การออกแบบและการทำให้เกิดผลของโครงข่ายเมชไร้สายกลางแจ้งพิสัยปานกลางด้วยโอเพนโฟลวในราสเบอร์รีพาย

นายโซ ยีเต็ด

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมไฟฟ้า ภาควิชาวิศวกรรมไฟฟ้า
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2561
ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

DESIGN AND IMPLEMENTATION OF MEDIUM-RANGE OUTDOOR WIRELESS MESH NETWORK WITH OPENFLOW IN RASPBERRY PI

Mr. Soe Ye Htet

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering Program in Electrical Engineering
Department of Electrical Engineering
Faculty of Engineering
Chulalongkorn University
Academic Year 2018

Thesis Title	DESIGN AND IMPLEMENTATION OF MEDIUM-RANGE OUTDOOR WIRELESS MESH NETWORK WITH OPENFLOW IN RASPBERRY PI
By	Mr. Soe Ye Htet
Field of Study	Electrical Engineering
Thesis Advisor	Associate Professor Chaodit Aswakul, Ph.D

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

.....Dean of the Faculty of Engineering
(Associate Professor Supot Teachavorasinkun, D.Eng.)

THESIS COMMITTEE

.....Chairman
(Assistant Professor Chaiyachet Saivichit, Ph.D)

.....Thesis Advisor
(Associate Professor Chaodit Aswakul, Ph.D)

.....Examiner
(Associate Professor Kultida Rojviboonchai, Ph.D)

.....External Examiner
(Associate Professor Sinchai Kamolphiwong, Ph.D)

โซ ยีเต็ด : การออกแบบและการทำให้เกิดผลของโครงข่ายเมชไร้สายกลางแจ้งพิสัยปานกลางด้วยโอเพนโฟลวในราสเบอร์รีพาย (Design and Implementation of Medium-Range Outdoor Wireless Mesh Network with OpenFlow in Raspberry Pi) อ.ที่ปรึกษาวิทยานิพนธ์หลัก: ดร.เชาวน์ดิศ อัศวกุล, 124 หน้า.

วิทยานิพนธ์ฉบับนี้ได้นำเสนอการออกแบบ และพัฒนาระบบทดสอบโครงข่ายไร้สายแบบเมชที่กำหนดโดยซอฟต์แวร์ (SDWMN) ด้วยวิธีการควบคุมอินแบน (in-band) มาใช้กับระบบเฝ้าติดตามการจราจรบนถนนพญาไทในช่วงจากถนนพระราม 1 และถนนพระราม 4 โนดเมชไร้สายสำหรับระบบทดสอบ SDWMN แบบกลางแจ้งนี้ประกอบด้วยกล่องแบบกันน้ำ 6 กล่อง, บอร์ดราสเบอร์รีพาย 6 ตัว,กล้อง 6 ตัว,แบตเตอรีสำรอง 6 ก้อน และคอมพิวเตอร์ Intel NUC 2 เครื่อง โดยใช้มาตรฐานอินเทอร์เน็ตไร้สายแอดฮอก IEEE 802.11 เพื่อส่งภาพที่ถ่ายจากราสเบอร์รีพายเกตเวย์ 2ตัวถูกติดตั้งที่ป้อมตำรวจจราจรและโนดโครงข่ายไร้สายแบบเมช 2 ตัวถูกติดตั้งไว้ที่สะพานลอยข้ามถนนพญาไทแต่ละแห่งโดยมีระยะทางระหว่างเกตเวย์ทั้งสองตัวเท่ากับ 1100 เมตรและมีระยะทางเฉลี่ยระหว่างสะพานลอยที่อยู่ติดกันเท่ากับ 250 ถึง 350 เมตร สาระสำคัญของวิทยานิพนธ์ฉบับนี้สรุปไว้ดังต่อไปนี้

ส่วนแรกคือ การออกแบบและพัฒนาส่วนประกอบทั้งหมดเพื่อเตรียมพร้อมสำหรับการติดตั้ง SDWMN จริง โดยส่วนของซอฟต์แวร์นั้นรวมถึงการติดตั้งสวิทช์เสมือน OpenVswitch, ตัวควบคุม RYU, ไดรเวอร์สำหรับอินเทอร์เน็ตไร้สายภายนอกในโนดโครงข่ายไร้สายแบบเมช และการจัดเส้นทางสำหรับ SDWMN ภายนอก การติดตั้งใช้เคอร์เนลลินุกซ์เวอร์ชัน 4.4 ร่วมกับไดรเวอร์สำหรับสายอากาศที่ใช้ในวิทยานิพนธ์นี้ ในส่วนของฮาร์ดแวร์กล่องกันน้ำถูกออกแบบมาสำหรับติดตั้งบนสะพานลอยบนถนนพญาไท

เส้นทางหลักและเส้นทางรองถูกสร้างขึ้นโดยกฎการส่งต่อ ที่กำหนดไว้ล่วงหน้าบนเส้นทางที่จำนวนฮอพต่ำที่สุด เส้นทางหลักจะถูกติดตั้งตามกฎต่าง ๆ ในขั้นตอนการเริ่มเปิดใช้งานโนดไร้สายทั้งหมด และเมื่อโนดเมชไร้สายบ่างโนดหยุดทำงาน เส้นทางรองจะถูกสร้างขึ้นตามกฎการส่งต่อสำรองที่กำหนดไว้ล่วงหน้าด้วยการใช้ข้อความมาตรฐานในการร้องขอให้ กำหนดค่าของโอเพนโฟลว

จากการวัดสถรรถนะโครงข่ายพบว่ามีทราฟฟิกควบคุมโอเพนโฟลวประมาณ 12 กิโลบิตต่อวินาทีเมื่อโนดเมชไร้สายทุกตัวเชื่อมต่ออยู่กับตัวควบคุม RYU และประมาณ 20 กิโลบิตต่อวินาทีเมื่อโนดเมชไร้สายตัวหนึ่งถูกตัดการเชื่อมต่อจากตัวควบคุม RYU

กรณีศึกษาที่มีโนดเมชไร้สายหยุดทำงานได้ถูกจำลองขึ้นโดยการรีบูตโนดเมชไร้สายด้วยผู้ทดลองเอง จากผลลัพธ์ที่ได้นั้นเวลาที่มากที่สุดที่ต้องใช้เพื่อกำหนดเส้นทางใหม่เป็น 46 วินาที สำหรับระนาบควบคุม และ 30 วินาที สำหรับระนาบข้อมูล ค่าระยะเวลาในการฟื้นฟูเส้นทางสำหรับความล้มเหลวในแต่ละกรณี จะขึ้นอยู่กับตำแหน่งทางกายภาพจริงที่ใช้ติดตั้ง SDWMN ภายนอก และขึ้นกับว่าโนดใดหยุดทำงาน

สุดท้ายวิทยานิพนธ์นี้ได้ทดสอบการรวมแอพพลิเคชัน และการเฝ้าติดตามการจราจรบนถนนกับโครงข่าย SDWMN นี้ โดยได้ให้บริการแก่ตำรวจจราจรเป็นจำนวน 16 ชั่วโมงในวันที่ 26 พฤศจิกายน พ.ศ. 2561 และได้มีการตรวจสอบสถานะของระนาบควบคุมในระหว่างการใช้งานจริงของระบบ ในการทดสอบในช่วงฤดูหนาวของประเทศไทยและพบว่าที่อุณหภูมิของโนดเมชไร้สายต่ำกว่าอุณหภูมิสูงสุดที่สามารถใช้งานได้คือ 85 องศาเซลเซียส เป็นผลให้งานในอนาคตคือการทดสอบใน ฤดูร้อนซึ่งอาจจะนำไปสู่การทดสอบความไม่เสถียรของโนดที่ขึ้นกับอุณหภูมิที่สูงขึ้นและส่งผลต่อการศึกษาความน่าเชื่อถือของระบบ SDWMN โดยระบบทดสอบโครงข่ายที่ได้พัฒนาขึ้นนี้จะเป็นพื้นฐานสำหรับการตรวจสอบการใช้งานต่าง ๆ ในอนาคตของ SDWMN ขนาดใหญ่ในโครงข่ายการเฝ้าระวังการจราจรบนถนนต่อไป

ภาควิชา วิศวกรรมไฟฟ้า ..... **ลายมือชื่อนิสิต** .....

สาขาวิชา วิศวกรรมไฟฟ้า ..... **ลายมือชื่อ อ.ที่ปรึกษาหลัก** .....

ปีการศึกษา 2561 .....

5970475121 : MAJOR ELECTRICAL ENGINEERING
KEYWORDS: OPENFLOW/ SOFTWARE-DEFINED NETWORKING/ RASPBERRY PI/ WIRELESS MESH NETWORK.

SOE YE HTET : DESIGN AND IMPLEMENTATION OF MEDIUM-RANGE OUTDOOR WIRELESS MESH NETWORK WITH OPENFLOW IN RASPBERRY PI. ADVISOR: ASSOC. PROF. CHAODIT ASWAKUL, Ph.D, 124 pp.

This thesis has designed and implemented the prototype of software-defined wireless mesh network (SDWMN) testbed with in-band control approach for road traffic monitoring system on Phaya Thai road between Rama 1 and Rama 4 roads. Wireless mesh nodes for this outdoor SDWMN testbed are composed of 6 waterproof boxes, 6 Raspberry Pi's, 6 cameras, and 6 power banks and 2 Intel NUC computers. Ad-hoc based IEEE 802.11 WiFi standard is used to send the captured image from Raspberry. Two gateways are installed at the traffic police boxes and two wireless mesh nodes are installed at each crossover bridge on Phaya Thai road. The total distance between two gateways is 1100 meters. On Phaya Thai road, the average distance between adjacent crossover bridges is 250-350 meters. In summary, the main contributions of this thesis are as follows.

Firstly, we have designed and developed all components in preparation for the actual installation SDWMN testbed. The software parts include the installation of OpenVswitch, RYU, driver for external WiFi adapter in all wireless nodes and routing for outdoor SDWMN. Linux kernel version 4.4 has been used with the driver for applied antenna in this thesis. A waterproof box is designed for installation on the crossover bridges on Phaya Thai road.

The primary route and the alternative route are built by predefined forwarding rules based on minimum hop path. The primary route is installed by predefined forwarding rules at bootstrapping stage in all wireless nodes and the alternative route is established by predefined backup forwarding rules from RYU controller when one of the wireless mesh nodes is failed with the usage of standard OpenFlow configuration request message.

Based on our measurement of network performance, OpenFlow control traffic requires around 12 kbit/sec when all wireless mesh node are connected to RYU controller and requires at least 20 kbit/sec when one of the wireless mesh nodes is disconnected from RYU controller.

The failure of wireless mesh node is investigated by manually rebooting the wireless mesh node. From our results, the required largest time to reroute for is 46 seconds and for data plane is 30 seconds. The actual restoration time for individual failure cases depends on the actual physical location where outdoor SDWMN is installed and the nodes that fail

Finally, we have integrated the intended traffic monitoring application and SDWMN network. We have provided to traffic police for usages of traffic monitoring system for 16 hours on 26th November 2018 and the status of control plane during this practical operation of a traffic monitoring system is investigated. During network operation with traffic monitoring application in winter season of Thailand, the temperature of wireless mesh node is lower than the maximum operable temperature which is 85-degree Celsius. Testing in the warmer seasons is left as a future work together with the testing of resultant temperature-dependent node inoperability and SDWMN reliability. The current network testbed will be a baseline for those future implementation verification of large-scale SDWMN of road traffic monitoring network.

Department : Electrical Engineering **Student's Signature** .....
Field of Study : Electrical Engineering **Advisor's Signature** .....
Academic Year : 2018

Acknowledgements

First of all, I wholeheartedly appreciate my advisor, Assoc. Prof. Dr. Chaodit Aswakul, for giving me chance to have the positive learning environment as his advisee and for his valuable guidance during my days at the Chulalongkorn University. At the beginning of my journey at Chulalongkorn University, I was not familiar with the networking and programming scenario. However, my advisor is always patient on me in teaching about software-defined networking and programming from the basic to advance level. Without his caring on my work, the day that I have to successfully say goodbye to the Chulalongkorn University will never reach to my life. I would like to express my gratitude to Asian Scholarship Program which provides the financial support for studying Master Engineering Degree in the Chulalongkorn University and I would like to thank the financial support for equipment preparation costs from the Wireless Network and Future Internet Research Unit of Chulalongkorn University.

The weekly group NRG-ASEEAWAYY meeting helps me to improve the skill of presentation and the suggestions during the presentation from my advisor and the group members from Wireless Network and Future Internet Research Unit improve my logical thinking in research. Those suggestions are really valuable not only for this research work, but also for my life. I will always keep every single suggestion in my mind.

I do appreciate the committee members for my thesis examination for your valuable points.

My journey at Chulalongkorn University is not easy. However, the motivation which is given by my advisor guides me to finish my journey of Master Engineering Degree at Chulalongkorn University. Moreover, I would like to thanks my parents, my girlfriend and my friends for their encouragement in my study life at Chulalongkorn University.

Contents

List of Tables

Page

List of figures

Page

Chapter 1

Introduction

1.1 Research Motivation

Nowadays, road traffic congestion becomes a major problem for the people in their daily life as the traffic density has been increasing year by year. In order to reduce the road traffic congestion, researchers have tried to set up the wireless sensor network or wireless mesh network or other types of networks to build the road traffic monitoring system. Under the category of a road traffic monitoring system, we have focused to build up the outdoor wireless network along the road in this thesis to support the future road traffic monitoring application such as the application of real-time video streaming. The wireless outdoor network of this work needs to be cost-effective, scalable and reliable.

In this case, IEEE 802.11 standard (WiFi) has been chosen for this work in order to minimize the operating cost in sending video or other possible data within the implemented outdoor wireless network. There are two options which can be utilized to establish the wireless network based on IEEE 802.11 standard which is an ad-hoc mode and an infrastructure mode. In an infrastructure based network, a wireless access point (AP) is required to build the wireless network and the range of wireless network can be extended by adding more APs. In contrast, an ad-hoc mode allows two or more devices such as laptops to be directly connected to each other without relying on the wireless access points. Therefore, an ad-hoc based wireless network is easy to be set up and extended without any AP. A wireless mesh network (WMN) is a multi-hop ad-hoc network and it can be established by connecting wireless nodes or routers to each other with the radio standards such as WiFi [4], ZigBee [12]. Since WMN can also eliminate the cost to purchase APs in expanding the network range, WMN becomes a cost-effective solution to create an outdoor wireless network to be deployed along a target congested road.

However, there are also challenges in WMN. The advantage of requiring no infrastructure based AP means that there is no central point to manage a WMN. To enable the routing in WMN, there are three types of distributed mesh routing protocols, namely, (i) proactive routing protocol (ii) reactive routing protocol and (iii) hybrid routing protocol [3].

Those distributed mesh routing protocols have a relatively restrictive functionality and their distributive nature of protocol configurations are difficult to be managed efficiently. In order to overcome the current limitation of WMN, software-defined networking (SDN) introduces the feature of centralized network management by separating the control and data planes with the open standard protocol such as OpenFlow [1]. By using the SDN in WMN, the network becomes manageable, controllable and easy to be modified. SDN based routing algorithms are easy to be configured or modified by using an available high-level programming language to automate the forwarding function from the control plane. This thesis aims at bringing the functionalities of SDN into the wireless mesh network and testing the real outdoor software defined wireless mesh network (SDWMN) testbed along an actual road. In this work, SDWMN testbed needs to be low cost and a Raspberry Pi 3 model B+ [13] becomes an interestingly appropriate option as the wireless mesh node hardware because it is a cheap and sufficiently powerful device. We design and implement the system with the intention for a road traffic monitoring system in this thesis by using an SDWMN of Raspberry Pi nodes to relay the road traffic video data from all the nodes to the gateway nodes locating at the ending points of connected WMN topology.

1.2 Problem Statement

According to the intended future application in this work, the proposed SDWMN testbed needs to be implemented in the outdoor environment along the targeted congested road. However, the past implementation of SDWMN [35, 36, 37, 38, 39, 40, 41, 42] have been set up in the indoor or only by emulated environments. While insightful results getting from the emulated environment can be easily repeated in a laboratory, the result can be varied by the type of emulated environments and can much differ from that in the actual outdoor scenario.

In this work, we have proposed to implement a prototype for an outdoor medium-range SDWMN testbed to be tested in real scenarios along the road. There are challenges to deploy our intended network testbed. The major focuses in this work are weather conditions, network reachability, and performance. The weather challenges are concerned with the hot-humid country's temperature and rain. The high temperature can cause the Rasp-

berry Pi's to be overheating and can be burnt. A heavy rain can degrade the wireless signal strength such as decreasing RSSI (received signal strength indicator) value. For this reason, the valid equipment hardware components must be chosen carefully in designing the outdoor SDWMN network testbed. Finally, with the testbed constructed, we are interested in testing network reachability and performance. Those challenges become our motivation to investigate the actual achievable performance of SDWMN testbed and practical measurement result from the real testbed is expectedly valuable for improving the proposed SDWMN testbed design.

1.3 Objective

The main objective of this thesis is to design and implement the medium-range outdoor wireless mesh network with OpenFlow-enabled Raspberry Pi's for a road traffic monitoring system. The network design criteria include the proper ISM WiFi frequency band selection, preparation for seasonal weather challenges, selection of antenna type, and the measurement location for the testbed performance. From the real implementation, the test includes the measurement of network reachability, TCP throughput, packet loss ratio, a temperature of Raspberry Pi and latency based on ICMP packets.

1.4 Scope of Thesis

The scope of this research are as follows:

1. Design of the outdoor medium-range SDWMN to prepare for the actual deployment scenario challenges.

2. Development of a simple fault-tolerant multi-hop routing scenario for failure of wireless mesh node which is suitable to the proposed outdoor SDWMN testbed.

3. Implementation of the real outdoor SDWMN testbed along the selected road and test TCP throughput, latency, packet loss ratio and the network reachability specified here by the maximum per-hop distance between wireless nodes as well as the maximum number of hops that can sustain the desired path throughput.

Chapter 2

Background and Literature Review

2.1 Wireless Mesh Network

Based on the IEEE 802.11 standard, a wireless network can be configured in two ways. The first one is an infrastructure based network and the second one is an ad-hoc based network. In an infrastructure based network, the wireless access point (AP) is required to establish the wireless connection between the wireless nodes such as laptops and routers. The range of the infrastructure based network can be extended by adding the APs. On the other hand, the wireless nodes are able to directly connect to each other without APs in an ad-hoc based network. An ad-hoc based network is easy to set up and it is inexpensive in extending the range of the network because there is no need for the APs in extending the network coverage. WMN is the multi-hop ad-hoc network and the typical architecture of WMN is demonstrated in Figure 2.1.

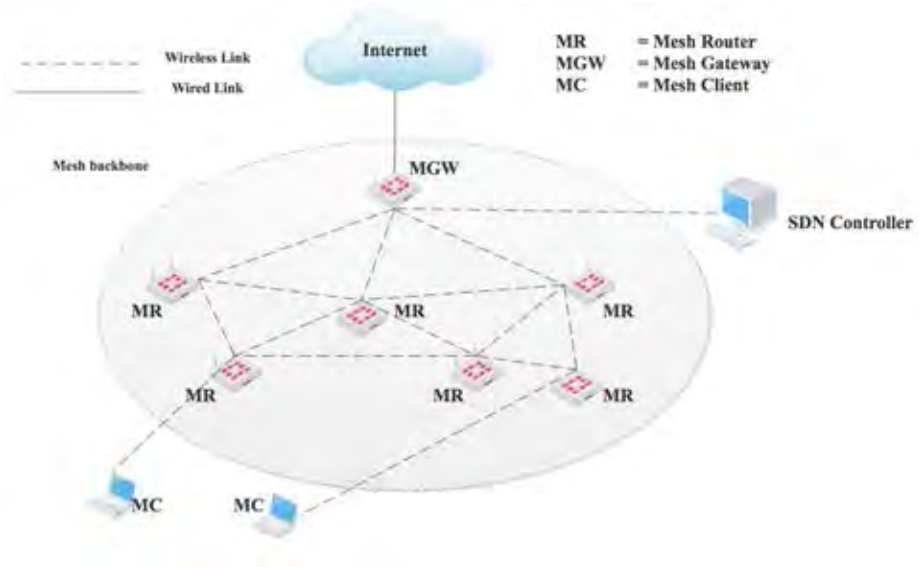

Figure 2.1: Typical architecture of WMN.

In Figure 2.1, WMN is composed of three kinds of wireless nodes which are mesh routers, mesh gateway and mesh clients. Since WMN is based on the ad-hoc mode, each mesh router and a mesh gateway can be directly connected to each other. Laptops, mobile phones can

be regarded as the mesh clients in this matter. A mesh gateway in Figure 2.1 is connected to the wired network for the internet connection. Various communication standards such as IEEE 802.15.4 (ZigBee), IEEE 802.11 (WiFi) can be applied to establish the WMN. There are a lot of traditional mesh routing protocols such as optimized link state routing protocol (OLSR) [28], ad-hoc on-demand distance vector (AODV) [29] implemented in the WMN for each mesh router to operate its responsibilities which are forwarding and routing the packets. The network must be designed by considering a topology robustness with self-healing characteristics as enabled by a proper mesh network routing. However, a conventional WMN with distributed routing protocols is generally difficult to be managed. This is because of the complex structure of wireless mesh topology and hence a manual configuration is often required upon significant network routing upgrades. Distributed routing also suffers from the lack of the network global view, and this could raise an issue on routing protocol efficiency. To cope with such difficulties, the feature of SDN is applied with an inherently enhanced network programmability. The detail of the SDN is explained in Section 2.2.

There are two types of topologies [25]:

1. Full mesh topology.

2. Partial mesh topology.

Figure 2.2 gives an example of the topology of a full WMN. If the wireless nodes such as mesh routers are fully interconnected to every other node, that topology is considered as the full wireless mesh topology [25]. Figure 2.3 gives another example of the topology of a partial WMN. If the wireless nodes are connected to each other but not to every other node, such a kind of topology is regarded as the partial mesh topology [25].

2.2 SDN

SDN [2] is regarded as the next generation of network architecture. In conventional networking, the control plane, data plane, and management plane are implemented in the hardware of forwarding elements. Here, the forwarding elements refer to the routers and switches. In SDN, the control plane and data plane are separated by an open standard protocol such as OpenFlow [1] and the control plane is implemented as the software in

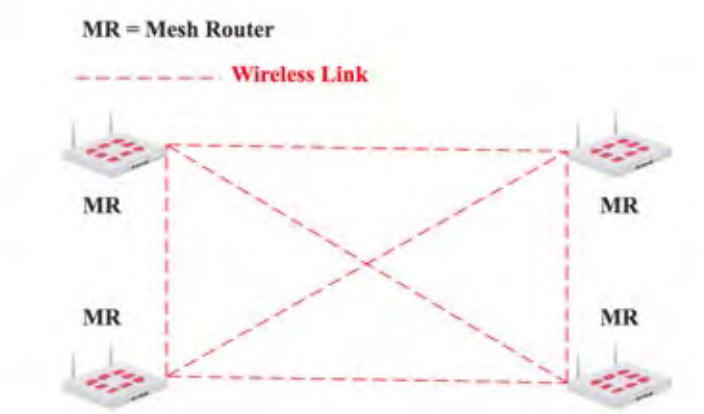

Figure 2.2: Typical topology of full WMN.

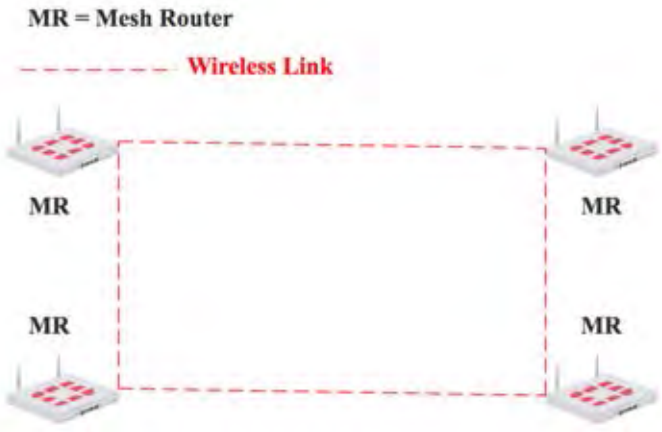

Figure 2.3: Typical topology of partial WMN.

a logically centralized SDN controller. Here, the centralized control plane is responsible for commanding SDN switches by giving instructions on how to send the packets and a data plane is responsible for forwarding incoming packets according to the instructions from the control plane. SDN switch is referred to as the OpenFlow-enabled forwarding elements and OpenFlow is an open standard protocol to enable the direct communication between the SDN controller and SDN switches. The intelligence of network is separated from the forwarding function and that the intelligence of network is located in the logically centralized SDN controller. In this matter, SDN controller does not need to be physically centralized but it must have the global view of the whole topology. Therefore, SDN can simplify the network operation. High-level programming languages such as Python, Java are allowed for the program development of software applications such as load balancing algorithm, routing algorithm to run on top of (or so-called at the northbound interface of) the SDN controller. Therefore, SDN network is easy to be managed and modified with those SDN applications which can be executed in the SDN controller. The principle of SDN is summarized in Figure 2.4.

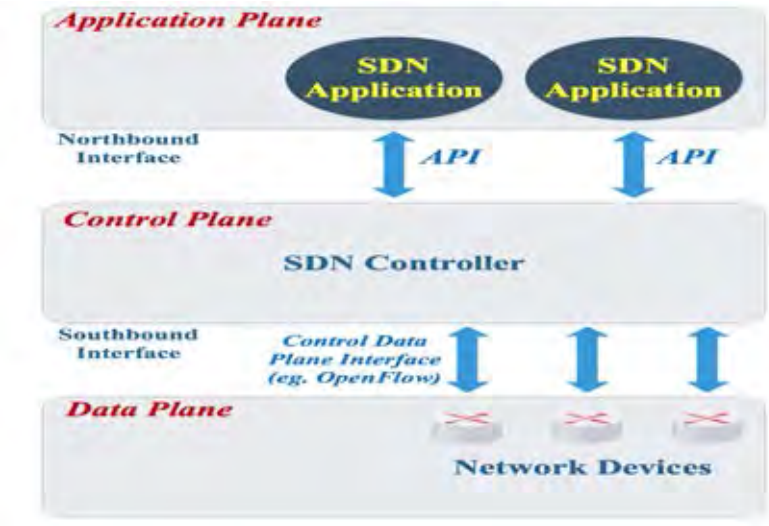

Figure 2.4: Principle of SDN [2].

Basically, SDN is composed of three layers.

1. Infrastructure Layer.

2. Control Layer.

3. Application Layer.

The lowest layer is the infrastructure layer which is also known as the forwarding layer and it is composed of SDN switches. The main difference between SDN switch and the traditional router is that SDN switch does not have its own routing logic. That means every SDN switch is not allowed to decide by itself how to forward each incoming packet. The SDN controller assigns the forwarding flow tables to SDN switches and SDN switches have to forward the packets according to the flow tables assigned by the SDN controller. The middle layer is the control layer and it is executed by the SDN controller. The upper layer is the application layer and it is composed of the SDN applications such as routing algorithm and load balancing. The interface between the control layer and the infrastructure layer is the southbound interface which is provided by the open standard interface such as OpenFlow [1]. SDN controller uses the southbound interface in order to install the forwarding rules into SDN switches. A northbound interface is located between the application layer and the control layer and that interface is used by the SDN application to run its service on the SDN controller through the application programming interface (API).

2.3 OpenFlow

OpenFlow [1] is the first open standard protocol that provides the southbound interface between the SDN controller and SDN switches. There are so far versions of OpenFlow protocols from version 1.0 to 1.5. OpenFlow version 1.0 is the default version and it is used in most SDN switches and SDN controllers. OpenFlow provides the communication interface for an SDN controller to instruct SDN switches on how to forward the packets by installing, deleting and modifying flow entries in SDN switches reactively or proactively. The flow table of SDN switch is a set of flow entries and flow entries are executed with match-action criteria. The main components of flow entry in a flow table is summarized in Table 2.1 from OpenFlow version 1.3, which will be used in this thesis.

Table 2.1: Main components of flow entry in a flow table [17].

Match Field	Priority	Counters	Instructions	Timeout	Cookie

Each flow table entry contains:

1. Match field : To filter the incoming packets by matching with the defined values in the matching fields.
2. Priority : The matching preference of table entries. When an SDN switch receives a packet, an incoming packet header is matched sequentially from the highest number of priority to the lowest number of priority.
3. Counters : The counter is used to count the number of the matched packet. The counter is updated with the number of matched packets.
4. Instructions : Action fields for each matched packet.
5. Cookie : Opaque data specified by the controller. Cookie value is used to filer the flow modification, flow statistics and flow deletion.

In the matching field, the ingress port and the specific packet header values are included. The ingress port is the port where the SDN switch receives the incoming packet. The specific packet headers in the matching fields are the destination MAC address, source MAC address, source IP address, destination IP address, TCP/UDP source port number, TCP/UDP destination port number, for example.

The options available in the action field of the flow table are used to instruct the SDN switches how they need to exactly handle the matched packet. Some available options in the action field and their functions are listed in Table 2.2 from OpenFlow version 1.3.

Table 2.2: Names of action fields of OpenFlow and their function [17].

Actions	**Function**
OUTPUT	Forward the packets on a specific port
DROP	Drop the packets
ALL	Forward packets out to all physical ports
CONTROLLER	Forward the packets to the controller as a packet in message
FLOOD	Forward packet out to all physical ports except to the ingress port
LOCAL	Forward the packets to the local port of the bridge
INPORT	Forward the packets to the ingress port

The architecture of OpenFlow switch compliant with OpenFlow version 1.3 is summarized in Figure 2.5. In this thesis, we use the OpenFlow version 1.3 because it can support

the features of multiple flow tables. The feature of multiple flow tables can provide the flexible OpenFlow switch pipeline. When the OpenFlow switch receives an incoming packet, that incoming packet is matched and processed in the operational precedence starting from the lowest number of tables e.g. table 0. The specific type of processing such as QoS, routing can be separately defined conveniently by dedicated flow tables such as table 1 for QoS, table 2 for routing. In this thesis, multiple flow tables are used for rewriting the packet header of the incoming packet, for relaying the incoming packet for multi-hop routing.

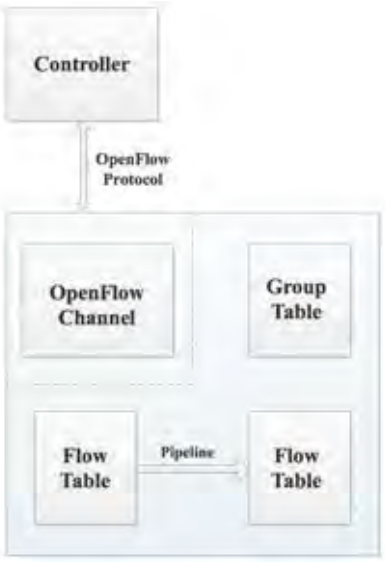

Figure 2.5: Architecture of OpenFlow switch compliant with OpenFlow version 1.3 [17].

2.4 WiFi Frequency Band Selection

We have considered to set up an outdoor SDWMN testbed based on 2.4 GHz and 5 GHz because those frequencies are unlicensed Industrial, Scientific and Medical (ISM) bands for WiFi. Therefore, the characteristics of 2.4 GHz ISM band and 5 GHz ISM band are mainly discussed in this subsection.

Under the standards of IEEE 802.11, there are generally five different IEEE 802.11 standards which are applied to 2.4 GHz ISM band and 5 GHz ISM band. Table 2.3 highlights the different frequency ranges of IEEE 802.11 standards.

The characteristics of the 2.4 GHz and 5 GHz ISM frequency bands are summarised as follows [45]:

Table 2.3: Wireless standards of IEEE 802.11.

IEEE 802.11 standard	ISM band
802.11b	2.4 GHz
802.11g	2.4 GHz
802.11a	5 GHz
802.11ac	5 GHz
802.11n	2.4/5 GHz

1. Channels 1 to 14 can be used (FCC allows only 11 channels) in 2.4 GHz ISM band. Among them, the maximum 3 non-overlapping channels can be applied [(1,6,11), (2,7,12), (3,8,13), (4,9,14), (5,10)]. In 5 GHz ISM band, the maximum of 23 non-overlapping channels can be applied. The number of non-overlapping channels in 2.4 GHz and 5 GHz ISM bands are based on 20 MHz channel bandwidth.

2. 2.4 GHz ISM band is widely used in Bluetooth, microwave oven, remote controller, cordless phone and this can possibly lead to the overcrowded situation. Since 23 non-overlapping channels can be applied in 5 GHz ISM band, there is less interference in 5 GHz ISM band.

3. Theoretically, 2.4 GHz can provide larger network coverage than network coverage that 5 GHz can because the higher the frequency, the shorter the range based on the same transmission power.

4. Based on 20 MHz channel width in 2.4 GHz and 5 GHz band, 5 GHz ISM band can provide more non-overlapping channels and a lower level of interference. There is also no overcrowded situation in 5 GHz band. Therefore, the performance of 5 GHz band is better than 2.4 GHz ISM band expectedly in general.

Selection of frequency range is one of the important factors to design the outdoor-based network to provide the stable and wide wireless connectivity. In this design, the location of the proposed outdoor SDMWN testbed will be located inside the urban area because the proposed testbed is intended for a road traffic monitoring system. If a proposed outdoor SDWMN testbed is implemented with 2.4 GHz ISM band, the Bluetooth devices from cars can potentially degrade a proposed outdoor SDWMN testbed. There is another point to be contemplated in the selection of frequency range. The allowable WiFi transmission power is

different according to the rules of the country. The maximum permitted WiFi transmission power in Thailand is listed in Table 2.4.

Table 2.4: Allowed WiFi transmission power in Thailand.

Frequency (GHz)	Allowed WiFi transmission power	Type of Network
2.400 - 2.500	0.1 W (EIRP)	Indoor/Outdoor
5.150 - 5.350	0.2 W (EIRP)	Indoor
5.470 - 5.725	1.0 W (EIRP)	Indoor/Outdoor
5.725 - 5.850	1.0 W (EIPR)	Indoor/Outdoor

Effective Isotropically Radiated Power or Equivalent Isotropic Radiated Power (EIRP) is equal to the output power of the transmitter minus cable loss plus antenna gain [27]. Note from Table 2.4 that there are higher maximum wireless transmission power allowed in 5 GHz band than in 2.4 GHz band. The wireless transmission power directly affects the network reachability. In summary, from all the reasonings aforementioned, this thesis proposes to utilize 5 GHz ISM band.

2.5 Type of Antenna

The type of WiFi antenna is another important factor for the outdoor network planning. There are two major types of WiFi antennas: (i) omnidirectional antenna and (ii) directional antenna. An omnidirectional antenna provides 360-degree horizontal radiation pattern. Therefore, the number of required omnidirectional antennas for a wireless node does not depend on the number of neighbor nodes due to its radiation pattern that can reach all the neighbor nodes in all surrounding directions. A directional antenna is used to provide the wireless signal in a specific direction. Since the transmission power is only needed to be consumed for the specific direction, the directional antenna can provide a longer per-hop distance based on the same EIRP transmission power allowed by law in comparison with the omnidirectional antenna. The number of required directional antenna is also based on the number of neighbor nodes for at least along a road the node must be able to relay both forward and backward to its neighbors. Therefore, the implementation cost of the network based on directional antennas will be higher than the network design which is based on omnidirectional antennas. In order the save the implementation cost, an omnidirectional

antenna is applied in the proposed outdoor medium-range SDWMN testbed.

2.6 Summary of Existing WMN Testbeds in Literature and Proposed SDWMN

This section provides a survey of how existing WMN testbeds have been implemented in the past and compares with the proposed SDWMN.

The small-scale Raspberry Pi based WMN testbed has been built in [30]. OpenWrt operating system is installed in each Raspberry Pi mesh node. The main purpose of the paper is in analyzing the performance of OLSR routing protocol in line-of-sight (LoS) and non-line-of-sight (NLoS) propagations in an indoor environment.

The outdoor campus WMN based on the 802.11b/g wireless interfaces are implemented on the roof of the university in [31]. Those wireless interfaces use fourth generation Atheros chipset based on AR5213 MAC/baseband. The target of this paper is to analyze the outdoor wireless link performance achievable by the 802.11b standard and the 802.11g standard.

Another outdoor campus WMN testbed has been established in [32]. In this case, Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N) [7] mesh routing protocol is used to set up the testbed. The contribution of this work is in investigating the performance of B.A.T.M.A.N routing protocol which is integrated at their proposed outdoor testbed.

The large-scale WMN testbed with static routing for road traffic control is proposed in [33]. The main objective of this work is to control the road traffic by handling the traffic light. The testbed is composed of 7 wireless mesh nodes each installed in a traffic light pole. Three different radios are applied (900 MHz, 2.4 GHz, 3.5 GHz) to construct the testbed. That testbed is located in the Sydney Central Bank Distinct which is an urban environment. The range between each mesh node is from 200 meters to 500 meters. Link characteristics of the testbed are analyzed in terms of latency.

QuRiNet has been set up in [34]. QuRiNet is the large-scale WMN testbed composed of 30 wireless mesh nodes. The testbed is located in the Quail Ridge Natural Reserve in California. The physical link distance between each wireless node is ranged from a hundred meters. The author has described the detail challenges to install QuRiNet in the outdoor

environment and explained how they have designed the QuRiNet to overcome the weather challenge, equipment challenge, site location challenge, and antenna selection challenge. Dynamic OLSR routing protocol is deployed to enable multi-hop routing for QuRiNet.

The proposed WMN testbeds [30, 32, 34] are fully distributed wireless networks and the intelligence of the networks is maintained in each mesh node. Routing in the WMN is achieved by the traditional routing protocols and those conventional routing protocols provide a different kind of behaviors. The behavior of traditional mesh routing protocols is difficult to be managed as the intelligence of the network is implemented distributively in each mesh node.

The first architecture of SDWMN has been proposed in [35] to overcome the limited functionality of the legacy routing protocol and the WMN has been managed from the logically centralized SDN controller. In-band SDN control mechanism is applied by using two separated virtual local area networks (VLANs) on the same network interface to set up the control channel and data channel, respectively. One VLAN is used to carry the control traffic and another VLAN is used to carry the data traffic. Traditional OLSR routing protocol is used for control traffic routing.

The traditional routing protocol is often used for data traffic in SDWMN as the backup plan in case of the centralized SDN controller is down. OLSR routing protocol has been applied in [36] for control traffic and data traffic if the SDN controller is not working in the proposed SDWMN framework in an NS3 [9]. The use case of [36] is the gateway balancing by implementing a round-robin gateway selection algorithm in the POX controller. Implementing traditional routing protocol inside the SDN based network may increase communication overhead.

The testbed of SDWMN based on Raspberry Pi is proposed in [37]. The testbed is set up in the laboratory with the WLAN APs and an ONOS (Open Network Operating System) controller [6]. Each AP is composed of three Raspberry Pi's for AP mode, station (STA) mode and Open Virtual Switch (OVS). An ONOS controller is connected to the AP via the wired network. The main purpose of the paper is in analyzing the performance of each AP.

A small-scale SDWMN is implemented in [38] inside the laboratory with 4 wireless

routers and an SDN controller. The out-of-band approach is applied by setting up the control plane with the wired network and the data plane with the wireless interfaces. The main objective of this paper is to propose the OpenFlow-based load balancing with the concept of the data flow path redirecting between links of the wireless mesh nodes.

The three-staged routing algorithm for SDWMN is proposed in [39]. The authors in [39] have used NS3 [9] and Mininet [16] to create the emulation environment for the testbed. For stage 1, an SDN controller tries to connect the switch with the basic routing by sending OF_Initial_Path_Request message. For stage 2, the switch sends OF_Inital_Path_Response message to the controller. From the message from stage 2, an SDN controller knows the information of neighbor nodes. For stage 3, an SDN controller installs the routing path based on the shortest path algorithm. In this paper, they focus on the connection between an SDN controller and switches and the connection between the switches.

Control overhead, CPU usage is investigated in an in-band control approach SDWMN testbed with full mesh and partial mesh topologies inside the building in [40]. Tun/Tap interface is applied to construct the in-band control. The topology in [40] is composed of 6 mesh routers and an SDN controller. OLSR routing protocol is used to construct the communication between mesh routers and an SDN controller.

IISTMeshNet testbed is implemented in [41] with two RYU controllers [19] and three mobile routers. ALIX.3d3 board [20] is used as a mobile router in the proposed testbed. In-band control approach is applied. The main purpose of this paper is to propose two dynamic multi-hop hand-off solutions based on the mobility of the OpenFlow-enabled routers. The first solution is the Round-Trip-Time (RTT) based hand-off scheme and the second solution is the Expected Transmission Count (ETX) based hand-off scheme.

Prediction-based link uncertainty solution in SD-WMN (PLUS-SW) is proposed in [42] to predict the link failure in the control plane and data plane on the case of node mobility by using the supervised learning model and to determine the optimal alternative route from the SDN controller. The simulation is done in the Network Simulation 3 (NS3) by building the network with 50 nodes in 300 m x 1500 m area.

Table 2.5 gives a comparative summary of the reviewed WMN and SDWMN systems. Thanks to the authors in [34] who have built the QuRiNet which is the outdoor large

WMN, the characteristic of the outdoor wireless network has been studied. A conventional outdoor WMN is set up in the urban environment in [33] for controlling road traffic. Indoor and emulated SDWMN testbeds are implemented in [35, 36, 37, 38, 39, 40, 41, 42]. The main difference between our work and those summarized works is that we will propose the real outdoor based SDWMN testbed along the road for traffic monitoring. OpenFlow based routing scenario is applied to establish a control plane and a data plane on a single physical interface. By using the features of SDN which we have discussed in Section 2.2 in building WMN testbed, our proposed SDWMN testbed is easy to be managed and modified. From the proposed outdoor SDWMN testbed, the investigation for the performance of real outdoor SDWMN testbed can be conducted.

Table 2.5: Summary of reviewed WMN testbeds and proposed SDWMN.

Papers	**OpenFlow Enabled**	**Type of Control**	**Testbed Environment**	**Routing Protocol**
[30]	No	Distributed	Indoor	OLSR
[31]	No	Distributed	Outdoor	802.11 b/g
[32]	No	Distributed	Outdoor	B.A.T.M.A.N
[33]	No	Distributed	Outdoor	Static
[34]	No	Distributed	Outdoor	OLSR
[35]	Yes	In-Band	Indoor	OLSR & SDN
[36]	Yes	In-Band	NS3	OLSR & SDN
[37]	Yes	Out-of-Band	Indoor	SDN
[38]	Yes	Out-of-Band	Indoor	SDN
[39]	Yes	Out-of-Band	NS3 & Mininet	SDN
[40]	Yes	In-Band	Indoor	OLSR & SDN
[41]	Yes	In-Band	Indoor	OLSR
[42]	Yes	In-Band	NS3	SDN
Proposed Work	Yes	In-Band	Outdoor	SDN

Chapter 3

Proof-of-concept Investigation of OpenFlow Based 2-hop Routing Scenario in Small-scale Preliminary Outdoor SDWMN Testbed on Phaya Thai Road [44]

Before we implement the final outdoor SDWMN testbed, we have implemented first the small-scale outdoor preliminary SDWMN testbed with two Raspberry Pi's and an Intel®NUC7i7BNH [14] in order to analyze the performance of OpenFlow based 2-hop routing. The preliminary testbed preparation in this research is concerned with the medium-range outdoor WMN based on OpenFlow-enabled Raspberry Pi, thanks to Raspberry Pi 3's cost-effectiveness and obtainable computational power within a compact form factor. The design intention is for studying the achievable link and path throughputs upon the medium achievable range of wireless connectivity based on off-the-shelf wireless ad-hoc link antenna. Our design is aimed at a potential future application towards a road network traffic monitoring, where each Raspberry Pi 3 serves both as a wireless signal relay as well as a sensor e.g. by attaching with a small camera to monitor road traffic conditions.

3.1 Design of Preliminary Small-scale Outdoor SDWMN Testbed

An example of future usage scenario of proposed outdoor SDWMN is depicted in Figure 3.1. The topology consists of 2 gateways and 6 mesh nodes. Each gateway is operated as a server to receive sensor data from mesh nodes and to push the data potentially into a data cloud. Here, a node running the gateway functionality can also run the SDN controller. Intel®NUC7i7BNH is used as a gateway and a Raspberry Pi 3 model B+ with Quad-Core CPU and 1-GByte RAM is used as a mesh node. Both the gateway and mesh nodes run the Ubuntu MATE operating system [15] version 16.04 (32 bit). In each gateway and mesh

node, a dual-band EDUP EP-AC1605 Wi-Fi USB adapter [24] with two omnidirectional antennas is installed. Since EDUP EP-AC1605 is a dual-band antenna, 2.4 GHz ISM band or 5.5 GHz ISM band can be selected. 5.5 GHz is applied in establishing the preliminary outdoor SDWMN testbed and the reason has been already discussed in Section 2.4. The high WiFi transmission power in 5.5 GHz according to Table 2.4 should allow a per-hop distance of at least between 100 meters and 500 meters, which is regarded here as the medium range in this design.

Figure 3.1: Typical installation scenario of outdoor SDWMN to monitor road network traffic (e.g. on Phaya Thai road, Bangkok).

A sample SDWMN testbed has been designed with a typical usage deployment area as exemplified by the Phaya Thai road segment in between Rama 1 road and Rama 4 road in Bangkok, for the convenience of future installation and testing preparation. Here, there are five crossover bridges which are the suitable locations to place, wherever possible, the Raspberry Pi's in order to avoid the line-of-sight obstacles such as trucks or buses which can block the wireless signal. The average distance between adjacent crossover bridges is around 250 meters. An exception is on the distance between Raspi 3 and Raspi 5 over 400 meters, which are not long enough to relay the signal from Raspi 3 to Raspi 5 directly. However, Raspi 4 can be placed on either side of the road in between Raspi 3 and Raspi 5.

Open Virtual Switch (OVS) [8] is installed in each mesh node and gateway to establish a connection between an SDN controller and the mesh nodes. OVS runs the standard OpenFlow protocol. RYU controller [19] is chosen in this thesis as the testbed's SDN controller. RYU controller is Python-based and can support up to OpenFlow version 1.5

with usage convenience features e.g. GUI, OpenStack support, REST API. RYU controller has been suggested as a good choice for small businesses and research applications [43].

For the implementation of the control plane for SDWMN, there is a challenge in that most of the mesh nodes cannot reach the gateway directly in one wireless hop. This is unlike the wired SDN, whereby a direct communication channel is easily dedicated to the establishment of a control interface. For SDWMN, there are two possible options to implement the control plane i.e. with in-band control and out-of-band control [26].

Out-of-band control requires the separately dedicated control network and data network. Since the outdoor SDWMN testbed is based on the IEEE 802.11 standard, at least two USB Wi-Fi adapters would be required in each of mesh nodes and gateways if the design is based on the out-of-band control approach [26]. An extra hardware cost for control network is reducible in the in-band SDN approach, whereby the control and data planes are implemented within the same physical interface at each node.

In the preliminary outdoor SDWMN testbed, therefore, the in-band SDN approach is used in order to reduce the hardware cost. The in-band control plane is here established by installing our properly defined forwarding rules to the OVS of each mesh node to relay address resolution protocol (ARP) packets and transmission control protocol (TCP) packets between the mesh node and the SDN controller. These forwarding rules have been pre-installed at the mesh node as a script that will be automatically executed every time that the mesh node is restarted.

From Figure 3.1, as a preliminary testing, a gateway (Gateway 1) has been installed at location 2, and two Raspberry Pi's (Raspi 1 and Raspi 2) have been installed at locations 3 and 4.

Figures 3.2 and 3.3 show the equipment of gateway and mesh node.

3.2 Implementation of OpenFlow Based 2-hop Simplified Routing

Generally, OVS can be configured with bridges and each bridge can consist of multiple ports [18]. Bridge in OVS is needed to be connected with an SDN controller in order to

Figure 3.2: Equipment of gateway in preliminary outdoor SDWMN testbed.

Figure 3.3: Equipment of mesh node in preliminary outdoor SDWMN testbed.

establish OpenFlow connection between SDN controller and OVS [18]. Here, a port in a bridge is regarded as OpenFlow port. OpenFlow port can be a logical port, a physical port, and a local reserved port [17]. The example of OpenFlow logical port is VLAN port. A physical port is a port that OVS define for a hardware interface such as wireless interface which is added to a bridge of OVS. LOCAL port represents local networking stack of the OVS and all network traffic coming to and from a bridge of OVS is required to pass through a LOCAL port.

The preliminary outdoor medium-range SDWMN testbed is shown in Figure 3.4.

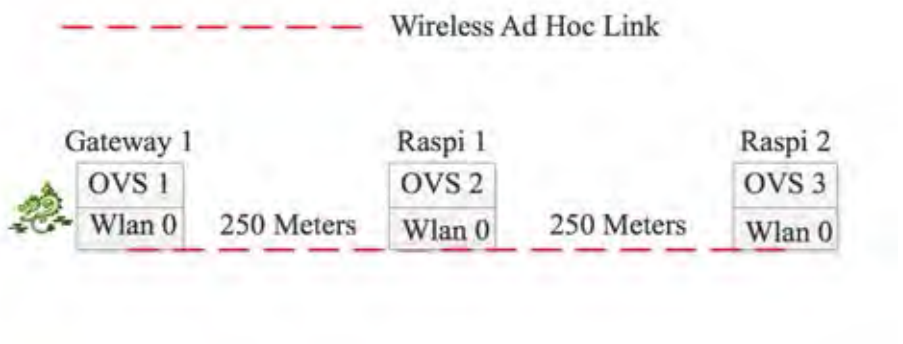

Figure 3.4: Architecture of preliminary outdoor medium-range SDWMN testbed.

In this work, a physical wireless network interface of Gateway 1, Raspi 1 and Raspi 2 is added to bridges of OVS 1, OVS 2 and OVS 3, respectively. OVS 2 in Raspi 1 enables the relay function of the wireless relay node between Gateway 1 and Raspi 2. All network traffic passing through those added physical wireless network interfaces is required to be handled by OVS 1, OVS 2 and OVS 3.

When an incoming packet arrives at the wireless network interface of a mesh node (Raspi 1 or Raspi 2), the mesh node will check a destination IP address of an incoming packet. If the destination IP address of an incoming packet is equal to the IP address of the mesh node, then that mesh node will respond to that incoming packet such as sending back reply packets (e.g. ARP reply, ICMP reply). If the destination IP address of an incoming packet is not equal to the IP address of the mesh node, then that mesh node forwards it to the next hop by rewriting the header of destination MAC address to the next hop's MAC address. There is a reason why destination MAC address needs to be modified into next

hop's MAC address to enable relay function. The problem of relaying the ARP packet in 2-hop simplified routing from Gateway 1 to Raspi 2 through Raspi 1 with the external USB WiFi Adapter in the preliminary outdoor SDWMN testbed is summarized in Figure 3.5.

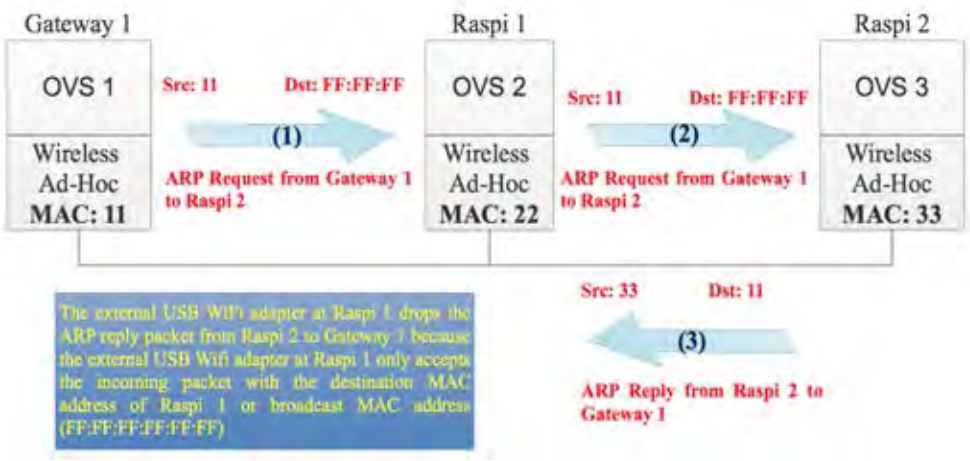

Figure 3.5: Problem of relaying ARP packet in 2-hop simplified routing from Gateway 1 to Raspi 2 with external USB WiFi adapter.

As shown in Figure 3.5, Gateway 1 sends the ARP request packet to Raspi 2 and the full message of ARP request packet is "Gateway 1's MAC address > FF:FF:FF:FF:FF:FF who has IP address (Raspi 2) tell IP address (Gateway 1)". The wireless network interface of Raspi 1 captures that packet and checks the destination IP address of that incoming packet.

In this example, the destination IP address of an arrival packet at the wireless interface of Raspi 1 is IP address of Raspi 2. Therefore, Raspi 1 forwards the arrival packet to Raspi 2. The wireless interface of Raspi 2 receives the ARP request packet from Gateway 1 and replies that ARP request packet by sending the ARP reply packet. The full message of ARP reply packet from Raspi 2 to Gateway 1 is "Raspi 2's MAC address" > Gateway 1's MAC address IP address (Raspi 2) is at MAC address of Raspi 2". The ARP reply packet from Raspi 2 to Gateway 1 is dropped by the wireless interface of Raspi 1 because the applied external USB WiFi adapter at the wireless interface of Raspi 1 only accepts an incoming packet with the destination MAC address being Raspi 1's MAC address or the broadcast MAC address FF:FF:FF:FF:FF:FF. Therefore, the wireless of Raspi 1 drops the ARP reply packet from Raspi 2 to Gateway 1. The problem of relaying an ARP packet in

this example is solved by rewriting the header of destination MAC address. The modified process of relaying the ARP packets in 2-hop routing from Gateway 1 to Raspi 2 with the external USB WiFi Adapter is summarized in Figure 3.6. We follow the same scenario not only for the ARP packet but also the other types of packets such as IP packet, TCP packet, UDP packet and so on.

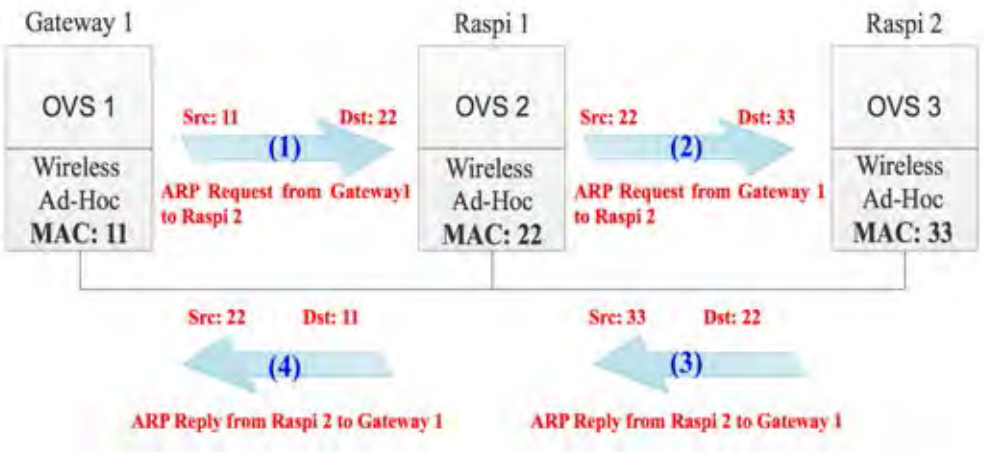

Figure 3.6: Relaying ARP packet in 2-hop simplified routing from Gateway 1 to Raspi 2 with external USB WiFi adapter by modifying destination MAC address.

In this work, we assume that each wireless mesh node and a gateway know each other's MAC address. The successful way of enabling the 2-hop routing scenario by modifying the destination MAC address with external USB WiFi Adapter is demonstrated in Figure 3.6 with the process of sending the ARP packets between Gateway 1 and Raspi 2.

Figure 3.7 shows the OVS forwarding rules which enable an in-band network in this preliminary testing. Figure 3.7(a) shows the forwarding rules of OVS 1, Figure 3.7(b) and 3.7(c) show the forwarding rules of OVS 2 and OVS 3. Among these figures, the forwarding rules in Figure 3.7(b) enable the relay function of a wireless relay node (Raspi 1) between Gateway 1 and Raspi 2. Here, the IP addresses of Gateway 1, Raspi 1 and 2 are 10.0.0.3, 10.0.0.1 and 10.0.0.2, respectively. Additionally, the MAC addresses of Gateway 1, Raspberry Pi's 1 and 2 are e8:4e:06:40:d3:4b, e8:4e:06:5f:47:59 and e8:4e:06:5e:6a:b1, respectively.

In OVS 1 of Gateway 1, flow table 0 is to check all packets arrive at the wireless network

```
gateway1@gateway1:~$ sudo ovs-ofctl dump-flows br0

cookie=0x0, duration=1915.557s, table=0,n_packets=1501, n_bytes=114247, idle_age=0,
priority=100,in_port=LOCAL actions=resubmit(,2)

cookie=0x0, duration=1915.539s, table=0,n_packets=41, n_bytes=5996,
idle_age=7,priority=50,in_port=1 actions=drop

cookie=0x0, duration=1915.546s, table=0,n_packets=985,n_bytes=108813,idle_age=5,
priority=80,ip,in_port=1,nw_dst=10.0.0.3 actions=LOCAL

cookie=0x0, duration=1915.543s, table=0,n_packets=223,n_bytes=9366,idle_age=0,
priority=80,arp,in_port=1,arp_tpa=10.0.0.3 actions=LOCAL

cookie=0x0, duration=1915.553s, table=2, n_packets=1501, n_bytes=114247, idle_age=0,
priority=100 actions=mod_dl_dst:e8:4e:06:5f:47:59,resubmit(,4)

 cookie=0x0, duration=1915.550s, table=4, n_packets=1501, n_bytes=114247, idle_age=0,
priority=100,in_port=LOCAL,dl_dst=e8:4e:06:5f:47:59 actions=output:1
```

(a)

```
admin2@admin2-desktop:~$ sudo ovs-ofctl dump-flows br0
cookie=0x0,duration=1499.804s,table=0,n_packets=81,n_bytes=3402,idle_age=7,priority=100,arp,in_port=
1,arp_tpa=10.0.0.2 actions=mod_dl_dst:e8:4e:06:5e:6a:b1,load:0->NXM_OF_IN_PORT[],resubmit(,2)

cookie=0x0,duration=1499.701s,table=0,n_packets=383,n_bytes=31300,idle_age=4,priority=80,ip,in_port=
1,nw_dst=10.0.0.2 actions=mod_dl_dst:e8:4e:06:5e:6a:b1,load:0->NXM_OF_IN_PORT[],resubmit(,2)

cookie=0x0,duration=1499.590s,table=0,n_packets=104,n_bytes=4368,idle_age=7,priority=70,arp,in_port=
1,arp_tpa=10.0.0.3 actions=mod_dl_dst:e8:4e:06:40:d3:4b,load:0->NXM_OF_IN_PORT[],resubmit(,4)

cookie=0x0,duration=1499.532s,table=0,n_packets=342,n_bytes=34770,idle_age=4,priority=70,ip,in_port=
1,nw_dst=10.0.0.3 actions=mod_dl_dst:e8:4e:06:40:d3:4b,load:0->NXM_OF_IN_PORT[],resubmit(,4)

cookie=0x0,duration=1499.415s,table=0,n_packets=82,n_bytes=3444,idle_age=71,priority=30,arp,in_port=
1,arp_tpa=10.0.0.1 actions=LOCAL

cookie=0x0,duration=1499.352s,table=0,n_packets=649,n_bytes=51831,idle_age=0,priority=30,ip,in_port=
1,nw_dst=10.0.0.1 actions=LOCAL

cookie=0x0,duration=1499.302s, table=0, n_packets=685, n_bytes=74103, idle_age=0,
priority=30,in_port=LOCAL actions=output:1

cookie=0x0, duration=1499.236s,table=0,n_packets=869,n_bytes=54920,idle_age=0,
priority=1,in_port=1actions=drop

cookie=0x0,duration=1499.647s,table=2,n_packets=464,n_bytes=34702,idle_age=4,priority=100,in_port=0
,dl_dst=e8:4e:06:5e:6a:b1 actions=output:1

cookie=0x0,duration=1499.471s,table=4,n_packets=440,n_bytes=38886,idle_age=4,priority=100,in_port=0
,dl_dst=e8:4e:06:40:d3:4b actions=output:1
```

(b)

```
soe1@soe1-desktop:~$ sudo ovs-ofctl dump-flows br0
cookie=0x0,duration=2057.859s,table=0,n_packets=722,n_bytes=65054,idle_age=0,priority
=100,in_port=LOCAL actions=resubmit(,2)

cookie=0x0,duration=2057.408s,table=0,n_packets=1147,n_bytes=71968,idle_age=1,priorit
y=10,in_port=1 actions=drop

cookie=0x0,duration=2057.580s,table=0,n_packets=542,n_bytes=43102,idle_age=0,priority
=80,ip,in_port=1,nw_dst=10.0.0.2 actions=LOCAL

cookie=0x0,duration=2057.494s,table=0,n_packets=112,n_bytes=4704,idle_age=11,priority
=80,arp,in_port=1,arp_tpa=10.0.0.2 actions=LOCAL

cookie=0x0,duration=2057.761s,table=2,n_packets=722,n_bytes=65054,idle_age=0,priority
=100,actions=mod_dl_dst:e8:4e:06:5f:47:59,resubmit(,4)

cookie=0x0,duration=2057.668s,table=4,n_packets=721,n_bytes=64919,idle_age=0,priority
=100,in_port=LOCAL,dl_dst=e8:4e:06:5f:47:59 actions=output:1
```

(c)

Figure 3.7: OVS forwarding rules of in-band controlled preliminary SDWMN at (a) OVS 1 (b) OVS 2 (wireless relay Node) (c) OVS 3.

interface of Gateway 1. In the flow table 0, two flow rules of “ip, in_port=1, nw_dst=10.0.0.3, actions=LOCAL” and “arp, in_port=1, arp_tpa=10.0.0.3, actions=LOCAL” will match ARP packets and IP packets with the destination IP address 10.0.0.3 (IP address of Gateway 1) arrive at the wireless network interface of Gateway 1. If the incoming ARP packets and IP packets are matched with those two flow rules, the matched packets are forwarded to the LOCAL port of a bridge in order to be responded by OVS 1. The flow rule in table 0 “in_port=LOCAL, actions=resubmit(,2)” submit all packets generated from OVS 3 to flow table 2. The flow rule with the lowest priority in flow table 0 “in_port=1, actions=drop” to drop the unmatched packet by the flow rules with higher priority in order to prevent a problem of infinite loop. In flow table 2, the flow rule “table=2, actions=mod_dl_dst: e8:4e:06:5f:47:59 (MAC address of Raspi 1)” set destination MAC address as e8:4e:06:5f:47:59 (Raspi 1’s MAC address) in all submitted packets from table 0. In flow table 4, the flow rule “table=4, in_port=LOCAL, dl_dst: e8:4e:06:5f:47:59 (Raspi 1’s MAC address), actions=output:1” is to forward the modified packet submitted by the flow table 2 to Raspi 1. The flow entries of OVS 3 in Raspi 2 are mostly equal with the flow entries of OVS 1 in Gateway 1.

The flow rules of OVS 2 in Raspi 1 are also composed of three flow tables which are table 0, table 1 and table 2 to enable the relay function. The main job of the flow table 0 in OVS 2 is to check all incoming packets at the wireless network interface of Raspi 1. The flow rules in OVS 2 distinguish a target of an incoming packet by checking a destination IP address of an incoming packet. In this configuration, there is only a single physical port in each OVS. In OVS, an ingress port number and an output port number needs to be different because OVS will drop a packet if a number of an ingress port and an output port are the same [18]. The definition of an ingress port and output port is discussed in Section 2.3. For example, if a flow entry is “in_port=1, actions=output:1”, then OVS will drop the packet which arrives at an ingress port number 1 even when we define the rules to forward the matched packets to an output port number 1.

The two flow rules in OVS 2 “arp, in_port=1, arp_tpa=“10.0.0.2” (Raspi 2’s IP address), actions=mod_dl_dst: e8:4e:06:5e:6a:b1, load:0>NXM_OF_IN_PORT[ ], resubmit(,2)” and “ip, in_port=1, nw_dst=“10.0.0.2” (Raspi 2’s IP address), actions=mod_dl_dst:

e8:4e:06:5e:6a:b1, load:0>NXM_OF_IN_PORT[ ], resubmit(,2)" match incoming ARP packets and IP packets with destination IP address 10.0.0.2 (Raspi 2's IP address) at the wireless network interface of Raspi 1. If an incoming packet is matched with those two flow rules, then the destination MAC address of that incoming packet is modified into destination MAC address of Raspi 2. Then, number of ingress port is changed to be different from the output port number and submit the modified packet into flow table 2. Likewise, the two flow rules in OVS 2 "arp, in_port=1, arp_tpa="10.0.0.3" (Gateway 1's IP address), actions=mod_dl_dst: e8:4e:06:40:d3:4b, load:0>NXM_OF_IN_PORT[ ], resubmit(,4)" and "ip, in_port=1, nw_dst="10.0.0.3" (Gateway 1's IP address), actions=mod_dl_dst:e8:4e:06:40:d3:4b, load:0>NXM_OF_IN_PORT[ ], resubmit(,4)" match incoming ARP packets and IP packets with destination IP address 10.0.0.3 (Gateway 1's IP address) at the wireless network interface of Raspi 1. If an incoming ARP packet or incoming IP packet is matched with those two rules, then destination MAC address of that incoming packet is modified into destination MAC address of Gateway 1. Then, ingress port number is changed to be different from the output port number and submits the modified packet into flow table 4. Another two flow rules in OVS 2 "arp, in_port=1, arp_tpa="10.0.0.1", actions=LOCAL" and "ip, in_port=1, nw_dst="10.0.0.1", actions=LOCAL" match an incoming IP packet and ARP packet with the destination IP address 10.0.0.1 (Raspi 1's IP address). The incoming IP packets and ARP packets with destination IP address 10.0.0.1 at the wireless network interface of Raspi 1 are forwarded to LOCAL port in order to be responded by OVS 2. There is an also drop action at the lowest flow entries in OVS 2 in order to prevent the case of an infinite loop. The flow entry in table 2 is to forward a modified packet submitted from flow table 0 to Raspi 2 and the flow entry in table 4 to forward a modified packet submitted from flow table 0 to Gateway 1.

The flow rules for ARP packet and IP packet in each OVS of Gateway 1, Raspi 1 and Raspi 2 enable in-band communication by forwarding the control packets and data packets over a single physical wireless interface. Control packets are TCP packets from mesh nodes (Raspi 1 and Raspi 2) to a specified port number of Gateway 1 which RYU controller uses. Here, port number 6633 of Gateway 1 is used by RYU controller to set up the control channel. Data packets in the preliminary outdoor SDWMN testbed are TCP

packets generated by iperf [21] to measure TCP throughput from mesh nodes to another port number of Gateway 1 which is not used by RYU controller and ICMP packets generated by ping program from mesh nodes to Gateway 1 in order to measure round-trip-time (RTT).

3.3 Measurement Result of Preliminary SDWMN Testbed's Performance

Iperf software has been used to measure the TCP throughput of the wireless route with 1 and 2 hops. Table 3.1 reports the results of TCP iperf from 3 runs, each with the measurement period of 100 seconds. In order to measure the average RTT, a ping program is used. Here, 200 ICMP packets have been generated for each run and the average RTT value from each run is shown in Table 3.2.

Tables 3.1 and 3.2 confirm that the OVS forwarding rules of in-band control can provide effectively the necessary control plane for the implementation of data packet forwarding. In addition, based on the expected target usage scenario of a road network traffic monitoring, the reported round-trip time results confirm that the current network settings can be applied beneficially in the future large-scale road traffic monitoring system. In practice, even for an automatic control of traffic signal light based on the wireless sensor network, a latency as high as a second should be acceptable. Future investigations are, however, needed to thoroughly confirm the usability of this SDWMN for that practical application.

Table 3.1: Average TCP throughput in preliminary outdoor SDWMN testbed (Mbit/sec).

	Average Throughput for 1 hop	**Average Throughput for 2 hop**
Test 1	10.9	7.1
Test 2	9.1	2.8
Test 3	10.7	2
Total Average Throughput	10.2	3.9

In this section, we have designed and implemented the preliminary outdoor SDWMN testbed for measuring the network throughput and round-trip time performance of an outdoor medium-range SDWMN testbed. In-band control approach has been implemented

Table 3.2: Average RTT in preliminary outdoor SDWMN testbed (ms).

	Average RTT for 1 hop	**Average RTT for 2 hops**
Test 1	13.318	45.84
Test 2	8.954	46.784
Test 3	7.017	38.077
Total Average RTT	9.763	43.567

successfully to save the extra hardware cost. The network covers only up to 2 hops simplified routing with 1 gateway. Our ongoing work is to increase the number of mesh nodes in between the two gateways, where OpenFlow-based routing adaptation needs to be verified. To enable the final SDWMN testbed robustness, a 24-hour test span is in the plan, whereby the effect of other environmental conditions (e.g. ambient temperature, time of day) can be further studied.

Chapter 4

Proposed Fault-Tolerant Multi-hop Routed SDWMN with Node Failure

In this chapter, the criteria for the implementation of the final proposed outdoor medium-range SDWMN testbed along the road with multi-hop routing scenario is described. Firstly, the design criteria of the proposed outdoor SDWMN testbed is the same as what we have implemented in the preliminary outdoor SDWMN testbed in Chapter 3. The main difference is that the number of wireless components will be increased and the extended plan to resist the seasonal challenges be added in the final proposed outdoor SDWMN testbed. The number of wireless mesh nodes is increased to six wireless mesh nodes of Raspberry Pi's and two gateways which are the Intel®NUC7i7BNH computers.

The proposed topology of the outdoor medium-range SDWMN testbed for road traffic monitoring over a bi-directional road of car lanes between two intersections is illustrated in Figure 4.1.

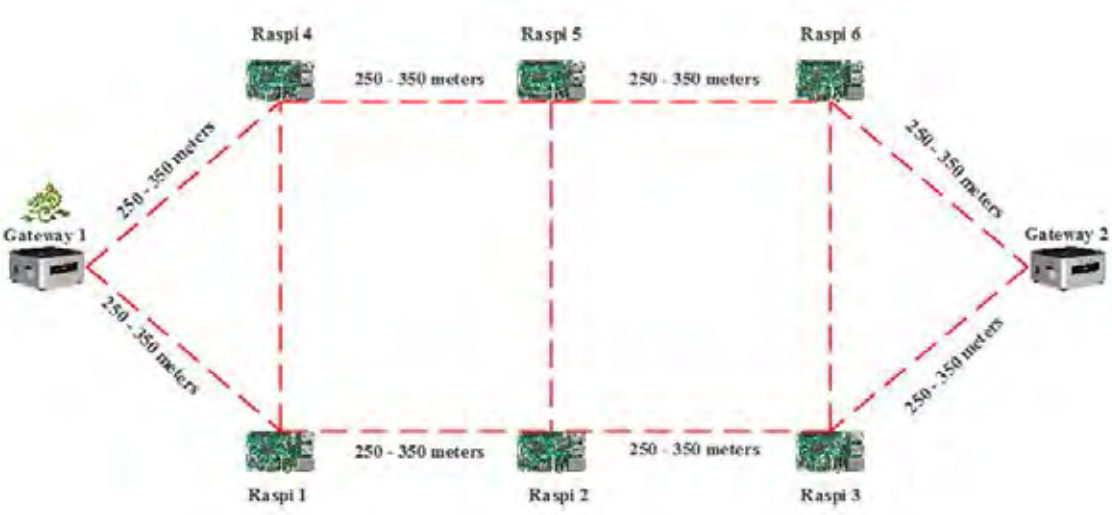

Figure 4.1: Topology of proposed outdoor SDWMN testbed for road traffic monitoring over bi-directional road of car lanes between two intersections.

4.1 Installation Preparation on Phaya Thai Road

The final outdoor SDWMN testbed is set up with 2 gateways and 6 wireless mesh nodes on Phaya Thai road segment between Rama 1 road and Rama 4 road in Bangkok. The total distance between two gateways is around 1100 meters. Each gateway is placed at the traffic police box. The topology of final outdoor SDWMN testbed in summarized in Figure 4.1. Between two traffic police boxes, there are three crossover bridges which are high enough to install the box for wireless mesh node and the average distance between each crossover bridge is 250 meters. RYU controller is installed at Gateway1.

The function of the wireless mesh node and the gateway are also the same as what we have discussed in the previous chapter. To recall their function, there are two main jobs for a wireless mesh node. The first job is to relay packets (control packet and data packet) which come from the other wireless mesh nodes to Gateway 1 or Gateway 2. The second job is to send data packets from the sensor such as the images from the attached camera of that wireless mesh node. The main job for a gateway is to receive the packets continuously from the wireless mesh nodes. The software tools for implementation of the proposed outdoor SDWMN testbed will be the same as that we have applied to implement the preliminary outdoor SDWMN testbed. The names of the software tools and the function of each software tool are recalled in Table 4.1.

Table 4.1: Software tools for implementing proposed outdoor SDWMN testbed.

Software	Function
Open vSwitch	Virtual OpenFlow Switch
RYU	SDN Controller Application
Ubuntu Mate	Linux Operating System
Ubuntu	Linux Operating System
Iperf	TCP/UDP Throughput Measurement Tool

We have described our experience in preparation from the software part for the final outdoor proposed SDWMN testbed. Installation the application of RYU controller, Open-Vswitch does not give any problems for us. However, the installation of the driver of EDUP EP-AC1605 into Linux devices gives a problem for us because the original driver which is supported by the company does not support for the Linux kernel version of 4.4. The modified driver version of EDUP EP-AC1605 can be downloaded from GitHub and that

modified driver version can be installed in Linux kernel version of 4.4. As Linux kernel version of 4.4, modified driver version of EDUP EP-AC1605 can be installed. However, Linux kernel version of Intel NUC device is 4.13 and we install a driver for EDUP EP-AC1605 antenna which can support Linux kernel version of 4.13. However, the problem we have faced is that shutting downtime for Intel NUC take at least 25 minutes when an external antenna is attached at Intel NUC. Therefore we downgrade the Linux kernel version from 4.13 to 4.4 at Intel NUC and that problem is solved. Based on our experience, Linux kernel version plays an important role to install a driver of an external antenna for a Linux-based operating system.

Another important thing in this work is the plan for the proposed outdoor SDWMN testbed to overcome the seasonal challenges, especially for the rainy season. Each gateway will be placed inside the traffic police box and therefore the gateways do not need to be waterproofed. A waterproof enclosure with an IP67 standard which needs to be wide enough to put a wireless mesh node and a power bank with 30000 mAh to feed the power to a wireless mesh node must be installed.

4.2 Implementation of Multi-hop Routing

After discussing the design criteria for the proposed outdoor SDWMN testbed, the detail procedures to implement the multi-hop routing scenario for the proposed outdoor SDWMN testbed are described.

Firstly, the detailed architecture of the proposed outdoor SDWMN testbed over a bi-directional road of car lanes between two intersections is summarized in Figure 4.2.

OVS 1,2,3,4,5,6 have been installed in the wireless mesh nodes and OVS 7,8 have been installed in the two gateways. A wireless interface of the wireless mesh nodes and the gateways is added to the OVS. The packet forwarding behavior of a wireless mesh node is summarized as a flowchart in Figure 4.3.

The application for routing running at the northbound interface of RYU controller must assign necessary OpenFlow forwarding rules to the wireless mesh nodes and the gateways in order to enable the in-band multi-hop routing. The control plane has been implemented in the same way that we have implemented at the preliminary outdoor SDWMN testbed

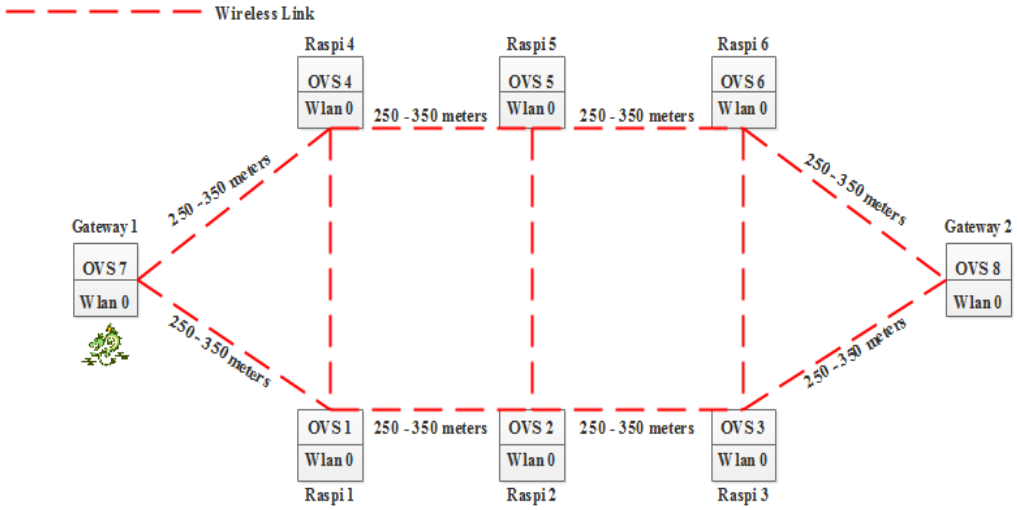

Figure 4.2: Architecture of proposed outdoor medium-range SDWMN testbed.

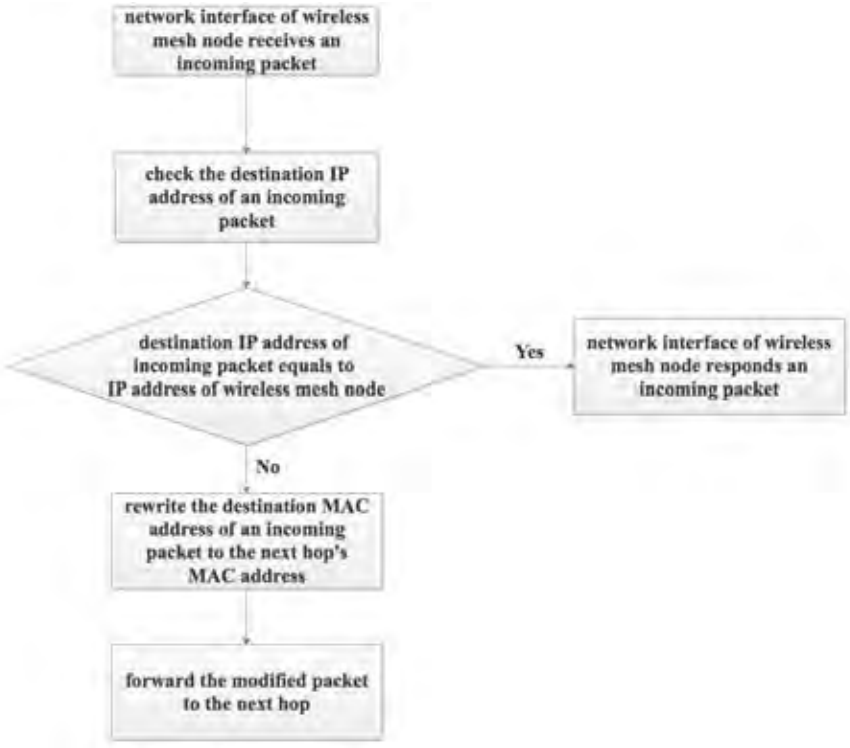

Figure 4.3: Forwarding procedure of wireless mesh node.

by installing pre-existing rules at each OVS of six wireless mesh nodes and two gateways.

Figure 4.4 illustrates the proposed outdoor SDWMN testbed with the primary routes for the control plane.

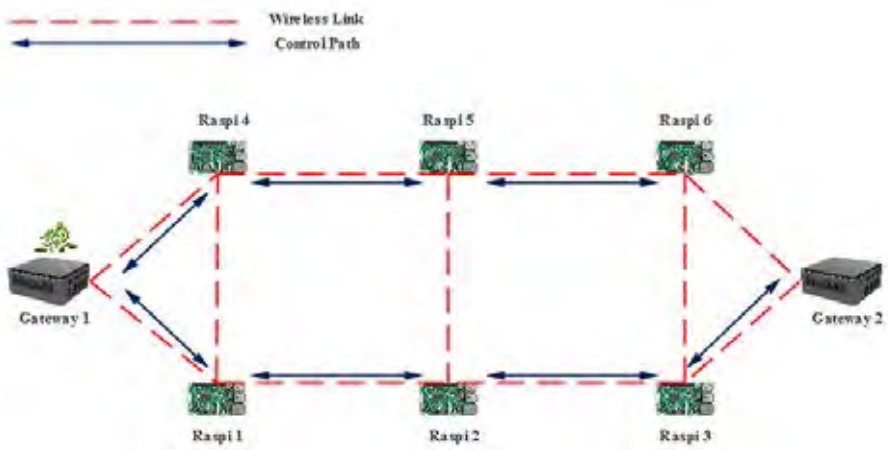

Figure 4.4: Illustration of routing for control plane in proposed outdoor SD-WMN testbed.

RYU controller is installed in Gateway 1. The routing information of the control plane in the proposed outdoor medium-range SDWMN testbed is described in Table 4.2.

Table 4.2: Routing information for control plane of proposed outdoor SDWMN testbed.

Routing path for control plane	Primary Route
Between RYU and Raspi 1	Raspi 1 - Gateway 1
Between RYU and Raspi 2	Raspi 2 - Raspi 1 - Gateway 1
Between RYU and Raspi 3	Raspi 3 - Raspi 2 - Raspi 1 - Gateway 1
Between RYU and Raspi 4	Raspi 4 - Gateway 1
Between RYU and Raspi 5	Raspi 5 - Raspi 4 - Gateway 1
Between RYU and Raspi 6	Raspi 6 - Raspi 5 - Raspi 4 - Gateway 1
Between RYU and Gateway 2	Gateway 2 - Raspi 3 - Raspi 2 - Raspi 1 - Gateway 1

In the proposed outdoor SDWMN testbed, the wireless mesh nodes will send the data packets such as a captured image from the attached camera in a Raspberry Pi continuously

to Gateway 1 and Gateway 2. Therefore the destination IP address of the data packets from each wireless mesh node is the IP address of Gateway 1 or IP address of Gateway 2. There are two gateways and six Raspberry Pi's in the proposed outdoor SDWMN testbed. Sending the data packets from the Raspberry Pi to only one gateway can lead to the traffic congestion at that gateway and therefore we separate nodes into two groups where each group includes one gateway and three wireless mesh nodes. For instance, Raspi 1, Raspi 2, Raspi 4, Gateway 1 is in one group and Raspi 3, Raspi 5, Raspi 6, Gateway 2 is in the other group. Each wireless mesh node sends the data to the nearest gateway. The shortest path between each wireless mesh node and gateway is not needed to be calculated by RYU controller as the distance between gateways and wireless mesh nodes have been already known. The routing principle for the data plane for the proposed outdoor SDWMN is summarized in Figure 4.5. The routing information of the primary route for data plane is summarized in Table 4.3.

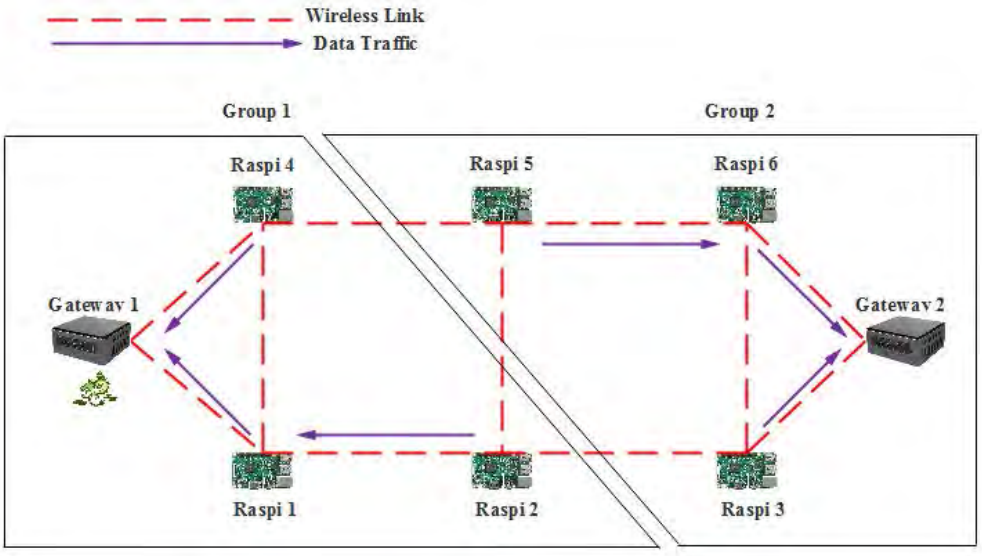

Figure 4.5: Illustration of routing for data plane in proposed outdoor SDWMN testbed.

The proposed outdoor SDWMN along the road is a static network and the location of the wireless mesh nodes and the gateways will be at the fixed location. Therefore, the case of the wireless mesh node mobility and the mobility-influenced changes of the topology will not be considered in this thesis.

Table 4.3: Routing information for data plane of proposed outdoor SDWMN testbed.

Switch ID	Primary route for data plane
Raspi 1	Raspi 1 - Gateway 1
Raspi 2	Raspi 2 - Raspi 1 - Gateway 1
Raspi 3	Raspi 3 - Gateway 2
Raspi 4	Raspi 4 - Gateway 1
Raspi 5	Raspi 5 - Raspi 6 -Gateway 2
Raspi 6	Raspi 6 - Gateway 2

4.3 Implementation of Restoration Mechanisms Upon Failure Scenario of Wireless Mesh Node

In this thesis, we also consider the simple necessary restoration mechanisms based on the failure of the wireless mesh node. The process of OpenFlow based rerouting based on the failure of a wireless mesh node from RYU controller contains:

1. The mechanism for RYU controller to detect the failure of the wireless mesh node.
2. The rerouting program which is implemented at the northbound interface of RYU controller will assign the OpenFlow forwarding rules reactively to the functioning wireless mesh nodes to establish alternative routes.
3. When the failed wireless mesh node is recovered back, RYU controller will assign back the primary route.

The mechanism for RYU controller to detect the failure of the wireless mesh node based on the messages of echo request and echo reply. Echo message is used to exchange the information of latency, bandwidth, liveness and echo request/reply message can be sent from either from an SDN controller or from an OpenFlow switch [17]. The exchange of echo request/reply message between an SDN controller and an OpenFlow switch is summarized in Figure 4.6.

In detecting the failure of a wireless mesh node, RYU controller sends the echo request message to a wireless mesh node to detect the liveness between RYU controller and wireless mesh nodes. The meaning of liveness between RYU controller and wireless mesh nodes is the active connection status between RYU controller and wireless mesh nodes. If wireless mesh

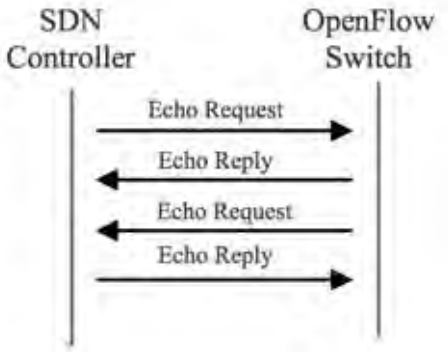

Figure 4.6: Echo message exchange between SDN controller and OpenFlow switch [17].

nodes cannot reply to the echo request message from RYU controller, then RYU controller will decide that the connection between RYU controller and the unreplying wireless mesh node is failed.

The rerouting program for the SDWMN network is implemented at the application layer of RYU controller to reroute the wireless mesh node when one of the wireless mesh nodes inside the SDWMN is failed. For rerouting purpose, RYU controller uses a set configuration request messages in order to install the necessary forwarding rules to build the alternative route. An example of rerouting scenario based on three wireless mesh nodes and one gateway is illustrated in Figure 4.7.

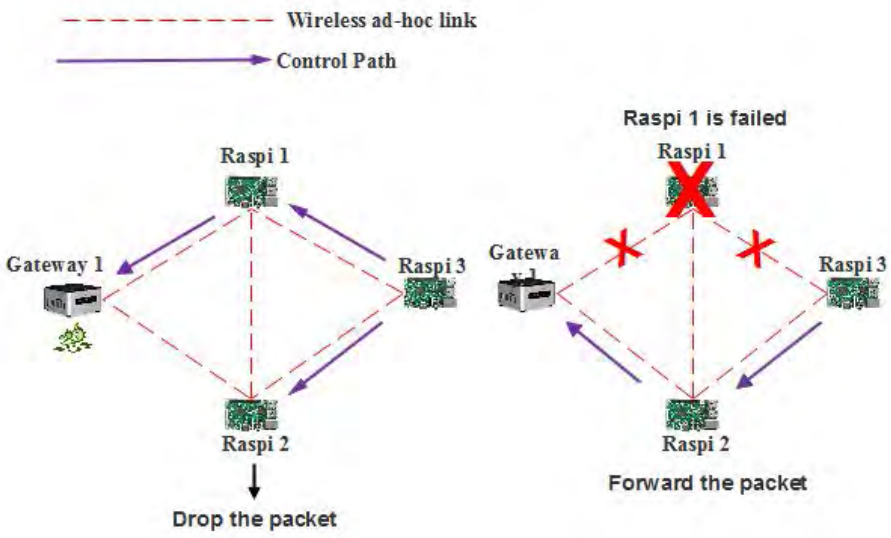

Figure 4.7: Example of node failure rerouting based on three wireless mesh nodes and one gateway.

Let Raspi 3, Raspi 2 and Raspi 1 represent the wireless mesh nodes and Gateway 1 represents a gateway. In this example, we consider the case of routing between Raspi 3 and Gateway 1. There are two possible primary routes between Raspi 3 and Gateway 1 which are Raspi 3 - Raspi 1 - Gateway 1 and Raspi 3 - Raspi 2 - Gateway 1.

In this example, the primary route is Raspi 3 - Raspi 1 - Gateway 1 for Raspi 3 and the alternative route for Raspi 3 is Raspi 3 - Raspi 2 - Gateway 1 when Raspi 1 is failed. The key idea behind the rerouting scenario is a configuration request message and hard_timeout. In order to establish the primary route in this example, the default forwarding rules at Raspi 1 to relay the control packet from Raspi 3 to Gateway 1 and assign the default forwarding rules at Raspi 2 to drop the control packet from Raspi 3 to Gateway 1. When all wireless mesh nodes such as Raspi 3, Raspi 2 and Raspi 1 in this example are connected with the RYU controller, RYU controller will keep silent without sending a configuration request message to the connected wireless mesh node. When Raspi 1 is failed, RYU controller sends a configuration request message to currently connected switch such as Raspi 2 in this example and assign the backup forwarding rules at Raspi 2 to forward the control packet from Raspi 3 to Gateway 1 with a specified amount of hard_timeout. Those backup forwarding rules need to be a higher priority than that of default forwarding rules. Due to the temporarily assigned forwarding rules at Raspi 2, Raspi 2 forwards the control packet from Raspi 3 to Gateway 1 to build the alternative route. When Raspi 1 is back to the operational stage, RYU controller will stop sending the configuration request message.

We have explained the example of rerouting scenario and the more detailed rerouting scenario for the proposed outdoor SDWMN testbed is discussed here. The forwarding rules for the primary route are installed at the bootstrapping stage of each wireless nodes and therefore, the primary route is established whenever wireless nodes including all wireless mesh nodes and two gateways are turned on.

The predefined primary routes let wireless nodes to send OFPT_HELLO messages and RYU controller will respond that OFPT_HELLO message to wireless mesh node when RYU controller receives that packet. After OFPT_HELLO message has been exchanged successfully between wireless mesh node and RYU controller, RYU controller decides that the connection between wireless mesh node and RYU controller has been established. The

scenario of exchanging the OFPT_HELLO messages between wireless mesh node and RYU controller is illustrated in Figure 4.8.

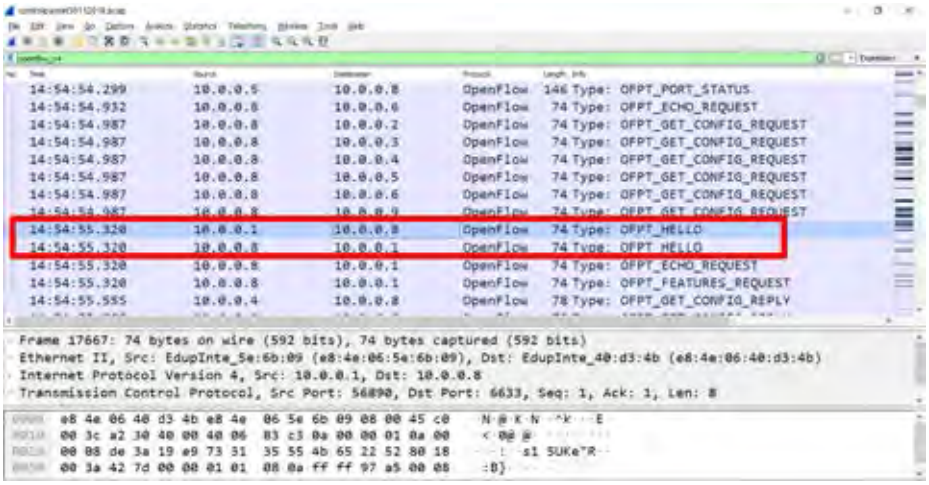

Figure 4.8: Illustration of exchanging OFPT_HELLO messages between wireless mesh node and RYU controller.

Once a wireless node has been connected with RYU controller, RYU controller puts that connected wireless nodes to the set of all nodes reachable by RYU controller. RYU controller keeps monitoring the connectivity status with wireless nodes by using echo request and echo reply message. In the current configuration, RYU controller sends an echo request message to all connected wireless nodes every 3 seconds. If wireless mesh nodes cannot reply to the echo request message from RYU controller for 4 retrial times, then RYU controller will decide that the connection between RYU controller and the unreplied wireless mesh node is failed or unreachable. The timeout for echo request interval is 5 seconds. Since echo request message will be sent to a wireless mesh node for every 3 seconds and therefore the fourth echo request message will be sent after 12 seconds of the first echo request message. The fourth echo request message will be expired in 5 seconds, the total required time for RYU controller to detect the failure is 17 seconds theoretically. If wireless mesh nodes are disconnected from RYU controller, RYU controller would delete the disconnected wireless nodes from the set of all nodes reachable by RYU controller. If all wireless mesh nodes are still connected with RYU controller, then RYU controller will simply need to keep sending only echo request message and listening to the echo reply message from each wireless node. If one or many of the wireless mesh nodes are disconnected from RYU controller, then RYU controller will send a configuration request message to every current connected switches

with RYU controller and assigns the necessary predefined forwarding rules to establish the alternative route with the purpose of rerouting. These new rules are treated here as the temporarily remedial rules because all the nodes are not moving and it is believed that the firstly predefined rules automatically assigned at the node' s boosting time must be nominally the best. Therefore, these new rules will be assigned the OpenFlow flow entry priority values which are higher than those of the predefined rules preinstalled at the boosting time. Consequently, with the presence of rerouting rules, the node will use these rerouting rules instead of using the predefined rules. In addition, since these new rules are treated merely as temporarily remedial of occasionally occurring node unreachability instances, these rerouting rules are assigned a relatively small value of hard_timeout. So, after the rule is installed for a longer time than that hard_timeout setting, the rule simply expires. The job of RYU controller therefore in our algorithm is to keep sending out those rerouting rules to all the switches periodically with the period that must be configured smaller than the hard_timeout settings. We have chosen to implement our rerouting in this way because we have to deal with the in-band control approach. So we must make sure that at least as the last resort, when all nodes are rebooted, the flow entries initially assigned must be a good starting plan to establish the control plane and the data plane at least in the case that all nodes are reachable SDN controller. If all wireless mesh nodes get connected back with RYU controller, RYU controller will stop sending a configuration request message to all currently connected switches. The algorithm of rerouting is discussed below and the assumption before starting rerouting algorithm is that all wireless mesh nodes get connected to RYU controller initially.

Algorithm: Rerouting Input : R_c= sdn controller

G = number of gateway nodes connected to R_c

N = number of wireless mesh nodes connected to R_c

$m_{request}$ = echo request message

m_{reply} = echo reply message

m_c = configuration request message

τ_e = echo request interval

τ_c = configuration request interval

τ_h = hard_timeout duration

u_{e_n} = unreplied echo request for each wireless mesh node

L_a = active node queue or set of all nodes reachable by R_c

Initialize : τ_e= 3s , $u_{e\_n} = 0$, τ_c = 8s, τ_h = 10s, L_a = N, G = 2, N = 6

1. Begin
2. For n = 1,. . . . . ,N Do
3. R_c sends $m_{request}$ to n every τ_e
4. if R_c receives m_{reply} from n then
5. mark n as connected active node i.e. put n into L_a and set $u_{e\_n} = 0$
6. else $u_{e\_n} += 1$
7. if $u_{e\_n} = 4$ then
8. delete n from L_a
9. R_c sends m_c to remaining active nodes in L_a every τ_c and install necessary forwarding rules to build the alternative route as predefined in Tables 4.4 and 4.5 with τ_h
10. End For
11. End

Since the in-band control approach is applied in the implementation of the outdoor medium-range SDWMN testbed, rerouting scenario from RYU controller needs to be considered not only for the control plane but also for the data plane. Consider the network topology in Figure 4.1 for the target testbed to be implemented.

Table 4.4 shows the rerouting information of the alternative routes to recover the control plane when primary routes for control plane are failed because of the failure of wireless mesh node. Table 4.5 shows the rerouting information of the alternative routes to restore the data plane when primary routes for the data plane are failed because of the failure of wireless mesh node. According to the routing information of primary route for control plane in Table 4.4, the failure of Raspi 6 does not affect it‘s neighbor wireless mesh node‘s control plane. Therefore, the failure of Raspi 6 is not considered in rerouting process for the control plane. The alternative routes for data plane in Table 4.5 are also the predefined

alternative routes to a nearest gateway. Failed node in Tables 4.4 and 4.5 is defined as a wireless mesh node which has no active connection with RYU controller. Affected node in Tables 4.4 and 4.5 is defined as a wireless mesh node which connects between RYU controller is disabled due to failed neighbor wireless mesh node. The alternative routes in Tables 4.4 and 4.5 are the predefined backup routes to recover the respective affected nodes due to the failure of neighboring wireless mesh node.

Table 4.4: Rerouting information with failure of wireless mesh node for control plane.

Failed node	Affected node	Alternative route
Raspi 1	Raspi 2	Raspi 2 - Raspi 5 - Raspi 4 - Gateway 1
Raspi 2	Raspi 3	Raspi 3 - Raspi 6 - Raspi 5 - Raspi 4 - Gateway 1
	Raspi 3	Raspi 3 - Raspi 6 - Raspi 5 - Raspi 4 - Gateway 1
Raspi 3	Gateway 2	Gateway 2 - Raspi 6 - Raspi 5 - Raspi 4 - Gateway 1
Raspi 4	Raspi 5	Raspi 5 - Raspi 2 - Raspi 1 - Gateway 1
Raspi 5	Raspi 6	Raspi 6 - Raspi 3 - Raspi 2 - Raspi 1 - Gateway 1
	Gateway 2	Gateway 2 - Raspi 6 - Raspi 5 - Raspi 4 Gateway 1

Table 4.5: Rerouting information with failure of wireless mesh node for data plane.

Failed node	Affected node	Alternative route
Raspi 1	Raspi 2	Raspi 2 - Raspi 5 - Raspi 4 - Gateway 1
Raspi 6	Raspi 5	Raspi 5 - Raspi 2 - Raspi 3 - Gateway 2

According to primary routes for data plane in Table 4.3, Raspi 4 and Raspi 3 do not need to relay the data packets from Raspi 2 and Raspi 5. If Raspi 4 and Raspi 5 are failed, then the data packets from Raspi 2 can still be sent through the route of Raspi 2 - Raspi 1 - Gateway 1 and the data packets from Raspi 5 can still be sent through the route of Raspi 5 - Raspi 6 - Gateway 2. If Raspi 2 and Raspi 5 fail, then the data packets from

Raspi 1, Raspi 4 can still be sent to Gateway 1 and the data packets from Raspi 3 and Raspi 6 can still be sent to Gateway 2. Only Raspi 1 needs to relay the data packets from Raspi 2 to Gateway 1 and Raspi 6 needs to relay the data packets from Raspi 5 to Gateway 2. If Raspi 1 is failed, then the data packets from Raspi 2 are rerouted to the alternative route which is Raspi 2 - Raspi 5 - Gateway 2 and the data packets from Raspi 5 is rerouted to the alternative route which is Raspi 5 - Raspi 3 - Gateway 1 if Raspi 6 is failed. The predefined alternative routes in Tables 4.4 and 4.5 are based on the shortest path scenario. In this work scope, the alternative routes to recover the control plane and data plane when a wireless mesh node is failed are only simple predefined alternative routes.

4.4 Monitoring Program for CPU Temperature of Wireless Mesh Node

The hardware specification of Raspberry Pi 3 is still limited and there is no CPU cooling system in the hardware of a Raspberry Pi. CPU temperature of a Raspberry Pi is suspected to be increased when applications are operated. The total temperature of a Raspberry Pi results from the addition of the device temperature and the ambient temperature. The maximum operable temperature of a Raspberry Pi is 85-degree Celsius and therefore the expected maximum temperature for system operation must be less than 80-degree Celsius with a safety margin of 5-degree Celsius. Since an ambient temperature is not controllable, the variation of actual operating temperature needs to be analyzed after the network is set up. Each Raspberry Pi will be placed inside a waterproof enclosure in the final testbed, and the CPU temperature of a Raspberry Pi is expected to be increased due to ambient temperature, especially in the summer season. The monitoring program for CPU temperature of wireless mesh node is implemented in each wireless mesh node in order to monitor the CPU temperature of a wireless mesh node due to ambient temperature by running the wireless mesh node for at least a continuous period of a whole day with 24 hours.

Temperature monitoring program which is summarized in 4.9 is implemented in each wireless mesh node. In the implemented temperature monitoring program, the wireless mesh node will be rebooted when the device temperature of the wireless mesh node is

```
import os
import time
def reboot():
    os.system('sudo reboot')
def test():
    os.popen("vcgencmd measure_temp >> /home/raspi5/Desktop/rrtresult/temp_26_11_2018.txt")
    os.popen("date >> /home/raspi5/Desktop/rrtresult/temp_26_11_2018.txt")
    temp=os.popen("vcgencmd measure_temp|cut -c6-9").readline()
    if temp<=str(80):
        print(temp)
        print("Raspberry Pi's Temperature is ok")
    else:
        time.sleep(10)
        os.popen("echo Device has bee restart >>/home/raspi5/Desktop/rrtresult/temp_26_11_2018.txt")
        if __name__ == "__main__":
            reboot()
try:
    while True:
        if __name__ == "__main__": #to exectute the defined function
            time.sleep(20)
            test()
except:
    print("Keyboard Error")
```

Figure 4.9: Temperature monitoring Python program installed in each wireless mesh node.

beyond 80-degree Celsius.

Chapter 5

Experiment of Final Outdoor SDWMN Testbed inside Campus

5.1 Setting of One-Hop Reachability Test inside Campus

The measurement of network reachability is taken along the road inside the campus area of the Chulalongkorn University and the position of the testing for wireless network reachability is summarized in Figure 5.1. The way of testing for the reachability of wireless

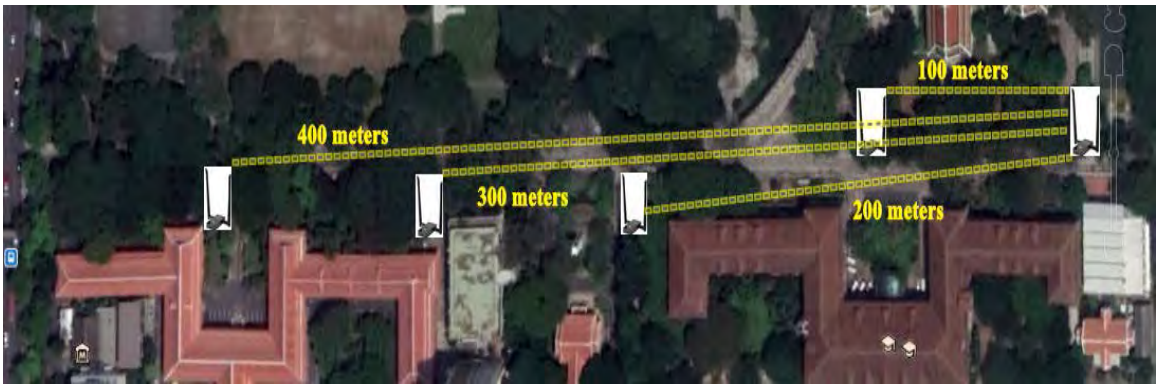

Figure 5.1: Testing scenario of wireless network reachability inside campus area of Chulalongkorn University.

network is summarized in Figures 5.2, 5.3, 5.4 and 5.5.

Figure 5.2: 100-metre wireless network reachability testing.

In 100 meters, two wireless mesh nodes are placed at the same side of the road as shown

Figure 5.3: 200-metre wireless network reachability testing.

Figure 5.4: 300-metre wireless network reachability testing.

Figure 5.5: 400-metre wireless network reachability testing.

in Figure 5.2 and there is an electric pole between two wireless mesh nodes which can block signal between two wireless mesh nodes. In 200 meters, two wireless mesh nodes are placed at the opposite side of the road and no trees between two wireless mesh nodes. In those two scenarios, the two wireless mesh nodes are placed on the small bush which is on the platform of the road as shown in Figure 5.2 and the height of the wireless mesh node inside the box from the ground is the same at every location. During the testing for 300 meters, one person has raised up the wireless mesh node instead of placing the wireless mesh node on the small bush. We tried to test TCP throughput in 400 meters as the same way what we have tested in 300 meters. Due to the large distance in 400 meters, TCP throughput and UDP throughput are the lowest among all testing experiments. The investigation has been conducted on Sunday 15th October inside the campus of Chulalongkorn University and therefore, there have been only a few cars which can block the signal during testing time. Two wireless mesh nodes have been configured at 5.66 GHz (132 channel) and the values of TCP and UDP throughput are summarized in Figures 5.7 and 5.8. The reason for choosing that channel is that 132 channel at that time has not been used by others according to the information of WiFi network analyzer from a smartphone.

5.2 Measurement Result of One-Hop Reachability

From the measurement result, the possible reason of causing the uncontrollable trend of throughput values are is a multipath fading within a campus. However, the aim for measuring the one-hop network reachability test is to know what is the maximum distance of one-hop link. The result from Figure 5.7 confirms that the obtained TCP throughput value at 400 meters which is still enough to support the intended future traffic monitoring application which requires 600 kbit/sec for sending captured image from Raspberry Pi‘s camera to traffic police box.

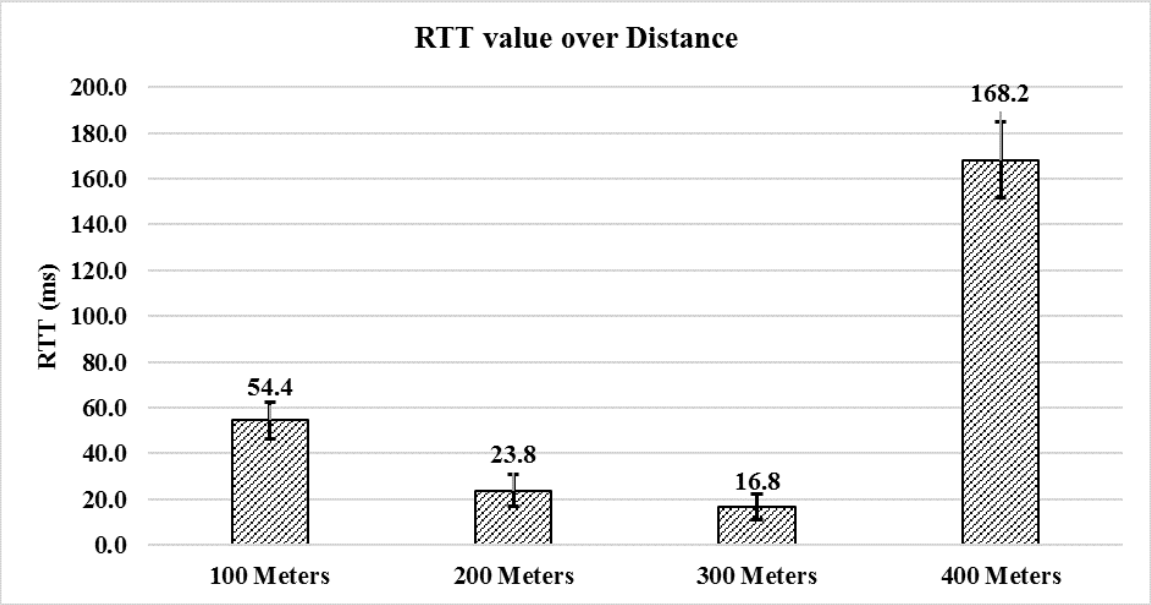

Figure 5.6: Comparison of 95-percent confidence interval for RTT in one-hop reachability.

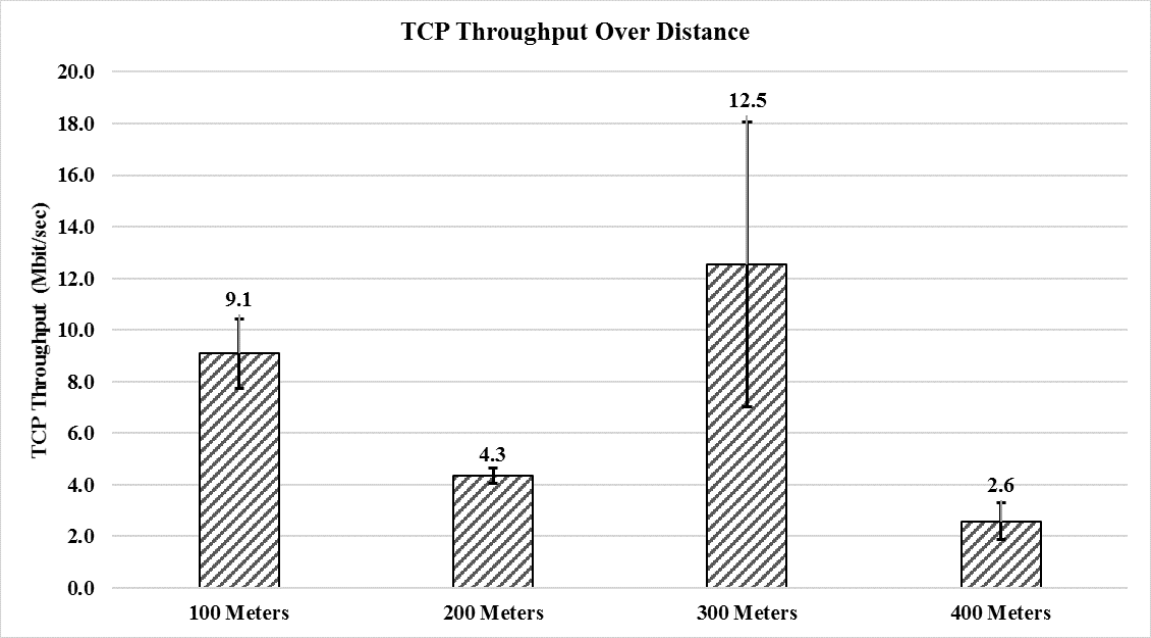

Figure 5.7: Comparison of 95-percent confidence interval for TCP throughput in one-hop reachability.

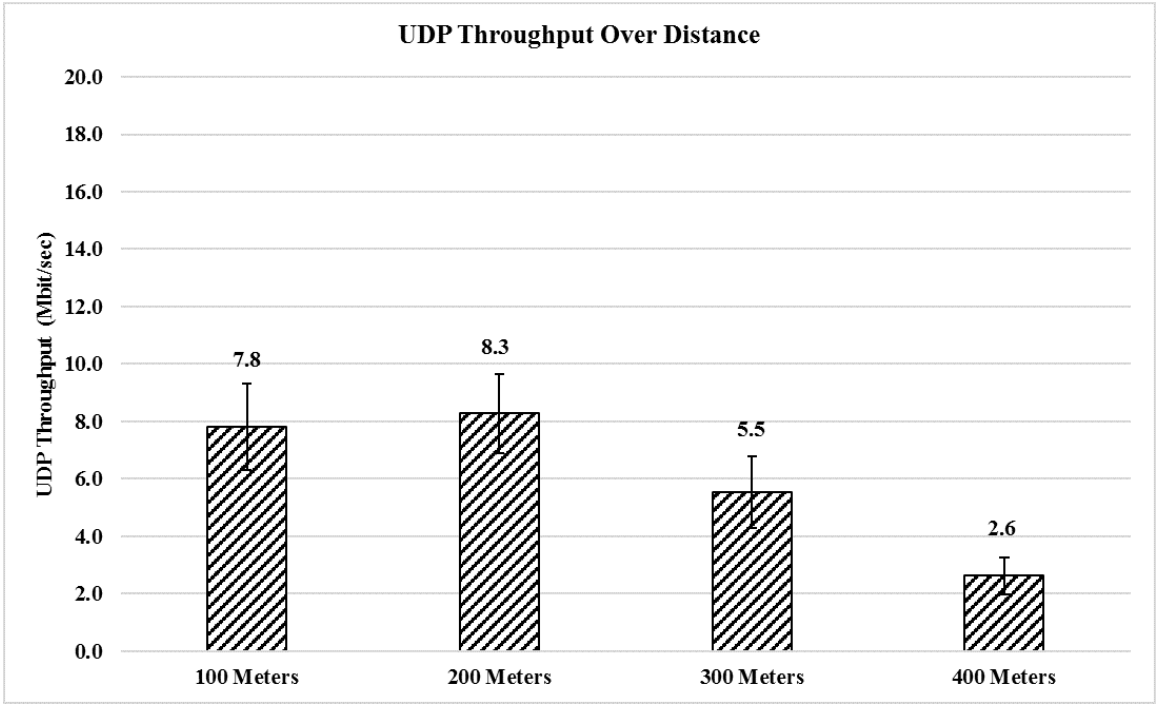

Figure 5.8: Comparison of 95-percent confidence interval for UDP throughput in one-hop reachability.

Chapter 6

Experiment of Final Outdoor SDWMN Testbed on Phaya Thai Road

6.1 Setting Up of Actual Testbed Component Installation

In this section, the steps of implementation for the real outdoor SDWMN testbed on Phaya Thai road between Rama 1 road and Rama 4 road is mainly discussed and Figure 6.1 shows the topology of real outdoor SDWMN testbed.

Figure 6.1: Topology of real outdoor SDWMN testbed on Phaya Thai road for road traffic monitoring network.

Figure 6.2 illustrates the way of attaching the waterproof box at the fence of the crossover bridge on Phaya Thai road.

Figure 6.2: Installation of waterproof box at fence of crossover bridge on Phaya Thai Road.

Figure 6.3: Installation of wireless mesh node inside waterproof box over crossover bridge on Phaya Thai Road.

Figure 6.3 shows the equipment of wireless mesh node inside the waterproof box. Inside every waterproof box, there is a Raspberry Pi, a power bank with 30000 mAh, and external omnidirectional antenna.

Figure 6.4: Crossover bridge on Phaya Thai Road.

Figure 6.4 is a picture of crossover bridge along the Phaya Thai road between Rama 1 road and Rama 4 road where wireless mesh nodes are installed. There are two wireless mesh nodes installed on each crossover bridge and the position of attached waterproof box at the crossover bridge is shown in Figure 6.5.

Two gateways in the outdoor SDWMN testbed are installed in two different traffic police boxes. Gateway 1 is installed at the traffic police box which is close to SamYan and Gateway 2 is installed at another traffic police box which is close to the MBK shopping

Figure 6.5: Position of wireless mesh node over crossover bridge on Phaya Thai Road.

center along the Phaya Thai road. Figure 6.6 shows the installation of Gateway 1 inside the building of traffic police box and Figures 6.7 and 6.8 represent the building of traffic box where Gateway 2 is located.

Figure 6.6: Traffic police box near SamYan MRT station on Rama 4 road where Gateway 1 is installed.

6.2 Measurement Result of Network Performance for Data Plane Traffic

In this section, the network performance of primary routes for the data plane in the outdoor SDWMN testbed is reported in terms of TCP throughput, UDP throughput, RTT and packet loss ratio. Measurement has been conducted during both daytime and nighttime in order to investigate the likely impact of vehicle presence crowd such as buses and cars on the road which is denser in daytime than nighttime. The period of a daytime experiment

Figure 6.7: Traffic police box near Chulalongkorn Soi 12 where Gateway 2 is installed.

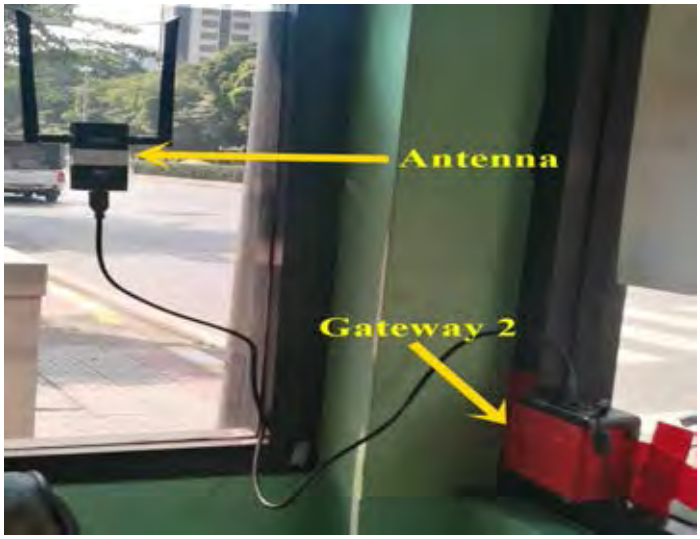

Figure 6.8: Installation of Gateway 2 in traffic police box.

is from 11 AM to 9 PM and the period of a nighttime experiment is from 10 PM to 8 AM. The time frame for daytime and nighttime testings are based on the standard responsible working hours of the shift of local traffic police.

The road traffic situation on Phaya Thai road segment between Rama 1 and Rama 4 road is exemplified in Figures 6.9 and 6.10.

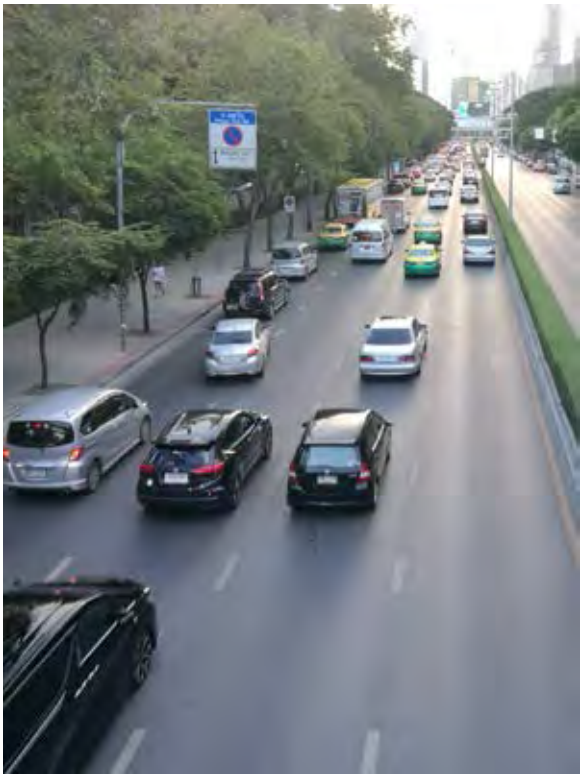

Figure 6.9: Traffic image captured with smart phone at daytime on Phaya Thai road at 4 PM.

The routing information for the data plane of the implemented testbed is recalled in this section. The routing information for the data plane is divided into two groups which are the group for Gateway 1 and the group for Gateway 2. In the group of Gateway 1, wireless mesh nodes of Raspi 1, Raspi 2 and Raspi 4 send the data packets to Gateway 1 and Raspi 3, Raspi 5 and Raspi 6 send the data packets to Gateway 2. The reason for grouping into two groups is for traffic load balancing.

The intended future traffic monitoring application is Kafka [10] and that application is based on TCP protocol for sending data packets such as image or video. Therefore, we mainly measure TCP throughput for each group of a gateway in order to make sure current network setting can support Kafka or not. We also have measured UDP throughput for a comparative reference.

Each measurement procedure is conducted by using the applications of iperf3 and ping. Iperf3 server has been run as a daemon in two gateways and an Iperf3 client is executed in each wireless mesh node. Each testing for TCP throughput, UDP Throughput and RTT based on ICMP packet with 1456 bytes sent for 3 minutes in a sequence which means iperf3

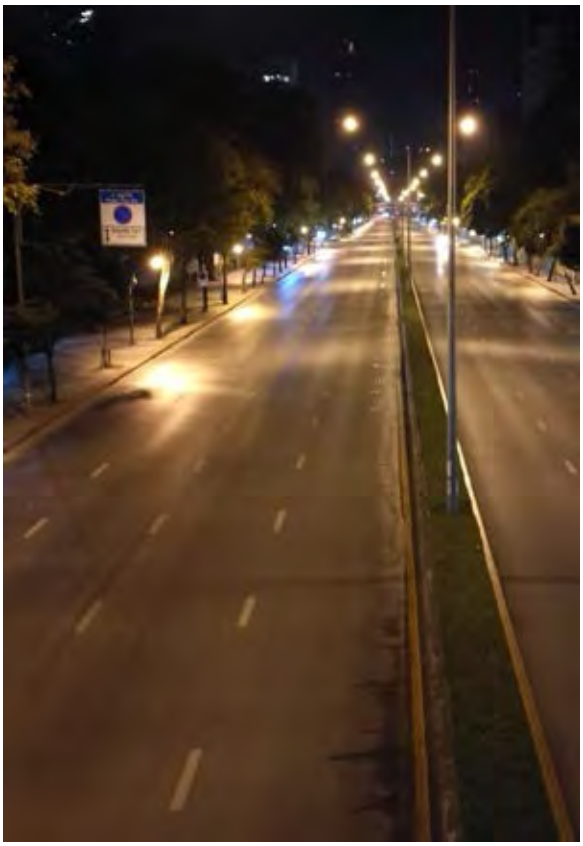

Figure 6.10: Traffic image captured with smart phone at nighttime on Phaya Thai road at 4 AM.

is run for TCP throughput for 3 minutes, UDP throughput for 3 minutes and ping is run for 3 minutes. We repeat the testing sequence 12 times.

Therefore, measuring the network performance at each wireless mesh node has taken at least 1 hour and 48 minutes. If iperf3 is run at all wireless mesh nodes at the same time, there will be some congestion being built up by the injected test traffics from all the nodes and the actual available value of network performance cannot be obtained correctly. Therefore, we measure the network performance at each wireless mesh node at a time. For example, when we complete the testing scenario at Raspi 1, measurement for network performance is started at Raspi 2. At least 10 hours are required to complete the testing scenario for all mesh nodes. During this test operation, we have noticed that the network interface of an attached external antenna has congested during the operation for measuring UDP throughput. Since there is no congestion control in UDP communication, testing the UDP throughput over medium-range wireless link has congested the external wireless network interface. However, we will not use the UDP protocol in the future intended traffic monitoring application.

Learning from the experiment of running the test for daytime operation, we change the testing sequence for nighttime operation. Particularly, we run TCP iperf3 first for 12 times at each of wireless mesh nodes with one node at a time. After TCP throughput measurement is finished, we run a ping program at each of wireless nodes for 12 times with one node at a time. As the last experiment, UDP measurement is conducted.

Figures 6.11, 6.12, 6.13 and 6.14 report the compared values of TCP throughput, UDP throughput and RTT during operation of daytime and nighttime as computed with 95-percent confidence interval.

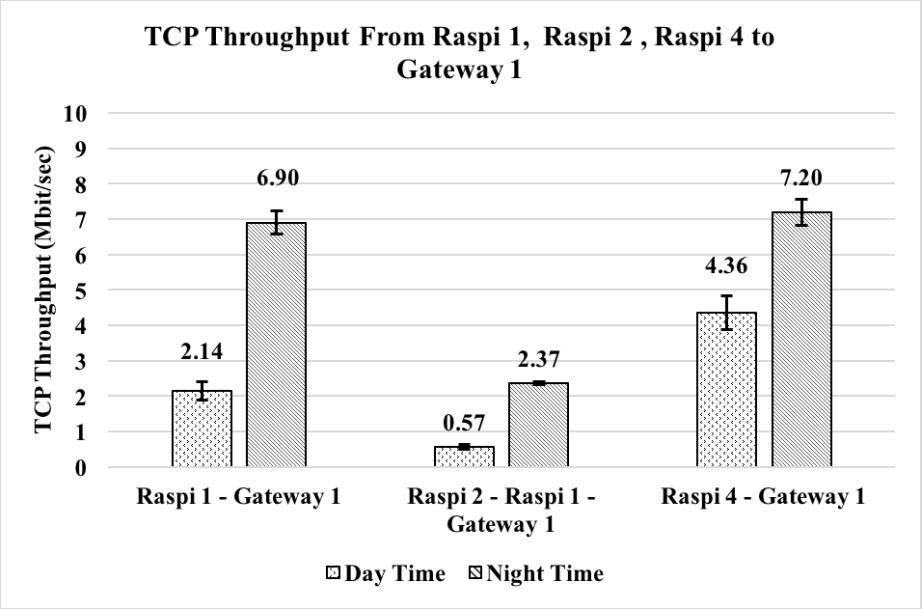

Figure 6.11: Comparison of 95-percent confidence interval for TCP throughput from Raspi 1, Raspi 2 and Raspi 4 to Gateway 1 between daytime and nighttime.

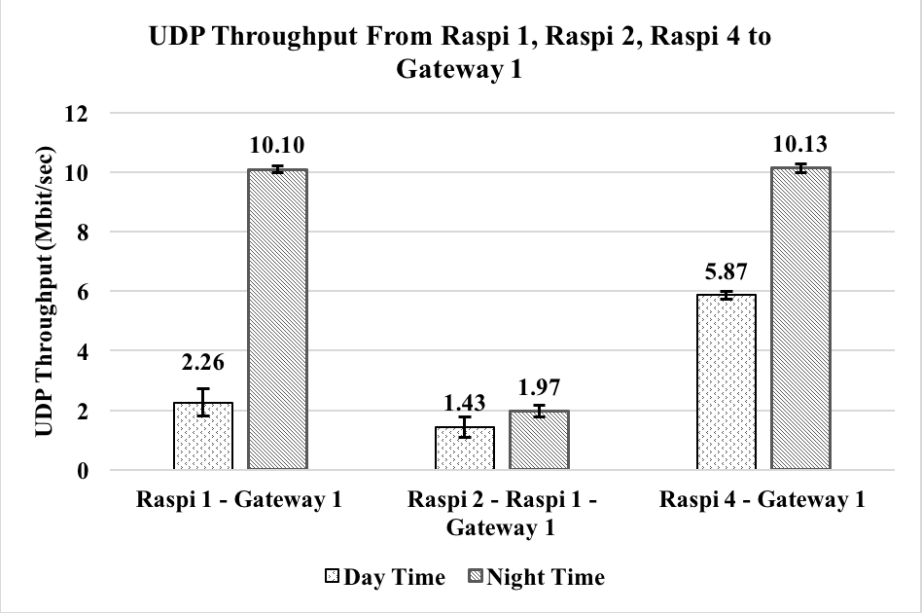

Figure 6.12: Comparison of 95-percent confidence interval for UDP throughput from Raspi 1, Raspi 2 and Raspi 4 to Gateway 1 between daytime and nighttime.

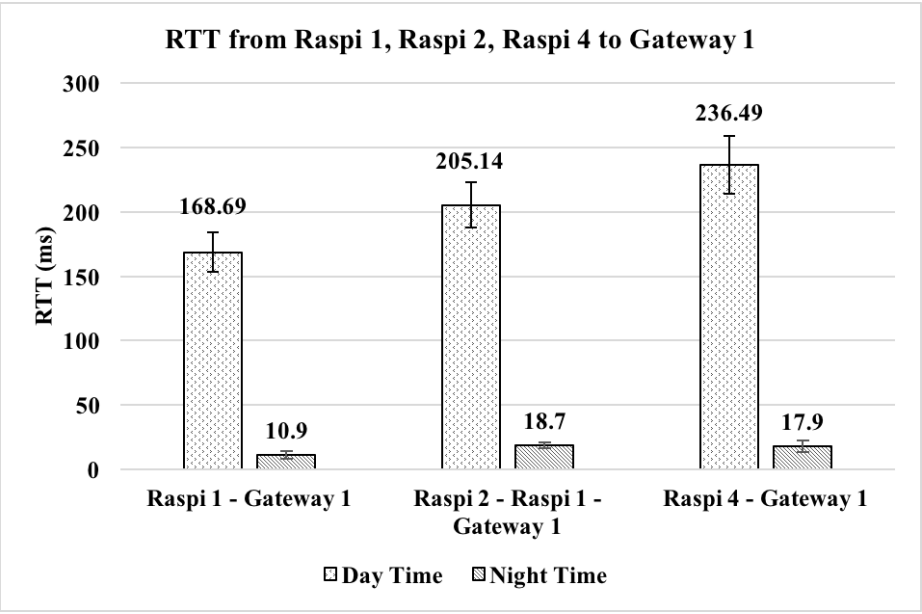

Figure 6.13: Comparison of 95-percent confidence interval for RTT from Raspi 1, Raspi 2 and Raspi 4 to Gateway 1 between daytime and nighttime.

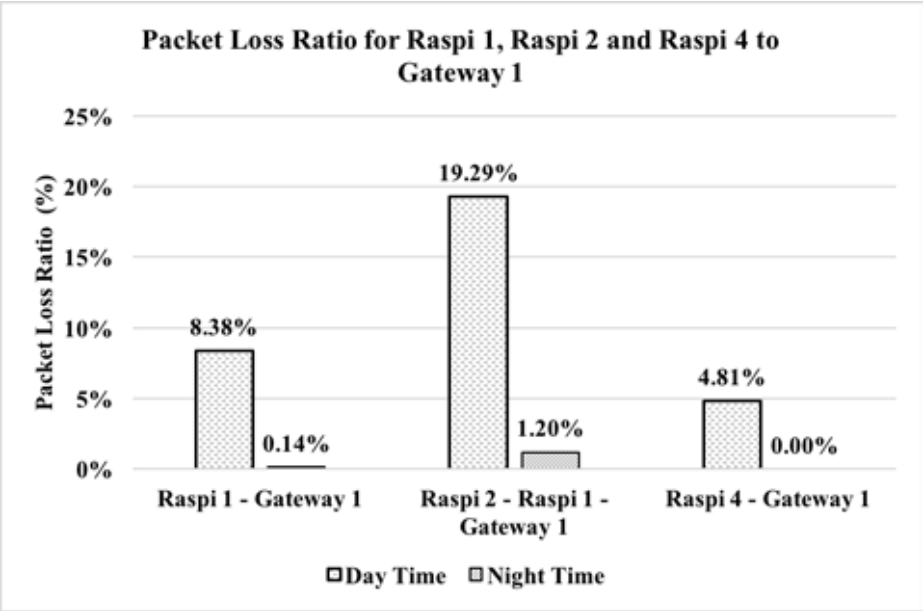

Figure 6.14: Comparison of packet loss ratio from Raspi 1, Raspi 2 and Raspi 4 to Gateway 1 between daytime and nighttime.

Figures 6.11 and 6.12 show that the implemented outdoor SDWMN provides better network performance at nighttime than what it can provide at daytime. Figure 6.13 confirms that more congested traffic situation at daytime can increase RTT of ICMP packet with 1456 bytes and packet loss ratio at daytime is higher than that of night time as seen from Figure 6.14. As the wireless external antenna at Gateway 1 is placed inside the building of traffic police box as shown in Figure 6.6, the signal between Gateway 1 and nearest wireless mesh nodes can be blocked while big cars such as buses are about to passing the intersection. From the real investigation result, the recommendation for future investigation is in locating the wireless antenna as high as possible such as placing the antenna at the roof of the traffic police box. In the group of Gateway 1, 2-hop communication from Raspi 2 to Gateway 1 provide lower than 600 kbit/sec which is an amount of bandwidth that traffic monitoring application is required, there can be a delay sending the captured images from the attached camera at Raspi 2 to Gateway 1. For other two nodes which are Raspi 1 and Raspi 2, the available TCP bandwidth can well support for Raspi 1 and Raspi 4 to send captured images to Gateway 1.

Similar measuremet has been executed for the group of Gateway 2 and the results are shown in Figures 6.15, 6.16, 6.17 and 6.18.

Before discuss the result of comparison for the group of Gateway 2, recall that the physical location between Raspi 3 and Gateway 2 is shown in Figure 6.19.

Along the route between Gateway 2 and Raspi 3, there are many trees at the side of the Phaya Thai road and the results of TCP throughput, UDP throughput between Raspi 3 and Gateway 2 in Figures 6.15 and 6.16 are the lowest in the nighttime. Likewise, RTT value is also the largest between Raspi 3 and Gateway 2 in both daytime and nighttime experiments. Due to many obstacles for the route between Raspi 3 and Gateway 2, obtained

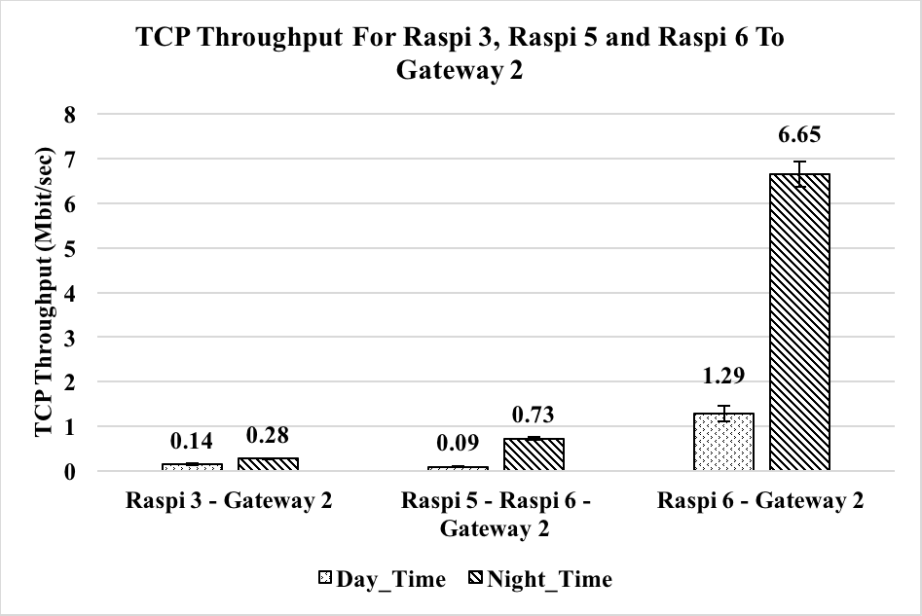

Figure 6.15: Comparison of 95-percent confidence interval for TCP throughput from Raspi 3, Raspi 5 and Raspi 6 to Gateway 2 between daytime and nighttime.

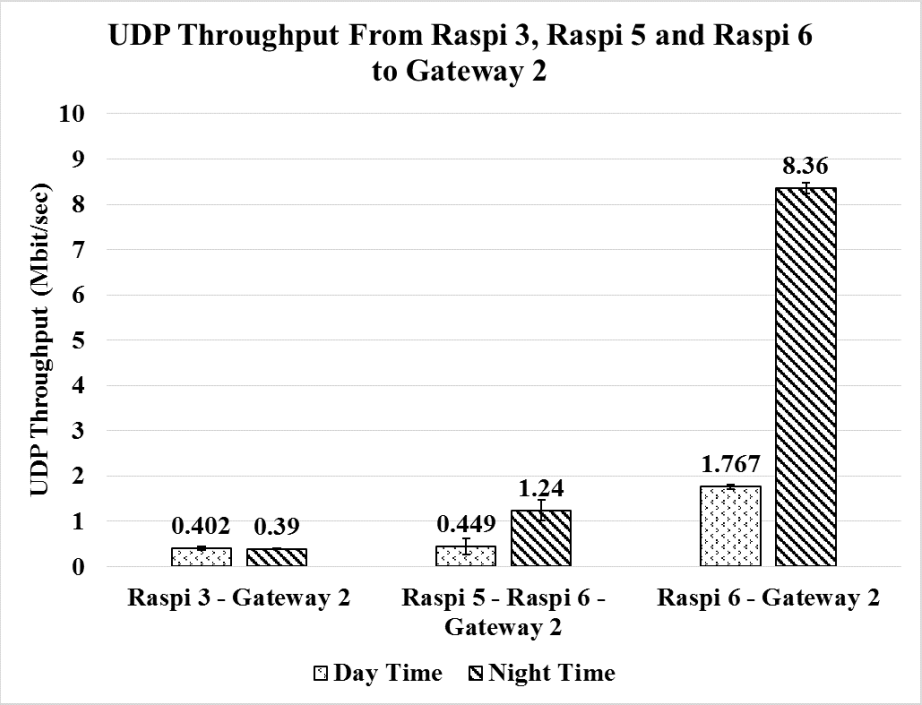

Figure 6.16: Comparison of 95-percent confidence interval for UDP throughput from Raspi 3, Raspi 5 and Raspi 6 to Gateway 2 between daytime and nighttime.

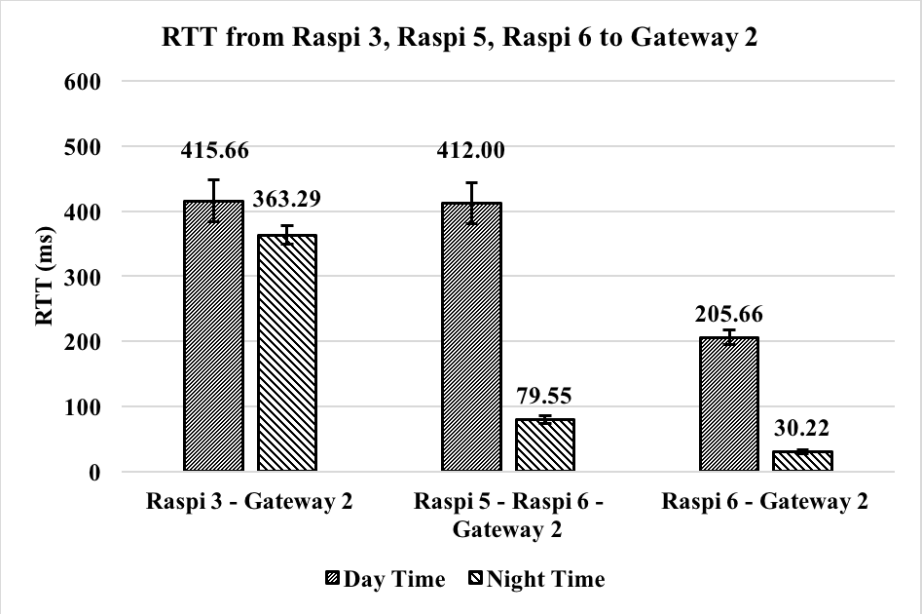

Figure 6.17: Comparison of 95-percent confidence interval for RTT from Raspi 3, Raspi 5 and Raspi 6 to Gateway 2 between daytime and nighttime.

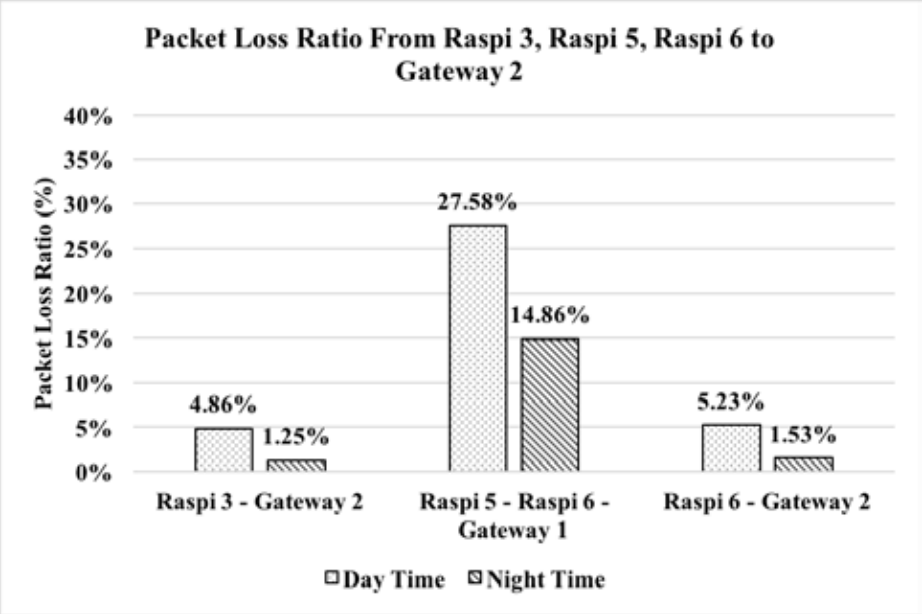

Figure 6.18: Comparison of packet loss ratio from Raspi, Raspi 5 Raspi 6 to Gateway 2 between daytime and nighttime.

Figure 6.19: Physical location between Raspi 3 and Gateway 2.

TCP throughput result is not sufficient to support for future traffic monitoring application.

Figure 6.20: Physical location between Raspi 6 and Raspi 5.

The measurement value for TCP throughput from Raspi 5 and Raspi 6 is only 90 kbit/sec which is very low to support necessary bandwidth for traffic monitoring application. The possible problem is that there has been a lot of disconnection between the wireless link between Raspi 5 and Raspi 6 as shown in Figure 6.20. The waterproof box in Figure 6.20 is Raspi 6 and Raspi 5 at the opposite side of the crossover bridge is at the same location. Therefore, we shift the location of Raspi 5 from the side of the crossover bridge to the middle of the crossover bridge for nighttime testing to avoid the obstacles. Since we moved the location of Raspi 5 in order to avoid interference of trees between the route of wireless link Raspi 5 and Raspi 6, we compare the obtained result as old location vs new location

in Figures 6.21, 6.22, 6.23 and 6.24.

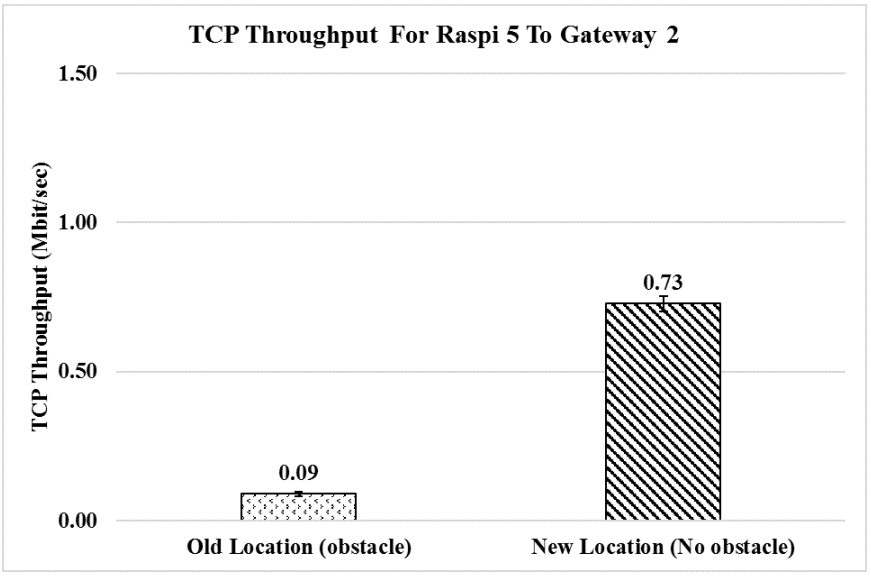

Figure 6.21: Comparison of 95-percent confidence interval for TCP throughput from Raspi 5 to Gateway 2 in old location and new location.

After we have moved the location of Raspi 5 from the side of the crossover bridge to the middle of the crossover bridge, TCP, UDP throughput at the new location is better than that of an old location. Adjusting the new location of Raspi 5 increases the network performance. Apart from that part, the comparison has been made between nighttime and daytime for the wireless link between Raspi 1 and Gateway 1, Raspi 2 and Gateway 1, Raspi 3 and Gateway 2, Raspi 4 and Gateway 1, Raspi 5 and Gateway 2 and Raspi 6 and Gateway 2.

From the experiment result, we have observed that the current measurement value of TCP throughput from Raspi 4 - Gateway 1, from Raspi 1 to Gateway 1, from Raspi 6 to Gateway 2 is enough for the whole day to support future traffic monitoring application which is required at 600 kbps. From Raspi 2 to Gateway 1, From Raspi 5 (new location) to Gateway 2, TCP and UDP throughput are enough when there is a light traffic condition but there can be a delay for traffic monitoring application in sending the captured images to the traffic box. The comparison of daytime vs nighttime values shows that network performance is better than at nighttime than daytime due to traffic density on the road especially, there can be only a few big cars at nighttime. Therefore, the position an antenna should be high enough to receive the better signal from wireless mesh node at each gateway in future investigation. The impact of trees on the throughput value between Raspi 3 and Gateway 2 is a good lesson for designing the routing for future work as trees can be an unavoidable obstacle for road traffic monitoring network. In the physical location for the group of Gateway 2 as per described in Figure 6.26, the distance between Raspi 3 and Gateway 2 and the distance between Raspi 6 and Gateway 2 are mostly same but there is a huge difference in obtained TCP, UDP throughput. For the group of Gateway 1, TCP and UDP throughput between Raspi 4 and Gateway 1 are larger in twice than the value

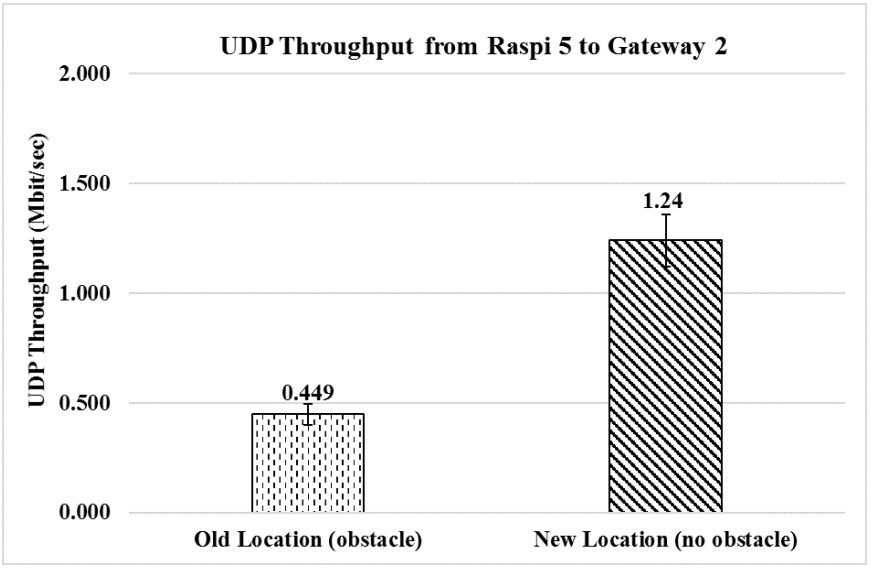

Figure 6.22: Comparison of 95-percent confidence interval for UDP throughput from Raspi 5 to Gateway 2 in old location and new location.

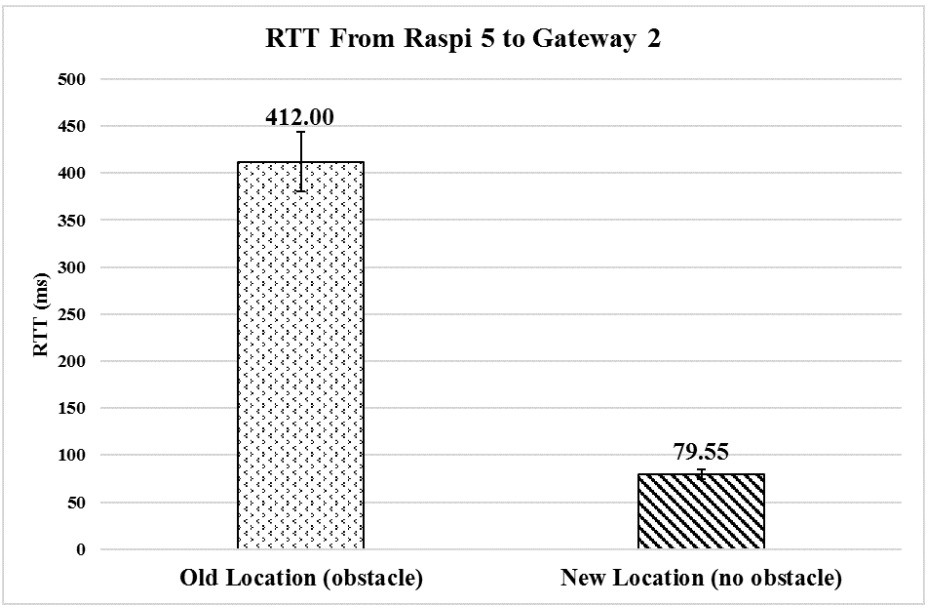

Figure 6.23: Comparison of 95-percent confidence interval for RTT from Raspi 5 to Gateway 2 in old location and new location.

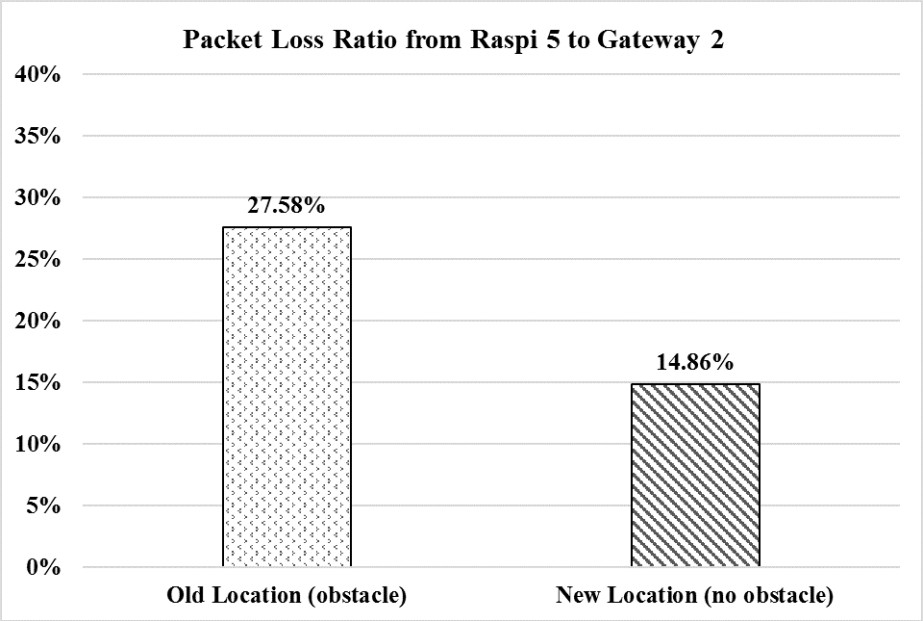

Figure 6.24: Comparison of packet loss ratio from Raspi 5 to Gateway 2 in old location and new location.

Figure 6.25: Physical location between Raspi 1, Raspi 4 and Gateway 1.

Figure 6.26: Physical location between Raspi 3, Raspi 6 and Gateway 2.

of TCP and UDP throughput between Raspi 1 and Gateway 1. Therefore, we recommend considering the routing pattern of zigzag instead of a straight line if wireless mesh node needs to be installed at the side of the road instead of being installed at the crossover bridges.

6.3 Practical Deployment Trial for Integrated SDWMN and Traffic Monitoring Application on Phaya Thai Road

In our final demonstration test case, we have installed and tested SDWMN with the intended road traffic monitoring application for 16 hours starting from 5 PM to 10:47 AM. During the operation, Raspi 2, Raspi 4 send the captured images of road traffic situation to Gateway 1 and Raspi 3, Raspi 5 send the captured images of road traffic situation to Gateway 2. The captured images are taken by the attached camera at the board of Raspberry Pi. The primary objective of this work is to provide the necessary network layer for that application and the status of outdoor SDWMN network during the operation of data plane is summarized in Figure 6.28. During this operation, wireless mesh nodes are often disconnected from RYU controller and the number of unreachable times of each wireless mesh node to RYU controller is summarized in Table 6.1 and the operation of traffic monitoring application over implemented SDWMN testbed is illustrated in Figure 6.27.

Figure 6.1 reports that the number of unreachable times from Gateway 2 to RYU controller is the highest. The control traffic from Gateway 2 needs to be relayed by Raspi 3, Raspi 2 and Raspi 1 in order to reach to RYU controller on the primary route. Moreover, the distance from Gateway 1 and Gateway 2 is 1100 meters which can cause an unstable connection. However, the number of unreachable time from wireless mesh nodes to RYU controller is not too much difference which means that up to 3-Hops connection for control plane can be applied well while 4-Hops connection (Gateway 2 to RYU controller) is not suitable to be applied in the implemented outdoor SDWMN testbed. The overall status of the control plane from RYU controller during the 16 hours operation is summarized in

Figure 6.27: Running traffic monitoring application over SDWMN testbed.

Table 6.1: Status of network during 16 hours of practical deployment operation.

Node	Number of Hops to RYU Controller	Number of Unreachable Time
Raspi 1	1	15
Raspi 2	2	29
Raspi 3	3	34
Raspi 4	1	16
Raspi 5	2	29
Raspi 6	3	26
Gateway 1	0	0
Gateway 2	4	394

Figure 6.28.

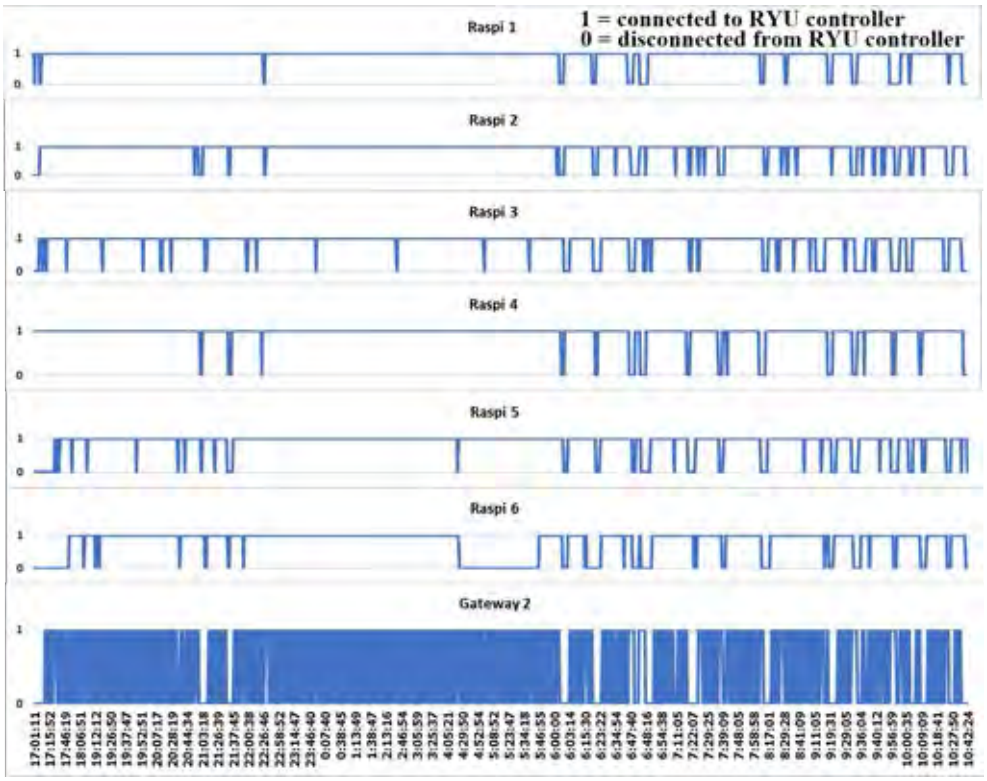

Figure 6.28: Practical operation status for 16 hours of outdoor SDWMN on Phaya Thai road

The overhead of OpenFlow traffic is measured by capturing OpenFlow traffic with Wireshark tool from 12:30 AM to 5 AM and overhead of OpenFlow traffic is summarized in Figure 6.31. The measurement of OpenFlow overhead traffic is calculated based on the captured OpenFlow protocol packets.

According to the summarized value of overhead OpenFlow traffic in Figure 6.31, the average overhead OpenFlow traffic is around 12 kbit/sec before 4:25:00 AM on 27th November 2018. After 4:25:00 AM, the average overhead OpenFlow traffic is jumped to around 20 kbit/sec and the increment is caused due to the failure of Raspi 6 at 4:27:00 AM. The duration of Raspi 6's unreachable to RYU controller was long. In the algorithm of rerouting RYU application, RYU sends configuration request message when one of the wireless mesh nodes is disconnected from RYU controller. Before 4:25:00 AM, all wireless mesh nodes are connected with RYU controller and RYU controller keeps in silence without sending any configuration request message to all alive wireless mesh node and therefore, the overhead is around 12 kbit/sec. Another observation is that the status of the control plane is quite stable before 6 AM. The density of the car could be very low at that time. However, Figure 6.28 shows that the connection between Raspi 1, Raspi 4 and RYU controller starts being fluctuated after 6 AM. The potential reason is an obstacle such as public bus which can block the signal between RYU controller and two neighbor nodes which are Raspi 1 and Raspi 4. Since the height of the attached antenna at Gateway 1 is not high, an obstacle

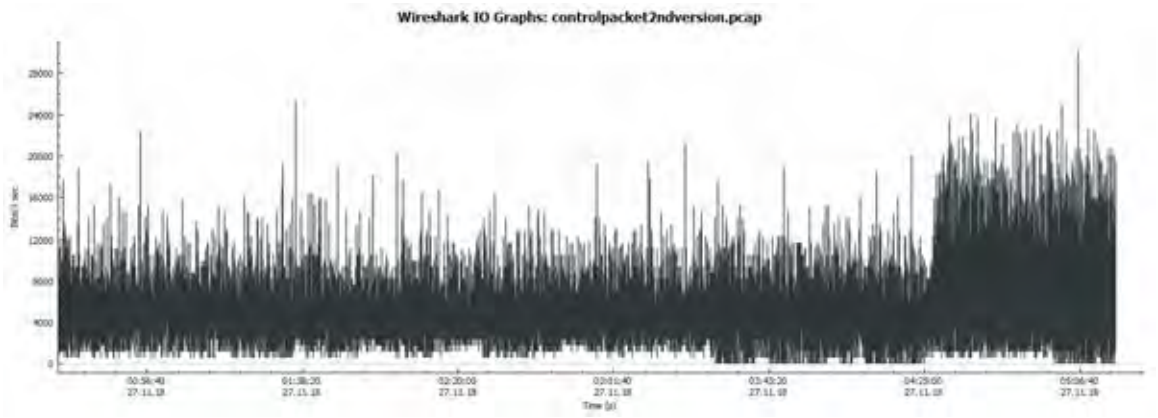

Figure 6.29: Overhead of OpenFlow traffic.

can easily interrupt the signal between RYU controller and two wireless mesh nodes (Raspi 1 and Raspi 4). In Section 6.2, RTT of ICMP packets between Raspi 1 and Gateway 1 and between Raspi 4 and Gateway 1 is higher in daytime than the values of RTT in the nighttime. Figure 6.30 is a picture which is captured at the location of Gateway 1 while the public bus in red color is passing through an intersection.

Figure 6.30: Captured image at location of Gateway 1 while public bus is passing through intersection which potentially blocks the line-of-sight of signal propagation in between the nearest wireless mesh nodes and Gateway 1.

6.4 Temperature Measurement of Wireless Mesh Node During Outdoor Network Operation

In this Section, the status of device temperature of wireless mesh node during outdoor network operation with traffic monitoring application is mainly discussed. Recalling the

value of maximum operable temperature for a raspberry pi is 85-degree Celsius. Therefore, the temperature of the wireless mesh node needs to be under 85-degree Celsius. We collect the temperature status of each wireless mesh node while the road traffic monitoring application is running on the outdoor SDWMN network. Temperature value of each wireless mesh node based on the day of 26th November in 2018 in Phaya Thai Road in Bangkok.

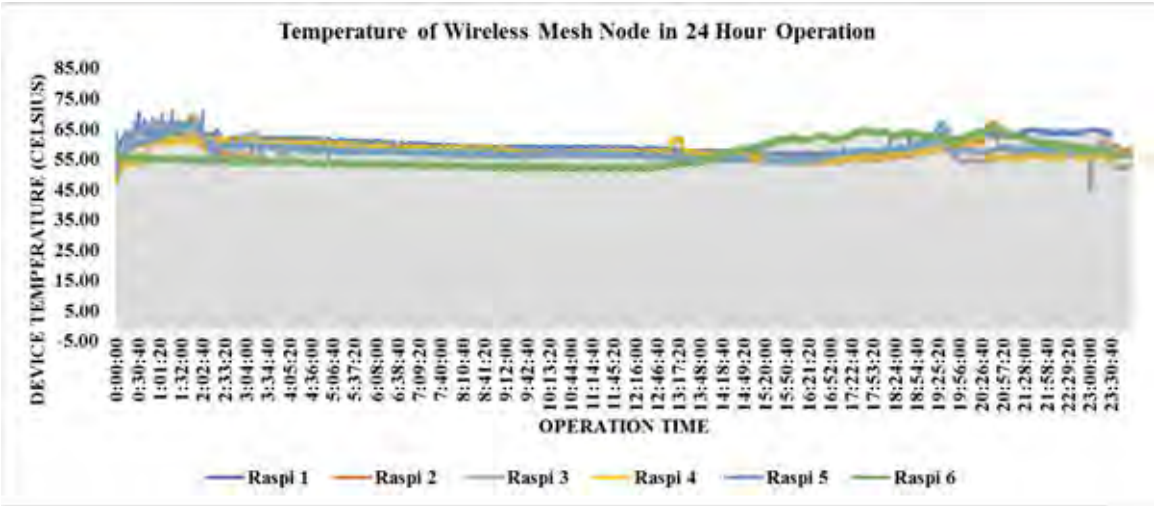

Figure 6.31: Status of temperature of wireless mesh node at outdoor network operation.

A temperature of a wireless mesh node while intended traffic monitoring application working with SDWMN is lower than the threshold level.

Figures 6.32 and 6.33 are screenshots of the information of the ambient temperature of 26th November 2018 and 27th November 2018 in Bangkok from https://www.timeanddate.com.

Ambient temperature during network operation is not hot and therefore wireless mesh node can run properly in the winter season in Thailand. Based on the actual measurement here, we have found that the selected Rasberry Pi hardware can tolerate the actual temperature during the real deployment. Therefore, our prepared Python watchdog program to reset the wireless mesh node when being overheated has not been triggered. As the result, the practical node failure due to temperature concern has not yet been realized in practice. And the disconnection of nodes from the RYU controller is mainly influenced instead by the wireless link connectivity. However, all our tests so far have been carried out in November, where the ambient temperature is considered the lowest annually. In the future work, it is recommended that the test should also be carried out in the summer so we could evaluate properly how this SDWMN system would function in a warmer condition. Then, the temperature-triggered software prepared in this research should be evaluated again.

Show weather for: 26 November 2018

Time	Conditions	Temp	Weather	Comfort Wind		Humidity	Barometer	Visibility
01:00 Mon, 26 Nov		26 °C	Clear	No wind		74%	1012 mbar	10 km
04:00		25 °C	Clear	No wind		78%	1010 mbar	10 km
07:00		24 °C	Passing clouds	No wind		76%	1012 mbar	10 km
10:00		30 °C	Passing clouds	2 km/h	—	59%	1014 mbar	10 km
13:00		33 °C	Passing clouds	2 km/h	↘	50%	1011 mbar	10 km
16:00		32 °C	Partly sunny	4 km/h	↘	50%	1009 mbar	10 km
19:00		30 °C	Passing clouds	4 km/h	—	59%	1011 mbar	10 km
22:00		28 °C	Passing clouds	No wind		57%	1013 mbar	10 km

Figure 6.32: Weather information from www.timeanddate.com for 26th November 2018 in Bangkok.

Show weather for: 27 November 2018

Time	Conditions	Temp	Weather	Comfort Wind		Humidity	Barometer	Visibility
01:00 Tue, 27 Nov		27 °C	Passing clouds	2 km/h	↘	57%	1012 mbar	10 km
04:00		26 °C	Passing clouds	2 km/h	—	63%	1012 mbar	10 km
07:00		26 °C	Partly sunny	2 km/h	↖	67%	1013 mbar	10 km
10:00		29 °C	Partly sunny	No wind		55%	1015 mbar	10 km
13:00		30 °C	Partly sunny	2 km/h	↖	56%	1012 mbar	10 km
16:00		28 °C	Overcast	No wind		72%	1012 mbar	10 km
19:00		27 °C	Passing clouds	4 km/h	—	73%	1013 mbar	10 km
22:00		25 °C	Passing clouds	No wind		72%	1015 mbar	10 km

Figure 6.33: Weather information from www.timeanddate.com for 27th November 2018 in Bangkok.

6.5 Measurement Result of Rerouting Performance

Since rerouting application is based on the failure of wireless mesh node, the testing for rerouting is conducted by rebooting each wireless mesh node manually for three times and restoration time for the affected wireless mesh node is analyzed. Firstly, the information of wireless mesh node is summarized in Table 6.2.

Table 6.2: MAC and IP addresses of wireless mesh nodes and gateways.

Wireless Mesh Node	MAC address	IP address
Raspi 1	e8:4e:06:5e:6b:09	10.0.0.1
Raspi 2	e8:4e:06:5f:47:59	10.0.0.2
Raspi 3	e8:4e:06:40:d3:7f	10.0.0.3
Raspi 4	e8:4e:06:40:d3:db	10.0.0.4
Raspi 5	e8:4e:06:40:dc:62	10.0.0.5
Raspi 6	e8:4e:06:40:94:20	10.0.0.6
Gateway 1	e8:4e:06:40:d3:4b	10.0.0.8
Gateway 2	e8:4e:06:5e:6a:b1	10.0.0.9

6.5.1 Case of Raspi 1's Failure

When Raspi 1 is failed, Raspi 2, Raspi 3 and Gateway 2 will be unreachable to Gateway 1 according to the information of predefined primary routes for the control plane. Raspi 1 is a relay node not only for the control plane but also for the data plane because Raspi 2 sends both control packets and data packets to Gateway 1 through the route Raspi 2 - Raspi 1 - Gateway 1. Likewise, Raspi 3 and Gateway 2 send the control packets through the routes Raspi 3 - Raspi 2 - Gateway 1 and Gateway 2 - Raspi 3 - Raspi 2 - Raspi 1 and Gateway 1, respectively.

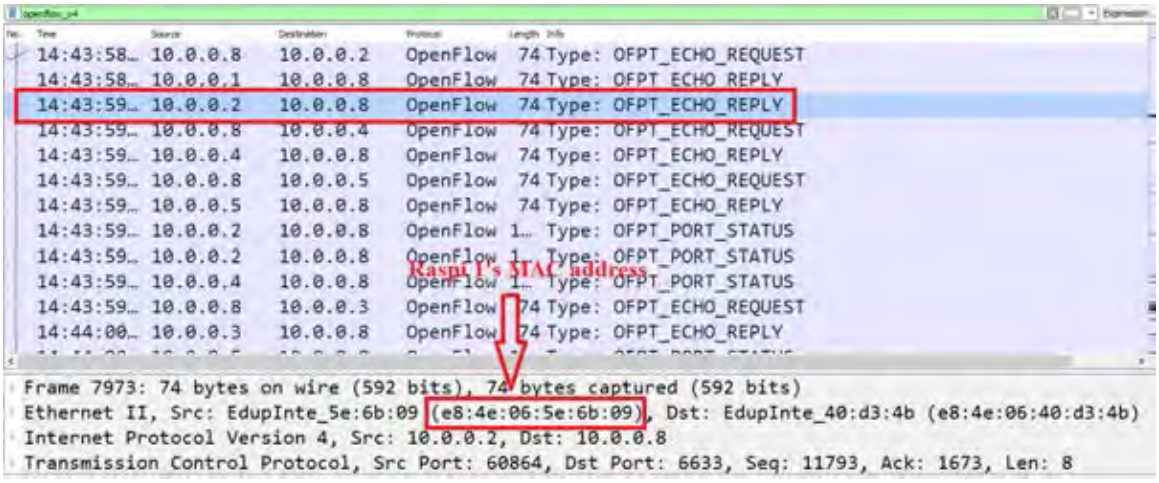

Figure 6.34: Information of received control packet from Raspi 2 to Gateway 1 through primary route.

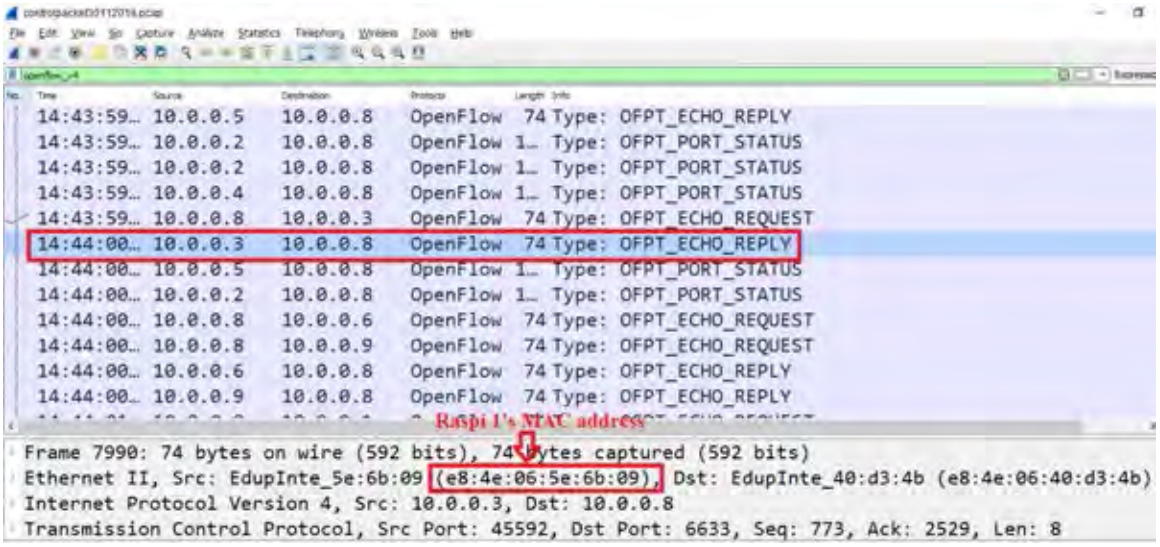

Figure 6.35: Information of received control packet from Raspi 3 to Gateway 1 through primary route.

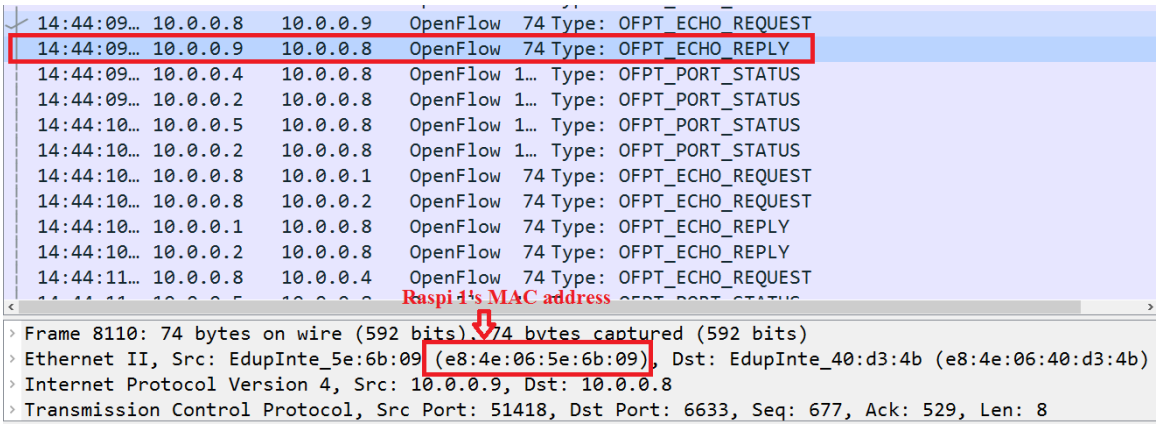

Figure 6.36: Information of received control packet from Gateway 2 to Gateway 1 through primary route.

Figures 6.34, 6.35 and 6.36 are the informations of packets which Gateway 1 receives from Raspi 2, Raspi 3 and Gateway 2 when Raspi 1 is in normal situation and those three figures show that Gateway 1 receives the packet from Raspi 2, Raspi 3 and Gateway 2 through MAC address of Raspi 1 on the predefined primary routes.

Table 6.3: Rerouting information of control plane for failure case of Raspi 1 in round 1.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 1	10.0.0.1	14:44:34			
Raspi 2	10.0.0.2	14:44:34	14:44:39	23 seconds	17 (detection_time) + 5 (rerouting_time)
Raspi 3	10.0.0.3	14:44:32	14:44:47	32 seconds	17 (detection_time) + 15 (rerouting_time)
Gateway 2	10.0.0.9	14:44:30	14:44:40	27 seconds	17 (detection_time) + 10 (rerouting_time)

Table 6.4: Rerouting information of control plane for failure case of Raspi 1 in round 2.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 1	10.0.0.1	14:54:25			
Raspi 2	10.0.0.2	14:54:26	14:54:32	23 seconds	17 (detection_time) + 6 (rerouting_time)
Raspi 3	10.0.0.3	14:54:27	14:54:31	21 seconds	17 (detection_time) + 4 (rerouting_time)
Gateway 2	10.0.0.9	14:54:24	14:54:32	25 seconds	17 (detection_time) + 8 (rerouting_time)

Referring to the restoration time values in Tables 6.3, 6.4 and 6.5, affected nodes (Raspi 2, Raspi 3 and Gateway 2) in the failure of Raspi 1 are restored within half a minute in most of the cases.

Configuration request message plays at the key role in the restoration of affected wireless mesh nodes. RYU controller assigns the necessary forwarding rules to establish the alternative routes with the purpose of rerouting by using the config request message and the way of sending configuration request message between RYU controller and wireless mesh node is described in Figure 6.37. In Figure 6.37, the wireless mesh node responds the configuration request message from RYU controller only when one of the wireless mesh nodes is disconnected from RYU controller.

The role of raspi 4 in this rerouting process is to relay the packets from Raspi 2, Raspi 3 and Gateway 2 to Gateway 1 and the relayed packets from Raspi 4 are captured with

Table 6.5: Rerouting information of control plane for failure case of Raspi 1 in round 3.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 1	10.0.0.1	14:59:28			
Raspi 2	10.0.0.2	14:59:25	14:59:41	33 seconds	17 (detection_time) + 5 (rerouting_time)
Raspi 3	10.0.0.3	14:59:28	14:59:29	18 seconds	17 (detection_time) + 1 (rerouting_time)
Gateway 2	10.0.0.9	14:59:28	14:59:30	19 seconds	17 (detection_time) + 2 (rerouting_time)

```
Switch ID 1152921504606846978),IP address is leaved ('10.0.0.2', 60864)
in Fri Nov 30 14:44:34 2018,0
Current Conneced Switches to RYU controller are [1152921504606846980L,
1152921504606846981L, 1152921504606846982L, 255421810004811L]
IP address ('10.0.0.8', 45432) sends OFPConfigReply message in Fri Nov
30 14:44:38 2018
case1
IP address ('10.0.0.4', 37352) sends OFPConfigReply message in Fri Nov
30 14:44:38 2018
case1
IP address ('10.0.0.5', 41386) sends OFPConfigReply message in Fri Nov
30 14:44:38 2018
case1
IP address ('10.0.0.6', 59900) sends OFPConfigReply message in Fri Nov
30 14:44:38 2018
case1
(Switch ID 1152921504606846978),IP address is connected ('10.0.0.2',
60872) in Fri Nov 30 14:44:39 2018,1
```

Figure 6.37: Configuration reply messages from wireless mesh nodes to RYU controller.

Wireshark tool at Gateway 1. In alternative route, Gateway 1 must receives the packets from Raspi 2, Raspi 3 and Gateway 2 from Raspi 4.

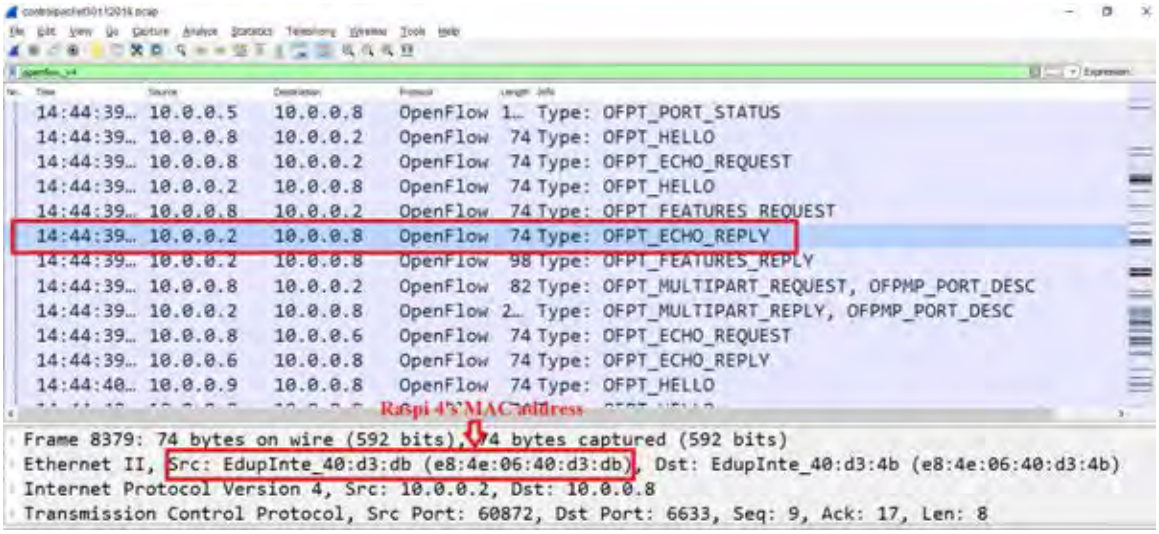

Figure 6.38: Control packet received at Gateway 1 from Raspi 2 through alternative route.

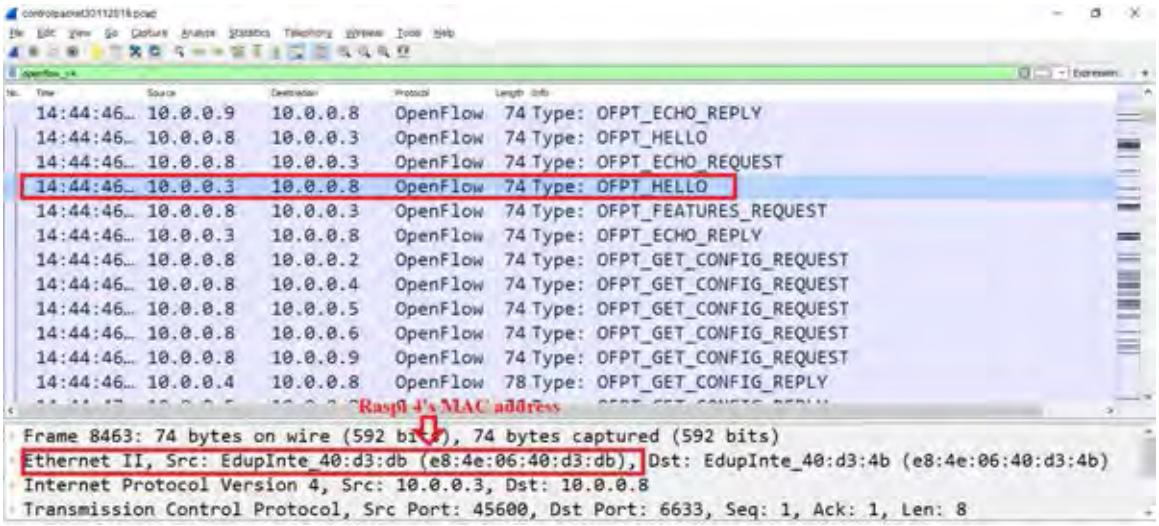

Figure 6.39: Control packet received at Gateway 1 from Raspi 3 through alternative route.

For the data plane, Raspi 1 relays a data packet from Raspi 2 to Gateway 1. If Raspi 1 is failed, Raspi 4 is responsible to relay the data packet to Gateway 1. The status of a received data packet at Gateway 1 from Raspi 2 when Raspi 1 is working and the time that Raspi 1 is failed is summarized in Figures 6.41 and 6.42, respectively.

In Figure 6.42, when Raspi 1 ia failed, Gateway 1 cannot receive a data packet from Raspi 2 at 16:44:17. After 29 seconds, data plane between Raspi 2 and Gateway 1 is rerouted from the primary route Raspi 2 - Raspi 1 - Gateway 1 to the alternative route Raspi 2 - Raspi 5 - Raspi 4 - Gateway 1. Then, Gateway 1 receives back a data packet from Raspi 2

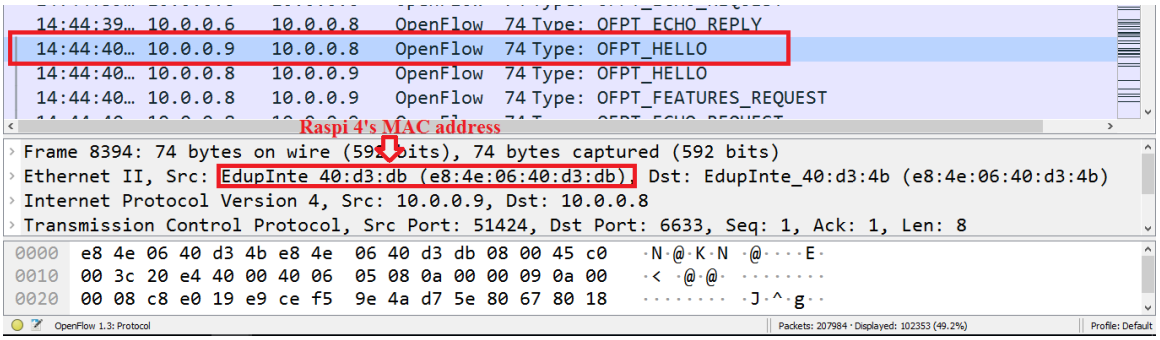

Figure 6.40: Control packet received at Gateway 1 from Gateway 2 through alternative route.

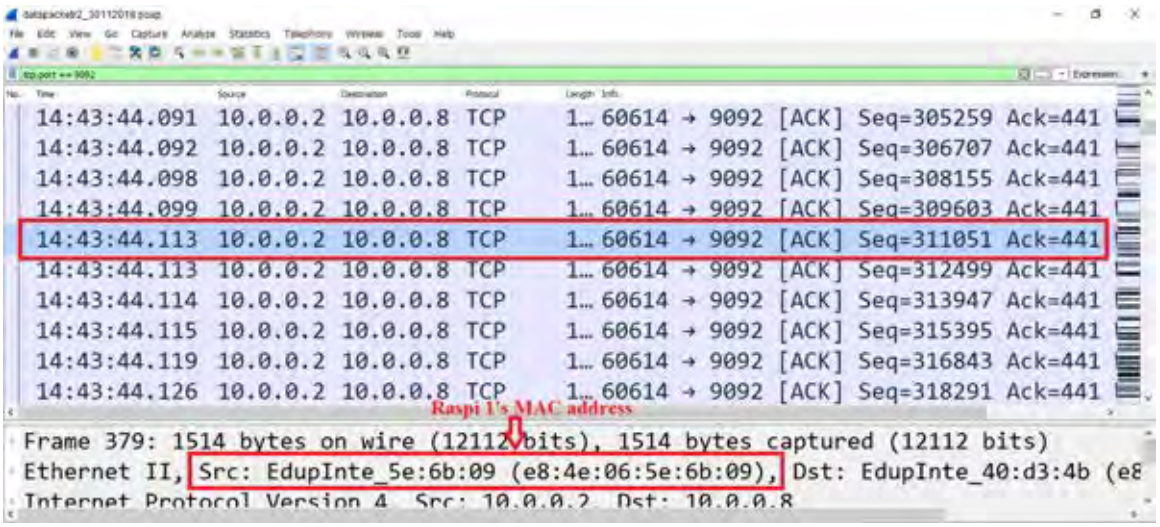

Figure 6.41: Control packet received at Gateway 1 from Raspi 2 through primary route.

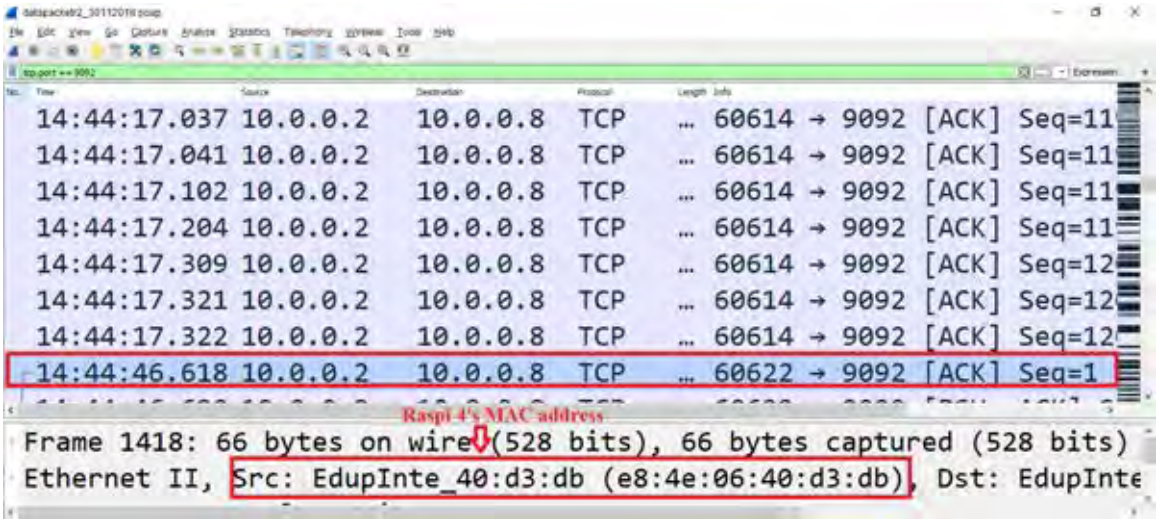

Figure 6.42: Control packet received at Gateway 1 from Raspi 2 through alternative route in round 1.

through Raspi 4. In round 1, total 29 seconds is required to reroute the data packets from Raspi 2 to Gateway 1.

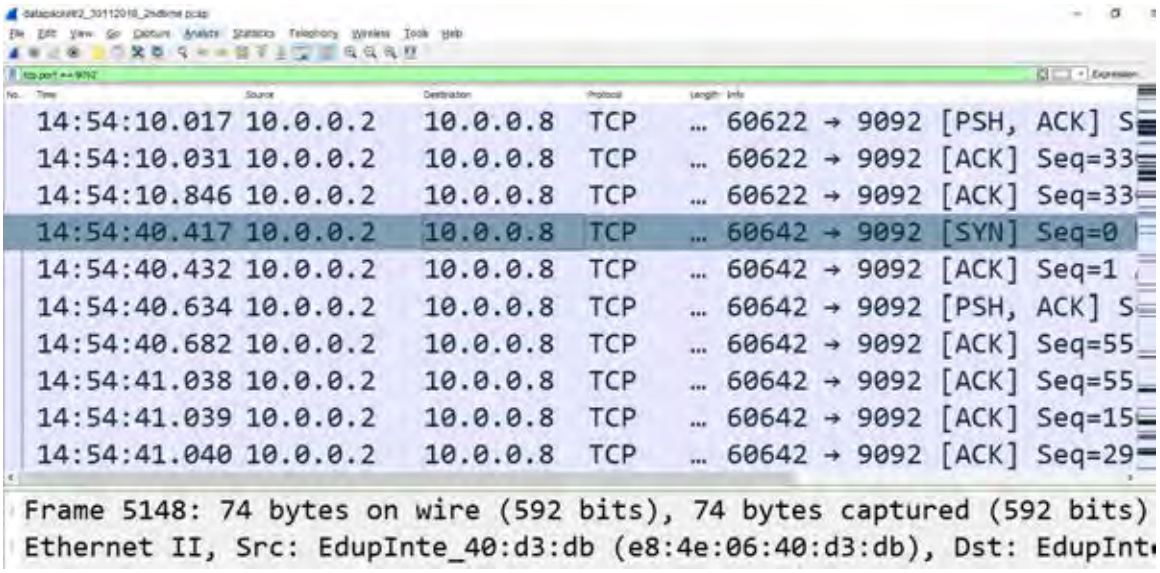

Figure 6.43: Control packet received at Gateway 1 from Raspi 2 through alternative route in round 2.

In round 2, Figure 6.43 shows that 30 seconds is required to reroute the data packets from Raspi 2 to Gateway 1 when Raspi 1 is failed.

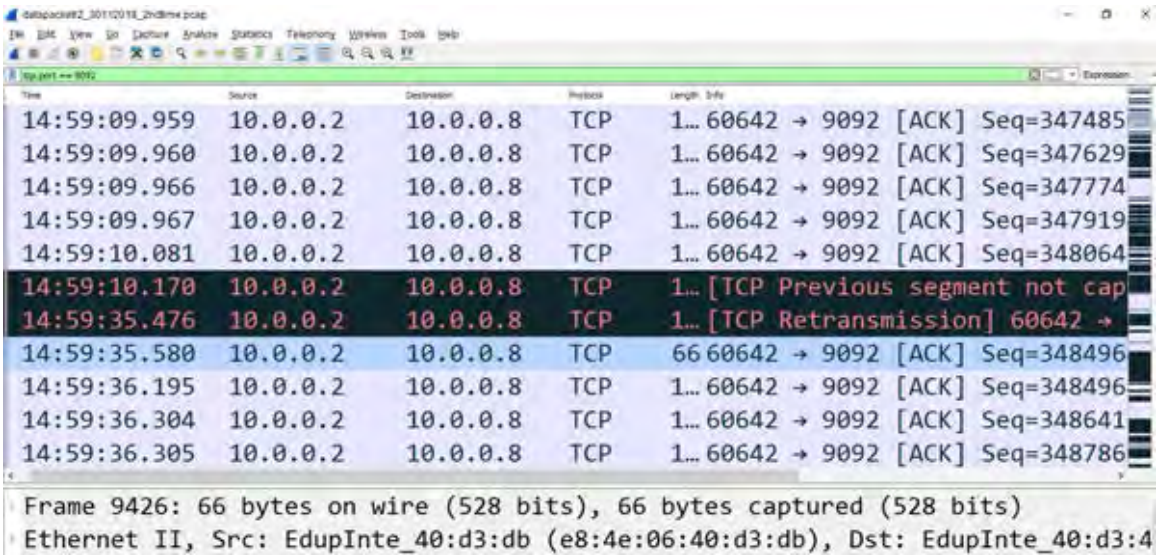

Figure 6.44: Control packet received at Gateway 1 from Raspi 2 through alternative route in round 3.

In round 3, Figure 6.44 shows that 25 seconds is required to reroute the data packets from Raspi 2 to Gateway 1 when Raspi 1 is failed.

For the case of failure of Raspi 1, the maximum required time for rerouting in all 3 rounds is 33 seconds.

6.5.2 Case of Raspi 2's Failure

Recalling the rerouting process of the failure of Raspi 2, the impacted wireless mesh nodes are Raspi 3 and Gateway 2 as Raspi 2 needs to relay the control packets from Raspi 3 and Gateway 2 to Raspi 1 for establishing the primary route.

Table 6.6: Rerouting information of control plane for failure case of Raspi 2 in round 1.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 2	10.0.0.2	16:50:45			
Raspi 3	10.0.0.3	16:50:44	16:50:58	31 seconds	17 (detection_time) + 14 (rerouting_time)
Gateway 2	10.0.0.9	16:50:43	16:50:50	24 seconds	17 (detection_time) + 7 (rerouting_time)

Table 6.7: Rerouting information of control plane for failure case of Raspi 2 in round 2.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 2	10.0.0.2	16:54:14			
Raspi 3	10.0.0.3	16:54:13	16:54:27	31 seconds	17 (detection_time) + 14 (rerouting_time)
Gateway 2	10.0.0.9	16:54:10	16:54:16	23 seconds	17 (detection_time) + 6 (rerouting_time)

Table 6.8: Rerouting information of control plane for failure case of Raspi 2 in round 3.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 2	10.0.0.2	16:56:55			
Raspi 3	10.0.0.3	16:56:54	16:57:09	32 seconds	17 (detection_time) + 15 (rerouting_time)
Gateway 2	10.0.0.9	16:56:52	16:56:58	23 seconds	17 (detection_time) + 6 (rerouting_time)

Tables 6.6, 6.7 and 6.8 confirm that all of the affected wireless mesh nodes are restored within 32 seconds when Raspi 2 is failed. Since Raspi 2 does not relay any data packets, rerouting is only considered for restoration of the control plane.

6.5.3 Case of Raspi 3's Failure

Raspi 3 relays the control packets from Gateway 2 to Gateway 1 to establish the OpenFlow control plane between Gateway 1 and Gateway 2. When Raspi 3 is failed, the traffic for control plane from Gateway 2 to Gateway 1 is rerouted from the primary route Gateway 2 - Raspi 3 - Raspi 2 - Raspi 1 - Gateway 1 to the alternative route Gateway 2 - Raspi 6 - Raspi 5 - Raspi 4 - Gateway 1. The information of rerouting for the failure of Raspi 3 is summarized in Tables 6.9, 6.10 and 6.11.

Table 6.9: Rerouting information of control plane for failure case of Raspi 3 in round 1.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 3	10.0.0.3	16:08:50			
Gateway 2	10.0.0.9	16:08:40	16:09:04	41 seconds	17 (detection_time) + 24 (rerouting_time)

Table 6.10: Rerouting information of control plane for failure case of Raspi 3 in round 2.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 3	10.0.0.3	16:11:35			
Gateway 2	10.0.0.9	16:11:33	16:12:02	46 seconds	17 (detection_time) + 29 (rerouting_time)

Table 6.11: Rerouting information of control plane for failure case of Raspi 3 in round 3.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 3	10.0.0.3	16:25:44			
Gateway 2	10.0.0.9	16:25:43	16:26:10	46 seconds	17 (detection_time) + 29 (rerouting_time)

The required time for rerouting the impacted wireless mesh node when Raspi 3 is failed is increased to 46 seconds. In the cases of failure of Raspi 1, 2 and 3 , Raspi 4 needs to relay the control packets from Raspi 2, 3 and Gateway 2 to Gateway 1 according to the predefined alternative routes because Raspi 4 is only the wireless mesh node in order to maintain the control plane between RYU controller and the remaining wireless nodes. The captured packets with Wireshark tool [5] at Gateway 1 from Raspi 4 during the rerouting process is the same which has been described in Figures 6.38, 6.39 and 6.40.

6.5.4 Case of Raspi 4's Failure

The scenario of rerouting when there is a failure at Raspi 4, Raspi 5 and Raspi 6 are similar to the scenario when there is a failure at Raspi 1, Raspi 2 and Raspi 3. When Raspi 4 is failed, Raspi 1 needs to relay the packets from Raspi 5 through the route Raspi 5 - Raspi 2 - Raspi 1 - Gateway 1 and relays the packets from Raspi 6 through the route Raspi 6 - Raspi 2 - Raspi 1 - Gateway 1. In this subsection ,the failure of Raspi 4 is considered for 3 times and the rerouting information is summarized in Tables 6.12, 6.13 and 6.14.

Table 6.12: Rerouting information of control plane for failure case of Raspi 4 in round 1.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 4	10.0.0.4	15:13:28			
Raspi 5	10.0.0.5	15:13:19	15:13:47	45 seconds	17 (detection_time) + 28 (rerouting_time)
Raspi 6	10.0.0.6	15:13:28	15:13:32	21 seconds	17 (detection_time) + 4 (rerouting_time)

Table 6.13: Rerouting information of control plane for failure case of Raspi 4 in round 2.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 4	10.0.0.4	15:17:47			
Raspi 5	10.0.0.5	15:17:48	15:18:01	30 seconds	17 (detection_time) + 13 (rerouting_time)
Raspi 6	10.0.0.6	15:17:45	15:17:59	31 seconds	17 (detection_time) + 14 (rerouting_time)

Table 6.14: Rerouting information of control plane for failure case of Raspi 4 in round 3.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 4	10.0.0.4	15:20:48			
Raspi 5	10.0.0.5	15:20:46	15:21:03	34 seconds	17 (detection_time) + 17 (rerouting_time)
Raspi 6	10.0.0.6	15:20:47	15:21:03	33 seconds	17 (detection_time) + 16 (rerouting_time)

In Rounds 1, 2 and 3, the required time for restoration for Raspi 5 and Raspi 6 when Raspi 4 is failed in Round 2 and Round 3 is around 34 seconds. In Round 1, the required

time for restoration take longer than other two rounds. The reason is that at Round 1, RYU controller decided the unreachable of Raspi 5 with only 3 unreplied echo request message and the connection stauts between Raspi 5 and Gateway 1 is captured with wireshark tool which is summarized in Figure 6.45.

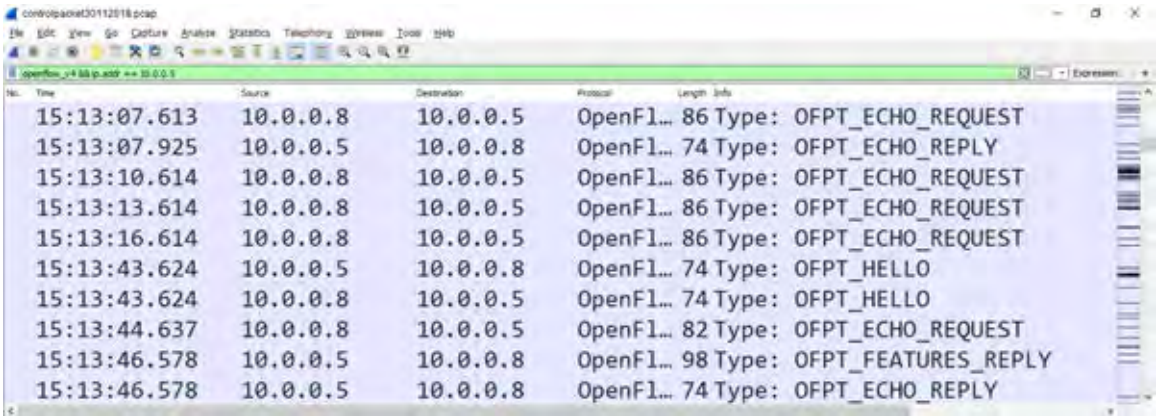

Figure 6.45: Control packet received at Gateway 1 from Raspi 5 through primiary route.

Figures 6.46 and 6.47 show the information of the received packet from Raspi 5 and Raspi 6 at Gateway 1. Source MAC address of those received packets from Raspi 5 and Raspi 6 through the primary route is a MAC address of Raspi 4 which means that Raspi 4 successfully relays the packets from Raspi 5 and Raspi 6 to Gateway 1.

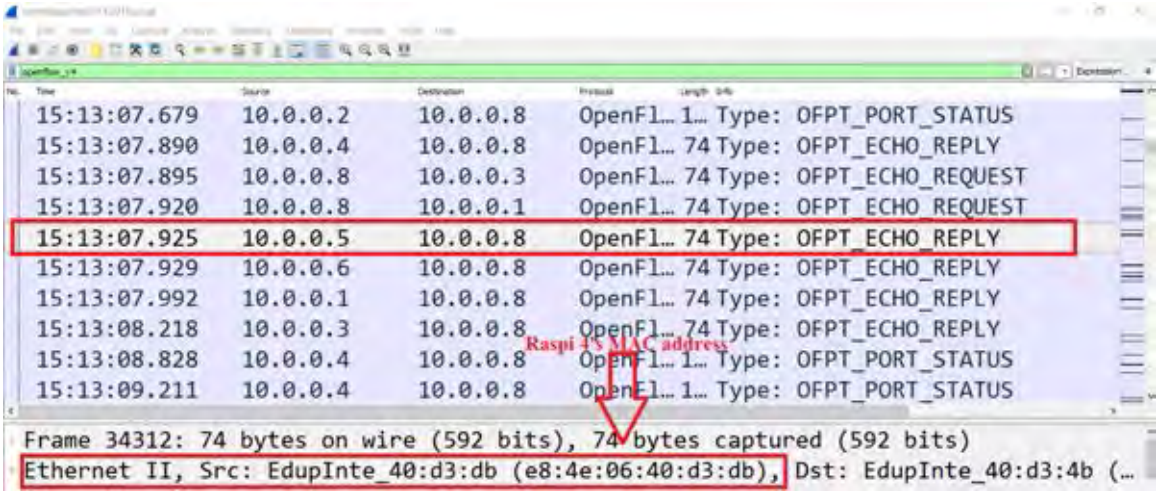

Figure 6.46: Control packet received at Gateway 1 from Raspi 5 through primiary route.

When Raspi 4 is failed, the control packet from Raspi 5 is rerouted through the alternative rouute which is Raspi 5 - Raspi 2 - Raspi 1 - Gateway 1. Likewise, the control packet from Raspi 6 is reoruted throught alternative route which is Raspi 6 - Raspi 3 - Raspi 2 - Raspi 1 - Gateway 1. Figures 6.48 and 6.49 shows that Gateway 1 receives the control packet from Raspi 5 and Raspi 6 from Raspi 1 when Raspi 4 is failed.

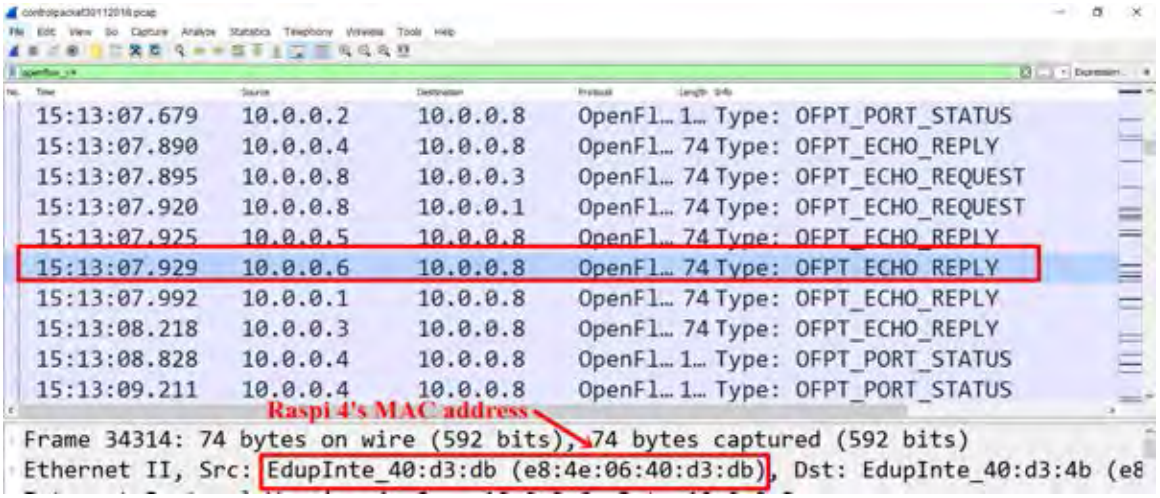

Figure 6.47: Control packet received at Gateway 1 from Raspi 6 through primiary route.

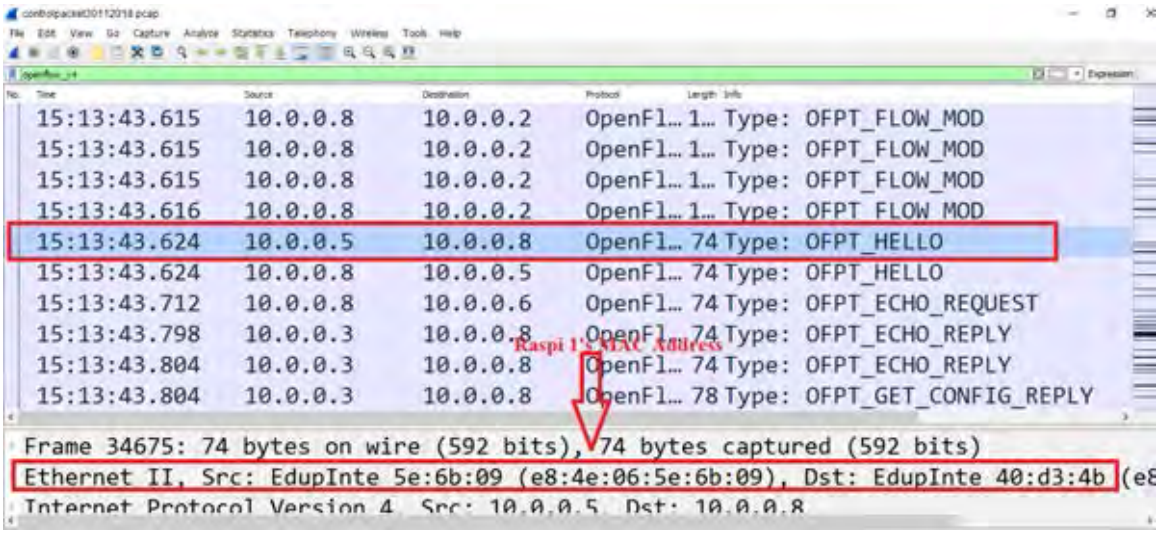

Figure 6.48: Control packet received at Gateway 1 from Raspi 5 through alternative route.

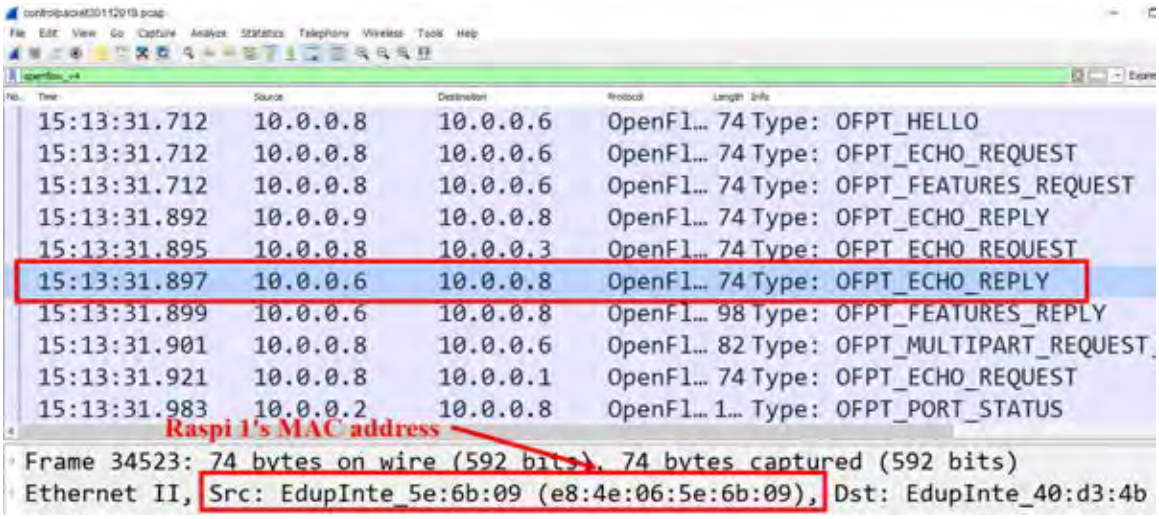

Figure 6.49: Control packet received at Gateway 1 from Raspi 6 through alternative route.

6.5.5 Case of Raspi 5's failure

In the primary route, Raspi 5 relays the control packet form Raspi 6 to Gateway 1 and therefore restoration for the control path between Raspi 6 and Raspi 5 is also required to be considered if Raspi 5 is failure stage. The rerouting informations of Raspi 5 are summarized in Tables 6.15, 6.16 and 6.17.

Table 6.15: Rerouting information of control plane for failure case of Raspi 5 in round 1.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 5	10.0.0.5	15:35:39			
Raspi 6	10.0.0.6	15:35:38	15:35:52	31 seconds	17 (detection_time) + 4 (rerouting_time)

6.5.6 Case of Raspi 6's failure

There is no impact for the control plane when Raspi 6 is failed because Raspi 6 does not need to relay any control packet to Gateway 1. However, Raspi 6 is used to relay the data packets from Raspi 5 to Gateway 2. Figure 6.50 shows the status of receiving incoming packets at Gateway 2 from Raspi 5 when Raspi 6 is in operational state ane Figure 6.51 shows the stutus of receiving incoming packets at Gateway from Raspi 5 with the situation of Raspi 6 is in failure state. When Raspi 6 is falied, Gateway 2 receives the data packet from Raspi 5 with the help from Raspi 3.

In Figures 6.50 and 6.51, Gateway 2 does not receive the data packets from Raspi 5 at 16:38:20 when Raspi 6 is failed and data packet from Raspi 5 is rerouted through the

Table 6.16: Rerouting information of control plane for failure case of Raspi 5 in round 2.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 5	10.0.0.5	15:40:40			
Raspi 6	10.0.0.6	15:40:38	15:40:54	33 seconds	17 (detection_time) + 16 (rerouting_time)

Table 6.17: Rerouting information of control plane for failure case of Raspi 5 in round 3.

Node	IP address	Down Time	Up Time	Restoration Time	Remark
Raspi 5	10.0.0.5	16:02:01			
Raspi 6	10.0.0.6	16:02:02	16:02:14	29 seconds	17 (detection_time) + 12 (rerouting_time)

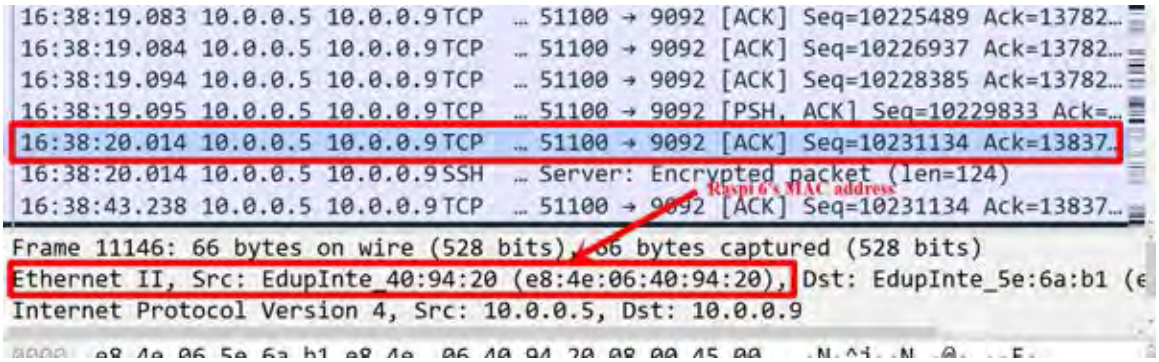

Figure 6.50: Status of received data packet from Raspi 5 at Gateway 2 while Raspi 6 is working.

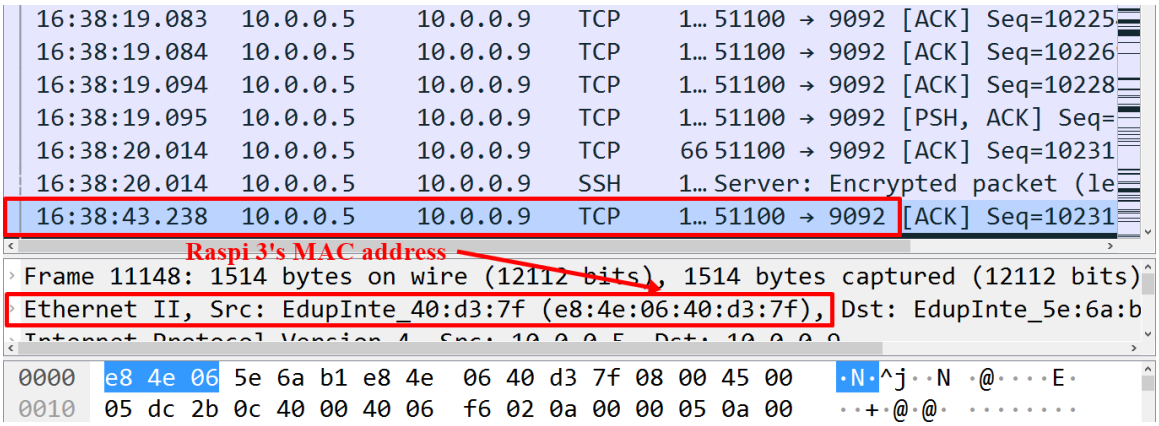

Figure 6.51: Status of received data packet from Raspi 5 at Gateway 2 when Raspi 6 is failed.

alternative route Raspi 5 - Raspi 2 - Raspi 3 - Gateway 2 and Gateway 2 receives back the data packets from Raspi 5 at 16:38:43 with the source MAC address is Raspi 3's MAC address. In round 1, total 23 seconds are required for rerouting.

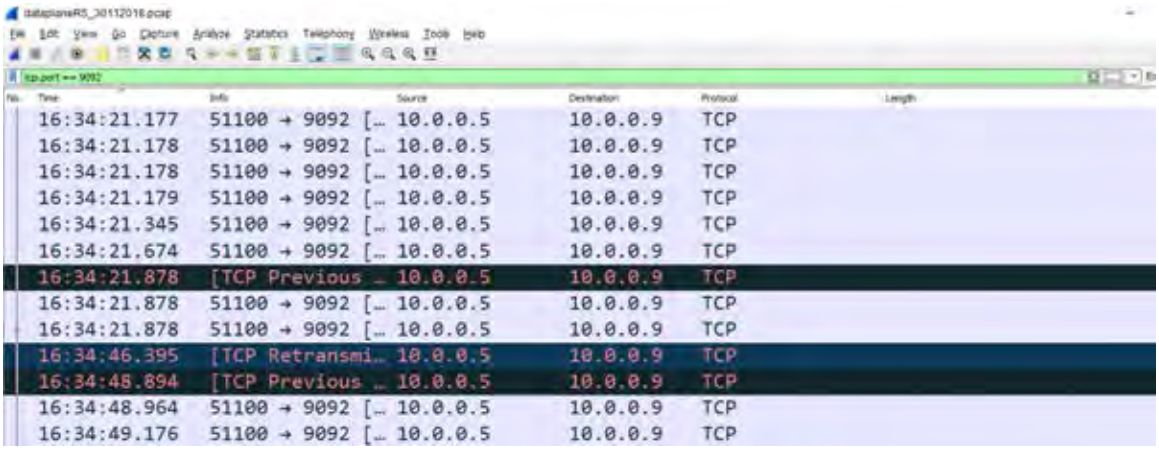

Figure 6.52: Status of received data packet from Raspi 5 at Gateway 2 when Raspi 6 is failed in round 2.

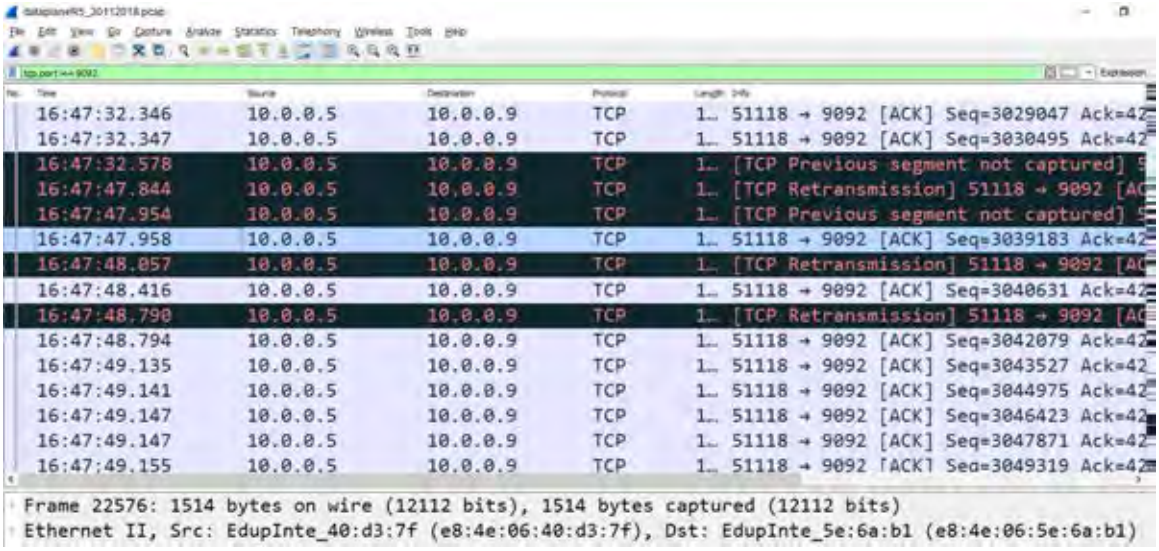

Figure 6.53: Status of received data packet from Raspi 5 at Gateway 2 when Raspi 6 is failed in round 3.

In round 2 and round 3, the total required time for rerouting the data packets from Raspi 2 are 25 seconds and 15 seconds respectively. Therefore, the maximum required time rerouting during experiment time is 25 seconds.

6.5.7 Summary of Rerouting Performance

In this work, rerouting is only based on the failure of the wireless mesh nodes. Another assumption for rerouting is that the wireless link between two wireless mesh nodes on

the same crossover bridge works properly before the process of rerouting is started. The maximum required time to reroute for control plane in all cases is 46 seconds. The maximum time for rerouting has occurred at the case of Raspi 3's Failure. In addition, the maximum required time to reroute for data plane is 30 seconds. The results of restoration time confirm that the predefined forwarding rules for the alternative routes work as intended. However, based on the physical location, the performance of rerouting can vary if rerouting is based on the predefined alternative routes. In our work, each wireless mesh node is placed high enough which has less possibility to be blocked line-of-sight by a car. However, the wireless link between wireless mesh nodes and Gateway 1 can be blocked by a big bus or the wireless link between wireless mesh nodes can also be blocked by cars if a wireless mesh node is not placed at the high place. Therefore, we recommend that the predefined rules for rerouting should be changed accordingly based on the physical location of outdoor SDWMN network or in the future an adaptive routing should be aimed at instead for a more robust deployment.

Chapter 7

Conclusion

In this thesis, we have designed the prototype of outdoor SDWMN testbed for road traffic monitoring network on Phaya Thai road between Rama 1 road and Rama 4 road by using Raspberry Pi. The main purpose is to apply the programmability of SDN to build the wireless network, where routing function can be programmed at the application layer of RYU controller. By using the strategy of mesh networking, captured images from Raspberry Pi's camera can be sent in near real-time through the wireless ad-hoc routes which can save the operational cost for sending data. In this prototype network, the routing functionalities are implemented as predefined forwarding rules for the primary route and alternative route which are based on the minimum-hop-path. The primary route is installed by predefined forwarding rules at the bootstrapping stage in all wireless nodes. The implemented rerouting application will assign the predefined backup rules to the respective wireless mesh nodes to build the alternative routes for rerouting by using standard OpenFlow configuration request messages. In-band control scenario is applied in SDWMN and therefore, the primary route and the alternative route are required to be considered for both the control plane and data plane over a single wireless network interface.

Firstly, we have designed and developed all components in preparation for the actual installation SDWMN testbed on Phaya Thai road. In the preparation, both software and hardware parts have been carried out. The software parts include the installation of Open-Vswitch, RYU, a driver for an external WiFi adapter in all wireless nodes and routing for outdoor SDWMN. Linux kernel version 4.4 has been used with the driver for an applied antenna in this thesis. A waterproof box is designed for installation on the crossover bridges on Phaya Thai road.

After preparation has been done, we set up the small-scale SDWMN testbed on Phaya Thai road between Rama 1 road and Rama 4 road. The total distance between two gateways is 1100 meters. On Phaya Thai road, the average distance between adjacent crossover bridges is 250-350 meters. Two gateways are installed at the traffic police boxes and two wireless mesh nodes are installed at each crossover bridge on Phaya Thai road.

The testing for network performance has been performed in order to investigate for the characteristic of SDN based outdoor wireless network. Firstly, we have measured TCP throughput, UDP throughput, ICMP packet with packet size 1456 bytes in 100 meters, 200 meters, 300 meters, and 400 meters. We make the measurement in different distances and the result confirms that one-hop distance at the outdoor SDWMN can be sufficiently increased up to 400 meters for road traffic monitoring application.

Secondly, the outdoor wireless characteristic has been investigated from the implemented outdoor SDWMN on Phaya Thai road. From an investigation result of performing the network performance for data plane traffic, we have learned about the impacts of obstacles such as buses, trees, and trucks which block the wireless signal. The results of daytime and

nighttime comparison show that we need to carefully design the pattern of routing based on the physical location of outdoor and placing the wireless nodes in order to avoid the obstacles to get the better network performance.

Thirdly, we have integrated the intended traffic monitoring application and SDWMN network and run the traffic monitoring application on SDWMN network for 16 hours on 26th November 2018. The status of the control plane while traffic monitoring application is being operated is quite stable but the control plane starts being fluctuated after 6 AM when the trend of the density of vehicle lead to be increased. Due to the low ambient temperature in the winter season of Thailand, device temperature of wireless mesh node can be operated for the whole day. The more investigation is required for the temperature of the wireless mesh node, especially in the summer season.

Finally, the investigation for rerouting experiment is tested by rebooting the wireless mesh nodes for three times. From the real measurement, the maximum time for rerouting the control plane is 46 seconds and maximum time for rerouting the data plane is 30 seconds. The predefined forwarding rules for the primary routes and the alternative routes are still effective for this such a kind of small-scale testbed. However, the forwarding rules should be changed to dynamic forwarding rules with the consideration of wireless link status to be more robust deployment when small-scale of existing SDWMN testbed is increase to a large-scale network. This work confirms that predefined routing can be operated well in a small-scale testbed. However, dynamic routing should be changed when the scale of the network is increased. In a large-scale network, the control plane status cannot be operated well if the number of hops is increased from RYU controller. Therefore, instead of placing RYU controller in only one gateway, RYU controller should be placed at the cloud or the scenario of using multiple RYU controller should be considered in a future work.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker and J. Turner, "OpenFlow: enabling innovation in campus networks," ACM SIGCOMM Computer Communication Review, pp. 69-74, 2008.

[2] Open Network Foundation,"Software Defined Networking: The new norm for networks," ONF White Paper, 2012.

[3] M. E. M. Campista, P. M. Esposito, I. M. Moraes, L. H. M. K. Costa, O. C. M. B. Duarte, U. F. D. G. Passos, C. V. N. D. Albuquerque, D. C. M. Saade and U. F. F. M. G. Rubinstein, "Routing metrics and protocols for wireless mesh networks," IEEE Network 22, No. 1, pp. 6-12, 2008.

[4] WiFi [Online]. Available from : https://www.wi-fi.org/ [July 8, 2018].

[5] WIRESHARK [Online]. Available from : https://www.wireshark.org/ [July 8, 2018].

[6] ONOS [Online]. Available from : https://onosproject.org/ [July 12, 2018].

[7] A. Neumann, C. E. Aichele and M. Lindner, "B.A.T.M.A.N. Status Report," [Online] . Available from : http://downloads.open-mesh.org/batman/papers/batman-status.pdf , 2007.

[8] Open vSwitch [Online]. Available from : http://openvswitch.org [April 10, 2018].

[9] NS3 [Online]. Available from : https://www.nsnam.org/ [July 12, 2018].

[10] KAFKA [Online]. Available from : https://kafka.apache.org/ [Dec 5, 2018].

[11] NS2 [Online]. Available from : https://www.isi.edu/nsnam/ns/ [July 12, 2018].

[12] ZigBee [Online]. Available from : https://www.zigbee.org/ [July 12, 2018].

[13] Raspberry Pi 3 [Online]. Available from : https://www.raspberrypi.org/ [August 18, 2018].

[14] Intel®NUC7i7BNH [Online]. Available from : https://www.intel.com/content/www/us/en/products/boards-kits/nuc/kits/nuc7i7bnh.html [August 18, 2018].

[15] Ubunut-Mate [Online]. Available from : https://ubuntu-mate.org/ [August 18, 2018].

[16] Mininet [Online]. Available from : http://mininet.org/ [July 12, 2018].

[17] Ben. P, B. Lantz and B. Heller, "OpenFlow switch specification, version 1.3.0," Open Networking Foundation (2012).

[18] Open Vswitch Release 2.9.2 [Online]. Available from : https://media.readthedocs.org/pdf/openvswitch/stable/openvswitch.pdf [August 19, 2018].

[19] RYU SDN Framework [Online]. Available from : https://osrg.github.io/ryu-book/en/Ryubook.pdf [April 20,2018].

[20] PC Engines Single Board Computer [Online]. Available from : http://www.pcengines.ch/alix3d3.htm [August 8,2018].

[21] iPerf [Online]. Available from : https://iperf.fr/ [July 13, 2018].

[22] P. Enns, "NETCONF Configuration Protocol," Juniper Networks, RFC 4741, 2006.

[23] OF-CONFIG 1.2: OpenFlow Management and Configuration Protocol. http://goo.gl/1JnFWN, 2014.

[24] EDUP EP-AC1605 [Online]. Available from : http://www.szedup.com/ [July 14, 2018].

[25] M. Seyedzadegan, M. Othman, B. M. Ali and S. Subramaniam, "Wireless mesh networks: WMN overview, WMN architecture.," International Conference on Communication Engineering and Networks IPCSIT, Vol. 19, pp. 12-18, 2011.

[26] M. Antikainen, T. Aura and M. Särelä, "Spook in your network: Attacking an SDN with a compromised OpenFlow switch," Nordic Conference on Secure IT Systems, pp. 229-244, 2014.

[27] D. B. Green and M. S. Obaidat, "An accurate line of sight propagation performance model for ad-hoc 802.11 wireless LAN (WLAN) devices," IEEE International Conference on Communications, pp. 3424-3428, 2002.

[28] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum and L. Viennot, "Optimized link state routing protocol for ad hoc networks," IEEE International Multi Topic Conference (IEEE INMIC 2001), pp. 62-68, 2001.

[29] C. E. Perkins and E. M. Royer, "Ad-hoc on-demand distance vector routing," 2nd IEEE Workshop on Mobile Computing Systems and Applications, pp. 90-100, 1999.

[30] T. Oda, D. Elmazi, M. Yamada, R. Obukata, L. Barolli and M. Takizawa, "Experimental results of a Raspberry Pi based WMN testbed in indoor environment: a comparison study of LoS and NLoS scenarios," 19th International Conference on Network-Based Information Systems (NBiS), pp. 9-14, 2016.

[31] G. Bianchi, F. Formisano and D. Giustiniano, "802.11 b/g link level measurements for an outdoor wireless campus network," International Symposium on World of Wireless, Mobile and Multimedia Networks, pp. 525-530, 2006.

[32] N. M. Anas, F. K. Hashim, H. Mohamad, M. H. Baharudin and M. P. Sulong, "Performance analysis of outdoor wireless mesh network using BATMAN advanced," IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp. 1-4, 2015.

[33] K. C. Lan, Z. Wang, R. Berriman, T. Moors and M. Hassan, "Implementation of a wireless mesh network testbed for traffic control," 16th International Conference on Computer Communications and Networks, pp. 1022-1027, 2007.

[34] D. Wu, D. Gupta and P. Mohapatra, "QuRiNet: A wide-area wireless mesh testbed for research and experimental evaluations," 2nd International Conference on COMmunication Systems and NETworks (COMSNETS 2010), pp. 1221-1237, 2010.

[35] P. Dely, K. Andreas and B. Nico, "Openflow for wireless mesh networks," 20th International Conference on Computer Communications and Networks (ICCCN), pp. 1-6, 2011.

[36] A. Detti, C. Pisa, S. Salsano and N. Blefari-Melazzi, "Wireless mesh software defined networks (wmSDN)," IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), pp.89-95, 2013.

[37] W. J. Lee, J. W. Shin, H. Y. Lee and M. Y. Chung, "Testbed implementation for routing WLAN traffic in software defined wireless mesh network," 8th International Conference on Ubiquitous and Future Networks(ICUFN), pp. 1052-1055, 2016.

[38] H. Yang, B. Chen and P. Fu, "OpenFlow-based load balancing for wireless mesh network," International Conference on Cloud Computing and Security, pp. 368-379, 2015.

[39] P. Patil, A. Hakiri, Y. Barve and A. Gokhale, "Enabling software-defined networking for wireless mesh networks in smart environments," IEEE 15th International Symposium on Network Computing and Applications (NCA), pp. 153-157, 2016.

[40] K. P. Arun, A. Chakraborty and B. S. Manoj, "Communication overhead of an openflow wireless mesh network," IEEE International Conference on Advanced Networks and Telecommuncations Systems (ANTS), pp. 1-6, 2014.

[41] A. V. Mamidi, S. Babu and B. S. Manoj, "Dynamic multi-hop switch handoffs in software defined wireless mesh networks," IEEE International Conference on Advanced Networks and Telecommuncations Systems (ANTS), pp. 1-6, 2015.

[42] K. Bao, J. D. Matyjas, F. Hu and S. Kumar, "Intelligent software-defined mesh networks with link-failure adaptive traffic balancing," IEEE Transactions on Cognitive Communications and Networking, Vol. 4, No. 2, pp. 266-276, 2018.

[43] O. Salman, I. H. Elhajj, A. Kayssi and A. Chehab, "SDN controllers: A comparative study," 18th Mediterranean Electrotechnical Conference (MELECON), pp. 1-6, 2016.

[44] S. Y. Htet, K. Leevangtou, P. M. Thet, K. Kawila, and C. Aswakul, "Design of Medium-Range Outdoor Wireless Mesh Network with Open-Flow Enabled Raspberry Pi,"

33rd International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC), pp. 192-195, 2018.

[45] IEEE 802.11 Working Group, "IEEE standard for information technology - telecommunications and information exchange between systems local and metropolitan area networks - specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications," IEEE Std 802.11 (2016).

Appendices

Appendix A

Network Configuration of Software-Defined Wireless Mesh Network (SDWMN)

```
#Network Configuration in Raspi 1 to set wireless interface as ad-hoc mode
sudo nano /etc/network/interfaces

auto lo
iface lo inet loopback
#Configure Wireless Ad-Hoc in Linux
auto wlan0
iface wlan0 inet static
address 10.0.0.1
netmask 255.0.0.0
wireless-channel 132
wireless-mode ad-hoc
wireless-essid 222
```

```
#Network Configuration in Raspi 2 to set wireless interface as ad-hoc mode
sudo nano /etc/network/interfaces

auto lo
iface lo inet loopback
#Configure Wireless Ad-Hoc in Linux
auto wlan0
iface wlan0 inet static
address 10.0.0.2
netmask 255.0.0.0
wireless-channel 132
wireless-mode ad-hoc
wireless-essid 222
```

```
#Network Configuration in Raspi 3 to set wireless interface as ad-hoc mode
sudo nano /etc/network/interfaces

auto lo
iface lo inet loopback
#Configure Wireless Ad-Hoc in Linux
auto wlan0
iface wlan0 inet static
address 10.0.0.3
netmask 255.0.0.0
wireless-channel 132
wireless-mode ad-hoc
wireless-essid 222
```

```
#Network Configuration in Raspi 4 to set wireless interface as ad-hoc mode
sudo nano /etc/network/interfaces

auto lo
iface lo inet loopback
#Configure Wireless Ad-Hoc in Linux
```

```
auto wlan0
iface wlan0 inet static
address 10.0.0.4
netmask 255.0.0.0
wireless-channel 132
wireless-mode ad-hoc
wireless-essid 222
```

```
#Network Configuration in Raspi 5 to set wireless interface as ad-hoc mode
sudo nano /etc/network/interfaces

auto lo
iface lo inet loopback
#Configure Wireless Ad-Hoc in Linux
auto wlan0
iface wlan0 inet static
address 10.0.0.5
netmask 255.0.0.0
wireless-channel 132
wireless-mode ad-hoc
wireless-essid 222
```

```
#Network Configuration in Raspi 6 to set wireless interface as ad-hoc mode
sudo nano /etc/network/interfaces

auto lo
iface lo inet loopback
#Configure Wireless Ad-Hoc in Linux
auto wlan0
iface wlan0 inet static
address 10.0.0.6
netmask 255.0.0.0
wireless-channel 132
wireless-mode ad-hoc
wireless-essid 222
```

```
#Network Configuration in Gateway 1 to set wireless interface as ad-hoc mode
sudo nano /etc/network/interfaces

auto lo
iface lo inet loopback
#Configure Wireless Ad-Hoc in Linux
auto wlan0
iface wlan0 inet static
address 10.0.0.8
netmask 255.0.0.0
wireless-channel 132
wireless-mode ad-hoc
wireless-essid 222
```

```
#Network Configuration in Gateway 2 to set wireless interface as ad-hoc mode
sudo nano /etc/network/interfaces

auto lo
iface lo inet loopback
#Configure Wireless Ad-Hoc in Linux
auto wlan0
iface wlan0 inet static
```

```
address 10.0.0.9
netmask 255.0.0.0
wireless-channel 132
wireless-mode ad-hoc
wireless-essid 222
```

Appendix B

Installing Necessary Package To Develop SDWMN

```
#Gateway1 (RYU controller)
#Installing RYU application in Ubuntu 16.04 (Kernel Version 4.4)
sudo apt-get install python-pip
sudo pip install ryu
#Update the installed packages
sudo apt-get update
#

\begin{lstlisting}
#In all wireless nodes
#Installing openvswitch in all wireless nodes
sudo apt-get install openvswitch-switch
#Update the installed packages
sudo apt-get update
```

```
#In all wireless mesh nodes
#Installation for EDUP EP-AC1605 in Raspberry Pi 3 (for Ubuntu Mate with Kernel
    Version 4.4.38-v7+)
Reference from https://github.com/jurobystricky/Netgear-A6210
sudo apt-get install git raspberrypi-kernel-headers
git clone https://github.com/jurobystricky/Netgear-A6210.git
cd Netgear-A6210
make
sudo make install
#Update the installed packages
sudo apt-get update
#For testing for TCP/UDP Iperf
sudo apt-get install iperf3
```

```
#In gateway1 and gateway 2
#Installation for EDUP EP-AC1605 in two gateways (for Ubuntu with Kernel Version 4.4)
Reference from https://github.com/jurobystricky/Netgear-A6210
git clone https://github.com/jurobystricky/Netgear-A6210.git
cd Netgear-A6210
make
sudo make install
#Update the installed packages
sudo apt-get update
#For testing for TCP/UDP Iperf
sudo apt-get install iperf3
```

Appendix C

Python Program for Monitoring Temperature of Wireless Mesh Node

```
#This program is written by Soe Ye Htet from Chulalongkorn University
#This program is to monitor the device temperature of wireless mesh node

import os #os package is to execute the linux command line in python program
import time

def reboot():#To reboot Raspberry Pi
    os.system('sudo reboot')

def test():#TO monitor the temperature
    os.popen("vcgencmd measure_temp >>
    /home/admin3/Desktop/rrtresult/temp26_11_2018.txt")
    os.popen("date >> /home/admin3/Desktop/rrtresult/temp26_11_2018.txt")
    temp=os.popen("vcgencmd measure_temp|cut -c6-9").readline()
    if temp<=str(80):
        print("Raspberry Pi's Temperature is ok")
    else:
        time.sleep(10)
  os.popen("echo Device has bee restart >>
    /home/admin3/Desktop/rrtresult/temp26_11_2018.txt")
        if __name__ == "__main__":
            reboot()
try:
    while True:#To execute forever
        if __name__ == "__main__":
            time.sleep(20)
            test()

except:
    print("Keyboard Error")
```

Listing C.1: Temperature monitoring program at wireless mesh node

```
#To execute the temperature program at bootstrapping stage
sudo crontab -e
@reboot sudo python /home/admin3/Desktop/pythonprogram/final/temperature.py &
#admin3 is the username of raspberry pi
```

Appendix D

Developing Predefined Forwarding Rules For Primary Route in OpenVswitch of Wireless Nodes

```
#To install the Primary OpenFlow Rules in Raspi 1
sudo nano /etc/rc.local
#rc.local file is to execute the linux command line in bootstrapping stage
sleep 3 #sleep is required to make sure the command line inside the rc.local file execute at the bootstrapping stage
sudo ovs-vsctl --if-exists del-br br0
#bridge is added to OpenVswitch
sudo ovs-vsctl add-br br0
sudo ovs-vsctl set bridge br0 other-config:datapath-id=1000000000000001
#Configure OpenVswitch in Userspace of Linux
sudo ovs-vsctl set bridge br0 datapath_type=netdev #Set OpenVswitch in userspace
#added wireless interface under bridge in OpenVswitch
sudo ovs-vsctl add-port br0 wlan0 -- set Interface wlan0 ofport_request=1
sudo ifconfig br0 10.0.0.1 netmask 255.0.0.0 up
sudo ifconfig wlan0 0
sudo iptables -A INPUT -i wlan0 -j DROP #For only OpenVswitch in userspace
sudo iptables -A FORWARD -i wlan0 -j DROP #For only OpenVswitch in userspace
#Connect to RYU controller
sudo ovs-vsctl set-controller br0 tcp:10.0.0.8:6633
sudo ovs-vsctl set controller br0 connection-mode=out-of-band
sudo ovs-vsctl set-fail-mode br0 secure
#Receive the incoming traffic to Raspi 1 (10.0.0.1) from Raspi 2, Raspi 4 and Gateway 1
sudo ovs-ofctl add-flow br0 arp,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:4b,arp_tpa=10.0.0.1,actions=LOCAL
sudo ovs-ofctl add-flow br0 arp,priority=100,in_port=1,dl_src=e8:4e:06:5f:47:59,arp_tpa=10.0.0.1,actions=LOCAL
sudo ovs-ofctl add-flow br0 arp,priority=100,in_port=1,arp_spa=10.0.0.4,actions=LOCAL
sudo ovs-ofctl add-flow br0 ip,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:4b,nw_dst=10.0.0.1,actions=LOCAL
sudo ovs-ofctl add-flow br0 ip,priority=100,in_port=1,dl_src=e8:4e:06:5f:47:59,nw_dst=10.0.0.1,actions=LOCAL
sudo ovs-ofctl add-flow br0 ip,priority=100,in_port=1,nw_src=10.0.0.4,actions=LOCAL
#Send the packet from Raspi 1 to other wireless node
sudo ovs-ofctl add-flow br0 arp,priority=100,in_port=LOCAL,arp_tpa=10.0.0.8,actions=output:1
sudo ovs-ofctl add-flow br0 arp,priority=100,in_port=LOCAL,arp_tpa=10.0.0.4,actions=output:1
sudo ovs-ofctl add-flow br0 arp,priority=90,in_port=LOCAL,arp_spa=10.0.0.1,actions="resubmit(,4)"
sudo ovs-ofctl add-flow br0 ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.8,actions=output:1
sudo ovs-ofctl add-flow br0 ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.4,actions=output:1
sudo ovs-ofctl add-flow br0 ip,priority=90,in_port=LOCAL,nw_src=10.0.0.1,actions="resubmit(,4)"
#Relay the incoming traffic to other wireless nodes not to Raspi 1
sudo ovs-ofctl add-flow br0 arp,priority=90,in_port=1,dl_src=e8:4e:06:5f:47:59,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0 arp,priority=90,in_port=1,dl_src=e8:4e:06:40:d3:4b,actions="resubmit(,4)"
```

```
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=1,dl_src=e8:4e:06:40:d3:4b,actions="resubmit(,4)"
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=1,dl_src=e8:4e:06:5f:47:59,actions="resubmit(,3)"
#Table 3 is to rewrite the destinatio MAC address into Gateway 1's MAC address
sudo ovs-ofctl add-flow br0
    table=3,actions=mod_dl_dst:e8:4e:06:40:d3:4b,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 4 is to rewrite the destinatio MAC address into Raspi 2's MAC address
sudo ovs-ofctl add-flow br0
    table=4,actions=mod_dl_dst:e8:4e:06:5f:47:59,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 5 is to forward to wireless interface
sudo ovs-ofctl add-flow br0 table=5,actions=output:1
#To prevent the infinite loop
sudo ovs-ofctl add-flow br0 priority=1,in_port=1,actions=drop
sudo ovs-vsctl set bridge br0 protocol=OpenFlow10,OpenFlow11,OpenFlow12,OpenFlow13
sudo sysctl -p
exit 0
```

```
#To install the Primary OpenFlow Rules in Raspi 2
sudo nano /etc/rc.local
#rc.local file is to execute the linux command line in bootstrapping stage
sleep 3
sudo ovs-vsctl --if-exist del-br br0
#bridge is added to OpenVswitch
sudo ovs-vsctl add-br br0
sudo ovs-vsctl set bridge br0 other-config:datapath-id=1000000000000002
#Configure OpenVswitch in Userspace of Linux
sudo ovs-vsctl set bridge br0 datapath_type=netdev
#added wireless interface under bridge in OpenVswitch
sudo ovs-vsctl add-port br0 wlan0 -- set Interface wlan0 ofport_request=1
sudo ifconfig wlan0 0
sudo ifconfig br0 10.0.0.2 netmask 255.0.0.0 up
sudo iptables -A INPUT -i wlan0 -j DROP #For only OpenVswitch in userspace
sudo iptables -A FORWARD -i wlan0 -j DROP#For only OpenVswitch in userspace
#Connect to RYU controller
sudo ovs-vsctl set-controller br0 tcp:10.0.0.8:6633
sudo ovs-vsctl set controller br0 connection-mode=out-of-band
sudo ovs-vsctl set-fail-mode br0 secure
sudo ovs-vsctl set bridge br0 stp_enable=true
#Receive the incoming traffic to Raspi 2 (10.0.0.2) from Raspi 1, Raspi 5 and Raspi 3
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:40:dc:62,arp_tpa=10.0.0.2,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:5e:6b:09,arp_tpa=10.0.0.2,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:7f,arp_tpa=10.0.0.2,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,arp_spa=10.0.0.10,arp_tpa=10.0.0.2,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:40:dc:62,nw_dst=10.0.0.2,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:5e:6b:09,nw_dst=10.0.0.2,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:7f,nw_dst=10.0.0.2,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,nw_src=10.0.0.10,nw_dst=10.0.0.2,actions=LOCAL
#Send the packet from Raspi 2 to other wireless node
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=LOCAL,arp_spa=10.0.0.2,actions="resubmit(,1)"
```

```
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.1,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.3,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.5,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=LOCAL,nw_src=10.0.0.2,actions="resubmit(,1)"
#Relay the incoming traffic to other wireless nodes not to Raspi 2
sudo ovs-ofctl add-flow br0
    arp,priority=90,in_port=1,dl_src=e8:4e:06:40:d3:7f,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    arp,priority=90,in_port=1,dl_src=e8:4e:06:5e:6b:09,actions="resubmit(,4)"
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=1,dl_src=e8:4e:06:40:d3:7f,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=1,dl_src=e8:4e:06:5e:6b:09,actions="resubmit(,4)"
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:7f,nw_dst=10.0.0.8,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:5e:6b:09,nw_dst=10.0.0.3,actions="resubmit(,4)"
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:5e:6b:09,nw_dst=10.0.0.9,actions="resubmit(,4)"
#Table 1 is to rewrite the destinatio MAC address into broadcast MAC address
sudo ovs-ofctl add-flow br0
    table=1,actions=mod_dl_dst:ff:ff:ff:ff:ff:ff,"resubmit(,5)"
#Table 2 is to rewrite the destinatio MAC address into Raspi 5's MAC address
sudo ovs-ofctl add-flow br0
    table=2,actions=mod_dl_dst:e8:4e:06:40:dc:62,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 3 is to rewrite the destinatio MAC address into Raspi 1's MAC address
sudo ovs-ofctl add-flow br0
    table=3,actions=mod_dl_dst:e8:4e:06:5e:6b:09,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 4 is to rewrite the destinatio MAC address into Raspi 3's MAC address
sudo ovs-ofctl add-flow br0
    table=4,actions=mod_dl_dst:e8:4e:06:40:d3:7f,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 5 is to forward to wireless interface
sudo ovs-ofctl add-flow br0 table=5,actions=output:1
#To prevent the infinite loop
sudo ovs-ofctl add-flow br0 priority=1,in_port=1,actions=drop
sudo sysctl -p
exit 0
```

```
#To install the Primary OpenFlow Rules in Raspi 3
sudo nano /etc/rc.local
sleep 3
sudo ovs-vsctl --if-exists del-br br0
#bridge is added to OpenVswitch
sudo ovs-vsctl add-br br0
sudo ovs-vsctl set bridge br0 other-config:datapath-id=1000000000000003
#Configure OpenVswitch in Userspace of Linux
sudo ovs-vsctl set bridge br0 datapath_type=netdev
#added wireless interface under bridge in OpenVswitch
sudo ovs-vsctl add-port br0 wlan0 -- set Interface wlan0 ofport_request=1
sudo ifconfig wlan0 0
sudo ifconfig br0 10.0.0.3 netmask 255.0.0.0 up
sudo iptables -A INPUT -i wlan0 -j DROP #For only OpenVswitch in userspace
sudo iptables -A FORWARD -i wlan0 -j DROP  #For only OpenVswitch in userspace
#Connect to RYU controller
sudo ovs-vsctl set-controller br0 tcp:10.0.0.8:6633
```

```
sudo ovs-vsctl set controller br0 connection-mode=out-of-band
sudo ovs-vsctl set-fail-mode br0 secure
sudo ovs-vsctl set bridge br0 protocol=OpenFlow10,OpenFlow11,OpenFlow12,OpenFlow13
#Receive the incoming traffic to Raspi 3 (10.0.0.3) from Raspi 2, Raspi 6 and
    Gateway 2
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:5f:47:59,arp_tpa=10.0.0.3,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:40:94:20,arp_tpa=10.0.0.3,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:5e:6a:b1,arp_tpa=10.0.0.3,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:5f:47:59,nw_dst=10.0.0.3,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:40:94:20,nw_dst=10.0.0.3,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:5e:6a:b1,nw_dst=10.0.0.3,actions=LOCAL
#Send the packet from Raspi 3 to other wireless node
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=LOCAL,arp_spa=10.0.0.3,actions="resubmit(,1)"
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.9,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.2,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.6,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=LOCAL,nw_src=10.0.0.3,actions="resubmit(,1)"
#Relay the incoming traffic to other wireless nodes not to Raspi 3
sudo ovs-ofctl add-flow br0
    arp,priority=90,in_port=1,dl_src=e8:4e:06:5f:47:59,arp_tpa=10.0.0.9,actions="resubmit(,4)"
sudo ovs-ofctl add-flow br0
    arp,priority=90,dl_src=e8:4e:06:5e:6a:b1,in_port=1,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=1,dl_src=e8:4e:06:5f:47:59,nw_dst=10.0.0.9,actions="resubmit(,4)"
sudo ovs-ofctl add-flow br0
    ip,priority=90,dl_src=e8:4e:06:5e:6a:b1,in_port=1,actions="resubmit(,3)"
#Table 1 is to rewrite the destinatio MAC address into broadcast MAC address
sudo ovs-ofctl add-flow br0
    table=1,actions=mod_dl_dst:ff:ff:ff:ff:ff:ff,"resubmit(,5)"
#Table 2 is to rewrite the destinatio MAC address into Raspi 6's MAC address
sudo ovs-ofctl add-flow br0
    table=2,actions=mod_dl_dst:e8:4e:06:40:94:20,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 3 is to rewrite the destinatio MAC address into Raspi 2's MAC address
sudo ovs-ofctl add-flow br0
    table=3,actions=mod_dl_dst:e8:4e:06:5f:47:59,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 4 is to rewrite the destinatio MAC address into Gateway 2's MAC address
sudo ovs-ofctl add-flow br0
    table=4,actions=mod_dl_dst:e8:4e:06:5e:6a:b1,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 5 is to forward to wireless interface
sudo ovs-ofctl add-flow br0 table=5,actions=output:1
#To prevent the infinite loop
sudo ovs-ofctl add-flow br0 priority=1,in_port=1,actions=drop
sudo sysctl -p
exit 0
```

```
#To install the Primary OpenFlow Rules in Raspi 4
sudo nano /etc/rc.local
sleep 3
```

```
sudo ovs-vsctl --if-exists del-br br0
#Bridge is added to OpenVswitch
sudo ovs-vsctl add-br br0
sudo ovs-vsctl set bridge br0 other-config:datapath-id=1000000000000004
#Configure OpenVswitch in Userspace of Linux
sudo ovs-vsctl set bridge br0 datapath_type=netdev
#Added wireless interface under bridge in OpenVswitch
sudo ovs-vsctl add-port br0 wlan0 -- set Interface wlan0 ofport_request=1
sudo ifconfig wlan0 0
sudo ifconfig br0 10.0.0.4 netmask 255.0.0.0 up
sudo iptables -A INPUT -i wlan0 -j DROP#For only OpenVswitch in userspace
sudo iptables -A FORWARD -i wlan0 -j DROP#For only OpenVswitch in userspace
#Connect to RYU controller
sudo ovs-vsctl set-controller br0 tcp:10.0.0.8:6633
sudo ovs-vsctl set controller br0 connection-mode=out-of-band
sudo ovs-vsctl set-fail-mode br0 secure
sudo ovs-vsctl set bridge br0 stp_enable=true
sudo ovs-vsctl set bridge br0 protocol=OpenFlow10,OpenFlow11,OpenFlow12,OpenFlow13
#Receive the incoming traffic to Raspi 4 (10.0.0.4) from Raspi 1, Raspi 5 and
    Gateway 1
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:4b,nw_dst=10.0.0.4,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:40:dc:62,nw_dst=10.0.0.4,actions=LOCAL
sudo ovs-ofctl add-flow br0 ip,priority=100,in_port=1,nw_src=10.0.0.1,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:4b,arp_tpa=10.0.0.4,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:40:dc:62,arp_tpa=10.0.0.4,actions=LOCAL
sudo ovs-ofctl add-flow br0 arp,priority=100,in_port=1,arp_spa=10.0.0.1,actions=LOCAL
#Send the packet from Raspi 4 to other wireless node
sudo ovs-ofctl add-flow br0
    arp,priority=95,in_port=LOCAL,arp_spa=10.0.0.4,actions="resubmit(,2)"
sudo ovs-ofctl add-flow br0
    ip,priority=95,in_port=LOCAL,nw_dst=10.0.0.8,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=95,in_port=LOCAL,nw_dst=10.0.0.1,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=LOCAL,nw_src=10.0.0.4,actions="resubmit(,4)"
#Relay the incoming traffic to other wireless nodes not to Raspi 4
sudo ovs-ofctl add-flow br0
    arp,priority=90,in_port=1,dl_src=e8:4e:06:40:dc:62,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=1,dl_src=e8:4e:06:40:dc:62,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    arp,priority=90,in_port=1,dl_src=e8:4e:06:40:d3:4b,actions="resubmit(,4)"
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=1,dl_src=e8:4e:06:40:d3:4b,actions="resubmit(,4)"
#Table 2 is to rewrite the destinatio MAC address into broadcast MAC address
sudo ovs-ofctl add-flow br0
    table=2,actions=mod_dl_dst:ff:ff:ff:ff:ff:ff,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 3 is to rewrite the destinatio MAC address into Gateway 1's MAC address
sudo ovs-ofctl add-flow br0
    table=3,actions=mod_dl_dst:e8:4e:06:40:d3:4b,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 4 is to rewrite the destinatio MAC address into Raspi 5's MAC address
sudo ovs-ofctl add-flow br0
    table=4,actions=mod_dl_dst:e8:4e:06:40:dc:62,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 5 is to forward to wireless interface
sudo ovs-ofctl add-flow br0 table=5,actions=output:1
```

```
sudo ovs-ofctl add-flow br0 priority=1,in_port=1,actions=drop
sudo sysctl -p
exit 0
```

```
#To install the Primary OpenFlow Rules in Raspi 5
sudo nano /etc/rc.local
sleep 3
sudo ovs-vsctl --if-exists del-br br0
#Bridge is added to OpenVswitch
sudo ovs-vsctl add-br br0
sudo ovs-vsctl set bridge br0 other-config:datapath-id=1000000000000005
#Configure OpenVswitch in Userspace of Linux
sudo ovs-vsctl set bridge br0 datapath_type=netdev
#Added wireless interface under bridge in OpenVswitch
sudo ovs-vsctl add-port br0 wlan0 -- set Interface wlan0 ofport_request=1
sudo ifconfig wlan0 0
sudo ifconfig br0 10.0.0.5 netmask 255.0.0.0 up
sudo iptables -A INPUT -i wlan0 -j DROP#For only OpenVswitch in userspace
sudo iptables -A FORWARD -i wlan0 -j DROP#For only OpenVswitch in userspace
#Connect to RYU controller
sudo ovs-vsctl set-controller br0 tcp:10.0.0.8:6633
sudo ovs-vsctl set controller br0 connection-mode=out-of-band
sudo ovs-vsctl set-fail-mode br0 secure
sudo ovs-vsctl set bridge br0 stp_enable=true
#Receive the incoming traffic to Raspi 5 (10.0.0.5) from Raspi 2, Raspi 4 and Raspi 6
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:5f:47:59,arp_tpa=10.0.0.5,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:db,arp_tpa=10.0.0.5,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:40:94:20,arp_tpa=10.0.0.5,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:5f:47:59,nw_dst=10.0.0.5,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:db,nw_dst=10.0.0.5,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:40:94:20,nw_dst=10.0.0.5,actions=LOCAL
#Send the packet from Raspi 5 to other wireless node
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.2,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.4,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.6,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=95,in_port=LOCAL,nw_src=10.0.0.5,actions="resubmit(,1)"
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=LOCAL,arp_spa=10.0.0.5,actions="resubmit(,1)"
#Relay the incoming traffic to other wireless nodes not to Raspi 5
sudo ovs-ofctl add-flow br0
    arp,priority=90,in_port=1,dl_src=e8:4e:06:40:94:20,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    arp,priority=90,in_port=1,dl_src=e8:4e:06:40:d3:db,actions="resubmit(,4)"
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=1,dl_src=e8:4e:06:40:94:20,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=1,dl_src=e8:4e:06:40:d3:db,actions="resubmit(,4)"
#Table 1 is to rewrite the destinatio MAC address into broadcast MAC address
sudo ovs-ofctl add-flow br0
```

```
    table=1,actions=mod_dl_dst:ff:ff:ff:ff:ff:ff,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 2 is to rewrite the destinatio MAC address into Raspi 2's MAC address
sudo ovs-ofctl add-flow br0
    table=2,actions=mod_dl_dst:e8:4e:06:5f:47:59,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 3 is to rewrite the destinatio MAC address into Raspi 4's MAC address
sudo ovs-ofctl add-flow br0
    table=3,actions=mod_dl_dst:e8:4e:06:40:d3:db,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 4 is to rewrite the destinatio MAC address into Raspi 6's MAC address
sudo ovs-ofctl add-flow br0
    table=4,actions=mod_dl_dst:e8:4e:06:40:94:20,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 5 is to forward to wireless interface
sudo ovs-ofctl add-flow br0 table=5,actions=output:1
#To prevent the infinite loop
sudo ovs-ofctl add-flow br0 priority=1,in_port=1,actions=drop
sudo ovs-vsctl set bridge br0 protocol=OpenFlow10,OpenFlow11,OpenFlow12,OpenFlow13
sudo sysctl -p
exit 0
```

```
#To install the Primary OpenFlow Rules in Raspi 6
sudo nano /etc/rc.local
sleep 3
sudo ovs-vsctl --if-exists del-br br0
#Bridge is added to OpenVswitch
sudo ovs-vsctl add-br br0
sudo ovs-vsctl set bridge br0 other-config:datapath-id=1000000000000006
#Configure OpenVswitch in Userspace of Linux
sudo ovs-vsctl set bridge br0 datapath_type=netdev
#Added wireless interface under bridge in OpenVswitch
sudo ovs-vsctl add-port br0 wlan0 -- set Interface wlan0 ofport_request=1
sudo ifconfig wlan0 0
sudo iptables -A INPUT -i wlan0 -j DROP#For only OpenVswitch in userspace
sudo iptables -A FORWARD -i wlan0 -j DROP#For only OpenVswitch in userspace
sudo ifconfig br0 10.0.0.6 netmask 255.0.0.0 up
#Connect to RYU controller
sudo ovs-vsctl set-controller br0 tcp:10.0.0.8:6633
sudo ovs-vsctl set controller br0 connection-mode=out-of-band
sudo ovs-vsctl set-fail-mode br0 secure
sudo ovs-vsctl set bridge br0 protocol=OpenFlow10,OpenFlow11,OpenFlow12,OpenFlow13
#Receive the incoming traffic to Raspi 6 (10.0.0.6) from Raspi 3, Raspi 5 and
    Gateway 2
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:7f,arp_tpa=10.0.0.6,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:40:dc:62,arp_tpa=10.0.0.6,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:5e:6a:b1,arp_tpa=10.0.0.6,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:7f,nw_dst=10.0.0.6,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:40:dc:62,nw_dst=10.0.0.6,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:5e:6a:b1,nw_dst=10.0.0.6,actions=LOCAL
#Send the packet from Raspi 6 to other wireless node
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=LOCAL,arp_spa=10.0.0.6,actions="resubmit(,1)"
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.9,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.5,actions=output:1
```

```
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.3,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=95,in_port=LOCAL,nw_src=10.0.0.6,actions="resubmit(,1)"
#To prevent duplicate message from Gateway 2 to Gateway 1
sudo ovs-ofctl add-flow br0
    arp,priority=95,in_port=1,dl_src=e8:4e:06:5e:6a:b1,arp_tpa=10.0.0.8,actions=drop
sudo ovs-ofctl add-flow br0
    ip,priority=95,in_port=1,dl_src=e8:4e:06:5e:6a:b1,nw_dst=10.0.0.8,actions=drop
#Relay the incoming traffic to other wireless nodes not to Raspi 6
sudo ovs-ofctl add-flow br0
    arp,priority=90,in_port=1,dl_src=e8:4e:06:5e:6a:b1,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    arp,priority=90,in_port=1,dl_src=e8:4e:06:40:dc:62,actions="resubmit(,4)"
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=1,dl_src=e8:4e:06:5e:6a:b1,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=1,dl_src=e8:4e:06:40:dc:62,actions="resubmit(,4)"
#Table 1 is to rewrite the destinatio MAC address into broadcast MAC address
sudo ovs-ofctl add-flow br0
    table=1,actions=mod_dl_dst:ff:ff:ff:ff:ff:ff,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 2 is to rewrite the destinatio MAC address into Raspi 3's MAC address
sudo ovs-ofctl add-flow br0
    table=2,actions=mod_dl_dst:e8:4e:06:40:d3:7f,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 3 is to rewrite the destinatio MAC address into Raspi 5's MAC address
sudo ovs-ofctl add-flow br0
    table=3,actions=mod_dl_dst:e8:4e:06:40:dc:62,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 4 is to rewrite the destinatio MAC address into Gateway 2's MAC address
sudo ovs-ofctl add-flow br0
    table=4,actions=mod_dl_dst:e8:4e:06:5e:6a:b1,"load:0->OXM_OF_IN_PORT[],resubmit(,5)"
#Table 5 is to forward to wireless interface
sudo ovs-ofctl add-flow br0 table=5,actions=output:1
#To prevent the infinite loop
sudo ovs-ofctl add-flow br0 priority=1,in_port=1,actions=drop
sudo sysctl -p
exit 0
```

```
#To install the Primary OpenFlow Rules in Gateway 1
sudo nano /etc/rc.local
sleep 3
sudo ovs-vsctl --if-exist del-br br0
#Bridge is added to OpenVswitch
sudo ovs-vsctl add-br br0
#Configure OpenVswitch in Userspace of Linux
sudo ovs-vsctl set bridge br0 datapath_type=netdev
sudo ovs-vsctl set bridge br0 other-config:datapath_id=1000000000000008
#Added wireless interface under bridge in OpenVswitch
sudo ovs-vsctl add-port br0 wlan0 -- set Interface wlan0 ofport_request=1
sudo ifconfig wlan0 0
sudo ifconfig br0 10.0.0.8 netmask 255.0.0.0 up
sudo iptables -A INPUT -i wlan0 -j DROP#For only OpenVswitch in userspace
sudo iptables -A FORWARD -i wlan0 -j DROP#For only OpenVswitch in userspace
#Connect to RYU controller
sudo ovs-vsctl set-controller br0 tcp:10.0.0.8:6633
sudo ovs-vsctl set controller br0 connection-mode=out-of-band
sudo ovs-vsctl set-fail-mode br0 secure
sudo ovs-vsctl set bridge br0 protocol=OpenFlow10,OpenFlow11,OpenFlow12,OpenFlow13
sudo ovs-vsctl set bridge br0 stp_enable=true
#Receive the incoming traffic to Raspi 3 (10.0.0.3) coming from Raspi 1 and Raspi 4
```

```
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:5e:6b:09,arp_tpa=10.0.0.8,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:5e:6b:09,nw_dst=10.0.0.8,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:db,nw_dst=10.0.0.8,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:db,arp_tpa=10.0.0.8,actions=LOCAL
#Send the packet from Gateway1 to other wireless node
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=LOCAL,arp_tpa=10.0.0.4,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=LOCAL,arp_tpa=10.0.0.5,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=LOCAL,arp_tpa=10.0.0.6,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    arp,priority=90,in_port=LOCAL,arp_spa=10.0.0.8,actions="resubmit(,2)"
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.4,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.5,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.6,actions="resubmit(,3)"
sudo ovs-ofctl add-flow br0
    ip,priority=90,in_port=LOCAL,nw_src=10.0.0.8,actions="resubmit(,2)"
#Table 2 is to rewrite the destinatio MAC address into Raspi 1's MAC address
sudo ovs-ofctl add-flow br0
    table=2,actions=mod_dl_dst:e8:4e:06:5e:6b:09,"resubmit(,4)"
#Table 3 is to rewrite the destinatio MAC address into Raspi 4's MAC address
sudo ovs-ofctl add-flow br0
    table=3,actions=mod_dl_dst:e8:4e:06:40:d3:db,"resubmit(,4)"
#Table 4 is to forward to wireless interface
sudo ovs-ofctl add-flow br0 table=4,actions=output:1
#To prevent the infinite loop
sudo ovs-ofctl add-flow br0 priority=1,in_port=1,actions=drop
sudo sysctl -p
exit 0
```

```
#To install the Primary OpenFlow Rules in Gateway 2
sudo nano /etc/rc.local
sleep 3
sudo ovs-vsctl --if-exist del-br br0
#Bridge is added to OpenVswitch
sudo ovs-vsctl add-br br0
sudo ovs-vsctl set bridge br0 other-config:datapath-id=1000000000000009
#Configure OpenVswitch in Userspace of Linux
sudo ovs-vsctl set bridge br0 datapath_type=netdev
#Added wireless interface under bridge in OpenVswitch
sudo ovs-vsctl add-port br0 wlan0 -- set Interface wlan0 ofport_request=1
sudo iptables -A INPUT -i wlan0 -j DROP
sudo iptables -A FORWARD -i wlan0 -j DROP
sudo ifconfig wlan0 0
sudo ifconfig br0 10.0.0.9 netmask 255.0.0.0 up
#Connect to RYU controller
sudo ovs-vsctl set-controller br0 tcp:10.0.0.8:6633
sudo ovs-vsctl set controller br0 connection-mode=out-of-band
sudo ovs-vsctl set-fail-mode br0 secure
#Receive the incoming traffic to Gateway 2 (10.0.0.9)
sudo ovs-ofctl add-flow br0
```

```
    ip,priority=100,in_port=1,dl_src=e8:4e:06:40:94:20,nw_dst=10.0.0.9,actions=LOCAL
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:7f,nw_dst=10.0.0.9,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,arp_spa=10.0.0.10,actions=LOCAL
sudo ovs-ofctl add-flow br0 ip,priority=100,in_port=1,nw_src=10.0.0.10,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:40:d3:7f,arp_tpa=10.0.0.9,actions=LOCAL
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=1,dl_src=e8:4e:06:40:94:20,arp_tpa=10.0.0.9,actions=LOCAL
#Send the packet from Gateway 2 to other wireless node
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.3,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.6,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=100,in_port=LOCAL,nw_dst=10.0.0.10,actions=output:1
sudo ovs-ofctl add-flow br0
    ip,priority=95,in_port=LOCAL,nw_src=10.0.0.9,actions="resubmit(,2)"
sudo ovs-ofctl add-flow br0
    arp,priority=100,in_port=LOCAL,arp_spa=10.0.0.9,actions="resubmit(,2)"
#Table 2 is to rewrite the destinatio MAC address into broadcast MAC address
sudo ovs-ofctl add-flow br0
    table=2,actions=mod_dl_dst:ff:ff:ff:ff:ff:ff,"resubmit(,4)"
#Table 5 is to forward to wireless interface
sudo ovs-ofctl add-flow br0 table=4,actions=output:1
#To prevent the infinite loop
sudo ovs-ofctl add-flow br0 priority=1,in_port=1,actions=drop
sudo sysctl -p
exit 0
```

Appendix E

Developing Rerouting RYU Program

```
#This program is written by Soe Ye Htet from Chulalongkorn University
#This program is for rerouting in outdoor SDWMN testbed in RYU controller
#
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER,
    DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib import hub
import time
import os
#Datapath ID of each wireless node
raspi1=1152921504606846977
raspi2=1152921504606846978
raspi3=1152921504606846979
raspi4=1152921504606846980
raspi5=1152921504606846981
raspi6=1152921504606846982
gateway1=255421810004811
gateway2=1152921504606846985

#MAC addresses of each wireless nodes
r1="e8:4e:06:5e:6b:09"
r2="e8:4e:06:5f:47:59"
r3="e8:4e:06:40:d3:7f"
r4 ="e8:4e:06:40:d3:db"
r5="e8:4e:06:40:dc:62"
r6="e8:4e:06:40:94:20"
gw2="e8:4e:06:5e:6a:b1"
gw1="e8:4e:06:40:d3:4b"

#IP addresses of each wireless node
gw1ip="10.0.0.8"
r1ip="10.0.0.1"
r2ip="10.0.0.2"
r3ip="10.0.0.3"
r4ip="10.0.0.4"
r5ip="10.0.0.5"
r6ip="10.0.0.6"
gw2ip="10.0.0.9"


class node_failure (app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self,*args,**kwargs):
        super(node_failure,self).__init__(*args,**kwargs)
        self.switch_table = {}
        self.datapaths = {}
        self.monitor_thread = hub.spawn(self._monitor)
        #require to send configuration request message in every 8 seconds

```

```
#Define the funtion to add flow rules
    def add_flow(self,datapath,table,priority,match,actions,hard):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        mod =
    parser.OFPFlowMod(datapath=datapath,table_id=table,command=ofproto.OFPFC_ADD,

    priority=priority,match=match,instructions=inst,hard_timeout=hard)
        datapath.send_msg(mod)

#Define the function to add flow rule with the action of gototable
    def add_gototable(self,datapath,table,n,priority,match,hard): #n is a number of
    table
        parser = datapath.ofproto_parser
        ofproto = datapath.ofproto
        inst = [parser.OFPInstructionGotoTable(n)]
        mod =
    parser.OFPFlowMod(datapath=datapath,table_id=table,command=ofproto.OFPFC_ADD,

    priority=priority,match=match,hard_timeout=hard,instructions=inst)
        datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        dp = ev.msg.datapath
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        self.logger.info("Switch_ID %s (IP address %s) is
    connected,1",dp.id,dp.address)

#Define the function to detect when wireless nodes connect to RYU controller or
    leave from RYU controller
    @set_ev_cls(ofp_event.EventOFPStateChange,[MAIN_DISPATCHER, DEAD_DISPATCHER])
    def _state_change_handler(self, ev):
        current_time = time.asctime(time.localtime(time.time()))
        datapath = ev.datapath
        if ev.state == MAIN_DISPATCHER:
            if datapath.id not in self.datapaths:
                self.logger.debug('register datapath: %016x', datapath.id)
                self.logger.info("(Switch ID %s),IP address is connected %s in
    %s,1",datapath.id,datapath.address,current_time)
                self.datapaths[datapath.id] = datapath
                self.logger.info("Current Conneced Switches to RYU controller are
    %s",self.datapaths.keys())
        elif ev.state == DEAD_DISPATCHER:
            if datapath.id in self.datapaths:
                self.logger.debug('unregister datapath: %016x', datapath.id)
                self.logger.info("(Switch ID %s),IP address is leaved %s in %s,0",
    datapath.id, datapath.address,current_time)
                del self.datapaths[datapath.id]
                self.logger.info("Current Conneced Switches to RYU controller are
    %s", self.datapaths.keys())

#Define the function to send configuraion request message in every second
    def _monitor(self):
        while True:
```

```
100             #To send configuration request message only when one of the wireless
    mesh nodes leave from RYU controller
101             if (raspi1 not in self.datapaths or raspi2 not in self.datapaths or
    raspi3 not in self.datapaths or
102                 raspi4 not in self.datapaths or raspi5 not in self.datapaths or
    raspi6 not in self.datapaths):
103                 for datapath in self.datapaths.values():
104                     self.send_get_config_request(datapath)
105             hub.sleep(8)
106
107 #Define the function for configuration request message
108     def send_get_config_request(self, datapath):
109         ofp_parser = datapath.ofproto_parser
110         req = ofp_parser.OFPGetConfigRequest(datapath)
111         datapath.send_msg(req)
112
113 #Define the function to add flow rules with configuration request message
114     @set_ev_cls(ofp_event.EventOFPGetConfigReply, MAIN_DISPATCHER)
115     def get_config_reply_handler(self,ev):
116         current_time = time.asctime(time.localtime(time.time()))
117         datapath = ev.msg.datapath
118         parser = datapath.ofproto_parser
119         self.logger.info('IP address %s sends OFPConfigReply message in %s',
    datapath.address, current_time)
120         if ((raspi1 not in self.datapaths and raspi2 in self.datapaths and raspi3 in
    self.datapaths and raspi4 in self.datapaths and raspi5 in self.datapaths and
    raspi6 in self.datapaths)
121             or (raspi1 not in self.datapaths and raspi2 not in self.datapaths and
    raspi3 in self.datapaths and raspi4 in self.datapaths and raspi5 in
    self.datapaths and raspi6 in self.datapaths)
122             or (raspi1 not in self.datapaths and raspi2 not in self.datapaths and
    raspi3 not in self.datapaths and raspi4 in self.datapaths and raspi5 in
    self.datapaths and raspi6 in self.datapaths)):
123             self.logger.info("case1")
124             local = datapath.ofproto.OFPP_LOCAL
125             if datapath.id == raspi5:
126                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r2,
    arp_tpa = gw1ip)
127                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay to Raspi 4
128
129                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r2,
    ipv4_dst = gw1ip)
130                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay to Raspi 4
131                 #These two rules make the route Raspi 2 to Raspi 4 from Raspi 5
    Raspi 2 - Raspi 5 - Raspi 4 - GW1
132
133                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r4,
    arp_tpa = r2ip)
134                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 4 is to
    relay to Raspi 2
135
136                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r4,
    ipv4_dst = r2ip)
137                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 4 is to
    relay to Raspi 2
138                 #These two rules make the route GW1 to Raspi 2 through the route GW1
    - Raspi 4 - Raspi 5 - Raspi 2
```

```
139
140             if datapath.id == raspi6:
141                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r3,
    arp_tpa = gw1ip)
142                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay to Raspi 5
143
144                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r3,
    ipv4_dst = gw1ip)
145                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay to Raspi 5
146
147                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=gw2,
    arp_tpa=gw1ip)
148                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay to Raspi 5
149
150                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=gw2,
    ipv4_dst=gw1ip)
151                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay to Raspi 5
152
153                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r5,
    arp_tpa = r3ip)
154                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 3
155
156                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r5,
    ipv4_dst = r3ip)
157                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 3
158
159                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r5,
    arp_tpa = gw2ip)
160                 self.add_gototable(datapath, 0, 4, 160, match, 10)#Table 4 is to
    relay Gateway 2
161
162                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r5,
    ipv4_dst = gw2ip)
163                 self.add_gototable(datapath, 0, 4, 160, match, 10)#Table 4 is to
    relay Gateway 2
164
165             elif datapath.id == gateway1: #Gateway1
166                 match = parser.OFPMatch(in_port=local,eth_type=0x0806,arp_tpa=r2ip)
167                 self.add_gototable(datapath, 0, 3, 160, match, 10) #Table 3 is to
    relay Raspi 4
168
169                 match = parser.OFPMatch(in_port=local, eth_type=0x0800,ipv4_dst=r2ip)
170                 self.add_gototable(datapath, 0, 3, 160, match, 10) #Table 3 is to
    relay Raspi 4
171
172                 match = parser.OFPMatch(in_port=local,eth_type=0x0806,arp_tpa=r3ip)
173                 self.add_gototable(datapath, 0, 3, 160, match, 10) #Table 3 is to
    relay Raspi 4
174
175                 match = parser.OFPMatch(in_port=local, eth_type=0x0800,ipv4_dst=r3ip)
176                 self.add_gototable(datapath, 0, 3, 160, match, 10) #Table 3 is to
    relay Raspi 4
177
```

```
178                 match = parser.OFPMatch(in_port=local, eth_type=0x0806, arp_tpa=r2ip)
179                 self.add_gototable(datapath, 0, 3, 160, match, 10)  # Table 3 is to
    relay Raspi 4
180
181                 match = parser.OFPMatch(in_port=local, eth_type=0x0800,
    ipv4_dst=r2ip)
182                 self.add_gototable(datapath, 0, 3, 160, match, 10)  # Table 3 is to
    relay Raspi 4
183
184                 match = parser.OFPMatch(in_port=local, eth_type=0x0806,
    arp_tpa=gw2ip)
185                 self.add_gototable(datapath, 0, 3, 160, match, 10)  # Table 3 is to
    relay Raspi 4
186
187                 match = parser.OFPMatch(in_port=local, eth_type=0x0800,
    ipv4_dst=gw2ip)
188                 self.add_gototable(datapath, 0, 3, 160, match, 10)  # Table 3 is to
    relay Raspi 4
189
190         elif ((raspi2 not in self.datapaths and raspi3 in self.datapaths and raspi1
    in self.datapaths and raspi4 in self.datapaths and raspi5 in self.datapaths and
    raspi6 in self.datapaths)
191               or (raspi2 not in self.datapaths and raspi3 not in self.datapaths and
    raspi1 in self.datapaths and raspi4 in self.datapaths and raspi5 in
    self.datapaths and raspi6 in self.datapaths)):
192             self.logger.info("Case 2")
193             local = datapath.ofproto.OFPP_LOCAL
194             if datapath.id == raspi6:
195                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r3,
    arp_tpa=gw1ip)
196                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay to Raspi 5
197
198                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r3,
    ipv4_dst=gw1ip)
199                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay to Raspi 5
200
201                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=gw2,
    arp_tpa=gw1ip)
202                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay to Raspi 5
203
204                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=gw2,
    ipv4_dst=gw1ip)
205                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay to Raspi 5
206
207                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r5,
    arp_tpa=r3ip)
208                 self.add_gototable(datapath, 0, 2, 160, match, 10)  # Table 2 is to
    relay Raspi 3
209
210                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r5,
    ipv4_dst=r3ip)
211                 self.add_gototable(datapath, 0, 2, 160, match, 10)  # Table 2 is to
    relay Raspi 3
212
```

```
213                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r5,
    arp_tpa=gw2ip)
214                 self.add_gototable(datapath, 0, 4, 160, match, 10) #Table 4 is to
    relay Gateway 2
215 
216                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r5,
    ipv4_dst=gw2ip)
217                 self.add_gototable(datapath, 0, 4, 160, match, 10) #Table 4 is to
    relay Gateway 2
218 
219             if ev.msg.datapath.id == gateway1: #Gateway1
220                 match = parser.OFPMatch(in_port=local,eth_type=0x0806,arp_tpa=r3ip)
221                 self.add_gototable(datapath, 0, 3, 160, match, 10) #Table 3 is to
    relay Raspi 4
222 
223                 match = parser.OFPMatch(in_port=local, eth_type=0x0800,ipv4_dst=r3ip)
224                 self.add_gototable(datapath, 0, 3, 160, match, 10) #Table 3 is to
    relay Raspi 4
225 
226                 match = parser.OFPMatch(in_port=local,eth_type=0x0806,arp_tpa=gw2ip)
227                 self.add_gototable(datapath, 0, 3, 160, match, 10) #Table 3 is to
    relay Raspi 4
228 
229                 match = parser.OFPMatch(in_port=local,
    eth_type=0x0800,ipv4_dst=gw2ip)
230                 self.add_gototable(datapath, 0, 3, 160, match, 10) #Table 3 is to
    relay Raspi 4
231 
232         elif (raspi3 not in self.datapaths and raspi1 in self.datapaths and raspi2
    in self.datapaths and raspi4 in self.datapaths and raspi5 in self.datapaths and
    raspi6 in self.datapaths):
233             self.logger.info("Case 3")
234             local = datapath.ofproto.OFPP_LOCAL
235             if datapath.id == raspi6:
236                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=gw2,
    arp_tpa=gw1ip)
237                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay to Raspi 5
238 
239                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=gw2,
    ipv4_dst=gw1ip)
240                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay to Raspi 5
241 
242                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r5,
    arp_tpa=gw2ip)
243                 self.add_gototable(datapath, 0, 4, 160, match, 10)#Table 4 is to
    relay Gateway 2
244 
245                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r5,
    ipv4_dst=gw2ip)
246                 self.add_gototable(datapath, 0, 4, 160, match, 10)#Table 4 is to
    relay Gateway 2
247 
248             if datapath.id == gateway1:
249                 match = parser.OFPMatch(in_port=local,eth_type=0x0806,arp_tpa=gw2ip)
250                 self.add_gototable(datapath, 0, 3, 160, match, 10) #Table 3 is to
    relay Raspi 4
251 
```

```
252                 match = parser.OFPMatch(in_port=local,
    eth_type=0x0800,ipv4_dst=gw2ip)
253                 self.add_gototable(datapath, 0, 3, 160, match, 10) #Table 3 is to
    relay Raspi 4
254 
255             elif ((raspi4 not in self.datapaths and raspi5 in self.datapaths and raspi6
    in self.datapaths and raspi1 in self.datapaths and raspi2 in self.datapaths and
    raspi3 in self.datapaths)
256                   or (raspi4 not in self.datapaths and raspi5 not in self.datapaths and
    raspi6 in self.datapaths and raspi1 in self.datapaths and raspi2 in
    self.datapaths and raspi3 in self.datapaths)
257                   or (raspi4 not in self.datapaths and raspi5 not in self.datapaths and
    raspi6 not in self.datapaths and raspi1 in self.datapaths and raspi2 in
    self.datapaths and raspi3 in self.datapaths)):
258                 self.logger.info("Case 4")
259                 local = datapath.ofproto.OFPP_LOCAL
260                 if datapath.id == raspi2:
261                     match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r5,
    arp_tpa = gw1ip)
262                     self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 1
263 
264                     match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r5,
    ipv4_dst = gw1ip)
265                     self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 1
266 
267                     match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r1,
    arp_tpa = r5ip)
268                     self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 5
269 
270                     match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r1,
    ipv4_dst = r5ip)
271                     self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 5
272 
273                     match = parser.OFPMatch(in_port=1, eth_type=0x0806, arp_spa=gw2ip,
    arp_tpa=r5ip)
274                     self.add_flow(datapath, 0, 160, match, [], 10)
275 
276                     match = parser.OFPMatch(in_port=1, eth_type=0x0800, ipv4_src=gw2ip,
    ipv4_dst=r5ip)
277                     self.add_flow(datapath, 0, 160, match, [], 10)
278 
279                 if datapath.id == raspi3:
280                     match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r6,
    arp_tpa = gw1ip)
281                     self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 2
282 
283                     match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r6,
    ipv4_dst = gw1ip)
284                     self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 2
285 
286                     match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r2,
    arp_tpa = r6ip)
```

```
287                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 6
288
289                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r2,
    ipv4_dst = r6ip)
290                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 6
291
292             if datapath.id == gateway1:
293                 match = parser.OFPMatch(in_port=local,eth_type=0x0806,arp_tpa=r6ip)
294                 self.add_gototable(datapath, 0, 2, 160, match, 10) #Table 2 is to
    relay Raspi 1
295
296                 match = parser.OFPMatch(in_port=local, eth_type=0x0800,ipv4_dst=r6ip)
297                 self.add_gototable(datapath, 0, 2, 160, match, 10) #Table 2 is to
    relay Raspi 1
298
299                 match = parser.OFPMatch(in_port=local,eth_type=0x0806,arp_tpa=r5ip)
300                 self.add_gototable(datapath, 0, 2, 160, match, 10) #Table 2 is to
    relay Raspi 1
301
302                 match = parser.OFPMatch(in_port=local, eth_type=0x0800,ipv4_dst=r5ip)
303                 self.add_gototable(datapath, 0, 2, 160, match, 10) #Table 2 is to
    relay Raspi 1
304
305         elif (raspi4 not in self.datapaths and raspi5 in self.datapaths and raspi6
    not in self.datapaths and raspi1 in self.datapaths and raspi2 in self.datapaths
    and raspi3 in self.datapaths):
306             local = datapath.ofproto.OFPP_LOCAL
307             if datapath.id == raspi2:
308                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r5,
    arp_tpa = gw1ip)
309                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 1
310
311                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r5,
    ipv4_dst = gw1ip)
312                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 1
313
314                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r1,
    arp_tpa = r5ip)
315                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 5
316
317                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r1,
    ipv4_dst = r5ip)
318                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 5
319
320                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r5,
    arp_tpa = gw2ip)
321                 self.add_gototable(datapath, 0, 4, 160, match, 10)#Table 4 is to
    relay Raspi 3
322
323                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r5,
    ipv4_dst = gw2ip)
324                 self.add_gototable(datapath, 0, 4, 160, match, 10)#Table 4 is to
    relay Raspi 3
```

```
325
326             match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r3,
    arp_tpa = r5ip)
327             self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 5
328
329             match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r3,
    ipv4_dst = r5ip)
330             self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 5
331
332         if datapath.id == raspi3:
333             match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r6,
    arp_tpa = gw1ip)
334             self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 2
335
336             match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r6,
    ipv4_dst = gw1ip)
337             self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 2
338
339             match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r2,
    arp_tpa = r6ip)
340             self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 6
341
342             match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r2,
    ipv4_dst = r6ip)
343             self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 6
344
345         if datapath.id == gateway1:
346             match = parser.OFPMatch(in_port=local,eth_type=0x0806,arp_tpa=r6ip)
347             self.add_gototable(datapath, 0, 2, 160, match, 10) #Table 2 is to
    relay Raspi 1
348
349             match = parser.OFPMatch(in_port=local, eth_type=0x0800,ipv4_dst=r6ip)
350             self.add_gototable(datapath, 0, 2, 160, match, 10) #Table 2 is to
    relay Raspi 1
351
352             match = parser.OFPMatch(in_port=local,eth_type=0x0806,arp_tpa=r5ip)
353             self.add_gototable(datapath, 0, 2, 160, match, 10) #Table 2 is to
    relay Raspi 1
354
355             match = parser.OFPMatch(in_port=local, eth_type=0x0800,ipv4_dst=r5ip)
356             self.add_gototable(datapath, 0, 2, 160, match, 10) #Table 2 is to
    rleay Raspi 1
357
358     elif ((raspi5 not in self.datapaths and raspi1 in self.datapaths and raspi2
    in self.datapaths and raspi3 in self.datapaths and raspi4 in self.datapaths and
    raspi6 in self.datapaths)
359           or (raspi5 not in self.datapaths and raspi6 not in self.datapaths and
    raspi1 in self.datapaths and raspi2 in self.datapaths and raspi3 in
    self.datapaths and raspi4 in self.datapaths)):
360         self.logger.info("Case 5")
361         local = datapath.ofproto.OFPP_LOCAL
362         if datapath.id == raspi3:
```

```
363                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r6,
    arp_tpa = gw1ip)
364                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 2
365
366                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r6,
    ipv4_dst = gw1ip)
367                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 2
368
369                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r2,
    arp_tpa = r6ip)
370                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 6
371
372                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r2,
    ipv4_dst = r6ip)
373                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 6
374
375             if datapath.id == gateway1:
376                 match = parser.OFPMatch(in_port=local,eth_type=0x0806,arp_tpa=r6ip)
377                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 1
378
379                 match = parser.OFPMatch(in_port=local, eth_type=0x0800,ipv4_dst=r6ip)
380                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 1
381
382         elif (raspi6 not in self.datapaths and raspi1 in self.datapaths and raspi2
    in self.datapaths and raspi3 in self.datapaths and raspi4 in self.datapaths and
    raspi5 in self.datapaths):
383             if datapath.id == raspi2:
384                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r5,
    arp_tpa = gw2ip)
385                 self.add_gototable(datapath, 0, 4, 160, match, 10)#Table 4 is to
    relay Raspi 3
386
387                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r5,
    ipv4_dst = gw2ip)
388                 self.add_gototable(datapath, 0, 4, 160, match, 10)#Table 4 is to
    relay Raspi 3
389
390                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r3,
    arp_tpa = r5ip)
391                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 5
392
393                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r3,
    ipv4_dst = r5ip)
394                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 5
395
396         elif ((raspi1 not in self.datapaths and raspi6 not in self.datapaths and
    raspi2 in self.datapaths and raspi3 in self.datapaths and raspi4 in
    self.datapaths and raspi5 in self.datapaths)
397             or (raspi1 not in self.datapaths and raspi2 not in self.datapaths and
    raspi3 not in self.datapaths and raspi6 not in self.datapaths and raspi4 in
    self.datapaths and raspi5 in self.datapaths)
```

```
398             or (raspi1 not in self.datapaths and raspi3 not in self.datapaths and
    raspi6 not in self.datapaths and raspi2 in self.datapaths and raspi4 in
    self.datapaths and raspi5 in self.datapaths)):
399             local = datapath.ofproto.OFPP_LOCAL
400             self.logger.info("Case 6")
401             if ev.msg.datapath.id == raspi2:  # Raspi2 To assign the flow rules at
    Raspi2 to reroute the control packet from raspi3 and gateway2 to gateway1
402                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r3,
    arp_tpa=gw1ip)
403                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 5
404
405                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r3,
    ipv4_dst=gw1ip)
406                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 5
407
408                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r5,
    arp_tpa=r3ip)
409                 self.add_gototable(datapath, 0, 4, 160, match, 10)#Table 4 is to
    relay Raspi 3
410
411                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r5,
    ipv4_dst=r3ip)
412                 self.add_gototable(datapath, 0, 4, 160, match, 10)#Table 4 is to
    relay Raspi 3
413
414                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r5,
    arp_tpa=gw2ip)
415                 self.add_gototable(datapath, 0, 4, 160, match, 10)#Table 4 is to
    relay Raspi 3
416
417                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r5,
    ipv4_dst=gw2ip)
418                 self.add_gototable(datapath, 0, 4, 160, match, 10)#Table 4 is to
    relay Raspi 3
419
420                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r3,
    arp_tpa = r5ip)
421                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 5
422
423                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r3,
    ipv4_dst = r5ip)
424                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 5
425
426
427             elif ev.msg.datapath.id == raspi5:
428                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r2,
    arp_tpa=gw1ip)
429                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 4
430
431                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r2,
    ipv4_dst=gw1ip)
432                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 4
433
```

```
434                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r4,
    arp_tpa=r2ip)
435                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 2
436
437                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r4,
    ipv4_dst=r2ip)
438                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 2
439
440                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r4,
    arp_tpa=r3ip)
441                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 2
442
443                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r4,
    ipv4_dst=r3ip)
444                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 2
445
446                 match = parser.OFPMatch(in_port=1, eth_type=0x0806, eth_src=r4,
    arp_tpa=gw2ip)
447                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 2
448
449                 match = parser.OFPMatch(in_port=1, eth_type=0x0800, eth_src=r4,
    ipv4_dst=gw2ip)
450                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 2
451
452             elif ev.msg.datapath.id == gateway1:  # Gateway1
453                 match = parser.OFPMatch(in_port=local,eth_type=0x0806,arp_tpa=r1ip)
454                 self.add_gototable(datapath, 0, 2, 160, match, 10) #Table 2 is to
    relay Raspi 1
455
456                 match = parser.OFPMatch(in_port=local, eth_type=0x0800,ipv4_dst=r1ip)
457                 self.add_gototable(datapath, 0, 2, 160, match, 10) #Table 2 is to
    relay Raspi 1
458
459                 match = parser.OFPMatch(in_port=local,eth_type=0x0806)
460                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 4
461
462                 match = parser.OFPMatch(in_port=local, eth_type=0x0800)
463                 self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi
464
465
466         elif ((raspi3 not in self.datapaths and raspi4 not in self.datapaths and
    raspi1 in self.datapaths and raspi2 in self.datapaths and raspi5 in
    self.datapaths and raspi6 in self.datapaths)
467                 or (raspi3 not in self.datapaths and raspi4 not in self.datapaths
    and raspi5 not in self.datapaths and raspi6 not in self.datapaths and raspi1 in
    self.datapaths and raspi2 in self.datapaths)
468                 or (raspi3 not in self.datapaths and raspi4 not in self.datapaths
    and raspi6 not in self.datapaths and raspi1 in self.datapaths and raspi2 in
    self.datapaths and raspi5 in self.datapaths)):
469             self.logger.info("Case 7")
470             local = datapath.ofproto.OFPP_LOCAL
```

```
471             if ev.msg.datapath.id ==raspi5: #Raspi5 Assign the flow rules at Raspi5 to relay the packet from raspi6 to gateway1
472                 match = parser.OFPMatch(in_port=1,eth_type=0x0806,eth_src=r6,arp_tpa=gw1ip) #Table 2 is to relay Raspi 2
473                 self.add_gototable(datapath,0,2,160,match,10)
474 
475                 match = parser.OFPMatch(in_port=1,eth_type=0x0800,eth_src=r6,ipv4_dst=gw1ip) #Table 2 is to relay Raspi 2
476                 self.add_gototable(datapath,0,2,160,match,10)
477 
478                 match = parser.OFPMatch(in_port=1,eth_type=0x0806,eth_src=r2,arp_tpa=r6ip) #Table 4 is to relay Raspi 6
479                 self.add_gototable(datapath,0,4,160,match,10)
480 
481                 match = parser.OFPMatch(in_port=1,eth_type=0x0800,eth_src=r2,ipv4_dst=r6ip) #Table 4 is to relay Raspi 6
482                 self.add_gototable(datapath,0,4,160,match,10)
483 
484                 match = parser.OFPMatch(in_port=1,eth_type=0x0806,eth_src=r2,arp_tpa=gw2ip) #Table 4 is to relay Raspi 6
485                 self.add_gototable(datapath,0,4,160,match,10)
486 
487                 match = parser.OFPMatch(in_port=1,eth_type=0x0800,eth_src=r2,ipv4_dst=gw2ip)#Table 4 is to relay Raspi 6
488                 self.add_gototable(datapath,0,4,160,match,10)
489 
490             elif ev.msg.datapath.id == raspi2:
491                 #Raspi2 To assign the flowrules at raspi2 to relay the control packet from raspi5,raspi6 to gateway1
492                 match = parser.OFPMatch(in_port=1,eth_type=0x0806,eth_src=r5,arp_tpa=gw1ip) #Table 3 is to relay Raspi 1
493                 self.add_gototable(datapath,0,3,160,match,10)
494 
495                 match = parser.OFPMatch(in_port=1,eth_type=0x0800,eth_src=r5,ipv4_dst=gw1ip)#Table 3 is to relay Raspi 1
496                 self.add_gototable(datapath,0,3,160,match,10)
497 
498                 match = parser.OFPMatch(in_port=1,eth_type=0x0806,eth_src=r1,arp_tpa=r5ip) #Table 2 is to relay Raspi 5
499                 self.add_gototable(datapath,0,2,160,match,10)
500 
501                 match = parser.OFPMatch(in_port=1,eth_type=0x0800,eth_src=r1,ipv4_dst=r5ip) #Table 2 is to relay Raspi 5
502                 self.add_gototable(datapath,0,2,160,match,10)
503 
504                 match = parser.OFPMatch(in_port=1,eth_type=0x0806,eth_src=r1,arp_tpa=r6ip) #Table 2 is to relay Raspi 5
505                 self.add_gototable(datapath,0,2,160,match,10)
```

```
506
507             match =
    parser.OFPMatch(in_port=1,eth_type=0x0800,eth_src=r1,ipv4_dst=r6ip)#Table 2 is
    to relay Raspi 5
508             self.add_gototable(datapath,0,2,160,match,10)
509
510             match =
    parser.OFPMatch(in_port=1,eth_type=0x0806,eth_src=r1,arp_tpa=gw2ip)#Table 2 is
    to relay Raspi 5
511             self.add_gototable(datapath,0,2,160,match,10)
512
513             match =
    parser.OFPMatch(in_port=1,eth_type=0x0800,eth_src=r1,ipv4_dst=gw2ip)#Table 2 is
    to relay Raspi 5
514             self.add_gototable(datapath,0,2,160,match,10)
515
516         elif ev.msg.datapath.id == raspi6: #Raspi 6
517             match =
    parser.OFPMatch(in_port=1,eth_type=0x0806,eth_src=gw2,arp_tpa=gw1ip)
518             self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 5
519
520             match =
    parser.OFPMatch(in_port=1,eth_type=0x0800,eth_src=gw2,ipv4_dst=gw1ip)
521
522             self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 5
523
524             match = parser.OFPMatch(in_port=1,eth_type=0x0806,eth_#Table 3 is to
    relay Raspi 5src=r5,arp_tpa=gw2ip)
525             self.add_gototable(datapath, 0, 3, 160, match, 10)
526
527             match =
    parser.OFPMatch(in_port=1,eth_type=0x0800,eth_src=r5,ipv4_dst=gw2ip)
528             self.add_gototable(datapath, 0, 3, 160, match, 10)#Table 3 is to
    relay Raspi 5
529
530         elif ev.msg.datapath.id == gateway1: #Gateway1
531             match = parser.OFPMatch(in_port=local,eth_type=0x0806,arp_tpa=r5ip)
532             self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 1
533
534             match = parser.OFPMatch(in_port=local,eth_type=0x0806,arp_tpa=r6ip)
535             self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 1
536
537             match = parser.OFPMatch(in_port=local,eth_type=0x0806,arp_tpa=gw2ip)
538             self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 1
539
540             match = parser.OFPMatch(in_port=local, eth_type=0x0800,
    ipv4_dst=r5ip)
541             self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 1
542
543             match = parser.OFPMatch(in_port=local, eth_type=0x0800,
    ipv4_dst=r6ip)
544             self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 1
```

```
545
546                 match = parser.OFPMatch(in_port=local, eth_type=0x0800,
    ipv4_dst=gw2ip)
547                 self.add_gototable(datapath, 0, 2, 160, match, 10)#Table 2 is to
    relay Raspi 1
```

Listing E.1: Run RYU Controller for Rerouting

```
1  #In order to detect the failure of wireless mesh node, echo request/reply message
       need to be enabled
2  #Need to enable the paramter in controller.py in the soucecode of RYU controller
3  #Source code of RYU controller can be installed by
4  git clone git://github.com/osrg/ryu.git
5  #Inside controller.py from source code and modify the parameters of echo request
       interval and maximum-unreplied-echo-request as per following
6  ])
7  CONF.register_opts([
8      cfg.FloatOpt('socket-timeout',
9                   default=5.0,
10                  help='Time, in seconds, to await completion of socket operations.'),
11     cfg.FloatOpt('echo-request-interval',
12                  default=3,
13                  help='Time, in seconds, between sending echo requests to a
       datapath.'),
14     cfg.IntOpt('maximum-unreplied-echo-requests',
15                default=4,#
16                min=0,
17                help='Maximum number of unreplied echo requests before datapath is
       disconnected.')
18 ])
19 #After modifying the source code run for ryu program
20 sudo ryu-manager sdwmn_rerouting.py
```

Appendix F

Setting Network Parameters In All Wireless Nodes

```
#In all wireless nodes
sudo nano /etc/sysctl.conf

net.core.rmem_default=8388608
net.core.wmem_default=500000
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
net.ipv4.tcp_rmem = 4096 87380 4194304
net.ipv4.tcp_wmem = 4096 87380 4194304
net.ipv4.tcp_mem = 8388608 8388608 8388608
net.ipv4.tcp_window_scaling=1
```

VITA

Soe Ye Htet was born in 1993 in Yangon, Myanmar. He received B.Eng degree in Electronic Engineering from West Yangon Technological University (WYTU), Myanmar, in 2014. From 2014 to 2016, he worked as a telecommunication engineer in Myanmar. He is a Master's degree student in the field of Wireless Network and Future Internet (STAR) Research Group at Department of Electrical Engineering, Chulalongkorn University, Thailand. From 2017 to present, he is a recipient of scholarship program for ASEAN countries, Chulalongkorn University, Thailand. His research interests include Future Internet Technology and Software Defined Networking.

List of Publications

[1] S. Y. Htet, K. Leevangtou, P. M. Thet, K. Kawila, and C. Aswakul, "Design of Medium-Range Outdoor Wireless Mesh Network with Open-Flow Enabled Raspberry Pi," 33rd International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC), pp. 192-195, 2018.

[2] A. M. Htut, S. Y. Htet, K. Leevangtou, K. Kawila, and C. Aswakul, "Testbed Design of Near Real-time Wireless Image Streaming with Apache Kafka for Road Traffic Monitoring," 33rd International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC), pp. 188-191, 2018.