header fields according to the mapping information. In our example, the NAT gateway will modify the source IP field from 10.0.0.2 to 100.64.0.1 and the source port field from 10000 to 20000, and then send out to the server 140.112.42.99. For the packets sent back from the server, the NAT gateway will modify the destination IP field and destination port field based on the pre-stored port mapping information and send the packets back to the original host 10.0.0.2. The procedure is detailed in Figure 1-2.

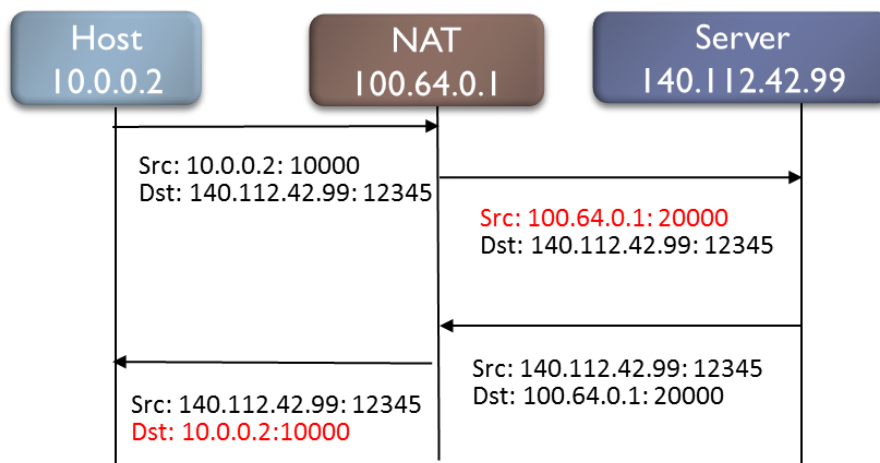


Figure 1-2 NAT function example

## 1.2 NAT 444 Architecture

Because of the IPv4 exhaustion problem, most of the large scaled network architectures nowadays adopt multiple layer NAT middle-boxes to minimize the need for the public IP addresses. For instance, to accommodate more new customers and more

emerging network devices, a lot of Internet Service Providers (ISP) choose to deploy the NAT 444 architecture in their internal network. The NAT 444 architecture is a two-layered NAT architecture consists of two kinds of NAT middle-boxes, the Carrier Grade NAT (CGNAT) and the Customer Premises Equipment (CPE), as illustrated in Figure 1-3.

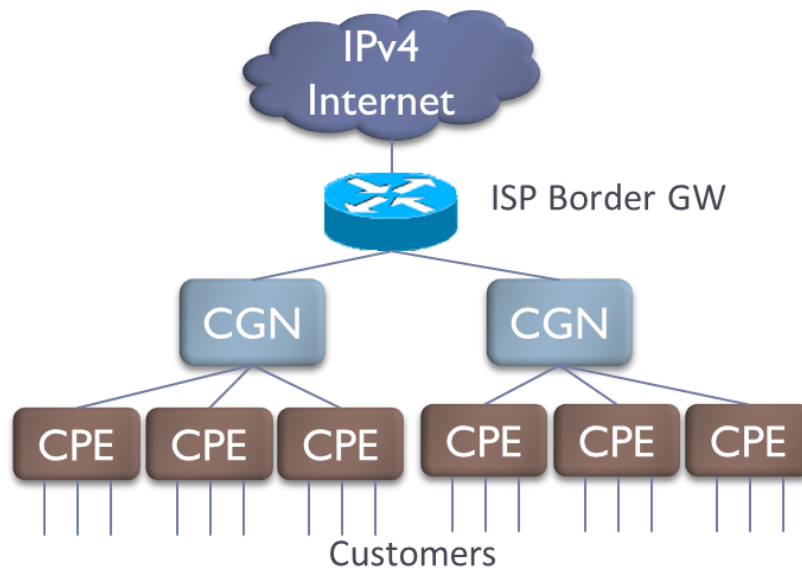


Figure 1-3 NAT 444 architecture

Under the NAT 444 architecture, the flows generated from the customers' equipment need to pass through at least two gateways before reaching the IPv4 Internet. The first gateway is the CPE, which is usually a terminal hardware co-located with Digital Subscriber Line (DSL), cable, or fiber optic modem at tenant's home or office by the ISP. The CPE provides the connectivity for user devices to access the core network, such as personal computers, cell phones, and IPTVs. The second one is the CGNAT, which is a large-scale NAT server deployed in the operator's network. The NAT 444 architecture could effectively reduce the requirement of the public addresses in the ISP network by aggregating many private IP addresses into fewer public IP addresses. Before the adoption of IPv6 becomes widely popular, the NAT 444 architecture is still the major solution to mitigate the IPv4 depletion problem.



### 1.3 Problems with Today's NAT

Despite the fact that the NAT 444 architecture is the most popular adopted solution to the IPv4 exhaustion problem, it is still not the sustainable solution. The NAT function introduces a variety of issues to the network. For example, as mentioned in section 1.2, the NAT 444 architecture contains multiple-layer NAT servers. In such a scenario, any traffic flow originating from the user devices will be masqueraded several times by the NAT middle-boxes before it reaches the wide area network. From the perspective of the ISP, the IP masquerading procedure results in difficulty to identify each data flow with modified packet header, so it is unlikely to perform the flow-based differentiated service in the ISP internal network. Therefore, the network managers could not have a complete and overall control of their internal network.

The NAT 444 architecture also inherits the disadvantages from the tradition NAT. Firstly, the CGNAT are usually delivered by dedicated and proprietary hardware. Therefore, it suffers from low scalability and is hard to scale up or scale out to fit the ever-changing network usage pattern. Besides, if a CGNAT is fully loaded, the others are not able to offload the congested traffic. Secondly, the traditional NAT is a stateful machine. Since the NAT server stores the dynamic IP/port mapping information in the memory, it cannot restore the information after experiencing a power outage or a server failure. Thus, it brings out the reliability issue. Lastly, the traditional NAT breaks the end-to-end principle. The NAT function gets involved with not only the network layer transmission of the packets but the transport layer as well. It will hide the transport layer information of subscribers' machines behind it. As a result, it causes difficulties to perform the peer-

to-peer applications such as VoIP and P2P file transfer.



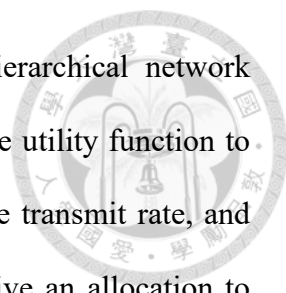
## 1.4 SDN-based QoS Aware NAT

To address the issues of the traditional NAT and obtain a more fine-grained overall control, we utilize the Software Defined Network (SDN) technology to re-implement the NAT function. We replace the dedicated NAT servers by the OpenFlow switches and employ a central SDN controller to control the behavior of these switches. The NAT function is delegated to the central controller and the port mapping information is stored in the SDN controller rather than the switches. Therefore, with the global network view provided by the SDN controller, the network operators can not only easier to manage the underlying network infrastructure but also control the network behavior in a flow-level manner. The source of a traffic flow can be identified no matter how many NAT layers it transverses so that the per-flows control for each subscriber is possible to enforce.

There are more benefits of the SDN-based NAT. For instance, since the port mapping information is kept in the SDN controller, it could be restored to the recovered server or another backup server even though a power outage or failure happened. The controller can install the copy of the mapping table to the working switch and concurrently sets up new forwarding rules to detour the affected flows. Thus, the SDN-based NAT could attain the stateless NAT and improve reliability. Besides, since the dedicated NAT servers are all replaced with the commercial OpenFlow switches, it makes it easier to manage and scale the network because they have a standardized structure to do mass production.

In our work, we also exploit the queue module supported in OpenFlow protocol to





achieve the flow-based QoS aware bandwidth allocation in a hierarchical network structure like the NAT 444 architecture. We adopt the concept of the utility function to measure the quality of experience of a data flow with respect to the transmit rate, and model the bandwidth allocation as an optimization problem to derive an allocation to maximize the total utility score while achieving the utility proportional fairness. We implement the SDN-based QoS aware NAT and conduct simulations on a Raspberry Pi platform with OpenvSwitch installed as an OpenFlow switch. Experiment results show that our implementation could achieve the function of NAT and the flow-level QoS enforcement.

## **1.5 Thesis Organization**

The rest of the paper is organized as follows. Chapter 2 introduces the literature related to our work. Chapter 3 describes our system architecture of the implementation of the SDN-based NAT and the QoS aware NAT respectively. Chapter 4 shows that how we model and analyze the bandwidth allocation problem and solve it by a heuristic greedy algorithm. Chapter 5 gives the simulation results of our proposed scheme. The conclusion and future work are discussed in Chapter 6.

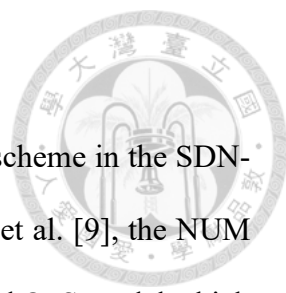
# Chapter 2

## Related Work



There is literature proposed utilizing the SDN technology to realize the NAT function to solve the existing problems. To the best of our knowledge, the concept of the SDN-based NAT is firstly proposed in [1]. Gijeong K. et al. [1] described an SDN based fully distributed NAT traversal scheme for IoT global connectivity. In the past, the NAT traversal problem is addressed by adding an external public relay server to handle the connection [2][3]. The proposed scheme in [1] used the SDN switches and the SDN controller to replace the NAT gateway and the relay server to distribute the workload of NAT processing to devices and reduce transmission delay by packet switching instead of packet modification. However, to modify packet headers in the user devices, the devices also need to be under control of the controller so that additional configurations are needed. The authors in [4] also proposed an SDN-based solution to resolve the NAT traversal problem and improve the current ALG (Application Layer Gateway) solution for SIP/IMS multimedia services in the All-IP mobile networks. Marnel P. et al. [5] proposed a horizontal model mobility management in a wireless network, which has the combined functions of NAT management and MME (Mobility Management Entity) in the control layer. In [6], the authors proposed the use of an SDN gateway to perform a flow-based security mechanism for IoT devices. Their scheme conducted the traffic patterns analysis to determine and block specific malicious flows instead of blocking all suspected traffic flows from a NAT gateway. Although the related works listed above proposed various possible applications of the SDN-based NAT, they are all lack of simulation or

implementation in real hardware.



In our work, we implement a QoS aware bandwidth allocation scheme in the SDN-based NAT. Since the seminal papers of Kelly et al. [7][8] and Low et al. [9], the NUM (Network Utility Maximization) framework has become a well-studied QoS model which expresses the network resource allocation as an optimization problem[10][11][12][13]. Our implementation work is primarily motivated from FlowQoS [14] and Contextual Router [15]. In FlowQoS [14], the authors proposed an SDN-based QoS approach for application-based bandwidth allocation on a home gateway. They classified the traffic flows into different application types on the fly by a light-weighted inspection tool [16], and then configured bandwidth traffic shaper of each type to achieve the differentiated resource allocation. However, the bandwidth allocated for each application types in [14] is pre-defined by users statically so that it could not adapt to the dynamic network environment. Contextual Router [15], also motivated from [14], implemented a dynamical resource allocation on a home router. They exploited utility functions to define the quality of experience of applications and modeling the resource bandwidth problem as an optimization problem whose objective is to maximize the total utility value. Whereas, the utility functions used in [15] are simple piecewise linear functions that could not address the actual needs of existing network applications. Moreover, the optimization problem did not consider the fairness issue either.

# Chapter 3

## System Architecture



In this chapter, we will describe how we implement the SDN-based NAT in detail. Figure 3-1 illustrates how we implement the SDN-based NAT in an NAT444-like architecture.

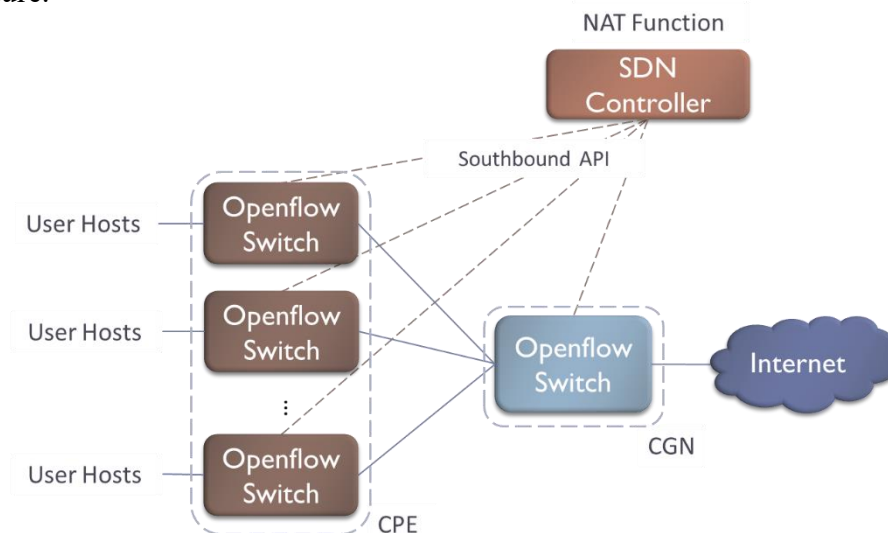


Figure 3-1 SDN-based NAT444 architecture

In the SDN-based NAT, the traditional NAT middle-boxes are replaced with the OpenFlow switches, and the central controller will take control of all these switches via the southbound protocol. We delegate the NAT function to the SDN controller so that the NAT port mapping policy will be determined and stored in the central controller. The controller will install the rules of packet modification and packet forwarding to the switches according to the NAT mapping information. The switches do not get involved with the policy determination but only execute the installed rules in the flow table when data packets arrived. In the rest of this chapter, we describe the features and the

implementation details of the proposed architecture.



### 3.1 SDN Controller

In our implementation, Ryu [17] is chosen as the SDN controller. It is a component-based software defined networking framework developed in Python language. Ryu supports fully OpenFlow protocols from version 1.0 to 1.5. It also provides a great variety of RESTful API for developers to make it easier to create new network applications. The Ryu controller architecture is shown in Figure 3-2.

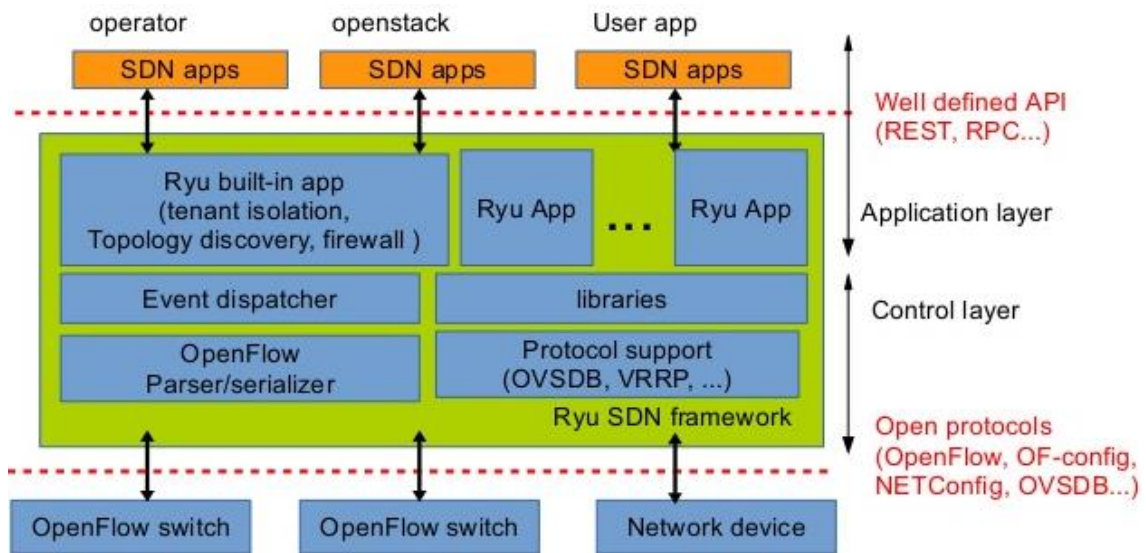


Figure 3-2 Ryu controller architecture

Ryu is developed as an event driven framework. When receiving a southbound message, an event corresponding to the message is dispatched and sent to the appropriate event handlers. The Ryu Apps developed by the network managers could register related event handlers to catch the events and further perform the desired control logic. In our work, we implement the NAT function as a Ryu application in Python language. The

application will catch the events indicating that there are new packets received from the data plane and apply the corresponding NAT functions.



## 3.2 Southbound API

Southbound API is the interface allows communications between the controller and the SDN switches. In our architecture, there are two kinds of Southbound API, OpenFlow [18] and OVSDB [19], in charge of different control messages respectively.

### 3.2.1 OpenFlow Protocol

The OpenFlow protocol, which is developed by the Open Networking Foundation (ONF), has become a standard of the southbound interface in recent years. It defines many kinds of control messages exchanged between the controller and the OpenFlow switches to realize fully operation and traffic engineering intelligence of the SDN network. We choose the version 1.3 since it is stable and commonly implemented on commercial hardware currently such as Pica8 and HP OpenFlow switches. In our architecture, the OpenFlow protocol is used to inform the controller the arrival of new packets at OpenFlow switches and install flow entries into the flow tables according to the NAT policy. Network application sitting on top of the controller can also leverage this protocol to obtain the status of the active flows in the network by analyzing the flow table.

### 3.2.2 OVSDB Protocol

The OVSDB protocol is the abbreviation of the Open vSwitch Database Management Protocol, which is used as a lightweight switch database server to store the

configuration information of an OpenFlow switch. In addition to the traffic engineering logic control, the protocol provides more physical hardware configurations that are not supported by the OpenFlow protocol. For example, the OVSDB protocol allows network managers to remotely create and delete ports, bridges, and queues of a switch. In our implementation, we configure the limited data rate of queues of OpenFlow switches as traffic shapers via the OVSDB protocol.

The difference of two southbound API used in our work is listed in Table 3-1.

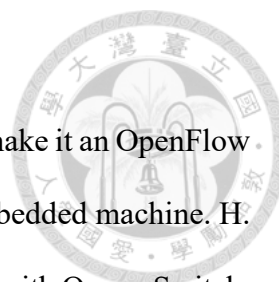
<b>Southbound API</b>	<b>Usage</b>	<b>Examples</b>
Openflow	Traffic engineering	Packet forwarding Packet modification
OVSDB	Switch configuration	Add/delete ports Queue management

Table 3-1 Southbound API

### 3.3 Raspberry Pi as an OpenFlow Switch

OpenFlow switches are the forwarding elements in the data plane under control of the SDN controller. When a packet arrives at an OpenFlow switch, the switch looks up the flow table to find a matching flow entry for the received packet and then execute the instructions in the rule; otherwise, the switch notifies the controller via the OpenFlow protocol to get installed related rules.

Open vSwitch [20] is an open source multilayer virtual switch commonly deployed in large-scale computing environments. It was firstly designed for forwarding data traffics between different virtual machines or even virtual machines and physical machines. Open vSwitch is also designed to be compatible with modern switching chipsets so that it can



be ported to the Linux-based platform.

In our work, we install Open vSwitch on a Raspberry Pi [21] to make it an OpenFlow switch. Raspberry Pi is a small, low-cost, and easily programmed embedded machine. H. Kim [22] also suggested Raspberry Pi as an SDN forwarding element with Open vSwitch. Since the Raspberry Pi is equipped with an ARM-based processor, we cross compile the source code of Open vSwitch by `gcc` tool and port it into the Raspberry Pi. Moreover, the Raspberry Pi only has one Ethernet RJ45 port, so we use USB to Ethernet converters to create more Ethernet ports, as shown in Figure 3-3.



Figure 3-3 Raspberry Pi

```
root@raspberrypi:/home/pi# ovs-vsctl show
cec602dc-850b-4f7a-86a1-0c64a868da8e
  Manager "ptcp:6632"
  Bridge "br0"
    Controller "tcp:192.168.96.105:6633"
      is_connected: true
    fail_mode: standalone
  Port "eth1"
    Interface "eth1"
  Port "br0"
    Interface "br0"
      type: internal
  Port "eth2"
    Interface "eth2"
  Port "eth0"
    Interface "eth0"
root@raspberrypi:/home/pi# _
```

Figure 3-4 OpenFlow switch configurations

The switch ports are bridged using the OVSDB tool to exchange the data packets between network interfaces. We also write a shell script running on the start-up of the switch. After a Raspberry Pi boots up, it automatically loads the Open vSwitch module, connects to the controller, and listens for OpenFlow and OVSDB messages. We detailed the configurations in Figure 3-4.

### 3.4 SDN-based NAT Function



In this section, we describe how we realize the SDN-based NAT function. When a new data flow arrives at a switch, the switch will check if the flow matches any forwarding rule in the flow table. If a flow entry is matched, then the switch directly execute the actions in the rule and send out data packets. Otherwise, the switch will generate a *PACKET\_IN* message and send it to the controller by the OpenFlow protocol along with a packet of the new flow. The controller determines and stores the NAT port mapping information according to the received packet. A NAT mapping table and a NAT port pool are also maintained in the controller to enable the policy. Finally, the controller derives a set of flow rules to install to the switch via OpenFlow *OFPFLOWMOD* method. We note that since the switches always send the unknown packets to the controller in the SDN, our proposed scheme does not increase the network overhead. The flow chart of the SDN-based NAT function is shown in Figure 3-5.

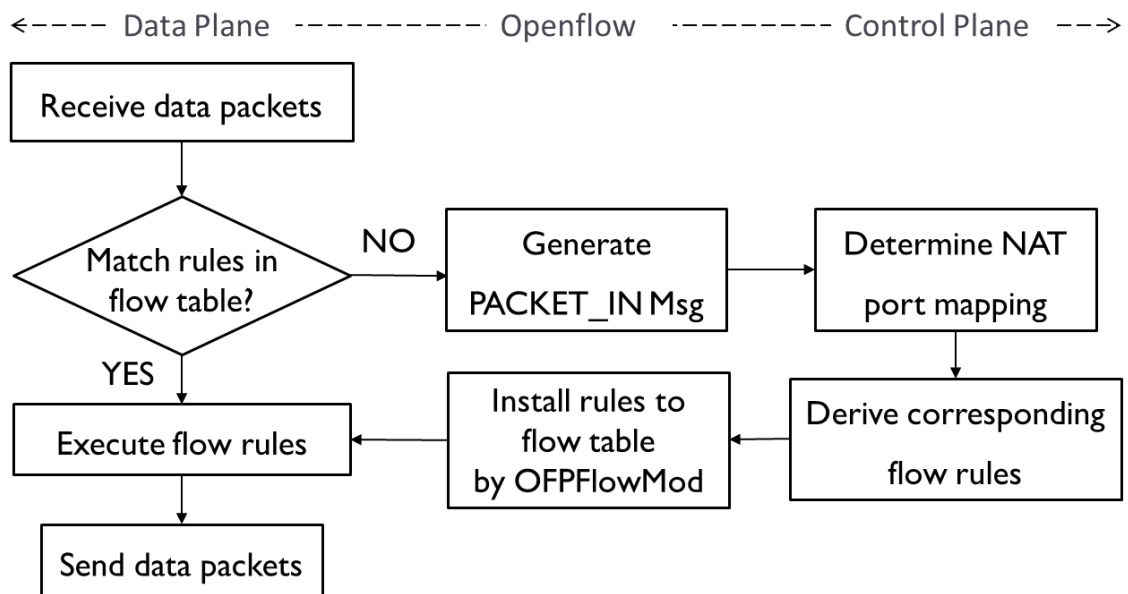


Figure 3-3 SDN-based NAT flow chart

We give an example of how the communication works. Consider a case as depicted in Figure 3-6, where a host with the IP 10.0.0.2 tries to communicate with a public server

with the IP 140.112.42.99 through the double-layered NAT architecture. When the first packet of a new traffic flow generated by the host 10.0.0.2 arrives at switch *S1*, since the new flow will match no rules in the flow table, switch *S1* will generate a *PACKET\_IN* message and send the message and the packet to the controller.

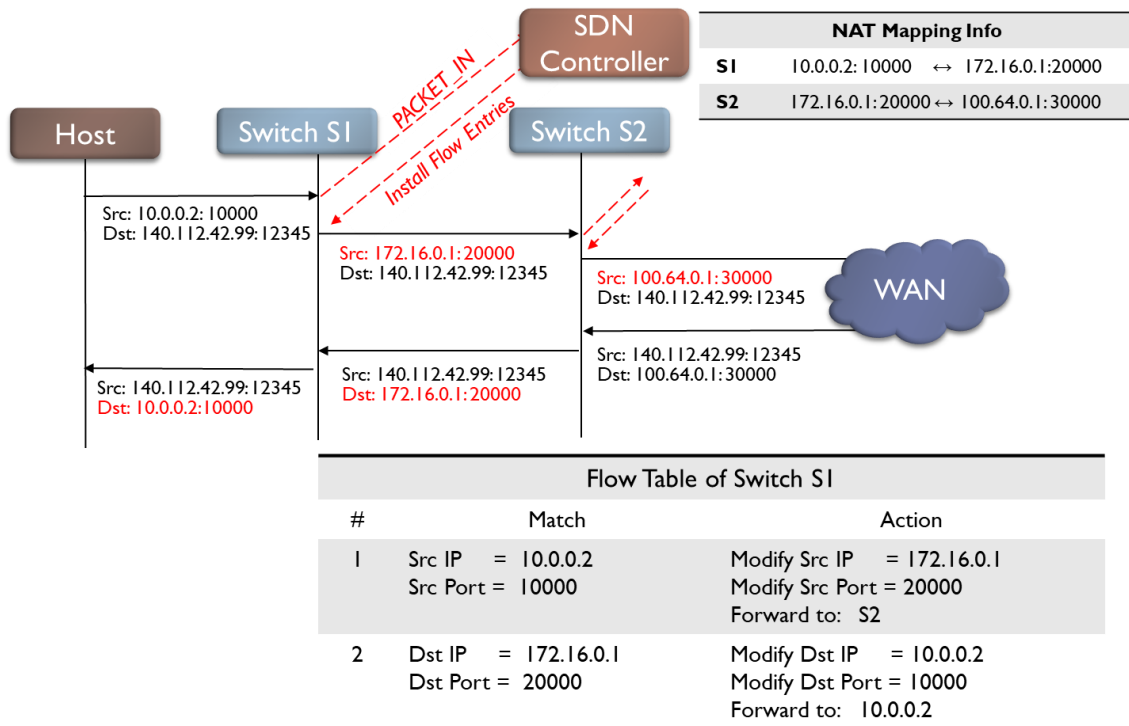
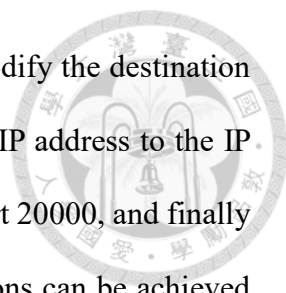


Figure 3-4 SDN-based NAT system architecture

In this example, the NAT logics that we implement in the controller makes a decision to map the flow from 10.0.0.2:10000 to the port number 20000 in switch *S1*. Then, the controller stores the mapping information in the mapping table, as the first entry of the flow table of switch *S1* listed at the bottom of Figure 3-6. The structure of the NAT mapping table is shown in Figure 3-7.

Switch #	Switch IP	NAT port	Src IP	Src port	Dst IP	Dst Port	protocol
S1	172.16.0.1	20000	10.0.0.2	10000	140.112.42.99	12345	TCP
S2	100.64.0.1	30000	172.16.0.1	20000	140.112.42.99	12345	TCP

Figure 3-5 NAT mapping table

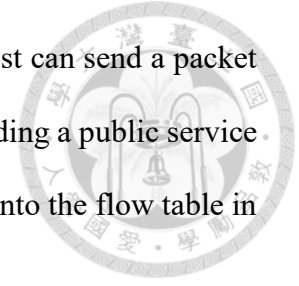


In this case, the forwarding rule includes these instructions: modify the destination MAC address to the MAC address of switch *S2*, modify the source IP address to the IP assigned to switch *S1*, modify the source port number to the NAT port 20000, and finally forward the packets to the switch *S2*. The process of the modifications can be achieved by *OFActionSetField* function provided by OpenFlow, which is able to modify the packet header fields from the data link layer to the transport layer. A similar procedure is conducted when the flow arrives at switch *S2*.

When the backward flow arrives at switch *S2* and the controller is informed, the controller will check the stored mapping table and look up a record to remap the original source IP and the source port of the flow. In our case that switch *S2* receives a flow with the destination IP 100.64.0.1 and the destination port number 30000, according to the mapping table, we could know that the flow is initially generated by the host with the IP 10.0.0.2 and the port number 10000. Thus, the NAT function could install rules to switch *S2* to modify the destination IP and the port number for the backward flow and send packets towards the host. We note that the ability to recognize the origination of a data flow is unattainable in traditional NAT since the mapping information is individually and locally stored. For example, when switch *S2* receives the data flow with the source IP/port 172.16.0.1:20000, switch *S2* does not know the flow is initially created by the host 10.0.0.1:10000. This unrevealed information causes difficulties of per-flows level network services.

If a data flow sent from the outside network to the inside network and there are no mapping records in the mapping table, it indicates the flow is initiated from an external server originally trying to connect to the devices in the local area network actively. In our work, we will ignore the flow for security concerns. The rationale behind the implementation here is the operation mode of the symmetric NAT. In the symmetric NAT,

only an external host that ever received a packet from an internal host can send a packet back. If an exception is needed in our implementation, such as providing a public service behind the NAT function, the related flow entries could be installed into the flow table in advance to enable the communication.



# Chapter 4

## QoS Enabled NAT



In this chapter, we discuss how we design and implement the QoS enabled SDN-based NAT. There are many QoS metrics studied and discussed in the literature, such as packet loss, jitter, round trip latency, and data transmission rate. In our work, motivated from [14][15], we utilize the queue module in the OpenFlow switches to implement the flow-based bandwidth allocation to approach the QoS differentiation. We adopt utility functions to measure the quality of experience of traffic flows with respect to the transmission rate, and model the bandwidth allocation as an optimization problem to derive an allocation maximizing the total utility score while considering the fairness criterion.

### 4.1 Design of QoS Enabled NAT

In this section, we discuss how we design the QoS enabled NAT architecture. The architecture is illustrated in Figure 4-1.

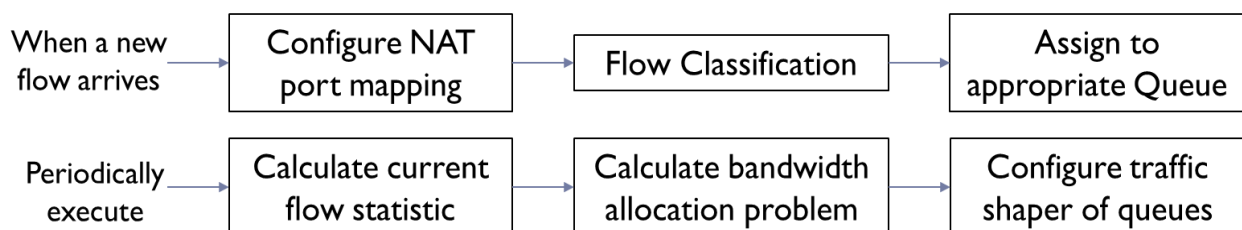


Figure 4-1 QoS Aware NAT architecture



#### 4.1.1 Flow Classification

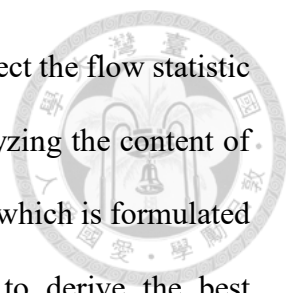
When a new data flow arrives at a switch, the NAT function will be applied first. Then we categorize the flow into a certain application type, where the network manager could define the application types in advance. From the perspective of implementation, it is unlikely to assign a queue to each data flow in the run time. Therefore, we implement a queue for an application type in the OpenFlow switches so that we could do traffic shaping for each type. As long as a flow is classified, the flow is assigned to the corresponding queue based on its type by the OpenFlow *OFActionSetQueue* method.

We adopt the classification method in [14]. The authors in [14] use a simple DNS classifier and a light-weighted packet inspection tool Libprotoident [16] for application identification. The method requires the well-known five-tuples and the other four fields of the packet, first four bytes sent, first four bytes received, first payload size sent, and first payload size received. Although the OpenFlow switches only support the traffic engineering from the network layer 2 to layer 4, the controller still could retrieve the layer 7 information from the packet payload to perform the classification. Previous work [23] shows that Libprotoident tool was properly able to classify 94% of 1,262,022 flows captured over 66 days.

We note that the traditional NAT function might masquerade the information of the five-tuples and result in the categorization errors. With the SDN-based NAT, the origin information of the five-tuples could be obtained and further increase the accuracy of classification.

#### 4.1.2 Rate Controller

To adapt to the dynamic network environment, we periodically configure the traffic



shaper of the queues to approach the QoS enforcement. First, we collect the flow statistic to know that how many active flows in the current network by analyzing the content of the flow tables. Second, we solve the bandwidth allocation problem, which is formulated as an optimization problem with the flow statistic information, to derive the best allocation method for each type in the present. The optimization problem will be discussed in section 4.2. Third, we configure the traffic shaper of the queues in the OpenFlow switches according to the optimal solution to enable the data rate limiting for each application type.

Even though the OpenFlow protocol supports assigning flows into different queues, the configuration of the queues is done out of the protocol. In our prototype, we conduct the settings via the OVSDB protocol as introduced in section 3.2.2.

## 4.2 Bandwidth Allocation Problem

We model the bandwidth allocation problem as a nonlinear programming optimization problem. Without loss of generality, we consider a hierarchical network topology just like the NAT 444 architecture as shown in Figure 4-2.

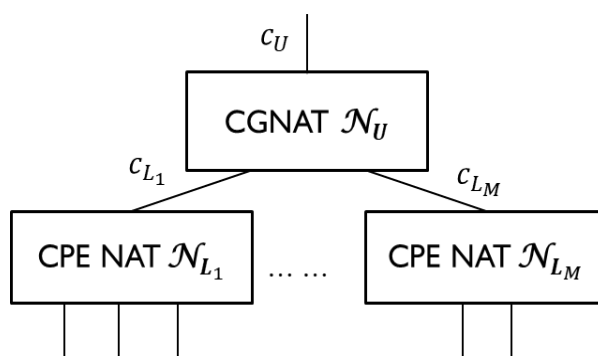


Figure 4-2 Two-layer network architecture

### 4.2.1 Basic Model

Let  $\mathcal{N} = \{\mathcal{N}_{L_1}, \mathcal{N}_{L_2}, \dots, \mathcal{N}_{L_m}\} \cup \{\mathcal{N}_U\}$  denotes the NAT middle boxes set in the

network consists of one CGNAT gateway and  $m$  CPEs, where  $\mathcal{N}_{L_i}$  is the  $i^{th}$  CPE gateway which directly connects to the user devices, and  $\mathcal{N}_U$  is the aggregated CGNAT server which connects to other CPEs. Let  $\mathbf{c} = \{c_{L_1}, c_{L_2}, \dots, c_{L_m}\} \cup \{c_U\}$  denotes the set of available capacity of the outer links of  $\mathcal{N}$ . Then we could derive an upper bound of the total bandwidth allocation that controller could assign to  $\mathcal{N}_{L_i}$  according to each data flows. For every  $\mathcal{N}_{L_i}$ , let  $f_{i,j}$  denotes the  $j^{th}$  traffic flow generated by a user connected to  $\mathcal{N}_{L_i}$ , and  $b_{i,j}$  represents the bandwidth allocation for  $f_{i,j}$  assigned by the controller, then first we have

$$\sum_{j=1}^{n_{L_i}} b_{i,j} \leq c_{L_i}, \forall i \in \{1, 2, \dots, m\} \quad (1)$$

, where  $n_{L_i}$  is the total number of flows belong to  $\mathcal{N}_{L_i}$ . Moreover, the total allocated bandwidth could not exceed the available bandwidth of the bottleneck link of  $\mathcal{N}_U$ , so we also have

$$\sum_{i=1}^m \sum_{j=1}^{n_{L_i}} b_{i,j} \leq c_U \quad (2)$$

In the flow classification procedure, each flow  $f_{i,j}$  would be categorized into a specific application type. Let  $T = \{t_1, t_2, \dots, t_K\}$  be the application type set defined by the network manager, where  $K$  is the total number of types. The flow classification process could be considered as a function  $\pi : \mathbf{f} \rightarrow T$  that could map a traffic flow  $f_{i,j}$  to an element  $t_k$  in set  $T$ . The utility function  $U : (t, b) \rightarrow \mathbf{R}^+ \cup \{0\}$  is also defined by the network manager and takes two value as its input parameters, the application type and the data rate of the flow. For a specific flow  $f_{i,j}$ , the corresponding utility value  $u_{i,j}$  is calculated as



$$u_{ij} = U(\pi(\mathbf{f}_{i,j}), b_{ij}), \forall i \in \{1, 2, \dots, m\} \text{ and } j \in \{1, 2, \dots, n_{L_i}\}. \quad (3)$$

In our implementation, we assign a queue to each application type. The implementation leads to the fact that the flows still need to compete with the other flows in the same queue for available bandwidth allocated to the application type. Therefore, we assume that the bandwidth allocated for any two flows assigned to the same queue of  $\mathcal{N}_{L_i}$  will be identical, as described in Assumption I.

**Assumption I:** For any two traffic flows under the same NAT server, say  $\mathbf{f}_{i,j}$  and  $\mathbf{f}_{i,j'}$  where  $j \neq j'$ , if they also belong to the same application type, then the bandwidth allocated for  $\mathbf{f}_{i,j}$  and  $\mathbf{f}_{i,j'}$  will be identical. That is, if  $\pi(\mathbf{f}_{i,j}) = \pi(\mathbf{f}_{i,j'})$ , then  $b_{ij} = b_{ij'}$ ,  $\forall i \in \{1, 2, \dots, m\}$ ,  $j$  and  $j' \in \{1, 2, \dots, n_{L_i}\}$ .

The Figure 4-3 above depicts the queues implementation in an OpenFlow switch.

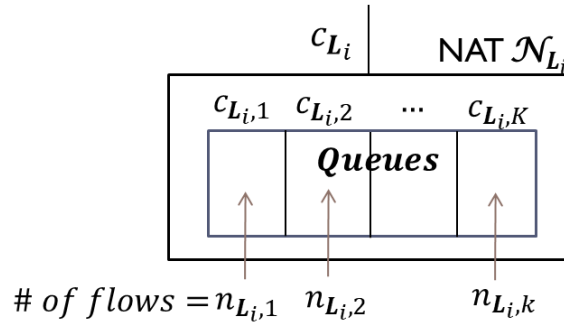


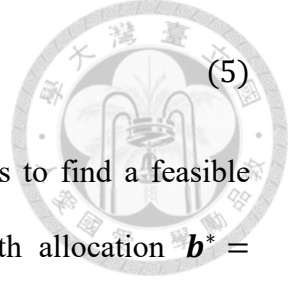
Figure 4-3 Queues implementation in a switch

Let  $c_{L_i,k}$  denotes the bandwidth assigned for type  $t_k$  in  $\mathcal{N}_{L_i}$  and  $n_{L_i,k}$  denotes the number of flows categorized into  $t_k$  in  $\mathcal{N}_{L_i}$ . The limitation of the bandwidth allocated to the queues of  $\mathcal{N}_{L_i}$  could be described as

$$\sum_{k=1}^K c_{L_i,k} \leq c_{L_i}, \forall i \in \{1, 2, \dots, m\} \quad (4)$$

Then, according to Assumption I, we have

$$b_{i,j} = \frac{c_{L_i,k}}{n_{L_i,k}}. \quad (5)$$



Finally, to achieve the best quality of experience, our target is to find a feasible bandwidth allocation to maximize total utility score. A bandwidth allocation  $\mathbf{b}^* = [b^*_{1,1}, b^*_{1,2}, \dots, b^*_{m,n_{L_m}}]$  is a feasible allocation if and only if it meets all the constraints (1) to (5) in the above model. Our objective function could be denoted as

$$\max \sum_{i=1}^m \sum_{j=1}^{N_{L_i}} u_{ij}. \quad (6)$$

#### 4.2.2 Utility Function

Take the NUM (Network Utility Maximization) problem [8][9] as a reference, we adopt utility functions to capture the bandwidth requirements of different applications. A utility function, which is defined by the network manager in advance, is designed to determine a utility value given an application type and a specifically allocated bandwidth. A variety of previous works [10][11][12][13] discussed how utility functions have effects on user's quality of experience and how to design these functions. In these works, most of them considered a utility function  $U$  has properties as follows.

- a)  $U$  is an increasing and continuous function
- b)  $U$  is a first and second differentiable function
- c)  $U$  is usually a sigmoidal-like function or a concave function

Essentially, all network applications could be classified into two categories, elastic and inelastic [24]. An elastic application is a TCP friendly application that could adapt its rate to maximize throughput in different network conditions [25]. Examples of such applications include E-mail, web access, and file transferring service. For this class of applications, an increasing concave function is ideal to model the utility as a function of

allocated bandwidth. The logarithm function is the most commonly used concave function to quantify the utility score. In general, we set two parameters for each application types respectively,  $b_{min}$  and  $b_{max}$ . For each applications, the utility score will be set to 0 if the assigned rate is less than  $b_{min}$ , and will be set to 1 if the assigned rate is larger than  $b_{max}$ . The general form of the utility function of an elastic service  $t$  is shown as follows.

$$U(t, b_{i,j}) = \begin{cases} \frac{\log(1 + k * b_{i,j})}{\log(1 + k * b_{max})}, & 0 \leq b_{i,j} < b_{max} \\ 1, & b_{i,j} \geq b_{max} \end{cases} \quad (7)$$

However, most of the network traffics in current networks are generated by real-time applications, such as video streaming, teleconferencing, and VoIP, which are considered as inelastic applications. Inelastic applications are generally delay and data rate sensitive and usually require a minimum transmission rate to enable the service. Therefore, existing works utilize the sigmoidal-like (logistic) function to model the utility function of inelastic applications. Similarly, the utility score will be set to 0 if the assigned rate is less than  $b_{min}$ , and will be set to 1 if the assigned rate is larger than  $b_{max}$ . The general form of the utility function of an inelastic service  $t$  could be described as follows.

$$U(t, b_{i,j}) = \begin{cases} 0 & , 0 \leq b_{i,j} < b_{min} \\ \frac{1}{1 + e^{-k(b_{i,j}-b_0)}} & , b_{min} \leq b_{i,j} \leq b_{max} \\ 1 & , b_{ij} > b_{max} \end{cases} \quad (8)$$

If the allocated bandwidth is close to  $b_{min}$ , we expect the utility score would be nearly close to 0, say  $\delta$ ; if the allocated bandwidth is approaching to  $b_{max}$ , we expect the utility score would nearly equal to 1, say  $1 - \delta$ . Substitute the parameters into (8) then we have

$$\begin{cases} U(t, b_{max}) = \frac{1}{1 + e^{-k(b_{max}-b_0)}} = 1 - \delta \\ U(t, b_{min}) = \frac{1}{1 + e^{-k(b_{min}-b_0)}} = \delta \end{cases} \quad (9)$$



Solve the simultaneous equations (9) then we could obtain the parameters of the sigmoid function as follows.

$$k = \frac{2 \log\left(\frac{1-\delta}{\delta}\right)}{b_{max} - b_{min}}, \quad b_0 = \frac{b_{max} + b_{min}}{2} \quad (10)$$

As the smaller  $\delta$  is set, the function is more like a simple step function. Note that sigmoid function is not a concave function, which might make the optimization problem become non-convex and local optimum solution might be in existence.

### 4.2.3 Utility Proportional Fairness

Fairness is a critical issue when it comes to the problems about allocating insufficient resources. An unfair bandwidth sharing might cause part of user data flows greedily occupy available resources and result in bandwidth starvation of the other flows. Many works studied how to allocate bandwidth fairly among the competing user flows to prevent flows from monopolizing limited resources. Kelly [7] had argued that utility fairness, rather than traditional bandwidth fairness, could more appropriately address the utility requirements of users and achieve better application layer fairness. H. Shi [26] had also argued that proportional fairness might be a better choice than well-known max-min fairness in the constrained rate allocation problems.

Utility proportional fairness is firstly defined in [27], and the definition is shown below.

**Definition I:** A bandwidth allocation  $x^* = \{x_1^*, x_2^*, \dots, x_R^*\}$  is utility proportional fair if it is feasible, and for any other feasible allocation  $x' = \{x_1', x_2', \dots, x_R'\}$  where  $x^* \neq$

$x'$ , we have

$$\sum_{r \in R} \frac{x'_r - x_r^*}{U_r(x_r^*)} \leq 0 \quad (11)$$



Utility proportional fairness could ensure the data flows to obtain the essential quality of service. In [28][29][30], the authors discussed how to derive an allocation with optimal total utility value while satisfying the condition of utility proportional fairness. Different from the objective functions of the traditional NUM problems, they considered maximizing the product of utility scores rather than the summation of utility scores, which is equivalent to maximizing the summation of the nature logarithm of utility scores. Therefore, we replace the objective function of (6) for maximizing the summation of the nature logarithm of utility scores as described in next section.

### 4.3 Heuristic Algorithm

To find a bandwidth allocation  $\mathbf{b}^*$  to maximize the total utility score while achieving utility proportional fairness, the optimization problem could be modeled as Problem (12), and the notations are described in Table 4-1.

$$\mathbf{b}^* = \arg \max \sum_{i=1}^m \sum_{j=1}^{n_{L_i}} \log[U(\pi(\mathbf{f}_{i,j}), b_{i,j})]$$

subject to

$$\sum_{j=1}^{n_{L_i}} b_{i,j} \leq c_{L_i}$$

$$\sum_{i=1}^M \sum_{j=1}^{n_{L_i}} b_{i,j} \leq c_U$$



$$\sum_{k=1}^K c_{L_i,k} \leq c_{L_i}$$

$$b_{i,j} = \frac{c_{L_i,k}}{n_{L_i,k}}$$

$$b_{i,j} \geq 0$$

$$\forall i \in \{1, 2, \dots, m\}, j \in \{1, 2, \dots, n_{L_i}\} \quad (12)$$

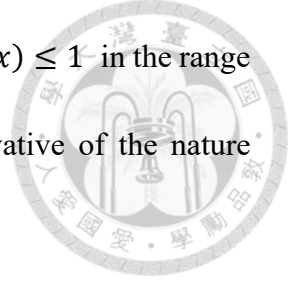
Notation	Description
$\mathcal{N}_U$	The aggregated CGNAT server which connects to the other CPEs
$\mathcal{N}_{L_i}$	The $i^{th}$ CPE gateway which directly connects to the user devices
$m$	The number of the CPE gateways
$n_{L_i}$	The number of flows belong to CPE gateway $\mathcal{N}_{L_i}$
$f_{i,j}$	The $j^{th}$ traffic flow generated by a user connected to $\mathcal{N}_{L_i}$
$b_{i,j}$	The bandwidth allocation for the flow $f_{i,j}$
$T$	The application type set defined in advance
$K$	The number of application types in $T$
$\pi$	The flow classification function $\pi : \mathbf{f} \rightarrow T$ , which could map a traffic flow $f_{i,j}$ to an element in set $T$
$U$	The utility function $U : (t, b) \rightarrow \mathbf{R}^+ \cup \{0\}$ , which gives a utility score for a flow with type $t$ and bandwidth $b$
$c_U$	The available capacity of the outer link of $\mathcal{N}_U$
$c_{L_i}$	The available capacity of the outer link of $\mathcal{N}_{L_i}$
$c_{L_i,k}$	The bandwidth assigned to type $t_k$ in $\mathcal{N}_{L_i}$
$n_{L_i,k}$	The number of flows categorized into $t_k$ in $\mathcal{N}_{L_i}$

Table 4-1 Notation table

### 4.3.1 Problem Analysis

The objective function in Problem (12) is to maximize the summation of nature logarithm of utility functions. Although our utility functions contain the sigmoid function, which is a non-concave function, the natural logarithms of our utility functions are all concave functions.

In elastic service case, the utility function  $U(x)$  is a logarithm function just like



Equation (7). Since  $U(x)$  is a strictly concave function and  $0 \leq U(x) \leq 1$  in the range of  $[0, b_{max}]$ , it follows  $\frac{dU(x)}{dx} > 0$  and  $\frac{d^2U(x)}{dx^2} < 0$ . The first derivative of the nature logarithm of the logarithm utility function could be calculated as

$$\begin{aligned} \frac{d \log(U(x))}{dx} &= \frac{1}{U(x)} \cdot \frac{dU(x)}{dx} \\ \left( \text{by } U(x) \geq 0 \text{ and } \frac{dU(x)}{dx} > 0 \right) &\geq 0 \end{aligned} \quad (13)$$

, and the second derivative could be calculated as

$$\begin{aligned} \frac{d^2 \log(U(x))}{dx^2} &= \frac{d}{dx} \left( \frac{1}{U(x)} \cdot \frac{dU(x)}{dx} \right) \\ &= \frac{1}{U^2(x)} \left[ \frac{d^2U(x)}{dx^2} \cdot U(x) - \left( \frac{dU(x)}{dx} \right)^2 \right] \\ \left( \text{by } U(x) \geq 0, \frac{dU(x)}{dx} > 0, \text{ and } \frac{d^2U(x)}{dx^2} < 0 \right) &\leq 0 \end{aligned} \quad (14)$$

The first and second derivative of  $U(x)$  in the range of  $[b_{max}, \infty]$  are both zero. Therefore, by  $\frac{d \log(U(x))}{dx} \geq 0$  and  $\frac{d^2 \log(U(x))}{dx^2} \leq 0$ , we know the nature logarithm of our elastic utility function is a concave function.

On the other hand, in the inelastic service case, the utility function  $U(x)$  is a sigmoid function as Equation (8). In the range of  $[b_{min}, b_{max}]$ , we have  $\delta \leq U(x) \leq 1 - \delta$  where  $\delta$  is a small positive value nearly close to zero. Although  $U(x)$  is a non-concave function,  $U(x)$  is still a strictly increasing function. Hence, we have  $\frac{dU(x)}{dx} > 0$ . The first derivative of the nature logarithm of the sigmoid function could be calculated as

$$\begin{aligned} \frac{d \log(U(x))}{dx} &= \frac{1}{U(x)} \cdot \frac{dU(x)}{dx} \\ &> 0 \\ \left( \text{by } U(x) > 0 \text{ and } \frac{dU(x)}{dx} > 0 \right) &\text{ could be calculated as} \end{aligned} \quad (15)$$



$$\begin{aligned}
& \frac{d^2 \log(U(x))}{dx^2} \\
&= \frac{1}{U^2(x)} \left[ \frac{d^2 U(x)}{dx^2} \cdot U(x) - \left( \frac{dU(x)}{dx} \right)^2 \right] \\
&= (1 + e^{-k(x-b_0)})^2 \cdot \left[ \frac{k^2 e^{-2k(x-b_0)} (1 - e^{k(x-b_0)})}{(1 + e^{-k(x-b_0)})^4} - \frac{k^2 e^{-2k(x-b_0)}}{(1 + e^{-k(x-b_0)})^4} \right] \\
&= \frac{-k^2 e^{-k(x-b_0)}}{(1 + e^{-k(x-b_0)})^2} \tag{16}
\end{aligned}$$

We have known that  $k = \frac{2 \log\left(\frac{1-\delta}{\delta}\right)}{b_{max}-b_{min}}$  in Equation (10), and since  $b_{max} > b_{min}$  and  $\delta > 0$ , we have  $k > 0$ . Therefore, by Equation (12), we also have  $\frac{d^2 \log(U(x))}{dx^2} < 0$ . Similarly, the first and second derivative of  $U(x)$  in the range of  $[b_{max}, \infty]$  and  $[0, b_{min}]$  are both zero. Therefore, by  $\frac{d \log(U(x))}{dx} \geq 0$  and  $\frac{d^2 \log(U(x))}{dx^2} \leq 0$ , the nature logarithm of an inelastic utility function is a concave function.

As the objective function in the Problem (12) is the summation of concave functions, the Problem (12) is a convex optimization problem. We note that the plateaus might exist in the optimization problem because of the non-strictly concavity.

### 4.3.2 Greedy Approach Algorithm

From the analysis in Section 4.3.1, we know that the Problem (12) is a convex optimization problem so that there are no local optimal solutions in this problem. Moreover, since we need to solve the problem periodically to adapt to the dynamically changing network environment, we tend to find an approach that could derive a solution as quickly as possible. Therefore, it is naïve to adopt a greedy approach to solve our bandwidth allocation problem. In our work, we exploit the concept of the greedy breadth first search algorithm in our approach. The pseudo code of our algorithm is shown as



below.



---

### Greedy Heuristic Algorithm

---

1.  $\text{opt\_alloc} \leftarrow$  choose Best Effort allocation as the initial solution
  2.  $\text{opt\_u} \leftarrow \text{CalculateUtility}(\text{opt\_alloc})$
  3.  $S \leftarrow \{\text{opt\_alloc}\}$
  4. **while**  $S \neq \emptyset$  **do**
  5.      $V \leftarrow$  the first element of  $S$
  6.      $d \leftarrow e^{\frac{|S|}{100}} - 0.9$
  7.      $S \leftarrow S \cup \text{Successors}(V, d)$
  8.      $\text{opt\_alloc}, \text{opt\_u} \leftarrow$  update optimal solution from  $S$
  9.      $S \leftarrow$  remove elements whose utility  $< \text{opt\_u}$
  10. **end while**
- 

We represent the allocation in our algorithm as a vector indicating the bandwidth allocated to each type of each CPE. At the initial stage, the initial solution could be any feasible solution satisfying the constraints in Problem (12). We choose the initial allocation that equally shares the available bandwidth to each flow solution. The reason is that this allocation manner is similar to the traditional best-effort delivery so that the solution we obtain will be definitely better than the result of the best effort delivery. Moreover, it is easy to calculate as well. The initial search space  $S$  contains the initial solution only. For every iteration of the algorithm, we pop out a solution  $V$  from the search space and derive the successors according to the distance  $d$ , which means find all the feasible solutions whose Manhattan distance to the solution  $V$  is equal to  $d$ . The distance  $d$  is designed to escape from the plateaus mentioned in section 4.3.1. Motivated from the well-known simulated annealing algorithm, we use an exponential function to adjust the value of  $d$  dynamically based on the current search space size. Finally, the current optimal solution is updated and only the solution whose utility score is equal to the current optimum will be inserted into the search space. The loop breaks when the search space is empty and the current optimum is outputted as the solution.

# Chapter 5

## Evaluation



In our experiment, we conduct simulations by some synthetic data to investigate our implementation.

### 5.1 Experiment Setup

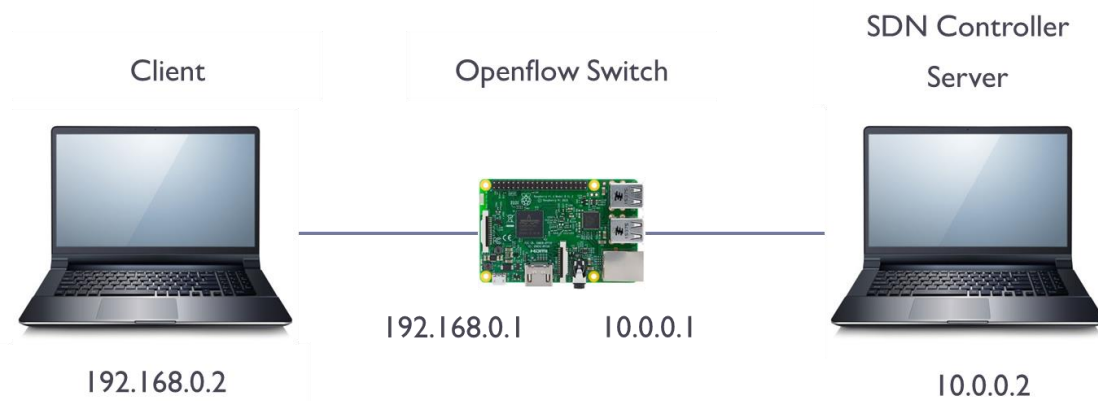


Figure 5-1 Experiment environment

Figure 5-1 our implementation of the SDN-based one-layered NAT prototype with the setting of IP addresses. The OpenFlow switch on the Raspberry Pi connects two laptops, which serves as the client and the application server respectively. The controller is co-located at the application server laptop running as a virtual machine to simplify the experiment structure. The Raspberry Pi connects two different network segments, the internal network 192.168.0.0/24 and the external network 10.0.0.0/24.

We define four application types in our simulations, WEB, FILE, VOIP, and VIDEO. The WEB type and the FILE type are the elastic applications with the logarithm utility

functions, and the VOIP type and the VIDEO type are the inelastic applications with the logistic utility functions. We set the value  $\delta$  as 0.001. The utility functions in terms of the allocated bandwidth are depicted in Figure 5-2 and the parameter settings are listed.

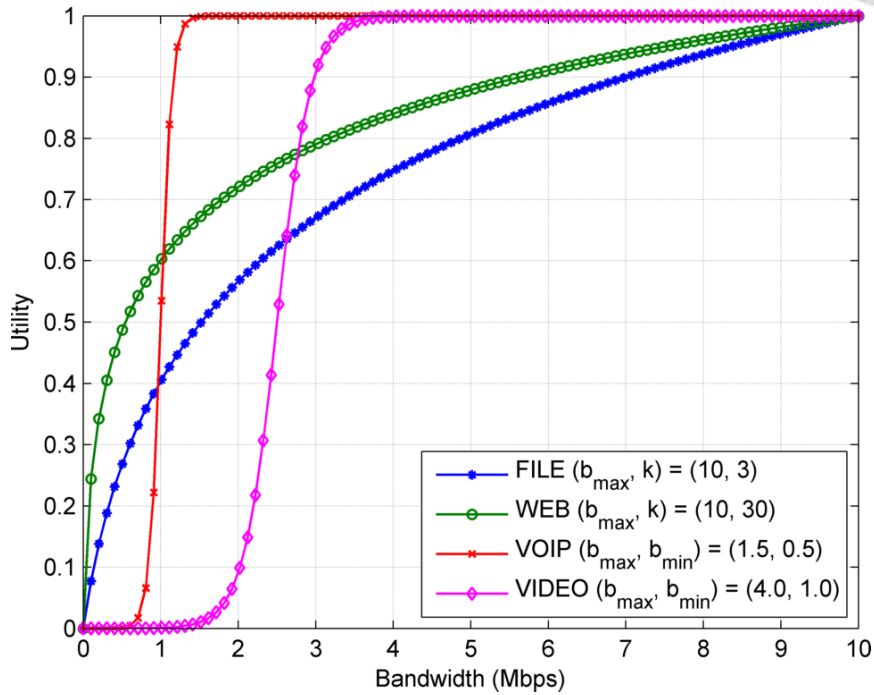
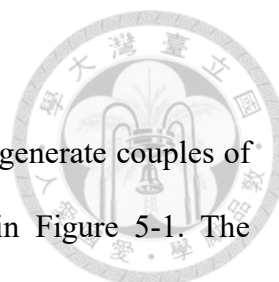


Figure 5-2 Utility functions

We simulate the network traffic flows using the traffic generator Iperf [31] to evaluate our implementation. The configuration interval of the controller is set as one second because that is the minimal *idle\_timeout* value of the flow entry defined in the OpenFlow protocol.

## 5.2 Experiment Results

In this section, we will show that our implementation of the SDN-based NAT function could operate correctly. Afterward, we generate synthetic data to evaluate the QoS aware function in our work.



## 5.2.1 NAT Function

To demonstrate the NAT function, we employ the Iperf tool to generate couples of TCP connections from the client to the server of the topology in Figure 5-1. The executions of Iperf client and server are shown in Figure 5-3 and 5-4.

```

C:\Users\TY\flow_gen>iperf -c 10.0.0.2 -t 10 -P 3
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 8.00 KByte (default)
-----
[292] local 192.168.0.2 port 49408 connected with 10.0.0.2 port 5001
[248] local 192.168.0.2 port 49406 connected with 10.0.0.2 port 5001
[232] local 192.168.0.2 port 49407 connected with 10.0.0.2 port 5001
[ ID] Interval      Transfer    Bandwidth
[248] 0.0-10.0 sec  7.00 MBytes  5.86 Mbits/sec
[292] 0.0-10.0 sec  8.13 MBytes  6.79 Mbits/sec
[232] 0.0-10.0 sec  9.63 MBytes  8.04 Mbits/sec
[SUM] 0.0-10.0 sec  24.8 MBytes  20.7 Mbits/sec

-----
dj184dja8@ubuntu:~/Application/gos$ iperf -s
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 4] local 10.0.0.2 port 5001 connected with 10.0.0.1 port 29520
[ 5] local 10.0.0.2 port 5001 connected with 10.0.0.1 port 57567
[ 6] local 10.0.0.2 port 5001 connected with 10.0.0.1 port 39924
[ ID] Interval      Transfer    Bandwidth
[ 6] 0.0-10.4 sec  8.13 MBytes  6.57 Mbits/sec
[ 4] 0.0-10.4 sec  7.00 MBytes  5.66 Mbits/sec
[ 5] 0.0-10.4 sec  9.63 MBytes  7.77 Mbits/sec
[SUM] 0.0-10.4 sec  24.8 MBytes  20.0 Mbits/sec
  
```

Figure 5-4 Iperf terminal in the client

Figure 5-3 Iperf terminal in the server

In Figure 5-3, three TCP connections are generated from the client with IP 192.168.0.2 and the port numbers are 49406, 49407, and 49408 respectively to the server listening on IP/port 10.0.0.2:5001. Whereas, in Figure 5-4, we observe that the source of the connections received at the server are from the IP 10.0.0.1 and the port numbers are 19520, 57567, and 39924. The results represent that our Raspberry Pi applies the NAT function on these connections successfully. Figure 5-5 shows the mapping table retrieved from the controller. Every entry in the mapping table records an IP address and port translation for a specific traffic flow. With this information, even if the location of the

NAT Mapping Table

NAT #	NAT port	Src IP	Src Port	Dst IP	Dst Port	Protocol
1	29520	192.168.0.2	49406	10.0.0.2	5001	6
2	57567	192.168.0.2	49407	10.0.0.2	5001	6
3	39924	192.168.0.2	49408	10.0.0.2	5001	6

Figure 5-5 The NAT mapping table in the controller

network manager is outside the network segment 192.168.0.0/24, they can still recognize the un-masqueraded information of these connections.



### 5.2.2 Synthetic Data

In this section, we generate synthetic data to evaluate the flow-based differentiated bandwidth allocation scheme in our work. In the synthetic data, we consider a simple scenario as shown in Figure 5-6. The blocks in Figure 5-6 represent how many different types of flows are created and the start/end time of these flows. For example, in this 90-second network script, there are three FILE-type flows simulating the large file transferring as the background network traffic. At the fifth second, the client generates five VIDEO-type flows and the flows are lasting for 30 seconds. We could also see that there are three FILE-type flows, four VOIP-flows, and five VIDEO-type flows competing for the available bandwidth at the 20<sup>th</sup> second.

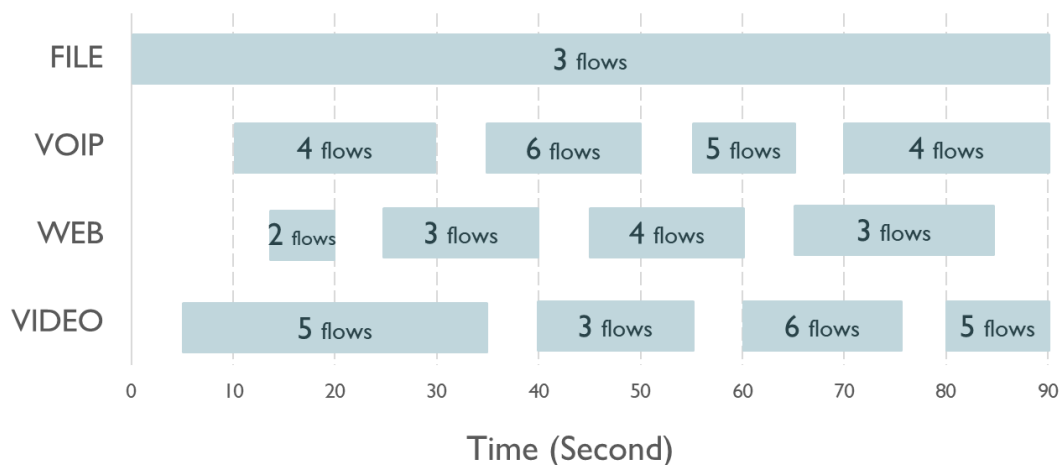


Figure 5-6 Synthetic data

The controller periodically solves the bandwidth allocation problem and configures the rate limiting of the traffic shapers every second. We suppose the total available bandwidth is 30 Mbps so that we could see the effect that the flows compete for available bandwidth with each other. Figure 5-7 shows how our implementation distributes the

available bandwidth to different application types of traffic in synthetic data I.

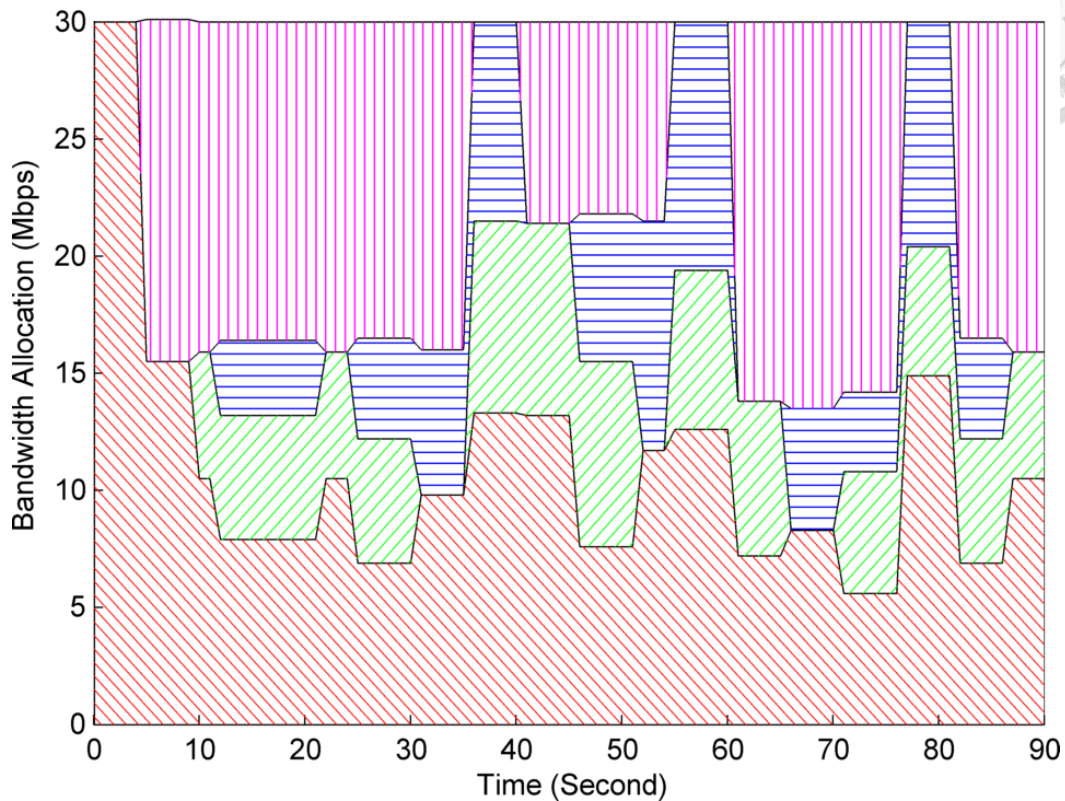
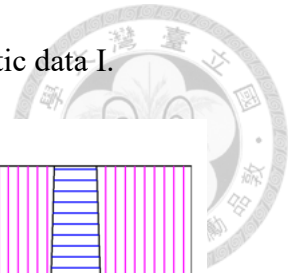


Figure 5-7 Bandwidth configuration result of synthetic data

In Figure 5-7, we could see that the bandwidth allocated for the FILE type decreases to about 16 Mbps at the fifth second since the VIDEO type flows are activated. The bandwidth allocated for the FILE type and the VIDEO type both decrease at the tenth second because of the VOIP-type flows. However, the decreasing margins of the FILE type and the VIDEO type at the tenth second are apparently different, which is the result of current optimal allocation for maximizing the total utility score among all flows. Our optimization problem judges that decreasing the bandwidth allocated for the FILE type more is more profitable than decreasing the bandwidth of the VIDEO type. We also see that, after some flows leave the network, the released bandwidth could be re-allocated to the existing flows again, as the situation at the thirty-fifth second.

In addition, we also measure the actual data transmission rate of each flow in the OpenFlow switch. We utilize the *byte\_count* field of the flow entry, which records the

total number of bytes transmitted from the time when the flow entry is installed, to calculate the data transmission rate. The comparison of the actual transmission rate and the configuration value in synthetic data for each type is depicted in Figure 5-8.

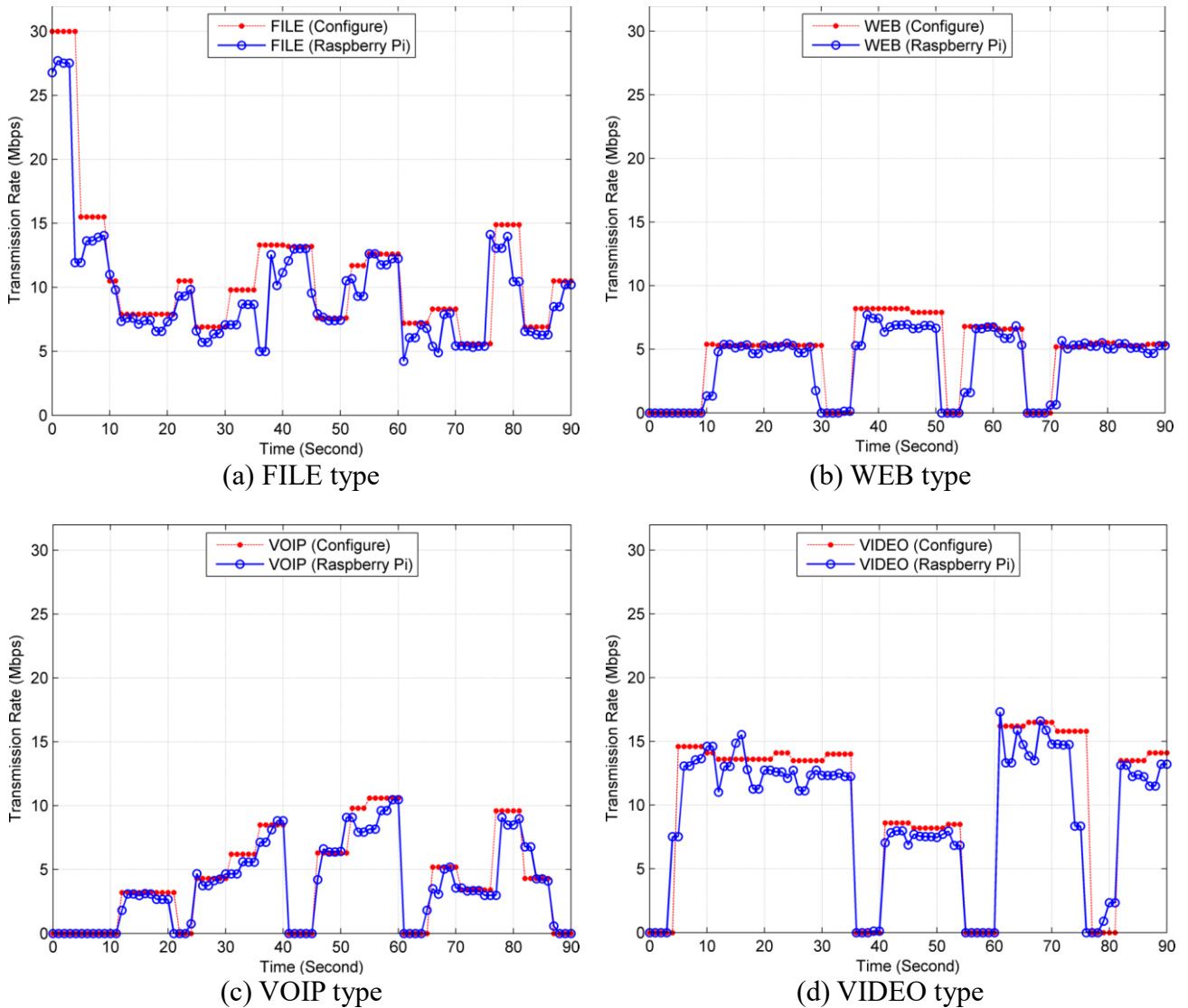
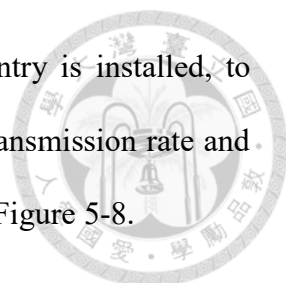


Figure 5-8 Comparison of transmission rate and assigned available bandwidth for different application types

From the experiment results, we could see the actual data transmission rate of each type quite fits the configured value by the controller well. In other words, our implementation is capable of achieving per-flows identification of the network traffic and can perform the flow-based management.

We further compare the theoretical utility score that could be obtained by the controller's configuration and the real utility score calculated from the actual transmission rate, and the result from the beginning to the 45<sup>th</sup> second is shown in Figure 5-9.

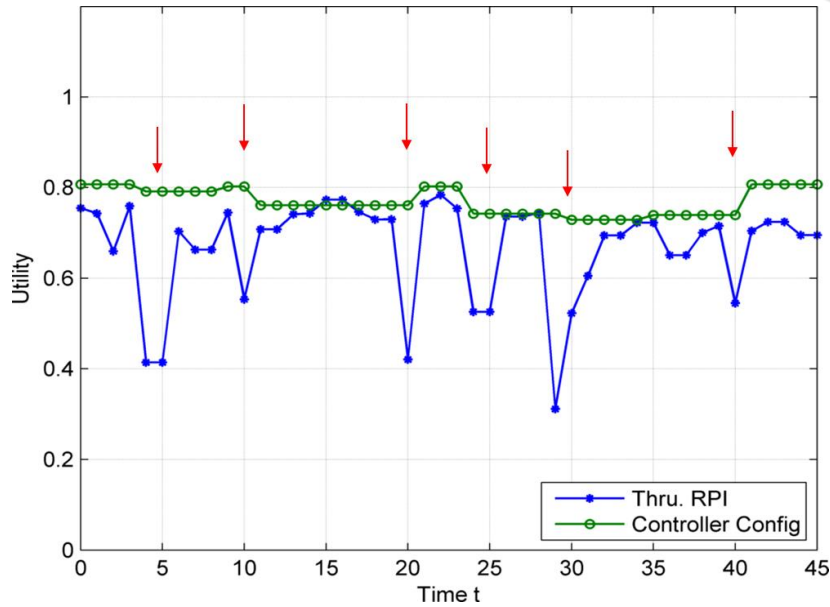


Figure 5-9 Utility scores comparison of synthetic data I

In Figure 5-9, at the time that the red arrows indicate, we observe that the actual utility score drops dramatically compared to the theoretical value. We find that the timing when the utility value drops is coherent with the timing when new flows are generated or existing flows terminate in Figure 5-6. When new flows are generated, the bandwidth allocation of the existing flows will be reduced to accommodate the new flows so that the congestion might be detected. When detecting the network congestion, according to the TCP protocol, the sender will decrease the congestion window to the half of the current amount and perform the slow start algorithm. Therefore, the actual transmit rate will be decreased suddenly and the utility score is dramatically declined.



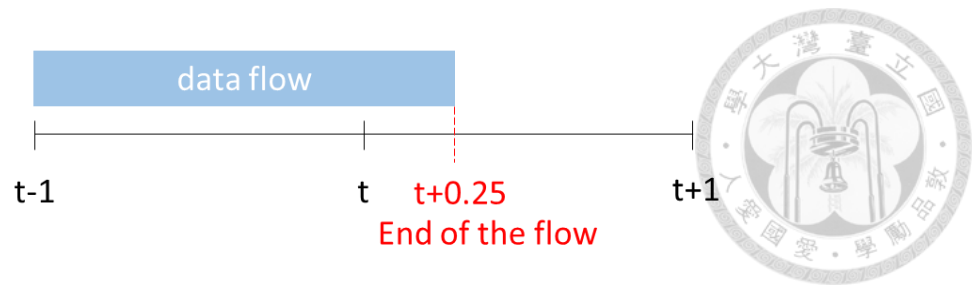


Figure 5-8 Utility drops due to flow ending

On the other hand, when a flow leaves the network, the utility score also drops because we could not estimate the exact time the flow leaves. Figure 5-10 shows such an example. In our work, the configure interval is set as one second, which is the minimal value of the *idle\_timeout* in the OpenFlow protocol. Suppose the actual data transmission rate of the flow in Figure 5-10 is always 10 Mbps, then the data rate we measure between time  $t-1$  and time  $t$  will be 10 Mbps. However, if the flow ends at the time  $t + 0.25$ , the flow entry will be deleted after one more second when the flow stops transmitting data. Therefore, when we try to access the flow table at the time  $t + 1$ , the flow entry will still in the flow table and we will consider that the flow still exists. Consequently, when we measure the data rate from time  $t$  to time  $t + 1$ , it will be only 2.5 Mbps and results in the decline of the utility.

We generate another synthetic data in which the network changes more frequently to conduct the simulation. Suppose the arrival of network traffic flows is a Poisson process with mean  $\lambda$  is 5 seconds, and the existing time of each flow is an exponential distribution with different mean values. For each event when a flow starts, the type of the flow could be one of the types we defined in section 5.1 in different probabilities. The probabilities and the exponential mean values are listed in Table 5-1, and the theoretical utility scores and the utility scores obtained from the simulation is compared in Figure 5-11.



App Types		Probability	Exponential mean $1/\mu$
Elastic	WEB	40%	10
Inelastic	VIDEO	30%	45
Inelastic	VOIP	20%	30
Elastic	FILE	10%	90

Table 5-1 Parameters setting in synthetic data II

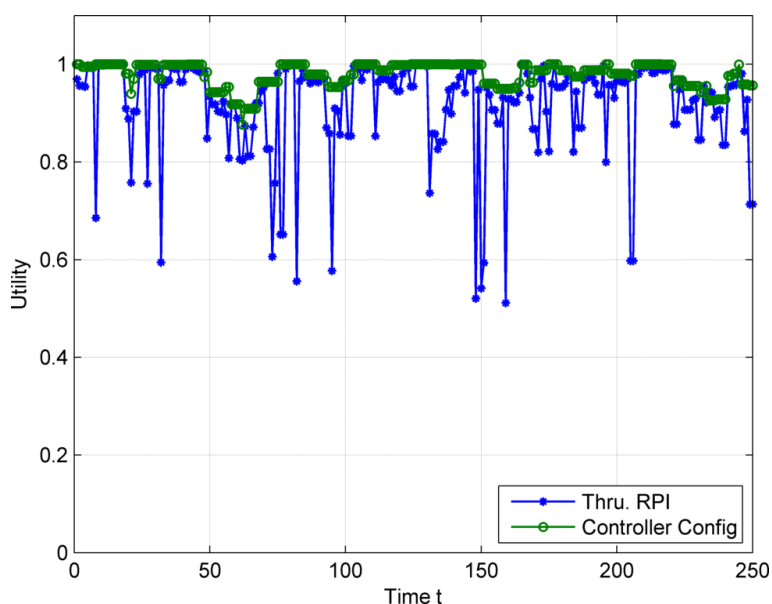


Figure 5-9 Utility scores comparison of synthetic data II

In Figure 5-11, we see that the total utility score oscillated more frequently than the result of the synthetic data I because the flows leave and arrives more frequently in the synthetic data II. We can also see that after the utility score decreases, it will return to the theoretical value soon at the next second if there are no flows changing in the network. Therefore, if the OpenFlow protocol could support a smaller timeout value in the future or a more precise flow detection mechanism is conducted, the performance could be further improved.

# Chapter 6

## Conclusion and Future Work

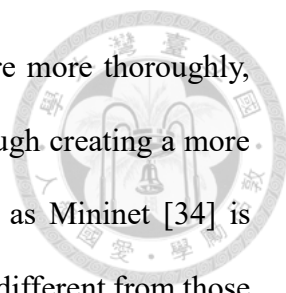


### 6.1 Conclusion

The contribution of this work is to design an SDN-based NAT architecture and implement it on a real hardware platform. We use a low-cost Raspberry Pi installed with Open vSwitch as the OpenFlow switch, and use Ryu as the controller in our implementation. With the global network view provided by SDN, we can achieve flow-level configuration to perform flow-level services that are disabled by the traditional distributed NAT. Furthermore, we implement a flow-based bandwidth allocation scheme in the SDN-based NAT. Considering both of the utility maximization and the utility proportional fairness, we formulate the bandwidth allocation as an optimization problem and solve it by a greedy heuristic algorithm. Then we use the solution to configure the traffic shapers in the OpenFlow switch to achieve the flow-based QoS enforcement. Finally, we conduct simulations in our prototype. The experiment results show that our implementation could achieve the NAT function properly and could identify each data flow to perform the flow-based QoS enforcement.

### 6.2 Future Work

Firstly, since the implementation of this work is a prototype, we consider a simple

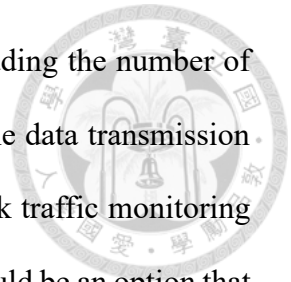


topology in our experiment. To investigate the proposed architecture more thoroughly, simulations in the more large-scale topologies will be needed. Although creating a more complicated network topology in the network emulator tools such as Mininet [34] is relatively easy, the network behaviors on a real testbed might still be different from those on the emulators. Besides, there are several limitations of the emulators. For example, a Mininet-based emulated network cannot exceed the CPU or the available bandwidth on a single physical server, and it cannot run non-Linux applications either currently. Therefore, our future work might consider building a more large-scale testbed to conduct the simulations.

Secondly, the SDN-based NAT architecture enables the logically central software management among the hardware-distributed OpenFlow switches. In the past, a dedicated NAT server is designed to apply the NAT function to the data flows that pass it all the time. Future works might consider using the SDN-based NAT architecture to enable the NAT functions in the OpenFlow switches dynamically. For instance, if the ISP detects the decreasing workload of an OpenFlow switch that substitutes a CGNAT, the ISP might consider disabling the NAT function of the switch or directly shutting down the switch to reduce the total power consumption in their internal network. Traditionally, the detouring work need the reconfiguration of CPE settings and it will involve great human efforts. With the help of the traffic engineering capability provided by SDN, the affected traffic flows could be easily redirected to another operating OpenFlow switch without disturbing the subscribers' traffic.

Thirdly, in our implementation and experiment, we observe that the utility score might decrease when the network changes. It is limited by the current OpenFlow restriction that the minimal *idle\_timeout* value is one second so that we cannot know the exact time when a flow ends. Our implementation could be improved if there are other


ways that can precisely detect the current flow in the network, including the number of current flows, the classified application type of current flows, and the data transmission rates of current flows. For example, sFlow [33] is a scalable network traffic monitoring tool in data networks with several packet sampling mechanisms. It could be an option that extends the functionality of sFlow to meet our requirements and integrate it into our implementation.



# Reference



- [1] Gijeong K. et al, "An SDN based fully distributed NAT traversal scheme for IoT global connectivity", In Proceeding of IEEE ICTC, 2015
- [2] J. Rosenberg et al., "STUN: Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)", IETF RFC 3489, March 2003
- [3] R. Mahy et al., "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", IETF RFC 5766, April 2010
- [4] Whai-En Chen and Bo-En Chen, "An Effective NAT Traversal Mechanism for SIP/IMS Services in SDN-Enabled All-IP Mobile Networks", In Journal of Wireless Personal Communications, Volume 84 Issue 3, 2015
- [5] Marnel Peradilla and Yunchan Jung, "Combined Operations of Mobility and NAT Management on the Horizontal Model of Software-Defined Networking", in Proceedings of ACM ICC 2016
- [6] P. Bull et al, "Flow Based Security for IoT Devices using an SDN Gateway", In Proceeding of IEEE Future Internet of Things and Cloud, 2016
- [7] F. P. Kelly, "Charging and rate control for elastic traffic," in proceedings of Europe Transactions on Telecommunications, vol. 8, pp. 33–37, Jan. 1997.
- [8] F. P. Kelly, A. Maulloo, and D. Tan, "Rate control in communication networks: Shadow prices, proportional fairness and stability," Journal of the Operational Research Society, pp. 237–252, 1998.
- [9] S. Low and D. Lapsley, "Optimization flow control: basic algorithm and convergence," in proceedings of IEEE/ACM Transactions on Networking, vol. 7, no. 6, pp. 861 –874, December 1999.

- 
- [10] J.-W. Lee, R.R. Mazumdar, N.B. Shroff, “Non-convex optimization and rate control for multi-class services in the Internet”, in proceedings of IEEE/ACM Transactions on Networking, Volume 13 Issue, 4, Aug. 2005
- [11] Georgios Tychogiorgos, Athanasios Gkelias, Kin K. Leung, “Distributed network resource allocation for multi-tiered multimedia applications”, in proceedings of IEEE INFOCOM 2015
- [12] Mo Ghorbanzadeh, Ahmed Abdelhadi, Ashwin Amanna, Johanna Dwyer, T. Charles Clancy, “Implementing an optimal rate allocation tuned to the user quality of experience”, in proceedings of IEEE ICNC 2015
- [13] Zhiruo Cao, E.W. Zegura, “Utility max-min: an application-oriented bandwidth allocation scheme”, in proceedings of IEEE INFOCOM 1999
- [14] M. S. Seddiki et al., “FlowQoS: QoS for the Rest of Us”, in Proceedings of ACM HOTSDN 2014
- [15] I. N. Bozkurt and Theophilus Benson, “Contextual Router: Advancing Experience Oriented Networking to the Home”, in Proceedings of ACM SOSR 2016
- [16] S. Alcock and R. Nelson, “Libprotoident: Traffic classification using lightweight packet inspection”, technical report, University of Waikato, 2012
- [17] Ryu. [Online]. Available: <https://osrg.github.io/ryu/>
- [18] OpenFlow, Open Network Foundation. [Online]. Available: <https://www.opennetworking.org>
- [19] OVSDB. [Online]. Available: <https://www.sdxcentral.com/projects/open-vswitch-database>
- [20] OpenvSwitch. [Online]. Available: <http://openvswitch.org/>
- [21] Raspberry Pi. [Online]. Available: <https://www.raspberrypi.org/>
- [22] H. Kim, J. Kim, Y.-B. Ko, "Developing a cost-effective openflow testbed for small-

scale software defined networking", Advanced Communication Technology (ICACT) 2014 16th International Conferenceon, pp. 758-761, 2014.

[23] V. Carela-Español, T. Bujlow, and P. Barlet-Ros. Is our ground-truth for traffic classification reliable? In M. Faloutsos and A. Kuzmanovic, editors, Passive and Active Measurement, volume 8362 of Lecture Notes in Computer Science, pages 98–108. Springer International Publishing, 2014

[24] W. Stallings, Data and Computer Communications, 9th ed. Pearson Custom Publishing, 2010

[25] IETF. <https://www.ietf.org/proceedings/52/slides/ieprep-1/tsld007.htm>

[26] H. Shi et al., “Fairness in Wireless Networks - Issues, Measures and Challenges”, IEEE Communications Surveys & Tutorials, vol. PP, pp. 1 – 20, 2013

[27] W.-H. Wang, M. Palaniswami, and S. H. Low, “Application-oriented flow control: Fundamentals, algorithms and fairness,” in proceedings of IEEE/ACM Transactions on Networking, vol. 14, no. 6, pp. 1282 –1291, December 2006

[28] T. Erpek et al., “An Optimal Application-Aware Resource Block Scheduling in LTE”, in proceedings of IEEE ICNC 2015

[29] A. AbdelHadi et al., “A Utility Proportional Fairness Approach for Resource Allocation of 4G-LTE”, IEEE ICNC Workshop CNC 2014

[30] Zaid Kbah et al., “Resource allocation in cellular systems for applications with random parameters”, in proceedings of IEEE ICNC 2016

[31] Iperf. [Online] Available: <https://iperf.fr/>

[32] Mininet. [Online] Available: <https://mininet.org/>

[33] sFlow. [Online] Available: <http://www.sflow.org/>