2021

Hybrid GNS3 and Mininet-WiFi emulator for survivable SDN backbone network supporting wireless IoT traffic

May Pyone Han
Faculty of Engineering

Hybrid GNS3 and Mininet-WiFi Emulator for Survivable SDN Backbone Network Supporting Wireless IoT Traffic

Miss May Pyone Han

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering in Electrical Engineering
Department of Electrical Engineering
FACULTY OF ENGINEERING
Chulalongkorn University
Academic Year 2021

อีมูเลเตอร์ผสมระหว่างจีเอ็นเอสสามกับมินิเน็ต-ไวไฟสำหรับโครงข่ายแกนหลักเอสดีเอ็นเพื่อรองรับทราฟฟิกไอโอทีไร้สาย

น.ส.เม โยน ฮัน

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมไฟฟ้า ภาควิชาวิศวกรรมไฟฟ้า
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2564

Thesis Title	Hybrid GNS3 and Mininet-WiFi Emulator for Survivable SDN Backbone Network Supporting Wireless IoT Traffic
By	Miss May Pyone Han
Field of Study	Electrical Engineering
Thesis Advisor	Associate Professor LUNCHAKORN WUTTISITTIKULKIJ, Ph.D.

Accepted by the FACULTY OF ENGINEERING, Chulalongkorn University in Partial Fulfillment of the Requirement for the Master of Engineering

..... Dean of the FACULTY OF ENGINEERING
(Professor SUPOT TEACHAVORASINSKUN, D.Eng.)

THESIS COMMITTEE

..... Chairman
(Associate Professor CHAODIT ASWAKUL, Ph.D.)

..... Thesis Advisor
(Associate Professor LUNCHAKORN WUTTISITTIKULKIJ, Ph.D.)

..... External Examiner
(Associate Professor Rardchawadee Silapunt, Ph.D.)

เม โยน ฮัน : อีมูเลเตอร์ผสมระหว่างจีเอ็นเอสสามกับมินิเน็ต-ไวไฟสำหรับโครงข่ายแกนหลักเอสดีเอ็นเพื่อรองรับทราฟฟิกไอโอทีไร้สาย. (Hybrid GNS3 and Mininet-WiFi Emulator for Survivable SDN Backbone Network Supporting Wireless IoT Traffic) อ.ที่ปรึกษาหลัก : ลัญฉกร วุฒิสิทธิกุลกิจ

วิทยานิพนธ์นี้ออกแบบและสร้างชุดจำลองการทดสอบสำหรับการเลือกเส้นทางที่คำนึงถึงค่าเวลาประวิงและความทนทานต่อความเสียหายสำหรับแทรฟฟิกของเซนเซอร์แบบไร้สายโดยใช้โครงข่ายที่กำหนดด้วยซอฟต์แวร์ (เอสดีเอ็น) ณ โครงข่ายแกนหลัก ในงานวิจัยนี้ ได้เสนอชุดจำลองการทดสอบโครงข่ายลูกผสมระหว่างจีเอ็นเอส 3 และมินิเน็ต-ไวไฟเพื่อสร้างโครงข่ายแกนหลักจำลองที่กำหนดด้วยซอฟต์แวร์ในจีเอ็นเอส 3 และจำลองไอพีรุ่น 6 บนโครงข่ายส่วนบุคคลกำลังต่ำ (6LoWPAN) ในมินิเน็ต-ไวไฟ มีการใช้เครื่องจักรเสมือนจำนวน 3 ชุดเพื่อการติดตั้งชุดจำลองการทดสอบโครงข่ายที่กำหนดด้วยซอฟต์แวร์แบบลูกผสม ในส่วนของแพลตฟอร์มมินิเน็ต-ไวไฟที่ใช้สร้างโครงข่ายเซนเซอร์ 6LoWPAN จำลองได้ติดตั้งบนเครื่องจักรเสมือน 2 ชุด และเครื่องจักรเสมือนอีกชุดเป็น จีเอ็นเอส-วีเอ็ม

ในโครงข่ายแกนหลักที่ใช้เอสดีเอ็นตามที่เสนอนั้น มีเส้นทางสำหรับส่งแทรฟฟิกจากโครงข่ายเซนเซอร์ทั้งสองแห่งไปยังโครงข่ายเซิร์ฟเวอร์รวมทั้งหมด 3 เส้นทาง สวิตช์เสมือนแบบเปิด (โอวีเอส) ที่รองรับโพรโทคอลโอเพินโฟลว์ถูกใช้ในการสร้างโครงข่ายแกนหลักเอสดีเอ็น กรอบการทำงานเอสดีเอ็นริวที่ใช้ไพธอนถูกนำมาใช้เป็นตัวควบคุมเอสดีเอ็นเชิงตรรกะซึ่งควบคุมโหนดโอวีเอสจำนวน 8 โหนดที่ตั้งอยู่บนเส้นทางทั้งสามเส้นทางโดยการเชื่อมต่อแบบนอกแถบ ในวิทยานิพนธ์ฉบับนี้ อัลกอริทึมการเลือกเส้นทางอาศัยพารามิเตอร์ค่าเวลาประวิง ค่าอัตราส่วนการสูญเสียแพ็กเกต และจำนวนฮอป ในการตัดสินใจเลือกเส้นทางที่เหมาะสุดสำหรับแทรฟฟิกจากเซนเซอร์ หรือแทรฟฟิกจากระนาบข้อมูล อัลกอริทึมการเลือกเส้นทางถูกพัฒนาขึ้นและใช้งานในตัวควบคุมริวแบบรวมศูนย์ โหนดขอบที่เชื่อมต่อกับโครงข่ายเซนเซอร์ทั้งสองโครงข่ายมีภารกิจหลัก 2 งาน (1) วัดค่าเวลาประวิง อัตราส่วนการสูญเสียแพ็กเกต และจำนวนฮอป (2) ส่งผลลัพธ์ที่ได้จากการวัดไปยังตัวควบคุมริวแบบรวมศูนย์ ข่าวสารที่ส่งโดยโหนดขอบมีความสำคัญสำหรับตัวควบคุมเอสดีเอ็นในการเลือกเส้นทางที่เหมาะสุดและติดตั้งเกณฑ์โอเพนโฟลว์ที่จำเป็นต่อโหนดโอวีเอสเพื่อสร้างระนาบข้อมูล

ในวิทยานิพนธ์ฉบับนี้ รายงานผลลัพธ์จากการวัดของอัลกอริทึมการเลือกเส้นทางที่คำนึงถึงค่าเวลาประวิง โดยมีการพิจารณาให้อัลกอริทึมการเลือกเส้นทางมีความทนทานต่อความเสียหาย ด้วยการใช้ผลการวัดของเวลาการเลือกเส้นทางใหม่ในสภาวะที่เส้นทางที่เหมาะสุดล้มเหลว ความสามารถในการโปรแกรมได้ของเอสดีเอ็นที่แยกระนาบข้อมูลออกจากระนาบควบคุมเป็นข้อประโยชน์สำคัญของเราเพราะพฤติกรรมการเลือกเส้นทางสามารถปรับได้ง่าย โดยเฉพาะอย่างยิ่งโครงข่ายเซนเซอร์ไอโอทีของงานวิจัยนี้

สาขาวิชา	วิศวกรรมไฟฟ้า	ลายมือชื่อนิสิต .....
ปีการศึกษา	2564	ลายมือชื่อ อ.ที่ปรึกษาหลัก .....

6272070121 : MAJOR ELECTRICAL ENGINEERING
KEYWORD: Software-Defined Networking, Internet of Things, GNS3, Mininet-WiFi, Delay Awareness Routing, Fault-Tolerant

May Pyone Han : Hybrid GNS3 and Mininet-WiFi Emulator for Survivable SDN Backbone Network Supporting Wireless IoT Traffic . Advisor: Assoc. Prof. LUNCHAKORN WUTTISITTIKULKIJ, Ph.D.

This thesis has designed and implemented an emulated testbed for fault-tolerant delay awareness routing for wireless sensor traffic by using software-defined networking (SDN) at the backbone network. In this work, the hybrid form of GNS3 and Mininet-WiFi emulation network testbed is proposed to build an emulated SDN-based backbone network in GNS3 and an emulated IPv6 over Low Power Personal Area Network (6LoWPAN) in Mininet-WiFi. Three virtual machines are used to set up the hybrid emulated SDN-based network testbed. The Mininet-WiFi platform which is used to build the emulated 6LoWPAN sensor network is installed in two virtual machines separately and the third virtual machine is GNS-VM.

In the proposed SDN-based backbone network, there are three available paths to carry the sensor traffic from two sensor networks to the server network, and Open Virtual Switch (OVS) supporting the OpenFlow protocol is used to establish an SDN-based backbone network. The python-based RYU SDN framework is used as the logically centralized SDN controller which controls eight OVS nodes located in three paths in an out-of-band connection. In this thesis, the routing algorithm is based on delay, packet loss ratio, and the number of hops parameters to decide the optimal path for the sensor traffic or the data plane traffic. The routing algorithm is developed and executed in the centralized RYU controller. There are two main tasks for the provider edge node connected to the two sensor networks (i) to measure the delay, packet loss ratio, and the number of hops (ii) to send the measurement result to the centralized RYU controller. The information which is sent by the provider edge node is important for the SDN controller to decide the optimal path and then install the necessary OpenFlow rules to the OVS node to establish the data plane.

In this thesis, the measurement result of delay aware routing algorithm is reported. Another consideration is that the implemented routing algorithm is fault-tolerant with the measurement result of rerouting time when the selected optimal path is failed. The programmability of SDN due to the separation of control and data planes is the key benefit for us as the routing behavior is easily customizable, especially for IoT sensor networks in this work.

Field of Study:	Electrical Engineering	Student's Signature .....
Academic Year:	2021	Advisor's Signature .....

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Assoc. Prof. Dr. Lunchakorn Wuttisittikulkij, for giving me the opportunity to study at Chulalongkorn University with his valuable advice and guidance. I am so grateful to Asian Scholarship Program for financial support to finish my master's degree in an interesting engineering field. I do appreciate the committee members for their judgments and points in my thesis examination.

I would like to express my gratitude to all seniors and friends in my research group for their kindness and help during my study time. Furthermore, I sincerely appreciate my senior Mr. Soe Ye Htet for his advice and help in my thesis. Last but not least, I feel so lucky and grateful to my supportive parents and siblings who love me unconditionally and encourage me throughout my long journey of student life.

May Pyone Han

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

Chapter 1

Introduction

1.1 Research Motivation

The Internet of Things (IoT) plays an important role in every sector of today's world society to promote the quality of life in many specific areas such as education, healthcare, agriculture, and transportation. IoT technology saves time, money, and energy due to enhancing a more flexible and scalable manner for human-to-human, human-to-machine, or machine-to-machine communication. Therefore, with the increased amount of data and devices in various network domains, IoT data management is becoming more and more difficult and remains a challenging topic for researchers and providers. Some of the challenges such as flexibility, scalability, heterogeneity, and energy saving are important to be considered for emerging IoT networks. Data is exchanged through different vendors and networks for diverse IoT domains, facing different issues such as latency, congestion, packet loss, and security problems. Therefore, it is essential to meet the required characteristics of a specific area that demands the Quality-of-Services (QoS), including delay, packet loss, or both.

Software-Defined Network (SDN), on the other hand, is very popular in networking for its programmability and flexibility for controlling and managing the network elements. Dynamically configuring and easily extending the network components are key to supporting a scalable manner for a huge variety of devices in the heterogeneous network. SDN allows building such a flexible environment by taking the responsibilities for network configuration through a programming language. SDN controller is capable of maintaining the data from the user or infrastructure level, also called data plane, through Southbound API (Southbound Application Programming Interface) for further data processing such as routing, load balancing, and monitoring. The optimization of the SDN controller for routing path through its global view of the network offers a special way of data transmission for a flexible and scalable traffic environment. Therefore, SDN has been extensively deployed in different domains to improve network performance.

The SDN paradigm has been proposed in IoT networks to change from traditional networks to more adaptable networks, solving the challenge of traditional IoT architecture. In the IoT domain, data management through the SDN core network makes it possible to choose the path with the lowest delay, minimum loss, maximum bandwidth, or higher security.

Furthermore, different types of IoT traffic can be prioritized and queued to be chosen for specific applications. Therefore, a heterogeneous IoT network environment can be enhanced by proposing SDN technology for flexibility, QoS improvement, data monitoring, and maintenance.

1.2 Problem Statement

In the delay-sensitive IoT domain, data management becomes a challenge to deal with a huge amount of data and devices. The heterogeneous network with several data paths is another challenge for data transmission. For reaching the desired destination or meeting the requirements of the specific area in different IoT networks, it is important to be chosen and designed for efficient routing. In the case of delay-sensitive and low-bandwidth applications, the delay parameter needs to be considered as the first priority to select the best path based on the value of delay. To handle various IoT applications through the heterogeneous network, it is still a challenge to propose an automatic delay awareness routing for easy deployment and management of IoT data.

1.3 Objective

There are two main objectives in this work which are (i) to propose the emulated testbed of SDN-based backbone network with 6LoWPAN IoT sensor network by integrating GNS3 and Mininet-WiFi emulators and (ii) to propose the fault-tolerant delay awareness routing for emulated SDN-based backbone network to handle IoT traffic. Delay awareness routing is proposed for an emulated SDN-based backbone network to handle emulated 6LoWPAN IoT network traffic because of the scalability of network implementation and cost-effectiveness. For example, the number of network nodes can be adjusted without purchasing the real hardware, therefore, fault-tolerant delay awareness routing is implemented in the emulated SDN-based backbone network. Similarly, the sensor nodes in emulated 6LoWPAN IoT sensor network can be connected as desired. The demand for performance of the data traffic can be different depending on the application usage scenario, for example, bandwidth and delay should be considered as the first priority for the video-streaming application scenario. In this work, it is assumed that the sensors that send UDP messages with low bandwidth to the destination node. Therefore, it is proposed that the delay-sensitive IoT traffic is handled by using the SDN features. The measurement parameters that are required for the SDN controller to decide the best path will be proposed in this work. Besides, the rerouting time that is needed when the node failure or path failure has occurred in the SDN-based backbone network will be reported in this thesis.

1.4 Scope of Thesis

The scope of this research is as follows:

1. Propose the hybrid emulated SDN-based backbone network testbed in GNS3 and emulated 6LoWPAN-based sensor network testbed in Mininet-WiFi.
2. Propose the SDN-based fault-tolerant delay awareness routing for delay-sensitive IoT traffic through the SDN-based backbone network.

Chapter 2

Background and Literature Review

2.1 Software-Defined Networking

SDN network architecture such as agile, centrally managed, and directly programmable brings the cost-effective and manageable network to a wide range of applications. Unlike the traditional network, SDN decouples the control plane and data plane, and the network management is enhanced by the programmable feature. In SDN, the control plane is logically centralized to handle the whole network from a global view. The network forwarding devices such as switches, and routers are placed in the data plane and responsible for forwarding data according to the instructions installed by the SDN controller. The SDN controller is the brain of the network which is logically centralized in the control plane and gives direct traffic flow instructions to the forwarding plane.

The three layers of the SDN are Infrastructure Layer, Control Layer, and Application Layer. The lowest layer is the infrastructure layer (so-called the data plane) where switches and routers are installed. Control Layer (also called the control plane) is where the controller is maintained to implement the flexible network flow rules into the switches based on the required parameters and adaptable information from the network devices. The top layer is the application layer which interacts with the network administrators to write the SDN applications for network configuration and maintenance.

The Southbound Interface (SBI) is used to deliver the instructions from the SDN controller to the network forwarding devices such as routers, switches which are located at the data plane layer. OpenFlow protocol can be regarded as the standard protocol for southbound interface and other protocols such as BGP (Border Gateway Protocol), and RESTCONF can also be used. Through this, the SDN controller is enabled to assign the flow rules to the dumb switches. On the other hand, the network administrator can develop the program to define the routing policy such as load-balancing algorithm by using high-level programming languages such as python, go, java, etc. The developed program by the network administrator will be communicated with the SDN controller through the application programming interface with the help of the northbound interface. The principle of SDN is shown in Figure 2.1.

2.2 OpenFlow

OpenFlow is the first open flow standard protocol proposed by Stanford University [30] and is now updated by the Open Networking Foundation (ONF). The default version is 1.0 and the latest version is 1.5. OpenFlow Protocol supports the southbound interface between the controller and the individual switch to directly access the forwarding plane of the network devices. Updating the flow entries such as installing, deleting, and modifying the flow entries in the OpenFlow-enabled network devices is easily done by exchanging respective OpenFlow protocol messages between the control plane and data plane. An OpenFlow Channel such as a secure SSL (Secure Sockets Layer) channel is used in every OpenFlow-enabled network element to connect with the external controller. Besides, one or more Flow Tables, a Group Table, and a Meter Table are implemented in the switches intending for packet lookups, packet forwarding and QoS features shaping.

The controller-switch communication depends on three main types of OpenFlow protocol exchanged messages that are Controller-to-Switch, Asynchronous, and Symmetric. The controller starts managing or querying the state of the switch through Controller-to-Switch messages. On the other hand, the SDN switch sends its state changes or notifications to the controller by using Asynchronous messages. The performance of the OpenFlow connection can be checked by using Symmetric messages in both controller and switch. The three types of OpenFlow protocol messages are summarized in Table 2.1. Every flow table of the switch handles a set of flow entries to guide the arriving packet to its specific destination and these flow entries are modified by the controller in two ways: reactively or proactively. Flow entries deal with the match-action criteria for incoming packets where the corresponding action is executed if the header of the flow entries is matched. Otherwise, these packets need to be sent to the controller for deciding and creating the required new flow rules or more actions.

In the OpenFlow network, control messages are sent either in an in-band mode or in an out-of-band mode. In an in-band mode, control messages are exchanged on the same channel used by the data plane traffic as a single network interface is shared for both control and data traffic. In an out-of-band mode, on the other hand, control messages are transferred on a separate channel. The OpenFlow version 1.3 has flow tables including Match Field, Priority, Counters, Instructions, Timeout, and Cookie as the main components for each flow entry. OpenFlow version 1.3 is used in this thesis and summarized in Table 2.1.

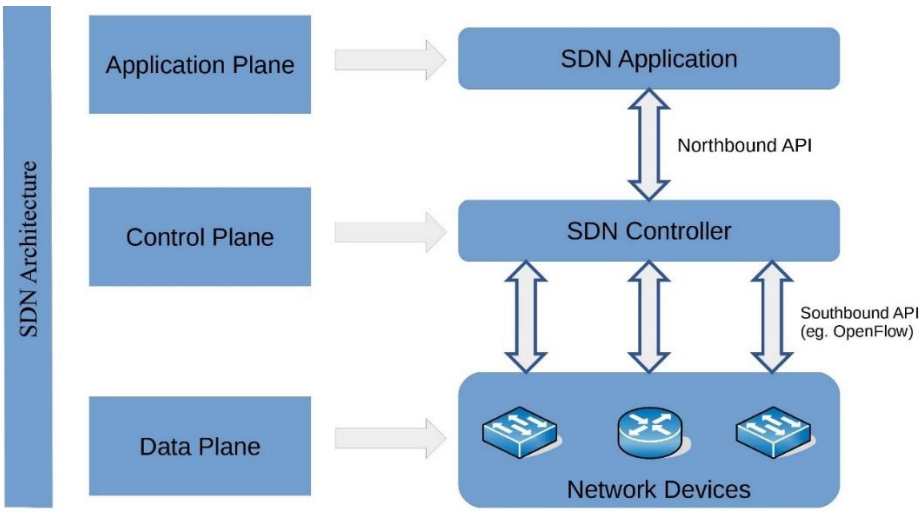


Figure 2.1: Principle of SDN.

Table 2.1: Contents of flow entry in OpenFlow version 1.3.

Match Field	Priority	Counters	Instructions	Timeout	Cookie

The contents of each flow entry are as below:

Match Field – It includes the value of the ingress port number (layer 1) and the header values for the upper three layers that define the source and destination addresses of MAC (Media Access Control), IP (Internet Protocol), and the TCP/UDP (Transmission Control Protocol/User Datagram Protocol) port numbers. These defined values filter the entering packets to match an exact flow in the switch.

Priority – Every entry in the flow table is assigned with a priority, as a result, an incoming packet header performs matching from the highest priority number to the lowest one in a sequenced order.

Counters – It counts the number of received packets, bytes, and duration for updating the statistical information about the matched packets of a particular flow.

Instructions – It handles the action field instructing the SDN switches to be applied to a specific flow for each matched packet. There are many available options in the action filed for respective matched flow instructions. Some of these options from OpenFlow version 1.3 are listed in Table 2.2.

Table 2.2: Action fields of OpenFlow version 1.3.

Actions	Function
OUTPUT	Forward the packets to the defined port
DROP	Drop the corresponding packets
ALL	Forward the packets to all other ports
CONTROLLER	Forward the packets to the controller
FLOOD	Forward the packets to all other ports except to the input/ingress port
LOCAL	Forward the packets to the local port
INPORT	Forward the packets to the input/ingress port

The OpenFlow version 1.3 is used in this thesis because it allows for more flow tables providing a flexible OpenFlow pipeline mechanism compared to the previous version. According to the pipeline mechanism, an incoming packet always interacts with a Flow Table starting from the lowest number (Table 0) to the highest one sequentially. The flow entry supports actions for an incoming packet and is executed from the highest priority to the lowest. In this case, if the flow entry in a Flow Table is not matched with the incoming packet, the so-called Table-Miss event happens in which the device sends its packet to the controller for the necessary actions. Furthermore, multiple flow tables define the different actions of the system such as QoS or routing in each flow table separately. Some of the OpenFlow messages are shown in Table 2.3 and the architecture of the OpenFlow switch is shown in Figure 2.2.

Table 2.3: Three main types of OpenFlow protocol messages (Controller-to-Switch, Asynchronous, and Symmetric).

CONTROLLER-TO-SWITCH	ACTIONS
Packet-Out	Inform the switch to forward the packet on a directed path
Modify-State	Add/delete/modify the flow entries in the Flow Table
Read-State	Collect statistics and configuration information from the switch
Features Request/Reply	Request for the switch features
Configuration Request/Reply	Request for the switch configuration

ASYNCHRONOUS	ACTIONS
Packet-In	Send the matched packets to the controller to be processed
Flow-Removed	Inform the controller about the removed flow entry because the time expired
Port Status	Inform the controller about the port configuration or state changes
Flow-Monitor	Inform the controller about the changes in the Flow Table
SYMMETRIC	**ACTIONS**
Hello	Exchange information between switch and controller
Echo Request/Reply	Used by the switch or controller to check the OpenFlow connection
Error	Used by the switch or controller to notify the problems

2.3 QoS Management in SDN

Quality-of-Service (QoS) acts as an important role to deliver services with specific network requirements such as delay, bandwidth, and packet loss. These QoS parameters (such as bandwidth, delay, and packet loss) are mentioned in Service Level Agreement (SLA) according to the user's demands. Along with the increase of network applications in different sectors, quality of service is a challenge for network service providers to guarantee network performance. In that case, Network Traffic Management is an essential key to satisfying the quality of service as well as reducing latency, and packet loss. Traditional network architecture is not flexible enough to handle the dynamic nature of heterogeneous devices in different network environments which demand urgent and adaptable requirements in terms of QoS. On the other hand, in SDN, flexible and programmable network features ensure QoS services for unpredictable network changes from time to time.

SDN controls the network centrally and logically which gives special advantages for monitoring the network traffic and collecting the statistical information to implement and analyze the required QoS level. The available QoS support mechanism in the OpenFlow protocol is meters and queues. OpenFlow 1.3 and later versions support a meter table in which meter entries are defined and consist of three fields: Meter ID, Meter Band, and Counters. The meter table supports rate-limiting of received packets by monitoring and controlling the ingress traffic rate for each flow entry. Meter band can be assigned in the table for further packet processing and then counter collects the statistical information of these

processed packets. Another QoS function is Queue configuration which processes the packets for output at a specified maximum or minimum limiting rate. Since the only two functions for QoS, queues, and meters, are provided in OpenFlow, new methods have been proposed for better QoS assurance for a wide range of modern applications. Therefore, SDN-based QoS frameworks are implemented in different network sectors and become solutions for future user demands.

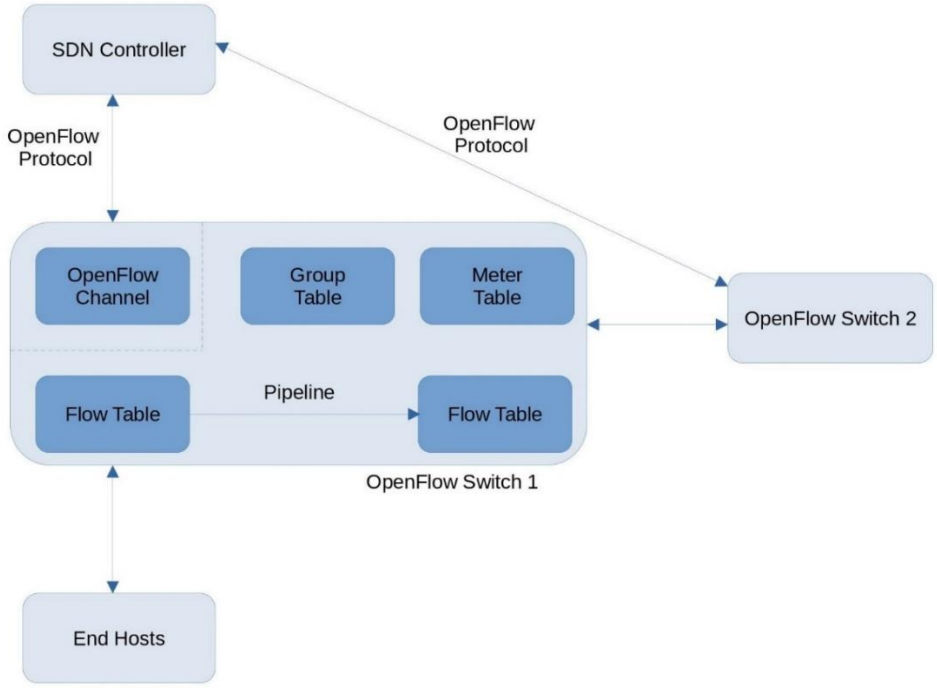


Figure 2.2: Architecture of OpenFlow switch (OpenFlow Version 1.3).

2.4 Internet of Things (IoT)

Nowadays, the Internet of Things (IoT) plays an essential role in different sectors such as smart cities, smart agriculture, smart transportation, smart healthcare, smart grids, and industrial automation, leading to the smart globe. The Internet enables billions of smart devices to communicate with each other for collecting and sharing information. Moreover, a wide variety of sensor objects make these devices smart and intelligent in every sector. As a result, IoT devices track real-time data and transfer these data over a network without human interactions. The tremendous number of IoT devices have been interconnected around the world for several advantages such as being cost-effective, time-saving, better monitoring, and automation.

An increasing number of IoT applications brings many new services and challenges such as security, scalability, flexibility, and quality of service for customer experiences. For

example, real-time transportation such as autonomous driving demands delay-sensitive and minimum packet loss data communication. Some IoT devices are bandwidth-hungry applications, 3D (three dimensions) videos, virtual reality, and augmented reality, which require higher bandwidth among others. In the healthcare sector, different QoS levels must be defined depending on the applications. For example, recording the health conditions of a patient needs to be given the lowest priority in terms of bandwidth and delay, on the other hand, telesurgery ranks the highest priority for critical and lossless communication. In the same way, in smart cities, smart homes, and every other sector, identifying and ensuring different QoS levels for diverse applications is the major consideration to meet the user experiences.

2.5 Architecture of IoT

There are different types of IoT architectures, including three-layer, five-layer, cloud-based, or fog-based IoT architecture. The most basic IoT architecture shown in Figure 2.3 consists of Perception Layer, Network Layer, Service Layer, and Application Layer. Cloud Computing and Fog Computing are introduced to IoT to enhance the performance and scalability of IoT, offering unlimited storage and real-time experience.

The Perception Layer is the physical layer of the IoT network which is also called the sensing layer. Many physical or virtual devices, as well as wired or wireless objects, are located to gather various data from the environment. These devices are sensors, actuators, RFID (Radio Frequency Identification) tags, or edge devices that connect with their specific domain. Most IoT devices are small, cheap, and low-power elements that are usually connected to a battery source for power.

The second layer is Network Layer, also called the access and transmission layer, where data transmission is mainly performed through the network. The network layer is where data is transferred from the sensing layer to the upper layer and vice versa. Data from various devices is collected and processed for further transmission based on specific communication technology. There are different types of communication networks used by IoT devices depending on their design and capabilities. This includes 5G (fifth-generation mobile network), LTE (Long Term Evolution), Ethernet, Wi-Fi (Wireless Fidelity), Zig-Bee, Bluetooth Low Energy (BLE), and 6LoWPAN. Each technology has its characteristics for power consumption, coverage area, data transmission rate, and cost.

The Service Layer, also called the processing layer, provides a system for storing, processing, and analyzing data. Data storage and information processing systems can be

either distributed or centralized. This is also the layer where middle-ware services are provided.

The Application Layer is the top layer of the IoT network which allows users to utilize and manage data depending on the business goal. The IoT applications are designed to deliver specific services through a user interface that can be easily accessed by users.

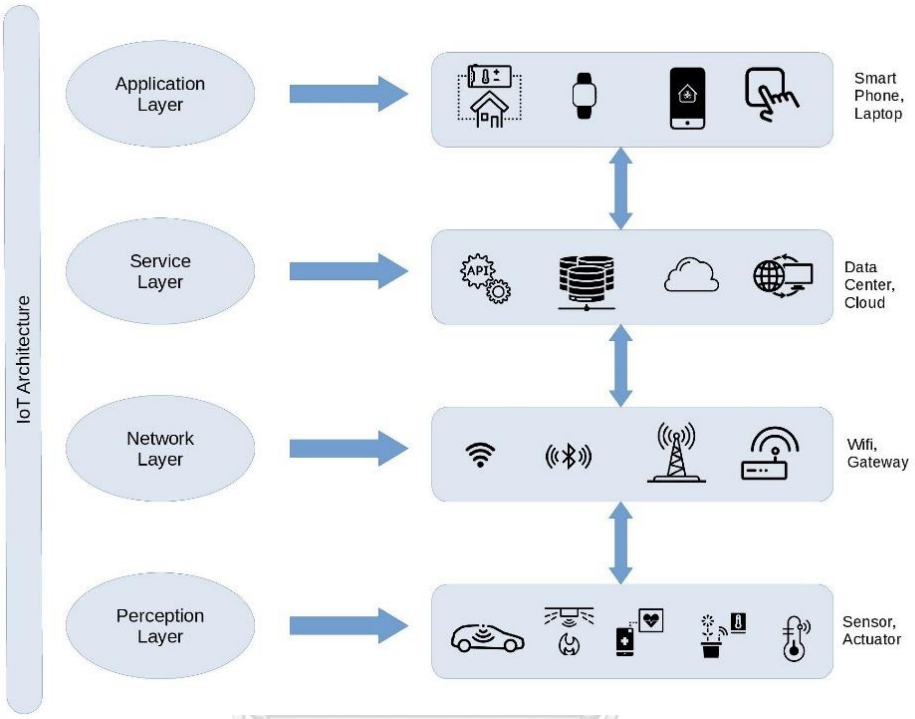


Figure 2.3: Architecture of IoT.

2.6 Wireless Sensor Network

Wireless Sensor Network (WSN) comprises many sensor nodes which monitor the environmental conditions and connect through different communication technologies. The WSN must be satisfied with some important characteristics such as low power, low cost, reliability, and easy maintenance. The nodes of WSN are also resource-constrained for speed, storage, or bandwidth limitations, however, recent WSN architecture supports further improvement for heterogeneous devices and limitations. A wide range of WSN applications has been proposed in several areas, including healthcare, industry, agriculture, and the environment. In WSN, two main communication technologies based on short-range and long-range can be categorized. Some short-range communication technologies include Bluetooth, Zig-Bee, BLE, and RFID. On the other hand, Long-Range (LoRa), Narrow Band IoT (NB-IoT), and Sigfox are long-range communication technologies. Depending on the requirements

of applications and characteristics of sensor nodes, each of them will be chosen for different purposes.

2.7 6LoWPAN

6LoWPAN stands for IPv6 over Low Power Personal Area Network introduced as an open standard by the International Engineering Task Force (IETF) based on IEEE 802.15.4. It supports a wide range of applications, including wireless sensor networks. 6LoWPAN covers many sensor nodes and allows internet connectivity in a large area. The transmission of IPv6 over IEEE 802.15.4 is enabled by an adaptation layer that is added between the data link layer and network layer as shown in Figure 2.4. The data of WSN is sent as packets in the form of IPv6 providing end-to-end IPv6 communication over IEEE 802.15.4. Some of its useful characteristics are low power, low data rate, and low cost. IPv6 gives special benefits such as small packet size and easy management as well as mobility, scalability, reliability, and availability. Interoperability is another main advantage of 6LoWPAN.

The adaptation layer is where the fragmentation or reassembly process is performed to fit the link layer and network layer. The minimum MTU (Maximum Transmission Unit) of IPv6 frame size is 1280 bytes whereas the maximum physical layer of IEEE 802.15.4 frame size is 128 bytes. Therefore, the adaptation layer handles the frame size adjustment to fit IPv6 with IEEE 802.15.4. Furthermore, the 6LoWPAN standard defines four types of frame headers: IPv6 compressed or not compressed IPv6 header, mesh header, broadcast header, and fragmentation header. Since 6LoWPAN gives many benefits to sensor networks with low power, low data rate, and small devices, it has been widely applied in various sectors such as industrial monitoring, home automation, smart grid, and different automation areas.

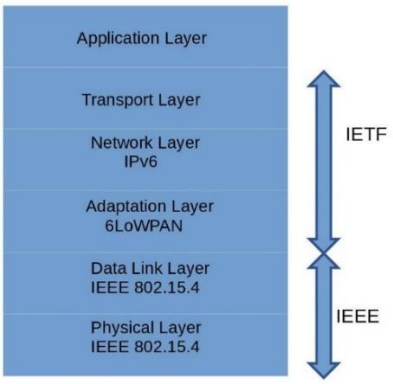


Figure 2.4: 6LoWPAN protocol stack.

2.8 Improvement of IoT Network with SDN

It has been challenging to manage and control the IoT network of a huge number of heterogeneous devices and data. To fulfill the characteristics and reduce the complexity of IoT networks, introducing the scalable and flexible network manner gives many opportunities for IoT devices to extend the network easily and quickly. As mentioned earlier, SDN architecture of programmable and adaptable network management conveys solutions to mitigate the complexity of IoT networks. Additionally, SDN improves the performance of IoT services in terms of QoS, Security, Routing, Load Balancing, and so on. Therefore, integrating the SDN into IoT sectors leads to a more sustainable IoT ecosystem by providing dynamic resource management and optimization capabilities. In the core network, optimized routing and device configuration for efficient data transmission is designed by SDN based on predefined rules and policies. Furthermore, device-to-device communication, as well as radio resource management, are also done by SDN. However, it is a new challenge to integrate the SDN network into an IoT environment to meet the desired specifics of IoT applications connected in different networks and sectors.

2.9 Literature Review for SDN-based QoS Management

The SDN-based QoS management has been proposed in many ways, dealing with topology discovery, traffic classification, traffic monitoring, and real-time data collection and path selection. The author in [1] has proposed Quality-of-Experience (QoE) management for intent-based SDN, measuring delay and packet loss of audio and video traffic in the Mininet environment. They defined the QoE limit to compare with the current measured value, and then frequently monitor the data quality based on the measured data. The ONOS Controller monitors and assesses the data through OpenFlow protocol messages by changing the network configurations automatically. Packet_Out and Packet_In OpenFlow messages are used to get a connection between switches or switches and controllers.

The multipath routing with the QoS management in SDN for Tactile internet traffic is presented in [2], targeting to reduce delay for such delay-sensitive traffic. The SDN environment is created in Mininet along with RYU Controller for testing different traffic sources. In QoS management, the author has distinguished three different types of traffic: tactile, video, and best-effort data, then provides the priority by using a queueing mechanism. In the routing section, real-time data is updated by the network monitoring module and used to calculate the path cost to optimize the best path with low bandwidth as well as low hops.

The author in [3] has proposed OpenHealthQ where different types of healthcare data are handled based on OpenFlow Queues, prioritizing the data on their throughput and delay requirement. The experiment is performed in Mininet-WiFi emulation to enable wireless sensor devices of SDN-based fog layer architecture. The RYU Controller is used as a centralized controller that connects to OpenFlow switches of the fog layer. The Iperf tool is used to send Differentiated Service Code Point (DSCP) traffic required for marking the traffic classification. The author has guaranteed the bandwidth for heterogeneous healthcare devices by implementing the dynamic feature of QoS services.

Another QoS improvement in [4] deals with minimizing the overall latency of all network links in the SDN network. The SDN Controller checks active sessions to calculate the delay of each link and compare it with a defined threshold value. Then, the author has applied the batch routing function for optimizing many video sessions arriving from time to time. Mininet emulator along with OpenDayLight Controller is chosen in their experiment and data traffic is produced by the Iperf tool.

The author in [5] has developed QoS-aware flow management in SDN, considering the bandwidth and delay parameters. They proposed a QoS-based routing algorithm based on the OpenDayLight controller. Network traffic is generated by the Iperf tool in the Mininet emulation test. Three independent modules are introduced in their system: Topology Discovery, Traffic Monitoring, and Path Selection. The structure of an entire network is defined in the topology discovery module, whereas in the traffic monitoring module, the packet capture technology is used to track the packets and calculate the dynamic bandwidth based on the collected statistics. In the path selection module, for every 5ms interval, port statistics are updated to compare with the threshold bandwidth, deciding for routing whether it is needed to alternate or not. Then they compare the resulting delay of the proposed method with the Dijkstra algorithm to show that their system has less delay. Therefore, there have been many studies on the SDN-based QoS framework, some of which are mentioned in this paper.

2.10 Literature Review for SDN-based QoS Management in an IoT Environment

In [6], the author has proposed edge-based 6LoWPAN-SDN architecture to reduce latency, packet loss, and heterogeneity. 6LE-SDNP (6LoWPAN-SDN Protocol) is developed to enable efficient communication between various devices. Besides, a hybrid-edge switch is designed to reduce complexity and heterogeneity, connecting SDN and 6LoWPAN devices.

In addition, to use the global SDN controllers, the author designed the SDN-based edge controller for each cluster of IoT networks. Instead of using the 6LoWPAN border router, the SDN controller performs an alternative way to operate as a more efficient border router for the 6LoWPAN IoT network. The performance of the proposed solution reduces latency and overhead as well as round trip time and packet loss when compared to the traditional 6LoWPAN network.

In [7], QoS constraints of the industrial network are solved by selecting SDN technology to prioritize and manage traffic flows in terms of delay and packet loss. The Mixed Flow Installation (MFI) method and the Proactive Flow Installation Rerouting (PFIR) method are introduced to achieve the optimal path for low-delay and low-loss traffic types. The proposed approach is considered for both wireless and wired networks in terms of guaranteeing QoS in data transmission. The simulation and real testbed show the result of the end-to-end delay of the framework compared to normal flow installation and achieve the target for the delay-sensitive industrial network.

In [8], the SDN paradigm is applied to the 6LoWPAN IoT network for testing and comparing the performance of 6LoWPAN devices in Mininet-IoT. The ONOS (Open Network Operating System) controller is introduced as a centralized gateway to collect data and route the path between 6LoWPAN network clusters through an edge router. In the research, the Wireshark Analyzer tool is used to measure network performance and the Iperf tool is utilized to generate data traffic to be able to measure throughput and packet loss. The two scenarios of different hosts and clusters are tested to compare jitter, packet loss, delay, throughput, and CPU (Central Processing Unit) usage.

In [9], the traditional IoT network is compared with Software-Defined IoT (SDIoT) architecture in terms of QoS performance. This comparative analysis focuses on jitter, latency, and throughput to highlight the efficiency of SDN contribution to data communication. The deployed SDN architecture also supports data interoperability and scalability among sharing various network devices and applications. The traditional IoT environment is designed in GNS3, on the other hand, the proposed SDIoT architecture is emulated in Mininet for analyzing the results according to QoS metrics. A Ping tool is used to measure the latency of data connection from the source to the destination. The results show the advantage of reducing network overheads in data communication which then increases the throughput of the network. However, the author also highlights a single point failure in the case of using a centralized control plane to handle the number of growing switches, nodes, and traffic.

In [10], SD-6LN is developed to incorporate the features of SDN in the existing 6LoWPAN network to resolve some challenges of the IoT network such as availability, reliability, and scalability. The SDN controller acts as the 6LoWPAN gateway node to check the quality of the network link, optimize the alternate path for link failure and update the flow entries or network topology. Mininet-WiFi simulator is utilized for experiment and comparison of two different paradigms of the traditional and proposed network. The Iperf tool generates the data stream for evaluating the performance factors, including average round trip time, packet loss, and jitter.

In [11], a traffic-aware QoS routing scheme is considered in the SDIoT network, dealing with delay-sensitive flows as well as loss-sensitive flows. The nature of SDN gives flexibility to the overall network to be able to maximize the network performance. The proposed scheme is emulated in Mininet by using the POX SDN controller. Two topologies (AttMpls topology and Goodnet topology) are tested and compared with existing schemes. A QoS routing algorithm based on Integer Linear Programming is introduced to optimize the best QoS path. The simulation gives a more feasible routing path to mitigate the path delay and number of QoS violated flows for delay-sensitive or loss-sensitive traffic. In the case of heterogeneous devices and traffic, it is also mentioned that jitter and low-level packet classification will be considered for plans of work.

In [12], end-to-end IoT traffic is managed by SDN to identify the real-time routing path, keeping low path latency for IoT data. Path resolving, delay tracking, and delay management are the three main functions of the proposed method. Path resolving and delay tracking functions are responsible for monitoring the path and operating in parallel. These two functions give information to delay management functions for further analyzing and controlling the network. Mininet emulation is used to build the topology with the OpenDayLight controller for conducting the test of QoS performance and the ping method is used to measure the path delay. It is highlighted that the solution reduces latency by 63.1 percent, compared with the routing based on the shortest-path algorithm.

In [13], network traffic monitoring by SDN is implemented for maximizing the overall performance of expanding networks. The single, liner, and tree topology are tested and controlled by the RYU controller in the Mininet emulator. The measurement parameters are throughput, jitter, bandwidth, and RTT (Round Trip Time) for QoS performance. A Ping tool is used to measure the delay whereas the Iperf tool generates UDP packets to measure the jitter and bandwidth utilization for three scenarios.

2.11 Literature Review for Delay Measurement of Network Traffic

In SDN, there are many methods to measure the delay of link or network path (end-to-end delay). Since the architecture of SDN separates the control plane from the data plane, delay measurement can be handled by the controller (control plane) or switch (data plane). Network measurement is performed in terms of the active method or passive method. The active measurement method means applying additional packets to monitor the network, on the other hand, passive measurement methods do not require probe packets since the delay is measured by observation. Both measurement schemes have drawbacks as well as advantages in the case of introducing overhead, reliability, accuracy, or complexity while monitoring network traffic or collecting statistics information.

In OpenNetMon [14] and SLAM [15], network latency is calculated by the SDN controller by injecting probe packets into the network. The first switch accepts this packet and sends it along the path to reach the destination. The last switch sends this packet back to the controller to estimate the delay of the path by calculating the difference between the total traveling time of the packet and switch-to-controller latency. This method gives real-time results since the controller continuously monitors all flows on delay, packet loss, or throughput. In LINK-MON [16], the OpenNetMon method is modified to monitor delay per link, reducing network overhead. This method supports monitoring in real-time for link delay and covers all links by applying Dijkstra's Algorithm. Overall network overhead is reduced when compared to OpenNetMon [14] where path delay is measured.

In TTL-Based Looping [17], end-to-end path delay is measured by the controller while the looping technique is applied to count the IP TTL. Therefore, a loop of packets with a specific Time-to-Live (TTL) is applied to the path. The OpenFlow switch decrements TTL while it transfers the packet in the loop, sending it back to the controller when TTL is zero. Then, the controller decides the latency according to TTL and iteration number. In Queue Length Method [17], the delay measurement is also estimated by using queue lengths at switches. This is another method that calculates delay in the control plane. Processing, propagation, and transmission delay are considered constant values to be able to detect queueing delay required for calculating the path delay.

In sFlow [18] and NetFlow [19], the special switches are used as agents to send the information of flow to the NetFlow collector periodically. Then, this information is analyzed by the collector to estimate the delay. Since the additional agents are implemented on hardware switches, the configuration of complexity and lack of scalability is put in the network architecture. On the other hand, in FlowTrace [21], the network path is traced in real-

time by probing packets from the data plane. FlowTrace [21] installs measurement rules in switches for further computing delay based on the table query algorithm. Besides, the method can be tested in real applications without changing physical switches due to using the OpenFlow protocol.

There are also many traditional methods for network measurement which are widely utilized in the networking environment. A Ping tool is the most traditional way of measuring RTT between the source node and destination node. ICMP (Internet Control Message Protocol) packets are identified as probe packets between sender and receiver to obtain the value of RTT. It can obtain the latency in various window sizes of packets.

Another traditional method is based on the timestamp between TCP second and third packet when the callee establishes a TCP connection. In more detail, the RTT gets from the timestamp between SYN-ACK packet and ACK packet of TCP three-way handshake. The other way is using TCP timestamp which adds an extra payload in the TCP header, however, it provides precise latency. Instead of using TCP timestamp, two-way UDP gives another chance to the traditional way of delay measuring. It detects the Round-Trip Delay (RTD) or One-Way Delay (OWD) by embedding the UDP timestamp in hosts, giving an accurate latency like the TCP timestamp.

There is also a modified way of the traditional method of latency measuring. The method in [22] measures both path latency and flow-setup latency and compares the real testbed with Mininet emulation. Latency is optimized by introducing some additional steps to the Ping tool to have an accurate delay measurement in the data plane. The study shows path latency or flow-setup latency of the physical testbed is larger than Mininet emulation, highlighting that Mininet has less reactivity than the real testbed regarding throughput, jitter, and packet loss.

Chapter 3

Proposed Network Topology and Methodology

3.1 Implementation of Emulated SDN-based Backbone Network with Emulated 6LoWPAN IoT Sensor Networks

GNS3 [28] is an open-source as well as enterprise-level network emulation software. GNS3 offers stable and realistic test case experiences with a testbed for professional network engineers. In addition, GNS3 can support multi-vendor network devices such as Huawei, Juniper, Cisco, and others. Moreover, GNS3 can also connect with real devices or other virtual machines which is a useful feature to create an emulated backbone network. Therefore, extensibility is the main advantage of GNS3. Other useful features are scalable, clustering, and para-virtualization. On the other hand, GNS3 has a drawback for wireless networking emulation, lacking wireless support, although it supports a physical wireless card.

Mininet-WiFi [29] is specially designed for wireless SDN, extending the Mininet which only supports wired network emulation of SDN. Therefore, unlike GNS3, Mininet-WiFi only allows access to the SDN emulation. The HWSIM driver is supported in Mininet-WiFi to offer wireless experiences in an SDN environment. Mininet-WiFi is a user-friendly emulator since python-based programming and configuring are allowed through its API. The test cases are implemented rapidly and easily through the command lines, rather than configuring each SDN/OpenFlow element. Mobility and propagation models are additional useful features of Mininet-WiFi. There are also downsides of Mininet-WiFi such as constraints to a large network, no clustering support, and no multi-vendor support.

In this thesis, the hybrid form of the GNS3 emulator and Mininet-WiFi emulator is used. GNS3 cannot support wireless emulation, therefore, the emulated wireless IoT sensor network is implemented in Mininet-WiFi emulator and the emulated SDN-based backbone network is built in GNS3 emulator. In this thesis, two network emulators (GNS3 and Mininet-WiFi) are used to propose the SDN-based delay aware routing for delay-sensitive IoT traffic in the emulated SDN-based backbone network with the emulated 6LoWPAN-based IoT sensor network. The proposed emulated topology is illustrated in Figure 3.1.

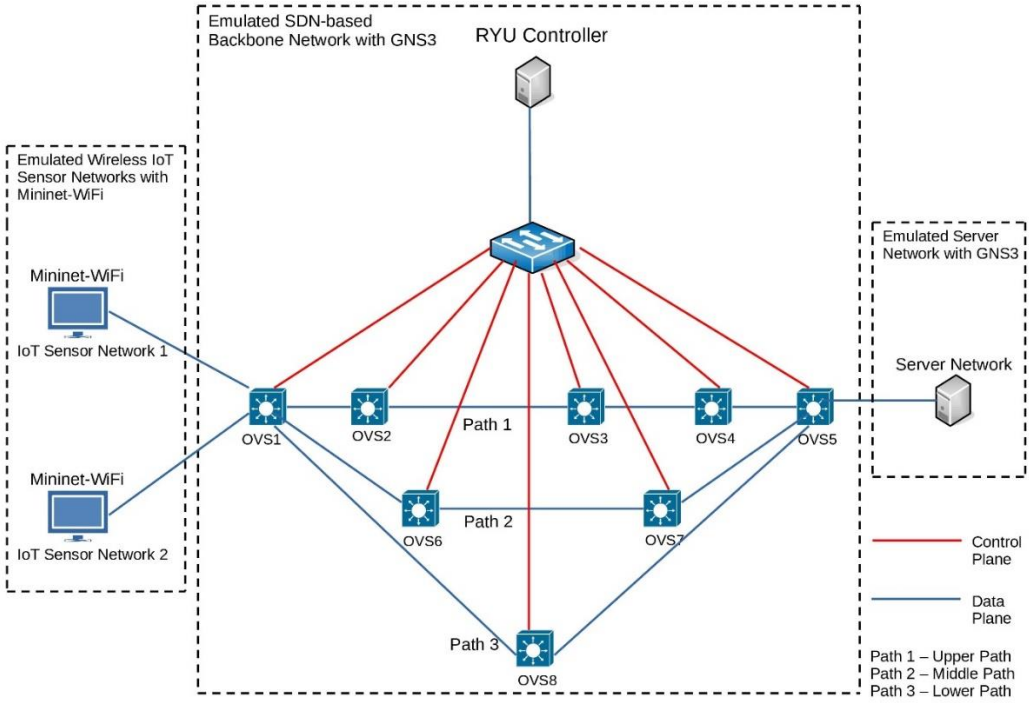


Figure 3.1: Proposed network topology of emulated SDN-based backbone network with 6LoWPAN IoT sensor networks and server network.

The RYU Controller acts as a centralized SDN controller in the control plane of the backbone network and the red lines in Figure 3.1 represent the control plane. The control plane represents the network connection between the RYU controller and OVS nodes. The data plane is drawn with the blue lines in Figure 3.1. The control plane and data plane are communicated through the Southbound API, OpenFlow protocol, supporting OpenFlow messages between controller and switches. The out-of-band connection mode is applied in the proposed topology as the different network interfaces are used for control and data planes.

Three virtual machines (VMs) will be installed to implement the proposed topology. The Mininet-WiFi package will be installed in two virtual machines and the remaining VM will be used for the GNS3 emulator. In the proposed topology, there are three main data networks: (i) IoT sensor network 1, (ii) IoT sensor network 2, and (iii) server network. The IoT sensor network is the network where emulated IoT sensors are located. The server network is the network where the emulated server is located. The server to receive the IoT traffic will be located in the server network. The IoT traffic from the IoT sensor networks 1 and 2 will be forwarded to the server network through the emulated SDN-based backbone network. A total of eight OVS nodes are used to create the emulated SDN-based backbone network. OVS 1 and OVS 5 represent the provider edge network nodes and the other OVS nodes are provider

network nodes or the internal network nodes. The provider edge network node is the network device that is connected with the edge network such as IoT wireless sensor networks and the server network. The provider network node or the internal network node is the network device that is only connected to the OVS node, which means that the provider network node is not connected with the edge network.

There are three available paths in the data plane which will carry the data traffic from the IoT sensor network to the server network. The three paths are labeled as the upper path, middle path, and lower path in this work. The number of hops is the highest in the upper path and lowest in the lower path. There are 6loWPAN-based IoT sensor networks 1 and 2 emulated in Mininet-WiFi 1 and Mininet-WiFi 2 which will send the IoT traffic to the server network through the data plane of the emulated SDN-based backbone network.

The virtual switch provided by the type2 VMware hypervisor, which is the VMware workstation in this work, will help to establish the connection between each virtual machine. Therefore, the two emulators will be connected through the virtual switch. Two sensor networks, server networks, and the SDN-based backbone network will be configured as different networks. Each OVS has a unique Data Path Identifier (DPID) and the RYU Controller will use the unique DPIDs of each OVS node to configure the necessary OpenFlow rules. The function of each network node is explained as follows. The job of the sensor node is to generate the IoT traffic with the standard of IEEE 802.15.4. The AP sensor node is used to convert the IEEE 802.15.4 to IEEE 802.3 to be enabled to forward the IoT traffic to provider OVS nodes. The function of the RYU controller is to receive the required network information from the OVS nodes for routing and to instruct the OVS nodes on how to forward the traffic. The job of the network node inside the server network is to receive the IoT traffic. The OVS 1 (provider edge network node) is responsible for measuring three network parameters including delay, packet loss ratio, and the number of hops. The OVS 1 then will send the values of measured parameters to the RYU controller. Therefore, the RYU controller can use the measured network parameter values in calculating the best path for the data traffic. TCP port 6633 is used to communicate the RYU controller with OVS.

Mininet-WiFi is where sensor network topology is easily created through the command line to configure nodes and links. The HWSIM module is supported to assign the 6LoWPAN sensor network with an access point (AP) sensor node, and wireless sensor nodes. AP sensor acts as a gateway and connects to every sensor node. AP sensor also connects with the wan0-eth0 virtual interface by using NAT (Network Address Translation) to route its network to the provider edge switch of the SDN backbone network. Therefore, the AP sensor node communicates with the provider edge network node through the wan0-eth0 virtual interface.

The sensor node sends its data to the server through the AP sensor node. The server receives and maintains the data sent from the sensor nodes of 6LoWPAN sensor networks 1 and 2 through the SDN-based backbone network. Figure 3.2 shows the topology of the emulated 6LoWPAN sensor network in Mininet-WiFi 1 and Mininet-WiFi 2.

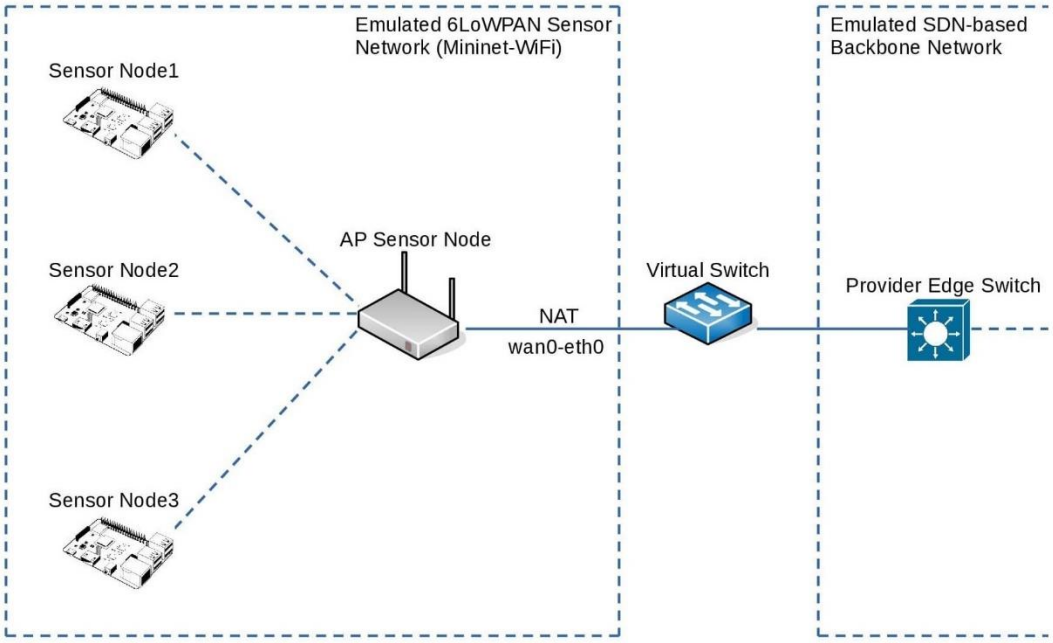


Figure 3.2: Virtual network connection between emulated 6LoWPAN IoT sensor network and provider edge network node of SDN-based backbone network.

3.2 Implementation of Delay Awareness Routing for IoT Traffic in SDN-based Fault-Tolerant Backbone Network

In the case of implementing the fault-tolerant delay awareness routing in this work, there are three main parameters required for the RYU controller to select the path. They are (i) delay of the path, (ii) packet loss ratio, and (iii) the number of hops. The three parameters are given different priorities: the delay of the path is defined as the first priority, the packet loss ratio is specified as the second priority, and the number of hops is assigned as the last priority. In this work, it is assumed that the sensor nodes of the 6LoWPAN sensor networks will send UDP text messages to the server networks as sensor data traffic which will not require high bandwidth usage. Another consideration is that the sensor data needs to be nearly sent from the 6LoWPAN sensor network to the server in real-time. Therefore, the delay of the path between two provider edge network nodes (OVS 1 and OVS 5) is considered as the first priority. The packet loss ratio and the number of hops are considered as the second priority

and third priority respectively. In this work, bandwidth will not be taken into account in selecting the best path by the RYU controller because the UDP text messages will only be carried along the path.

For measuring the three parameters, the provider edge network node (OVS 1) is responsible for all three paths (upper path, middle path, and lower path) because OVS 1 receives the data traffic from sensor network 1 and sensor network 2. ICMPv6 packets will be used to measure the path delay between two provider edge network nodes (OVS 1 and OVS 5). The packet size of the ICMPv6 packet will be adjusted with the packet size of the generated IoT traffic from the sensor node to provide the correct delay information for the RYU controller in route selection. From this measuring, the values of the packet loss ratio regarding three predefined paths can also be recognized.

The number of hops can be measured by using the Time-To-Live (TTL) information of the ICMPv6 packet. In the upper path, there are three OVS nodes to relay the data traffic from OVS 1 to OVS 5. Before the data traffic is relayed, TTL will be reduced at each OVS 2, OVS 3, and OVS 4. In this way, the number of hop information can be collected.

Then, the provider edge network node (OVS 1) encapsulates the measured values of three parameters into the UDP packet and sends the UDP packet to the RYU controller. On the other hand, the RYU controller recognizes each OVS switch by requesting the DPID through SBI by using OpenFlow echo request and OpenFlow echo reply packets. The RYU controller then receives the UDP packet with encapsulated parameter values from OVS 1. The RYU controller then decapsulates the UDP packet to get the parameter values used in decision-making for routing.

The RYU controller will use a UDP packet for two purposes. The first purpose is to obtain the measurement values of the three parameters required in selecting the best path for data traffic. The second purpose is to detect the path failure in the data plane.

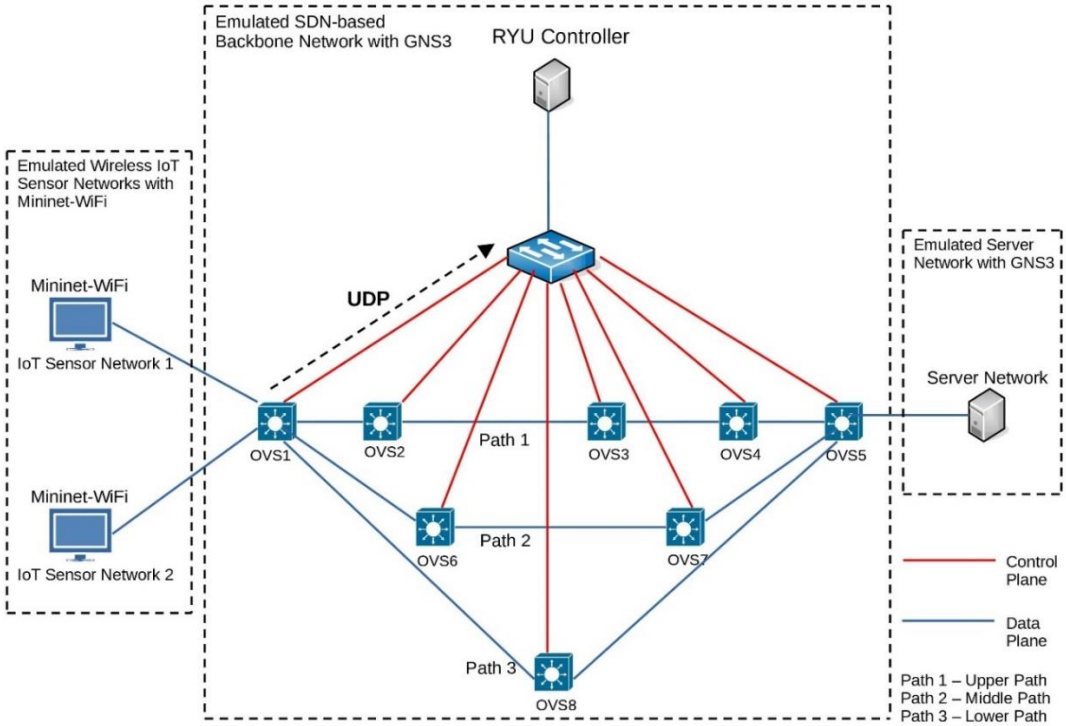


Figure 3.3: The traffic flow of UDP packet between provider edge network node and RYU controller.

In the routing scenario, the RYU controller firstly compares the delay value and decides to select the path with a minimum delay. The RYU controller then installs the flow rules into the provider edge network nodes (OVS 1 and OVS 5) to carry the traffic on the path with the lowest delay. The selected path is maintained for a specific amount of time to allow the traffic flow. The RYU controller checks the delay values again after the allowed period to keep the path at a minimum delay value. If the delay values of the three paths are the same, the RYU controller will consider the packet loss ratio as a second priority and choose the path with the least packet loss ratio value. Otherwise, if the comparing results are still the same, the number of hops is the last parameter to be checked by the RYU controller to choose the best path for data traffic.

In the fault-tolerant, the failure cases of the control plane as well as the data plane will be considered. In the control plane, the detection of network node (OVS) failure is considered in this work. The liveness of the OVS node will be detected by the RYU controller by using ICMP packets. In this case, the node failure is detected by sending and receiving the ICMP request and reply packets between the RYU controller and each OVS node to check whether the node is still connected or not. The traditional way of checking for the liveness of the OpenFlow session will be applied in the control plane.

In the data plane, the failure of the path between the provider edge nodes (OVS 1 and OVS 5) is considered in this work. In this case, the traditional method such as Link Layer Discovery Protocol (LLDP) for link failure or path failure detection can be used. However, it will not be applied in this thesis. The path failure will be checked based on the measurement values of three parameters (path delay, packet loss ratio, and the number of hops) between provider edge switches (OVS 1 and OVS 5). OVS 1 receives the measurement values of Path 1, Path 2, and Path 3 if there is no failure in the data plane. If OVS 1 does not receive any measurement values from Path 1, the situation can be considered as a failure in Path 1. As a result, OVS 1 will simply put the null value in the measurement variable for Path 1 and send the value to the RYU controller. Therefore, the RYU controller will recognize that the path is failed by receiving the message with the null value from the OVS 1. As a consequence, the RYU controller will remove Path 1 in the decision-making of routing for the best path. In this way, the failure of Path 2 and Path 3 can also be detected.

In the traditional method of LLDP detection, the LLDP packet is required to be generated by the RYU controller and sent to the OVS. Then, the OVS will forward this received packet to other OVS for failure detection. Therefore, there is some consumed bandwidth for the generated LLDP packet. In this work, path detection is done by checking the measurement values that are encapsulated in the UDP packet at the RYU controller. By using this method, additional bandwidth consumption for path failure detection will be saved.

Furthermore, in the data plane, when the selected path is failed, the RYU controller will not consider this path as the best path again. In this case, the main job of the RYU controller is to decide the best one among the remaining paths for routing path selection. For example, if Path 1 is failed, Path 2 or Path 3 will be chosen by the RYU controller to reroute the IoT traffic based on three main parameters, including delay of the path, packet loss ratio, and the number of hops.

The three main parameters, which are the delay of the path, packet loss ratio, and the number of hops, will be measured by OVS 1 and sent to the RYU controller every 20 seconds. Therefore, the RYU controller can check the failure of the path every 20 seconds and reroute if there is a failure in the data plane.

Chapter 4

Implementation of Testbed Environment

4.1 Implementation of SDN-based Backbone Network

In each OVS node of the SDN-based backbone network, the bridge 0 (br0) is configured with the IP address (192.168.1.5) of the RYU controller along with the TCP port number (6633). The RYU controller is configured with an out-of-band connection mode in each OVS node to separate the network interface of the control plane from the data plane. The br0 is set in the secure mode**.** The network interface which is used for the control plane is not required to be controlled by the RYU controller and is removed from each OVS bridge to protect against unwanted looping. The control plane network is 192.168.1.0/24. The eth0 of each OVS is connected to the RYU controller. The IPv4 addresses of OVS nodes are 192.168.1.1 for OVS 1, 192.168.1.2 for OVS 2, 192.168.1.3 for OVS 3, 192.168.1.7 for OVS 4, 192.168.1.4 for OVS 5, 192.168.1.8 for OVS 6, 192.168.1.9 for OVS 7, and 192.168.1.6 for OVS 8. Among three paths between OVS 1 and OVS 5 (provider edge nodes), OVS 1, OVS 3, and OVS 4 are internal nodes in the upper path, OVS 6 and OVS 7 are provider nodes in the middle path, and OVS 8 is a provider node in the lower path which are connected through eth1 and eth2.

The OVS 1 and OVS 5 are configured with the 2004::/64 network at eth1 to build the upper path in the data plane. For the middle path, the eth5 of OVS 1 and OVS 5 are assigned with the 2005::/64 network. For the lower path, the eth2 of OVS 1 and OVS 5 are configured with the 2006::/64 network. In each internal OVS node, there are OpenFlow rules to transfer the traffic from the 6LoWPAN IoT sensor networks 1 and 2 between provider edge nodes. Since the data plane of the SDN-based backbone network is built in the IPv6 network, each internal OVS node needs to carry the IPv6 data traffic between the provider edge nodes. Therefore, the OpenFlow rules that forward the IPv6 data traffic must be installed in each internal node. In the upper path, the forwarding OpenFlow rules are installed in OVS 2, OVS 3, and OVS 4. The OVS 6, OVS 7 in the middle path, and OVS 8 in the lower path are installed with the Openflow rules. The OpenFlow rules to route the sensor traffic (UDP or ICMPv6) are installed in OVS 1 and OVS 5 by the RYU controller after choosing the best path.

The OVS 1 is connected to Mininet-WiFi 1 through eth3 and to Mininet-WiFi 2 through eth6. The IPv6 address for eth3 is 2001::20 to reach the 2001::/64 network of Mininet-WiFi 1

and the IPv6 address for eth6 is 2002::20 to reach the 2002::/ 64 network of Mininet-WiFi 2. The gateway addresses of 2001::100 and 2002::100 are specified in OVS 1 to connect to the 6LoWPAN IoT sensor networks 1 and 2. The eth3 of OVS5 is configured with the IPv6 address of 2007::10 to reach the 2007::/64 network of the server.

Table 4.1: Network addresses of SDN-based backbone network.

Control Plane Network	Data Plane Network		
	Upper Path	Middle Path	Lower Path
192.168.1.0/24	2004::/64	2005::/64	2006::/64

Table 4.2: Network addresses of edge network.

Edge Networks		Sensor Networks		Server Network
Mininet-WiFi 1	Mininet-WiFi 2	Sensor Network 1	Sensor Network 2	
2001::/64	2002::/64	2003::/64	2009::/64	2007::/64

Table 4.3: IPv4 or IPv6 address of each node in the SDN-based backbone network and edge network.

No.	Network Nodes	Interface	Control Plane Network (IPv4)	Data Plane Network (IPv6)	Edge Network (IPv6)	Server Network (IPv6)
1.	RYU controller	eth0	192.168.1.5 /24			
2.	OVS 1 (provider edge node connected to Mininet-WiFi 1 and Mininet-WiFi 2)	eth0	192.168.1.1 /24			
		eth1		2004::20 /64		
		eth2		2005::20 /64		
		eth5		2006::20 /64		
		eth3			2001::20 /64	
		eth6			2002::20 /64	
3.	OVS 2 (provider node in the upper	eth0	192.168.1.2 /24			

	path)					
4.	OVS 3 (provider node in the upper path)	eth0	192.168.1.3 /24			
5.	OVS 4 (provider node in the upper path)	eth0	192.168.1.7 /24			
6.	OVS 5 (provider edge node connected to server network)	eth0	192.168.1.4 /24			
		eth1		2005::20 /64		
		eth2		2006::20 /64		
		eth5		2006::20 /64		
		eth3				2007::20 /64
7.	OVS 6 (provider node in the middle path)	eth0	192.168.1.8 /24			
8.	OVS 7 (provider node in the middle path)	eth0	192.168.1.9 /24			
9.	OVS 8 (provider node in the lower path)	eth0	192.168.1.6 /24			
10.	Mininet-WiFi 1	ens32			2001::100 /64	
11.	Mininet-WiFi 2	ens36			2002::100 /64	
12.	Server	eth0				2007::10 /64

4.2 Implementation of 6LoWPAN IoT Sensor Networks

In Mininet-WiFi 1, the 6LoWPAN IoT sensor network 1 is built with 10 sensor nodes and one AP node by using the HWSIM module. The interfaces (sensor1-pan0 to sensor10-pan0) of 10 sensor nodes are configured with the IPv6 addresses from 2003::1/64 to 2003::10/64. The AP node is assigned with the IPv6 address of 2003::60/64 at the ap1-pan0 interface. Through the AP node, 10 sensors are connected to the 2003::/64 network. The Mininet-WiFi 1 is configured with 2001::100 to reach the 2001::/64 network of OVS 1. The AP node is

assigned with the gateway address of 2001::20 to route the traffic between sensor network 1 and the SDN-based backbone network.

In Mininet-WiFi 2, the 6LoWPAN IoT sensor network 2 is created with 10 sensor nodes and one AP node. The IPv6 addresses of 10 sensor nodes are from 2009::1/64 to 2009::10/64 configured at sensor1-pan0 to sensor10-pan0. The ap1-pan0 of the AP node is assigned with the 2009::60/64 IPv6 address. The ens36 interface of Mininet-WiFi 2 is configured with 2002::100 to connect to the 2002::/64 network of OVS 1. The gateway address of the AP node is 2002::20 to route the traffic between sensor network 2 and the SDN-based backbone network. Sensor networks 1 and 2 measure the end-to-end delay (from the sensor to the server) by using ICMPv6 packets. The end-to-end delay is measured in each sensor based on the optimal path of the backbone network which is chosen by the RYU controller. The UDP sensor messages from sensor networks 1 and 2 are sent to the server through the optimal path of the SDN-based backbone network.

Table 4.4: IPv6 addresses of each node in 6LoWPAN sensor networks 1 and 2.

Sensor Networks	Network Nodes	Interface	IPv6 address
Sensor Network 1	Sensor 1	sensor1-pan0	2003::1/64
	Sensor 2	sensor2-pan0	2003::2/64
	Sensor 3	sensor3-pan0	2003::3/64
	Sensor 4	sensor4-pan0	2003::4/64
	Sensor 5	sensor5-pan0	2003::5/64
	Sensor 6	sensor6-pan0	2003::6/64
	Sensor 7	sensor7-pan0	2003::7/64
	Sensor 8	sensor8-pan0	2003::8/64
	Sensor 9	sensor9-pan0	2003::9/64
	Sensor 10	sensor10-pan0	2003::10/64
	AP 1	ap1-pan0	2003::60/64
Sensor Network 2	Sensor 1	sensor1-pan0	2009::1/64
	Sensor 2	sensor2-pan0	2009::2/64
	Sensor 3	sensor3-pan0	2009::3/64
	Sensor 4	sensor4-pan0	2009::4/64
	Sensor 5	sensor5-pan0	2009::5/64
	Sensor 6	sensor6-pan0	2009::6/64
	Sensor 7	sensor7-pan0	2009::7/64
	Sensor 8	sensor8-pan0	2009::8/64
	Sensor 9	sensor9-pan0	2009::9/64
	Sensor 10	sensor10-pan0	2009::10/64
	AP 1	ap1-pan0	2009::60/64

4.3 Installation of OpenFlow Rules in OVS Nodes

To forward the ICMPv6 traffic as well as UDP traffic between OVS 1 and OVS 5, there are the OpenFlow rules defined in the OVS nodes. In the upper path, OVS 2, OVS 3, and OVS 4 are installed with the match-action flow rules. For UDP sensor messages, the flow rule "in_port=2, priority=5, eth_type=0x86dd, ipv6_dst=2007::20, udp_dst=12345, actions=3" in OVS 2 , OVS 3, OVS 4 matches the incoming packet at input port 2 for ethernet type, IPv6 destination address, and UDP destination port. Since UDP sensor messages are sent over the IPv6, the ethernet type for IPv6 address, 0x86dd, is checked. Then the IPv6 destination address of the server (2007::20) and the UDP destination port of the server (12345) are matched to transfer the incoming UDP messages on the output port 3. For the ICMPv6 packet, IPv6 neighbor solicitation and IPv6 neighbor advertisement are specified to query the source link-layer address and target link-layer address. For IPv6 neighbor solicitation, the two flow rules "in_port=2, priority=10, eth_type=0x86dd, ip_proto=58, icmp_type=135, actions=3" and "in_port=3, priority=10, eth_type=0x86dd, ip_proto=58, icmp_type=135, actions=2" are installed in OVS 2, OVS 3, and OVS 4. In the flow rules, the ethernet type for IPv6 is 0x86dd, the protocol number of the header for ICMPv6 is 58, and the IPv6 neighbor solicitation for ICMPv6 is 135.

For IPv6 neighbor advertisement in OVS 2, OVS 3, and OVS 4, the two flow rules "in_port=2, priority=10, eth_type=0x86dd, ip_proto=58, icmp_type=136, actions=3" " (port 2 is eth0 or eth1 and actions 3 is also which port) and "in_port=3, priority=10, eth_type=0x86dd, ip_proto=58, icmp_type=136, actions=2" are set to match the IPv6 neighbor advertisement of 136 for ICMPv6 packets. After the link-layer addresses for source and destination are identified, the ICMPv6 packets are allowed to transfer on output port with two flow rules "in_port=2, priority=5, eth_type=0x86dd, ip_proto=58, actions=3" and " in_port=3, priority=5, eth_type=0x86dd, ip_proto=58, actions=2" in each OVS node. Similarly, OVS 6 and OVS 7 in the middle path and OVS 8 in the lower path need to be installed with the same flow rules as OVS 2, OVS 3, and OVS 4 to transfer UDP and ICMPv6 traffic between the provider edge nodes (OVS 1 and OVS 5).

For decrementing the TTL value for each hop along the path, OVS 2, OVS 3, and OVS 4 requires to be installed with two flow rules "in_port=2, ipv6_src=2004::20, ipv6_dst=2004::10, priority=210, eth_type=0x86dd, actions=dec_ttl,3" and "in_port=3, ipv6_src=2004::10, ipv6_dst=2004::20, priority=210, eth_type=0x86dd, actions=dec_ttl,2". For the upper path, the IPv6 address 2004::/64 is assigned to match the source IPv6 address and target IPv6 address. If the IPv6 addresses are checked for the ICMPv6 packet which is

specified with the ethernet type 0x86dd, the value of TTL is decreased by 1 at OVS 2, OVS 3, and OVS 4. Likewise, the OVS 6, and OVS 7 of the middle path use the same flow rules with 2005::/64 network addresses " in_port=2, ipv6_src=2005::20, ipv6_dst=2005::10, priority=210, eth_type=0x86dd, actions=dec_ttl,3" and "in_port=3, ipv6_src=2005::10, ipv6_dst=2005::20, priority=210, eth_type=0x86dd, actions=dec_ttl,2" to decrement the number of hops in each OVS node. In the lower path, the OVS 8 is assigned with the 2006::/64 network, therefore the flow rules "in_port=2, ipv6_src=2006::20, ipv6_dst=2006::10, priority=210, eth_type=0x86dd, actions=dec_ttl,3" and "in_port=3, ipv6_src=2006::10, ipv6_dst=2006::20, priority=210, eth_type=0x86dd, actions=dec_ttl,2" are installed.

```
/etc/network # ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=1523.009s, table=0, n_packets=522, n_bytes=61596, idle_age=6,
priority=210,ipv6,in_port=2,ipv6_src=2004::20,ipv6_dst=2004::10 actions=dec_ttl,output:3

cookie=0x0, duration=1523.004s, table=0, n_packets=522, n_bytes=61596, idle_age=6,
priority=210,ipv6,in_port=3,ipv6_src=2004::10,ipv6_dst=2004::20 actions=dec_ttl,output:2

cookie=0x0, duration=1543.033s, table=0, n_packets=110, n_bytes=9188, idle_age=16,
priority=10,icmp6,in_port=2,icmp_type=135 actions=output:3

cookie=0x0, duration=1543.030s, table=0, n_packets=109, n_bytes=9110, idle_age=16,
priority=10,icmp6,in_port=3,icmp_type=135 actions=output:2

cookie=0x0, duration=1543.027s, table=0, n_packets=36, n_bytes=2808, idle_age=22,
priority=10,icmp6,in_port=2,icmp_type=136 actions=output:3

cookie=0x0, duration=1543.024s, table=0, n_packets=53, n_bytes=4422, idle_age=22,
priority=10,icmp6,in_port=3,icmp_type=136 actions=output:2

cookie=0x0, duration=1543.035s, table=0, n_packets=23, n_bytes=1565, idle_age=15,
priority=5,ipv6,in_port=2,ipv6_dst=2007::20 actions=output:3

cookie=0x0, duration=1543.021s, table=0, n_packets=10, n_bytes=1020, idle_age=397,
priority=5,icmp6,in_port=2 actions=output:3

cookie=0x0, duration=1543.019s, table=0, n_packets=2, n_bytes=140, idle_age=3,
priority=5,icmp6,in_port=3 actions=output:2
```

(a) OVS 2

```
/etc/network # ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=1627.912s, table=0, n_packets=558, n_bytes=65844, idle_age=11,
priority=210,ipv6,in_port=2,ipv6_src=2004::20,ipv6_dst=2004::10 actions=dec_ttl,output:3

cookie=0x0, duration=1627.909s, table=0, n_packets=558, n_bytes=65844, idle_age=11,
priority=210,ipv6,in_port=3,ipv6_src=2004::10,ipv6_dst=2004::20 actions=dec_ttl,output:2

cookie=0x0, duration=1647.931s, table=0, n_packets=115, n_bytes=9610, idle_age=26,
priority=10,icmp6,in_port=2,icmp_type=135 actions=output:3

cookie=0x0, duration=1647.928s, table=0, n_packets=114, n_bytes=9524, idle_age=44,
priority=10,icmp6,in_port=3,icmp_type=135 actions=output:2

cookie=0x0, duration=1647.925s, table=0, n_packets=38, n_bytes=2964, idle_age=44,
priority=10,icmp6,in_port=2,icmp_type=136 actions=output:3

cookie=0x0, duration=1647.922s, table=0, n_packets=55, n_bytes=4570, idle_age=26,
priority=10,icmp6,in_port=3,icmp_type=136 actions=output:2

cookie=0x0, duration=1647.934s, table=0, n_packets=23, n_bytes=1565, idle_age=123,
priority=5,ipv6,in_port=2,ipv6_dst=2007::20 actions=output:3

cookie=0x0, duration=1647.919s, table=0, n_packets=11, n_bytes=1090, idle_age=111,
priority=5,icmp6,in_port=2 actions=output:3

cookie=0x0, duration=1647.917s, table=0, n_packets=1, n_bytes=70, idle_age=111,
priority=5,icmp6,in_port=3 actions=output:2
```

(b) OVS 3

```
/etc/network # ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=1681.645s, table=0, n_packets=580, n_bytes=68440, idle_age=0,
priority=210,ipv6,in_port=2,ipv6_src=2004::20,ipv6_dst=2004::10 actions=dec_ttl,output:3

cookie=0x0, duration=1681.643s, table=0, n_packets=580, n_bytes=68440, idle_age=0,
priority=210,ipv6,in_port=3,ipv6_src=2004::10,ipv6_dst=2004::20 actions=dec_ttl,output:2

cookie=0x0, duration=1701.662s, table=0, n_packets=118, n_bytes=9860, idle_age=14,
priority=10,icmp6,in_port=2,icmp_type=135 actions=output:3

cookie=0x0, duration=1701.660s, table=0, n_packets=117, n_bytes=9774, idle_age=43,
priority=10,icmp6,in_port=3,icmp_type=135 actions=output:2

cookie=0x0, duration=1701.657s, table=0, n_packets=39, n_bytes=3042, idle_age=48,
priority=10,icmp6,in_port=2,icmp_type=136 actions=output:3

cookie=0x0, duration=1701.654s, table=0, n_packets=47, n_bytes=3682, idle_age=14,
priority=10,icmp6,in_port=3,icmp_type=136 actions=output:2

cookie=0x0, duration=1701.666s, table=0, n_packets=23, n_bytes=1565, idle_age=179,
priority=5,ipv6,in_port=2,ipv6_dst=2007::20 actions=output:3

cookie=0x0, duration=1701.652s, table=0, n_packets=8, n_bytes=688, idle_age=167,
priority=5,icmp6,in_port=2 actions=output:3

cookie=0x0, duration=1701.650s, table=0, n_packets=6, n_bytes=556, idle_age=36,
priority=5,icmp6,in_port=3 actions=output:2
```

(c) OVS 4

```
/etc/network # ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=1687.303s, table=0, n_packets=584, n_bytes=68912, idle_age=1,
priority=210,ipv6,in_port=2,ipv6_src=2005::20,ipv6_dst=2005::10 actions=dec_ttl,output:3

cookie=0x0, duration=1687.298s, table=0, n_packets=583, n_bytes=68794, idle_age=1,
priority=210,ipv6,in_port=3,ipv6_src=2005::10,ipv6_dst=2005::20 actions=dec_ttl,output:2

cookie=0x0, duration=1707.330s, table=0, n_packets=116, n_bytes=9688, idle_age=0,
priority=10,icmp6,in_port=2,icmp_type=135 actions=output:3

cookie=0x0, duration=1707.326s, table=0, n_packets=117, n_bytes=9790, idle_age=17,
priority=10,icmp6,in_port=3,icmp_type=135 actions=output:2

cookie=0x0, duration=1707.323s, table=0, n_packets=46, n_bytes=3588, idle_age=17,
priority=10,icmp6,in_port=2,icmp_type=136 actions=output:3

cookie=0x0, duration=1707.320s, table=0, n_packets=56, n_bytes=4800, idle_age=0,
priority=10,icmp6,in_port=3,icmp_type=136 actions=output:2

cookie=0x0, duration=1707.333s, table=0, n_packets=58, n_bytes=3944, idle_age=134,
priority=5,ipv6,in_port=2,ipv6_dst=2007::20 actions=output:3

cookie=0x0, duration=1707.318s, table=0, n_packets=14, n_bytes=1492, idle_age=767,
priority=5,icmp6,in_port=2 actions=output:3

cookie=0x0, duration=1707.314s, table=0, n_packets=4, n_bytes=296, idle_age=373,
priority=5,icmp6,in_port=3 actions=output:2
```

(d) OVS 6

```
/etc/network # ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=1742.013s, table=0, n_packets=600, n_bytes=70800, idle_age=11,
priority=210,ipv6,in_port=2,ipv6_src=2005::20,ipv6_dst=2005::10 actions=dec_ttl,output:3

cookie=0x0, duration=1742.011s, table=0, n_packets=600, n_bytes=70800, idle_age=11,
priority=210,ipv6,in_port=3,ipv6_src=2005::10,ipv6_dst=2005::20 actions=dec_ttl,output:2

cookie=0x0, duration=1762.032s, table=0, n_packets=123, n_bytes=10266, idle_age=5,
priority=10,icmp6,in_port=2,icmp_type=135 actions=output:3

cookie=0x0, duration=1762.029s, table=0, n_packets=133, n_bytes=11334, idle_age=5,
priority=10,icmp6,in_port=3,icmp_type=135 actions=output:2

cookie=0x0, duration=1762.027s, table=0, n_packets=57, n_bytes=4694, idle_age=10,
priority=10,icmp6,in_port=2,icmp_type=136 actions=output:3

cookie=0x0, duration=1762.024s, table=0, n_packets=43, n_bytes=3362, idle_age=10,
priority=10,icmp6,in_port=3,icmp_type=136 actions=output:2

cookie=0x0, duration=1762.038s, table=0, n_packets=69, n_bytes=4693, idle_age=42,
priority=5,ipv6,in_port=2,ipv6_dst=2007::20 actions=output:3

cookie=0x0, duration=1762.021s, table=0, n_packets=2, n_bytes=140, idle_age=173,
priority=5,icmp6,in_port=2 actions=output:3

cookie=0x0, duration=1762.019s, table=0, n_packets=3, n_bytes=226, idle_age=566,
priority=5,icmp6,in_port=3 actions=output:2
```

(e) OVS 7

```
/etc/network # ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=1881.793s, table=0, n_packets=642, n_bytes=75756, idle_age=5,
priority=210,ipv6,in_port=2,ipv6_src=2006::20,ipv6_dst=2006::10 actions=dec_ttl,output:3

cookie=0x0, duration=1881.790s, table=0, n_packets=642, n_bytes=75756, idle_age=5,
priority=210,ipv6,in_port=3,ipv6_src=2006::10,ipv6_dst=2006::20 actions=dec_ttl,output:2

cookie=0x0, duration=1901.814s, table=0, n_packets=232, n_bytes=19352, idle_age=3,
priority=10,icmp6,in_port=2,icmp_type=135 actions=output:3

cookie=0x0, duration=1901.810s, table=0, n_packets=218, n_bytes=18204, idle_age=3,
priority=10,icmp6,in_port=3,icmp_type=135 actions=output:2

cookie=0x0, duration=1901.808s, table=0, n_packets=66, n_bytes=5172, idle_age=3,
priority=10,icmp6,in_port=2,icmp_type=136 actions=output:3

cookie=0x0, duration=1901.805s, table=0, n_packets=56, n_bytes=4632, idle_age=3,
priority=10,icmp6,in_port=3,icmp_type=136 actions=output:2

cookie=0x0, duration=1901.819s, table=0, n_packets=474, n_bytes=32250, idle_age=1,
priority=5,ipv6,in_port=2,ipv6_dst=2007::20 actions=output:3

cookie=0x0, duration=1901.803s, table=0, n_packets=26, n_bytes=3040, idle_age=352,
priority=5,icmp6,in_port=2 actions=output:3

cookie=0x0, duration=1901.798s, table=0, n_packets=1, n_bytes=70, idle_age=1401,
priority=5,icmp6,in_port=3 actions=output:2
```

(f) OVS 8

Figure 4.1: OpenFlow rules in OVS nodes of the SDN-based backbone network at (a) OVS 2 (b) OVS 3 (c) OVS 4 (d) OVS 6 (e) OVS 7 (f) OVS 8.

4.4 Routing Algorithm

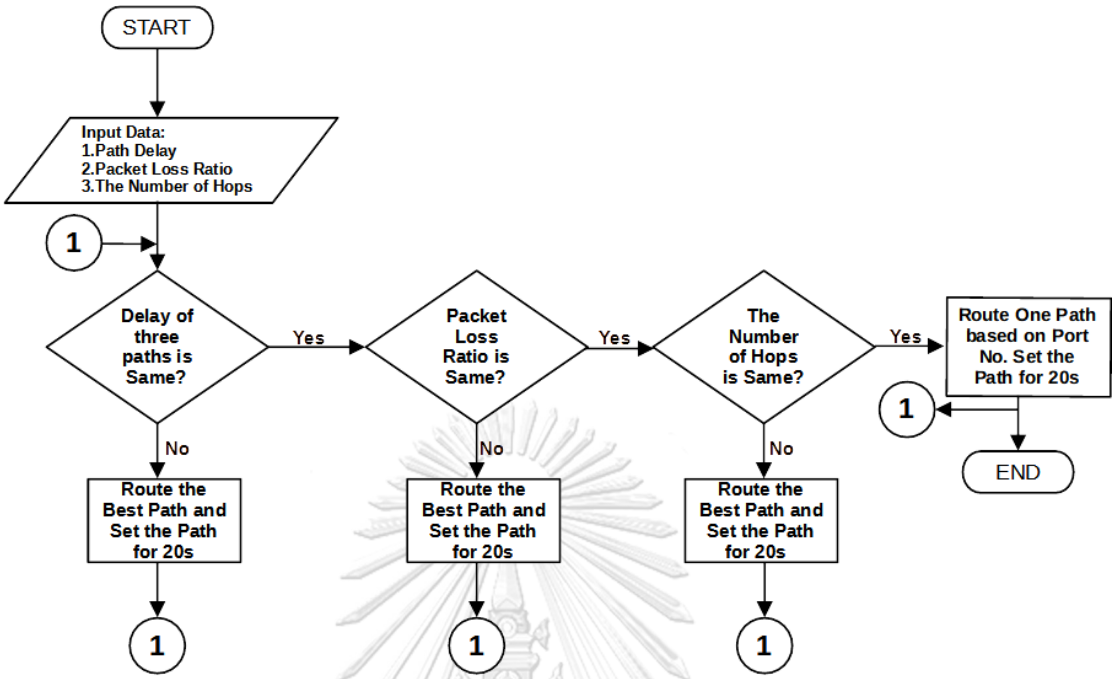


Figure 4.2: Flowchart diagram for routing algorithm

Routing Algorithm for Path Failure Detection and Rerouting

R_c = sdn controller

W = list of working paths in the SDN-based backbone network

V_{min} = minimum value of parameters

τ_p = path detection interval

τ_v = parameter measurement interval

Input:

v_u = encapsulated UDP message from the upper path measurement

v_m = encapsulated UDP message from the middle path measurement

v_l = encapsulated UDP message from the lower path measurement

Measurement Parameters:

d_u = delay of the upper path

d_m = delay of the middle path

d_l = delay of the lower path

r_u = packet loss ratio of the upper path

r_m = packet loss ratio of the middle path

r_l = packet loss ratio of the lower path

h_u = number of hops of the upper path

h_m = number of hops of the middle path

h_l = number of hops of the lower path

Step 1. Begin

Initialization: τ_v = 5s , τ_p = 20s

Step 2. while True

R_c decapsulates v_u , v_m , v_l received from OVS 1 in every τ_v

Step 3. if v_u = '0' and v_m = '0' and v_l = '0'

set W = [0,0,0]

else if v_u = '0' and v_m = '0' and v_l != '0'

set W = [0,0,1]

else if v_u = '0' and v_m != '0' and v_l = '0'

set W = [0,1,0]

else if v_u != '0' and v_m = '0' and v_l = '0'

set W = [1,0,0]

else if v_u = '0' and v_m != '0' and v_l != '0'

set W = [0,1,1]

else if v_u != '0' and v_m = '0' and v_l != '0'

set W = [1,0,1]

else if v_u != '0' and v_m != '0' and v_l = '0'

set W = [1,1,0]

else if v_u != '0' and v_m != '0' and v_l != '0'

set W = [1,1,1]

Step 4. if W = [0,0,0]

All paths are unavailable.

else if W = [1,1,1]

v_u is decapsulated to d_u , r_u , h_u

v_m is decapsulated to d_m , r_m , h_m

v_l is decapsulated to d_l , r_l , h_l

if d_u != d_m != d_l

V_{min} = min(d_u , d_u , d_m)

else if r_u != r_m != r_l

V_{min} = min(r_u , r_m , r_l)

else if h_u != h_m != h_l

V_{min} = min(h_u , h_m , h_l)

else if W = [0,1,1]

v_m is decapsulated to d_m , r_m , h_m

v_l is decapsulated to d_l , r_l , h_l

if d_m != d_l

V_{min} = min(d_u , d_m)

else if r_m != r_l

V_{min} = min(r_m , r_l)

else if h_m != h_l

V_{min} = min(h_m , h_l)

else if W = [1,0,1]

v_u is decapsulated to d_u , r_u , h_u

v_l is decapsulated to d_l , r_l , h_l

if d_u != d_l

V_{min} = min(d_u , d_l)

else if r_u != r_l

V_{min} = min(r_u , r_l)

else if h_u != h_l

V_{min} = min(h_u , h_l)

else if W = [1,1,0]

v_u is decapsulated to d_u , r_u , h_u

v_m is decapsulated to d_m , r_m , h_m

if d_u != d_m

V_{min} = min(d_u , d_m)

else if r_u != r_m

V_{min} = min(r_u , r_m)

else if h_u != h_m

V_{min} = min(h_u , h_m)

else if W = [0,0,1]

v_l is decapsulated to d_l , r_l , h_l

V_{min} = d_l

else if W = [0,1,0]

v_m is decapsulated to d_m , r_m , h_m

V_{min} = d_m

else if W = [1,0,0]

v_u is decapsulated to d_u , r_u , h_u

$V_{min} = d_u$

Step 6. if V_{min} is equal to d_u or r_u or h_u

Upper Path is chosen by R_c

Install OpenFlow rules into provider edge nodes.

if V_{min} is equal to d_m or r_m or h_m

Middle Path is chosen by R_c

Install OpenFlow rules into provider edge nodes.

if V_{min} is equal to d_l or r_l or h_l

Lower Path is chosen R_c

Install OpenFlow rules into provider edge nodes.

sleep for τ_p

goto Step 2

Step 7. End

Node Failure Detection Algorithm

R_n = sdn controller

N = number of OVS nodes in SDN-based backbone network

O_a = list of active OVS nodes connected to R_c

τ_d = node detection interval

$m_{request}$ = ICMP request packets sent to OVS node

m_{reply} = ICMP reply packets received from OVS node

Step 1: Begin

Initialize: O_a = [ ] , τ_d = 5s

Step 2. while True

For n = 1,…, N Do

R_c sends $m_{request}$ to n^{th} OVS node

if R_c receives m_{reply} from n^{th} OVS node then

append n^{th} OVS node to O_a

sleep for τ_d

goto Step 2

Step 3. End

4.5 Hardware and Software Specifications of Testbed Environment

Table 4.5: Hardware specifications of host machine.

Host Machine	Processor	RAM	System Type	OS
ASUS VivoBook X512DA	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GH	16 GB	64-bit operating system, x64 based processor	Windows 10

Table 4.6: Software specifications of virtual machines.

1.	VMware Workstation 15 pro	Version	15.5.0
		Hypervisor	Type 2
2.	GNS3 VM	GNS3 server version	2.2.23
		VM version	0.11.1
		RAM	4 GB
		Number of processors	2
3.	Mininet-WiFi VM 1	RAM	2 GB
		Number of processors	2
		OS	Ubuntu 20.04.3 LTS (Focal Fossa)
4.	Mininet-WiFi VM 2	RAM	2 GB
		Number of processors	2
		OS	Ubuntu 20.04.3 LTS (Focal Fossa)

Table 4.7: Software specifications of SDN-based backbone network.

1.	OpenFlow virtual switch	Open vSwitch	Type	Docker Container
			Server	GNS3 VM
			OS	Alpine Linux v3.3
2.	RYU controller	Ubuntu Docker Guest	Type	Docker container
			Server	GNS3 VM
			OS	Ubuntu 16.04.2 LTS (Xenial Xerus)
3.	Server	Ubuntu Docker Guest	Type	Docker container
			Server	GNS3 VM
			OS	Ubuntu 16.04.2 LTS (Xenial Xerus)

In the SDN-based backbone network, the RYU controller is installed on Ubuntu docker guest which uses containerization technology. In containerization technology, the applications are deployed in the container which runs on Linux and shares the kernel of the host machine. The container runs a discrete process, making it lightweight. Some of the advantages of Docker is

1. Flexible: The complex applications can be containerized.
2. Lightweight: The container shares the host kernel.
3. Interchangeable: Update and upgrade can easily be deployed.
4. Portable: The docker can run locally or deploy to the cloud.
5. Scalable: The docker supports automatically distributed replicas of containers.

Chapter 5

Testing and Measurement Result of Proposed Topology

5.1 Routing Path Selection

For routing measurement, the two scenarios are considered. The first scenario is the situation with no failure in the SDN-based backbone network. The second scenario is when there is a single path failure in the SDN-based backbone network. The UDP sensor messages are sent from the 6LoWPAN IoT networks 1 and 2 to the server. There are 10 sensors in each 6LoWPAN IoT network, therefore, the UDP messages of 20 sensors are sent to the server through the SDN-based backbone network. For routing the best path in the SDN-based backbone network, the RYU controller checks the measured parameter values of the three paths including the upper path, middle path, and lower path, which are received from OVS1. The RYU controller decides the optimal path every 20 seconds for sensor traffic. The testing is measured 20 times to compare the number of selected routing paths in the SDN-based backbone network. In Figure 5.1, the graph for the number of selected routing paths when there is no path failure in the SDN-based backbone network is illustrated. As shown in Figure 5.1, the lower path is mostly chosen as the optimal path during the measurement period. According to the proposed delay-awareness routing algorithm, the lower path has the highest chance to be selected as the optimal path in the topology summarized in Figure 3.1.

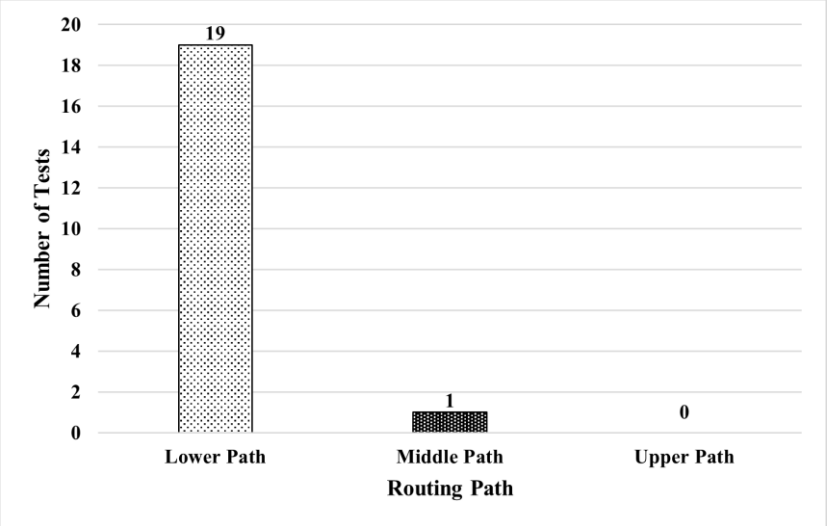


Figure 5.1: Comparison of routing paths chosen by the RYU controller when there is no path failure in the SDN-based backbone network.

In Figure 5.2, three different scenarios are tested for the situation of a single path failure. The number of tests is 20 times, and the results of optimal path selection are compared. The upper path is failed in the first scenario, the middle path is failed in the second scenario, and the lower path is failed in the last scenario. Therefore, there are only two available paths in each scenario to compare the optimal routing path selection by the RYU controller. When the upper path is failed, the remaining paths are the middle path and the lower path. The lower path is chosen more than the middle path. The reason is that there is one more hop in the middle path than the lower path. Therefore, the required time for the packet to reach the destination through the lower path is less than that of passing through the middle path. In the case of the middle path failure, a similar case also happens since the upper path has three hops to reach the server. When the lower path is failed, the middle is mostly chosen as the optimal path since there are fewer hops in the middle path than in the upper path which takes less time to carry packets to the server. The overall testing results that the lower path has the lowest delay, and the upper path has the highest delay in the proposed topology. Therefore, the result in Figure 5.2 shows that the path failure detection from the RYU controller is successful and the delay-awareness routing algorithm works properly in the SDN-based backbone network.

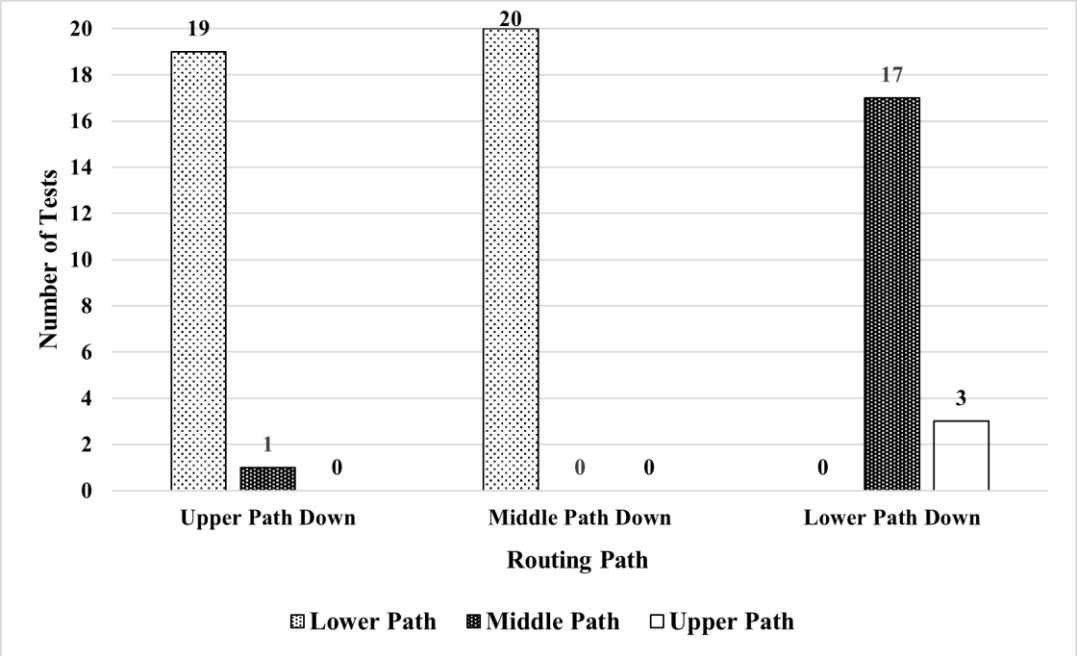


Figure 5.2: Comparison of routing paths chosen by the RYU controller when there is a single path failure in the SDN-based backbone network.

5.2 Reroute Time

The rerouting time is measured when the initial selected optimal path is failed in the backbone network. Figure 5.3 shows the UDP packet captured with the Wireshark tool at the server side to measure the required rerouting time. In this scenario, the time that the server receives the last UDP packet from the sensor node with IPv6 address 2001::100 through the initially selected path is 15:13:17.979. When the selected path is down, there is no incoming packet to the server during the time between 15:13:18.382 and 15:13:32.664. After the RYU controller selects another available path as the optimal path, then the server receives back the UDP packet at 15:13:36.539. The rerouting time is calculated from the absolute time difference between the last packet received from the optimal path and the first packet received from the rerouted path.

Rerouting Time = | (the time that the server receives the last UDP packet from the initial optimal path) - (the time that the server receives the first UDP packet back from the rerouted optimal path after the failure of the initial optimal path) |

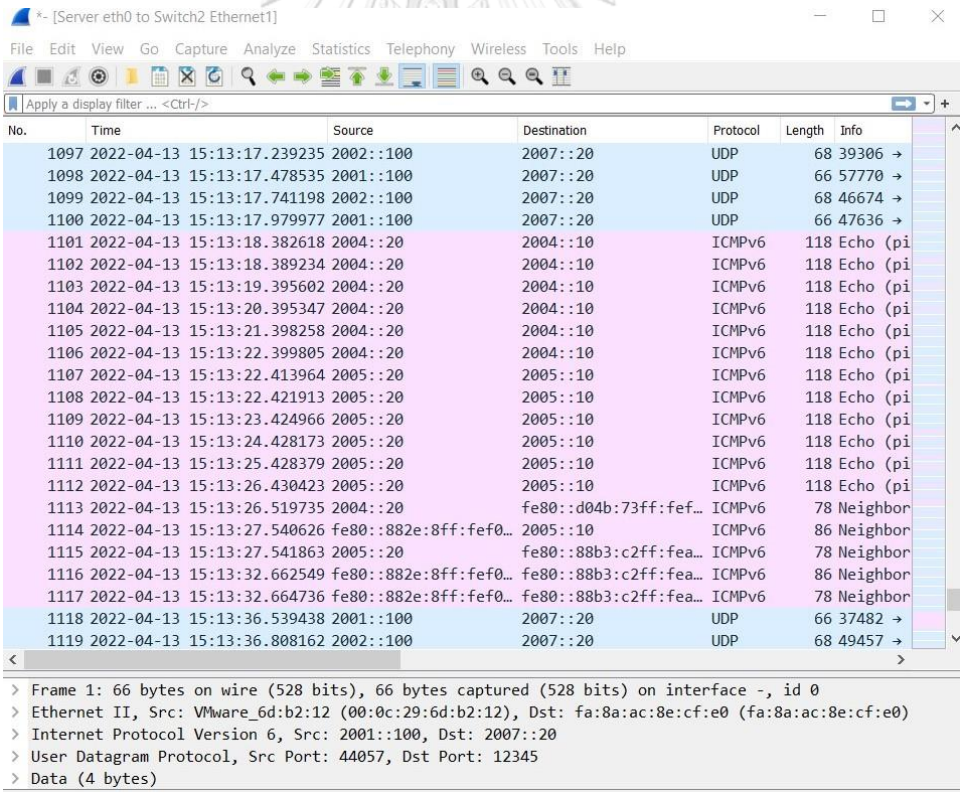


Figure 5.3: UDP packet captured in server to measure rerouting time with Wireshark tool.

The graph of the average rerouting time resulting from 10 times of testing is shown in Figure 5.4. When the lower path is failed, the required average time is around 20 seconds which means that the RYU controller reroutes the other path within the desired measurement of time. When both the lower path and middle path are failed, the RYU controller reroutes to the upper path within an average maximum time of 25 seconds. The proposed reroute scenario maintains the failure path in an acceptable amount of time which is around 20 to 25 seconds.

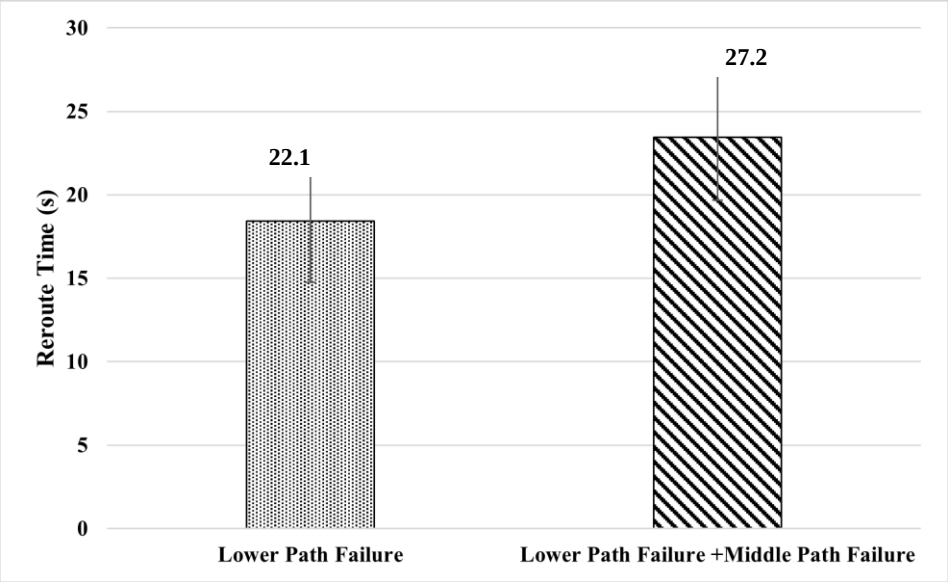


Figure 5.4: Average reroute time with 95-percent confidence interval for 6LoWPAN IoT traffic.

The reroute time of the RYU controller in the case of lower path failure is shown in Table 5.1. The average rerouting time that is resulted from 10 times of testing is about over 18 seconds. Besides, the required reroute time for the case of both the lower path and the middle path failures is about 24 seconds which is shown in Table 5.2. The RYU controller takes more time in the case of two path failure than a single path failure, however, the average rerouting time is the range of the proposed topology of the backbone network.

Table 5.1: Reroute time for lower path failure.

Test No.	Down Time (seconds)	Up Time (seconds)	Reroute Time (seconds)
1.	03:58:18.003	03:58:40.282	22.279
2.	04:34:24.958	04:34:41.473	16.515
3.	04:41:53.236	04:42:04.092	10.856
4.	04:44:48.509	04:45:08.764	20.255
5.	04:52:46.738	04:53:06.210	19.472
6.	04:55:47.023	04:56:09.366	22.343
7.	04:59:38.725	04:59:49.809	11.084
8.	05:02:09.385	05:02:37.427	28.042
9.	05:07:08.386	05:07:30.753	22.367
10.	05:09:16.036	05:09:37.063	21.027
Average Reroute Time (seconds)			18.424 s

Table 5.2: Reroute time for both lower path and middle path failures.

Test No.	Down Time (seconds)	Up Time (seconds)	Reroute Time (seconds)
1.	05:17:52.006	05:18:15.206	23.2
2.	05:30:20.114	05:30:44.041	23.927
3.	05:33:34.333	05:33:59.313	24.980
4.	05:35:36.224	05:35:59.498	23.274
5.	05:37:48.330	05:38:10.943	22.613
6.	05:49:45.858	05:50:05.144	19.286
7.	05:54:22.814	05:54:50.576	27.762
8.	06:02:38.792	06:03:01.015	22.223
9.	06:04:55.531	06:05:18.675	23.144
10.	06:07:17.203	06:07:51.277	34.074
Average Reroute Time (seconds)			23.448 s

5.3 End-to-End Delay Measurement

The results of the end-to-end average delay of each sensor from Mininet-WiFi 1 and Mininet-WiFi 2 to the server are shown in Table 5.3 and Table 5.4. The end-to-end delay through the optimal path is measured when there is no traffic in the SDN-based backbone network. The ICMP packet is used to measure the average end-to-end delay from the sensor to the server through the optimal routing path chosen by the RYU controller in the SDN-based backbone network.

Table 5.3: End-to-end average delay of each sensor from sensor network 1 to the server when there is no traffic in the SDN-based backbone network.

Sensors (Sensor Network 1)	End-to-end average delay (milliseconds)
Sensor 1	3.285
Sensor 2	2.350
Sensor 3	2.969
Sensor 4	3.775
Sensor 5	2.331
Sensor 6	2.764
Sensor 7	2.860
Sensor 8	2.849
Sensor 9	3.551
Sensor 10	2.655

Table 5.4: End-to-end average delay of each sensor from sensor network 2 to server when there is no traffic in the SDN-based backbone network.

Sensors (Sensor Network 2)	End-to-end average Delay (milliseconds)
Sensor 1	2.776
Sensor 2	2.886
Sensor 3	3.174
Sensor 4	3.078
Sensor 5	2.895
Sensor 6	2.767
Sensor 7	2.796
Sensor 8	3.806
Sensor 9	2.804
Sensor 10	2.367

Table 5.5 and Table 5.6 show the measurement results of the end-to-end average delay of sensors from sensor networks 1 and 2 to the server for three cases when there is sensor traffic from both sensor networks 1 and 2 in the SDN-based backbone network. When there is no path failure, the RYU controller chooses the optimal path with a minimum delay which is mostly the lower path in the SDN-based backbone network. Therefore, in the case of no path failure, the end-to-end delay measurement has the minimum value most of the time in contrast to the case of lower path failure or both lower path and middle path failures. In the case of a lower path failure, the end-to-end delay value is increased since the RYU controller chooses mostly the middle path rather than the upper path. When there are failures in both the lower path and middle path in the SDN-based backbone network, the RYU controller chooses only

the upper path for routing the sensor traffic, which causes the end-to-end delay value highest most of the time when compared to the lower path and middle path. Therefore, it can be concluded that the proposed routing algorithm has been working to route the IoT sensor traffic on the optimal path.

Table 5.5: End-to-end average delay of each sensor from sensor network 1 to the server in case of no path failure, one path failure, and two path failures in the SDN-based backbone network.

Sensors (Sensor Network 1)	End-to-end average delay (milliseconds)		
	No Path Failure	Lower Path Failure	Lower Path and Middle Path Failures
Sensor 1	2.541	3.247	4.094
Sensor 2	2.588	3.373	3.846
Sensor 3	2.502	4.676	3.436
Sensor 4	2.981	3.385	4.319
Sensor 5	2.911	3.469	3.728
Sensor 6	2.691	3.435	4.168
Sensor 7	2.957	3.216	4.221
Sensor 8	2.910	4.011	5.151
Sensor 9	2.921	3.620	5.435
Sensor 10	2.734	3.653	4.734
Average (ms)	2.773	3.608	4.313

Table 5.6: End-to-end average delay of each sensor from sensor network 1 to the server in case of no path failure, one path failure, and two path failures in the SDN-based backbone network.

Sensors (Sensor Network 2)	End-to-end average delay (milliseconds)		
	No Path Failure	Lower Path Failure	Lower Path and Middle Path Failures
Sensor 1	2.896	3.466	5.379

Sensor 2	2.928	3.591	5.911
Sensor 3	2.509	3.475	5.896
Sensor 4	3.482	4.061	6.865
Sensor 5	2.797	3.460	5.702
Sensor 6	3.635	4.122	6.617
Sensor 7	2.777	4.269	4.761
Sensor 8	3.097	3.547	5.457
Sensor 9	3.046	4.059	6.305
Sensor 10	2.970	3.894	6.022
Average (ms)	3.013	3.794	5.891

Chapter 6

Conclusions

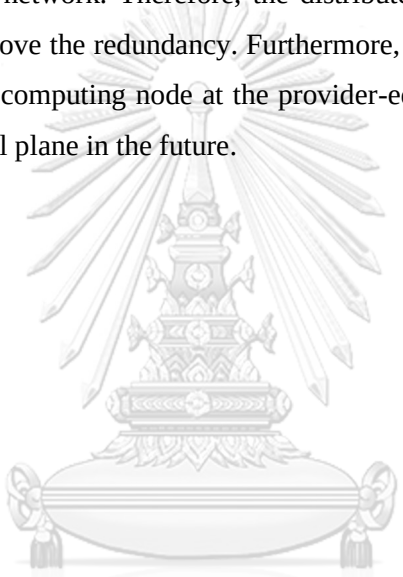
In this thesis, the hybrid form of implementation for edge network in Mininet-WiFi and core network in GNS3 has been successfully proposed. Firstly, in the SDN-based core network, the proposed network topology is built with eight OVS nodes that support the OpenFlow protocol, three different paths, and one centralized RYU controller. The network parameters of three paths are measured between provider edge OVS nodes. In OVS 1 which is connected to sensor networks, the measurement values are decapsulated into the packets and sent to the RYU controller. The RYU controller successfully installs the flow rules into the provider edge nodes to route the best path by following the implemented routing algorithm. Node failure detection, path failure detection, and rerouting for path failure are also tested.

Secondly, in edge networks, the 6LoWPAN IoT sensor network is created and connected through an AP node in Mininet-WiFi VM. Another important step is to connect different VM to establish the hybrid form of the testbed and to route the traffic from sensors of the edge network to the provider edge node of the core network. The GNS3 VM and Mininet-WiFi VM are connected through a virtual switch, therefore, the traffic from sensors is sent to the core network.

Thirdly, the end-to-end delay is directly measured from the sensors to the server. In this case, the RYU controller installs the OpenFlow rules with specific MAC addresses of core networks, and edge networks to route the traffic of sensors to the server and vice versa. The end-to-end delay is successfully and dynamically measured on the optimal path chosen by the RYU controller in the SDN-based core network.. In IPv6 addresses, the additional neighbor solicitation and neighbor advertisement for ICMPv6 are required to be matched to take action to decrement the number of hops in each OVS node. In the next step, the sensor messages are created in the form of UDP packets and sent to the server through a UDP port.

Fourthly, two different sensor networks are created in two Mininet-WiFi VMs to connect with the SDN-based backbone network. Two sensor networks send messages simultaneously to the server on the optimal path, likewise, the end-to-end delay is measured on the same path. In the last step, OVS node failure detection is tested. The ICMP packet is used by the RYU controller to check whether the OVS node is active or not. In overall testing, the selected routing path as well as rerouting time of the RYU controller, and end-to-end delay measurement are tested and recorded for the proposed network.

In this thesis, one main advantage is the contribution of a hybrid form of GNS3 and Mininet-WiFi emulated testbed. From this contribution, the testbed can collaborate with other vendors as well as real or virtual machines to explore more research in the future. This testbed is worked on a small-scale network with centralized monitoring and controlling. Failure of the SDN controller scenario is not considered in this work because there is a single RYU controller in the proposed testbed to control and manage the backbone network. From the point of view of redundancy, a single controller scenario is not suitable because the feature of controllability will be lost when an SDN controller is failed. Moreover, the delay between SDN-enabled network nodes such as OVS and the SDN controller needs to be considered in the large-scale backbone network. Therefore, the distributed multi-SDN controller scenario should be utilized to improve the redundancy. Furthermore, a slave SDN controller should be implemented as the edge computing node at the provider-edge network to reduce bandwidth consumption at the control plane in the future.



REFERENCES

[1] M. Medvetskyi, M. Beshley, and M. Klymash, “A Quality of Experience Management Method for Intent-Based Software-Defined Networks,” 16th International Conference on the Experience of Designing and Application of CAD Systems (CADSM), pp. 59–62, 2021.

[2] D. Lumbantoruan, Z. Fan, A. Mihailovic, and A.Hamid Aghvami, “Provision of Tactile Internet Traffic in IP Access Networks Using SDN-Enabled Multipath Routing,” 27th International Conference on Telecommunications (ICT), 2020.

[3] P. Bardalai, N. Medhi, B. Bargayary, and D. K. Saikia, “OpenHealthQ: OpenFlow based QoS Management of Healthcare Data in a Software-Defined Fog Environment,” IEEE International Conference on Communications (ICC), pp. 1-6, 2021.

[4] M. Saadatpour, T. Shabanian, and M. Behdadfar, “QoS Improvement in SDN Using Centralized Routing Based on Feedback,” International Conference on Information Networking (ICOIN), pp. 132-136, 2021.

[5] N. Albur, S. Handigol, S. Naik, M. M. Mulla, and D.G. Narayan, “QoS-aware Flow Management in Software-Defined Network,” 12th International Conference on Computational Intelligence and Communication Networks (CICN), pp.215-220, 2020.

[6] R. K. Das, N. Ahmed, F. H. Pohrmen, A. K. Maji, and G. Saha, “6LE-SDN: An Edge-Based Software- Defined Network for Internet of Things,” IEEE Internet of Things Journal, vol. 7, pp. 7725-7733, 2020.

[7] N. N. Josbert, W. Ping, M. Wei, M. S. A. Muthanna, and A. Rafiq, “A Frame for Managing Dynamic Routing in Industrial Networks Driven by Software-Defined Networking Technology,” IEEE Access Journal, vol. 9, pp. 74343-74359, 2021.

[8] D. Y. Setiawan, S. N. Hertiana, and R. M. Negara, “6LoWPAN Performance Analysis of IoT Software-Defined-Network-Based Using Mininet-IoT,” IEEE International Conference on Internet of Things and Intelligence System (IoTaIS), pp. 60-65, 2021.

[9] P. E. Numan, K. M. Yusof, J. B. Din, M. N. B. Marsono, U. S. Dauda, S. Nathaniel, and F. K. O, “Quality of Service Evaluation of Software-Defined Internet of Things Network,” ELEKTRIKA- Journal of Electrical Engineering, pp. 65–75,2021.

[10] R. K. Das, A. K. Maji, and G. Saha, “SD-6LN: Improved Existing IoT Framework by Incorporating SDN Approach,” International Conference on Innovative Computing and Communications, pp. 599-606, 2021.

[11] N. Saha, S. Bera, and S. Misra, “Sway: Traffic-Aware QoS Routing in Software-Defined IoT,” IEEE Transactions on Emerging Topics in Computing, vol. 9, pp. 390-401, 2021.

[12] J. M. Llopis, J. Pieczerak, and T. Janaszka, “Minimizing Latency of Critical Traffic through SDN,” IEEE International Conference on Networking, Architecture and Storage (NAS), pp. 1-6, 2016.

[13] O. M. A. Alssaheli, Z. Z. Abidin, N. A. Zakaria, and Z. A. Abas, “Implementation of Network Traffic Monitoring using Software-Defined Networking Ryu Controller,” WSEAS Transactions on Systems and Control Journal, vol. 16, pp. 270-277, 2021.

[14] N. L. M. vanAdrichem, C. Doerr, and F.A. Kuipers, “OpenNetMon: Network Monitoring in OpenFlow Software-Defined Networks,” IEEE Network Operations and Management Symposium (NOMS), pp. 1-8, 2014.

[15] C. Yu, C. Lumezanu, A. Sharma, Q. Xu, G. Jiang, and H. V. Madhyastha, “Software-Defined Latency Monitoring in Data Center Networks,” Springer International Publishing, pp. 360-372, 2015.

[16] A. M. Allakany, and K. Okamura, “Latency Monitoring in Software-Defined Networks,” 12th International Conference on Future Internet Technologies, pp. 1-4, 2017.

[17] D. Sinha, K. Haribabu, and S. Balasubramaniam, “Real-Time Monitoring of Network Latency in Software Defined Networks,” IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS), pp. 1-3, 2015.

[18] P. Phaal, and M. Lavine, “sFlow Version 5,” [Online]. Available from: http://www.sflow.org. Accessed June 2021

[19] RFC 3954, “Cisco Systems NetFlow Services Export Version 9,” [Online]. Available from: http://tools.ietf.org. Accessed June 2021

[20] K. Phemius, and M. Bouet, “Monitoring Latency with OpenFlow,” 9th International Conference on Network and Service Management (CNSM), pp. 122-125, 2013.

[21] S. Wang, J. Zhang, T. Huang, J. LIU, and F. R. YU, “FlowTrace: Measuring Round-Trip Time and Tracing Path in Software-Defined Networking with Low Communication Overhead,” Frontiers of Information Technology & Electronic Engineering Journal, vol. 18, pp. 206-219, 2021.

[22] A. Mosa, and A. Sadi, “Developing an Asynchronous Technique to Evaluate the Performance of SDN HP Aruba switch and OVS,” Springer International Publishing, pp. 569-580, 2019.

[23] S. Ghosh, S. A. Busari, T. Dagiuklas, M. Iqbal, R. Mumtaz, J. Gonzalez, S. Stavrou, and L. Kanaris, “SDN-Sim: Integrating System Level Simulator with Software Defined Network,” IEEE Communications Standards Magazine, vol. 4, pp. 18-25, 2020.

[24] 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.

[25] Y. Li, X. Su, A. Y. Ding, A. Lindgren, X. Liu, C. Prehofer, J. Riekki, R. Rahmani, S. Tarkoma, and P. Hui, “Enhancing the Internet of Things with Knowledge-Driven Software-Defined Networking Technology: Future Perspectives,” Sensors Journal, vol. 20, pp. 34-59, 2020.

[26] I. Alam, K. Sharif, F. Li, Z. Latif, M. M. Karim, B. Nour, S. Biswas, and Y. Wang, “IoT Virtualization: A Survey of Software Definition and Function Virtualization Techniques for Internet of Things,” arXiv 2019, arXiv:1902.10910.

[27] A. Triantafyllou, P. Sarigiannidis, and T. D. Lagkas, “Network Protocols, Schemes, and Mechanisms for Internet of Things (IoT): Features, Open Challenges, and Trends,” Wireless Communications and Mobile Computing Journal, vol. 2018, 24 pages, 2018.

[28] GNS3. [Online]. Available from: https://gns3.com. Accessed April 2022.

[29] Mininet-WiFi. [Online]. Available from: https://mininet-wifi.github.io. Accessed April 2022.

[30] Open Network Foundation, “Software-Defined Networking: The new norm for networks,” ONF White Paper, 2012.

[31] RYU SDN Framework. [Online]. Available from: https://osrg.github.io/ryu-book/en/Ryubook.pdf. Accessed April 2022.

[32] Open vSwitch. [Online]. Available from: http://openvswitch.org. Accessed April 2022.

Appendices

Appendix A

Network Configuration of SDN-based Backbone Network

```
#Network Configuration in OVS 1
nano /etc/network/interfaces
#Static config for eth0 to connect to the RYU controller
auto eth0
iface eth0 inet static
address 192.168.1.1
netmask 255.255.255.0
#Static config for eth1 to set the IPv6 address for the upper path
auto eth1
iface eth1 inet6 static
address 2004::20
netmask 64
#Static config for eth5 to set the IPv6 address for the middle path
auto eth5
iface eth5 inet6 static
address 2005::20
netmask 64
#Static config for eth2 to set the IPv6 address for the lower path
auto eth2
iface eth2 inet6 static
address 2006::20
netmask 64
#Static config for eth3 to connect to the Mininet-WiFi 1
auto eth3
iface eth3 inet6 static
address 2001::20
netmask 64
gateway 2001::100
#Static config for eth6 to connect to the Mininet-WiFi 2
auto eth6
iface eth6 inet6 static
```

```
address 2002::20
netmask 64
gateway 2002::100

#Network Configuration in OVS 2
nano /etc/network/interfaces
#Static config for eth0 to connect to the RYU controller
auto eth0
iface eth0 inet static
address 192.168.1.2
netmask 255.255.255.0

#Network Configuration in OVS 3
nano /etc/network/interfaces
#Static config for eth0 to connect to the RYU controller
auto eth0
iface eth0 inet static
address 192.168.1.3
netmask 255.255.255.0

#Network Configuration in OVS 4
nano /etc/network/interfaces
#Static config for eth0 to connect to the RYU controller
auto eth0
iface eth0 inet static
address 192.168.1.7
netmask 255.255.255.0

#Network Configuration in OVS 5
nano /etc/network/interfaces
#Static config for eth0 to connect to the RYU controller
auto eth0
iface eth0 inet static
address 192.168.1.4
netmask 255.255.255.0
```

```
#Static config for eth1 to set the IPv6 address for the upper path
auto eth1
iface eth1 inet6 static
address 2004::10
netmask 64
#Static config for eth5 to set the IPv6 address for the middle path
auto eth5
iface eth5 inet6 static
address 2005::10
netmask 64
#Static config for eth2 to set the IPv6 address for the lower path
auto eth2
iface eth2 inet6 static
address 2006::10
netmask 64
#Static config for eth2 to connect to the server
auto eth2
iface eth2 inet6 static
address 2007::10
netmask 64

#Network Configuration in OVS 6
nano /etc/network/interfaces
#Static config for eth0 to connect to the RYU controller
auto eth0
iface eth0 inet static
address 192.168.1.8
netmask 255.255.255.0

#Network Configuration in OVS 7
nano /etc/network/interfaces
#Static config for eth0 to connect to the RYU controller
auto eth0
iface eth0 inet static
address 192.168.1.9
```

```
netmask 255.255.255.0

#Network Configuration in OVS 8
nano /etc/network/interfaces
#Static config for eth0 to connect to the RYU controller
auto eth0
iface eth0 inet static
address 192.168.1.6
netmask 255.255.255.0

#Network Configuration in RYU controller
nano /etc/network/interfaces
# Static config for eth0
auto eth0
iface eth0 inet static
address 192.168.1.5
netmask 255.255.255.0

#Network Configuration in Server
nano /etc/network/interfaces
# Static config for eth0
auto eth0
iface eth0 inet6 static
address 2007::20
netmask 64

#Network Configuration in Mininet-WiFi 1
#Static config for ens32
address 2001::100
netmask 64

#Network Configuration in Mininet-WiFi 1
#Static config for ens36
address 2002::100
netmask 64
```

Appendix B

Installation of Necessary Package

#Ubuntu Docker Guest (RYU Controller)

apt-get update

apt-get install python

apt-get install python-pip

apt-get install git

git clone https://github.com/faucetsdn/ryu.git

pip install setuptools = = 33.1.0

pip install pbr = = 2.1.0

pip install pip = = 9.0.0

pip install dnspython = = 1.16.0

pip install oslo.config = = 5.0.0

pip install tinyrpc = = 1.0.1

pip install eventlet = = 0.22.0

pip install ovs = = 2.6.0

pip install ryu

cd ryu

pip install ryu

apt-get update

#OpenvSwitch (Alpine Linux)

apk –no-cache add git

git clone https://github.com/kytos/python-oprnflow.git

apk add --update –no-cache curl py-pip

apk add –update py-pip

apk add py-pip

apk add –no-cache python3

apk add py-setuptools

apk update

#Ubuntu Docker Guest (Server)

apt-get update

apt-get install python

apt-get install python-pip

apt-get update

Appendix C
Establishment of Connection between Data Plane and Control Plane and Installation of OpenFlow Rules in Provider OVS Nodes

```
#Connecting OVS 1 to the RYU controller
ovs-vsctl set-controller br0 tcp: 192.168.1.5:6633
ovs-vsctl set-controller br0 connection-mode=out-of-band
ovs-vsctl set-fail-mode br0 secure
ovs-vsctl del-port br0 eth0
ovs-vsctl set bridge br0 stp-enable=true

#Connecting OVS 2 to the RYU controller
ovs-vsctl set-controller br0 tcp: 192.168.1.5:6633
ovs-vsctl set-controller br0 connection-mode=out-of-band
ovs-vsctl set-fail-mode br0 secure
ovs-vsctl del-port br0 eth0
ovs-vsctl set bridge br0 stp-enable=true
#Installing predefined forwarding OpenFlow rules in OVS 2
nano /etc/network/flows.sh
ovs-ofctl del-flows br0
./rules.sh
sleep 10
./dec.sh
nano /etc/network/rules.sh
ovs-ofctl add-flow br0
in_port=2,priority=5,eth_type=0x86dd,ipv6_dst=2007::20,udp_dst=12345,actions=3
ovs-ofctl add-flow br0
in_port=2,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=135,actions=3
ovs-ofctl add-flow br0
in_port=3,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=135,actions=2
ovs-ofctl add-flow br0
in_port=2,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=136,actions=3
```

```
ovs-ofctl add-flow br0
in_port=3,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=136,actions=2
ovs-ofctl add-flow br0 in_port=2,priority=5,eth_type=0x86dd,ip_proto=58,actions=3
ovs-ofctl add-flow br0 in_port=3,priority=5,eth_type=0x86dd,ip_proto=58,actions=2
nano /etc/network/dec.sh
ovs-ofctl add-flow br0
in_port=2,ipv6_src=2004::20,ipv6_dst=2004::10,priority=210,eth_type=0x86dd,actions=dec_
ttl,3
ovs-ofctl add-flow br0
in_port=3,ipv6_src=2004::10,ipv6_dst=2004::20,priority=210,eth_type=0x86dd,actions=dec_
ttl,2

#Connecting OVS 3 to the RYU controller
ovs-vsctl set-controller br0 tcp: 192.168.1.5:6633
ovs-vsctl set-controller br0 connection-mode=out-of-band
ovs-vsctl set-fail-mode br0 secure
ovs-vsctl del-port br0 eth0
ovs-vsctl set bridge br0 stp-enable=true
#Installing predefined forwarding OpenFlow rules in OVS 3
nano /etc/network/flows.sh
ovs-ofctl del-flows br0
./rules.sh
sleep 10
./dec.sh
nano /etc/network/rules.sh
ovs-ofctl add-flow br0
in_port=2,priority=5,eth_type=0x86dd,ipv6_dst=2007::20,udp_dst=12345,actions=3
ovs-ofctl add-flow br0
in_port=2,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=135,actions=3
ovs-ofctl add-flow br0
in_port=3,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=135,actions=2
ovs-ofctl add-flow br0
in_port=2,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=136,actions=3
ovs-ofctl add-flow br0
in_port=3,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=136,actions=2
```

```
ovs-ofctl add-flow br0 in_port=2,priority=5,eth_type=0x86dd,ip_proto=58,actions=3
ovs-ofctl add-flow br0 in_port=3,priority=5,eth_type=0x86dd,ip_proto=58,actions=2
nano /etc/network/dec.sh
ovs-ofctl add-flow br0
in_port=2,ipv6_src=2004::20,ipv6_dst=2004::10,priority=210,eth_type=0x86dd,actions=dec_
ttl,3
ovs-ofctl add-flow br0
in_port=3,ipv6_src=2004::10,ipv6_dst=2004::20,priority=210,eth_type=0x86dd,actions=dec_
ttl,2

#Connecting OVS 4 to the RYU controller
ovs-vsctl set-controller br0 tcp: 192.168.1.5:6633
ovs-vsctl set-controller br0 connection-mode=out-of-band
ovs-vsctl set-fail-mode br0 secure
ovs-vsctl del-port br0 eth0
ovs-vsctl set bridge br0 stp-enable=true
#Installing predefined forwarding OpenFlow rules in OVS 4
nano /etc/network/flows.sh
ovs-ofctl del-flows br0
./rules.sh
sleep 10
./dec.sh
nano /etc/network/rules.sh
ovs-ofctl add-flow br0
in_port=2,priority=5,eth_type=0x86dd,ipv6_dst=2007::20,udp_dst=12345,actions=3
ovs-ofctl add-flow br0
in_port=2,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=135,actions=3
ovs-ofctl add-flow br0
in_port=3,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=135,actions=2
ovs-ofctl add-flow br0
in_port=2,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=136,actions=3
ovs-ofctl add-flow br0
in_port=3,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=136,actions=2
ovs-ofctl add-flow br0 in_port=2,priority=5,eth_type=0x86dd,ip_proto=58,actions=3
ovs-ofctl add-flow br0 in_port=3,priority=5,eth_type=0x86dd,ip_proto=58,actions=2
```

```
nano /etc/network/dec.sh
ovs-ofctl add-flow br0
in_port=2,ipv6_src=2004::20,ipv6_dst=2004::10,priority=210,eth_type=0x86dd,actions=dec_
ttl,3
ovs-ofctl add-flow br0
in_port=3,ipv6_src=2004::10,ipv6_dst=2004::20,priority=210,eth_type=0x86dd,actions=dec_
ttl,2

#Connecting OVS 5 to the RYU controller
ovs-vsctl set-controller br0 tcp: 192.168.1.5:6633
ovs-vsctl set-controller br0 connection-mode=out-of-band
ovs-vsctl set-fail-mode br0 secure
ovs-vsctl del-port br0 eth0
ovs-vsctl set bridge br0 stp-enable=true

#Connecting OVS 6 to the RYU controller
ovs-vsctl set-controller br0 tcp: 192.168.1.5:6633
ovs-vsctl set-controller br0 connection-mode=out-of-band
ovs-vsctl set-fail-mode br0 secure
ovs-vsctl del-port br0 eth0
ovs-vsctl set bridge br0 stp-enable=true
#Installing predefined forwarding OpenFlow rules in OVS 6
nano /etc/network/flows.sh
ovs-ofctl del-flows br0
./rules.sh
sleep 10
./dec.sh
nano /etc/network/rules.sh
ovs-ofctl add-flow br0
in_port=2,priority=5,eth_type=0x86dd,ipv6_dst=2007::20,udp_dst=12345,actions=3
ovs-ofctl add-flow br0
in_port=2,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=135,actions=3
ovs-ofctl add-flow br0
in_port=3,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=135,actions=2
```

```
ovs-ofctl add-flow br0
in_port=2,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=136,actions=3
ovs-ofctl add-flow br0
in_port=3,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=136,actions=2
ovs-ofctl add-flow br0 in_port=2,priority=5,eth_type=0x86dd,ip_proto=58,actions=3
ovs-ofctl add-flow br0 in_port=3,priority=5,eth_type=0x86dd,ip_proto=58,actions=2
nano /etc/network/dec.sh
ovs-ofctl add-flow br0
in_port=2,ipv6_src=2005::20,ipv6_dst=2005::10,priority=210,eth_type=0x86dd,actions=dec_
ttl,3
ovs-ofctl add-flow br0
in_port=3,ipv6_src=2005::10,ipv6_dst=2005::20,priority=210,eth_type=0x86dd,actions=dec_
ttl,2

#Connecting OVS 7 to the RYU controller
ovs-vsctl set-controller br0 tcp: 192.168.1.5:6633
ovs-vsctl set-controller br0 connection-mode=out-of-band
ovs-vsctl set-fail-mode br0 secure
ovs-vsctl del-port br0 eth0
ovs-vsctl set bridge br0 stp-enable=true
#Installing predefined forwarding OpenFlow rules in OVS 7
nano /etc/network/flows.sh
ovs-ofctl del-flows br0
./rules.sh
sleep 10
./dec.sh
nano /etc/network/rules.sh
ovs-ofctl add-flow br0
in_port=2,priority=5,eth_type=0x86dd,ipv6_dst=2007::20,udp_dst=12345,actions=3
ovs-ofctl add-flow br0
in_port=2,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=135,actions=3
ovs-ofctl add-flow br0
in_port=3,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=135,actions=2
ovs-ofctl add-flow br0
in_port=2,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=136,actions=3
```

```
ovs-ofctl add-flow br0
in_port=3,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=136,actions=2
ovs-ofctl add-flow br0 in_port=2,priority=5,eth_type=0x86dd,ip_proto=58,actions=3
ovs-ofctl add-flow br0 in_port=3,priority=5,eth_type=0x86dd,ip_proto=58,actions=2
nano /etc/network/dec.sh
ovs-ofctl add-flow br0
in_port=2,ipv6_src=2005::20,ipv6_dst=2005::10,priority=210,eth_type=0x86dd,actions=dec_
ttl,3
ovs-ofctl add-flow br0
in_port=3,ipv6_src=2005::10,ipv6_dst=2005::20,priority=210,eth_type=0x86dd,actions=dec_
ttl,2

#Connecting OVS 8 to the RYU controller
ovs-vsctl set-controller br0 tcp: 192.168.1.5:6633
ovs-vsctl set-controller br0 connection-mode=out-of-band
ovs-vsctl set-fail-mode br0 secure
ovs-vsctl del-port br0 eth0
ovs-vsctl set bridge br0 stp-enable=true
#Installing predefined forwarding OpenFlow rules in OVS 8
nano /etc/network/flows.sh
ovs-ofctl del-flows br0
./rules.sh
sleep 10
./dec.sh
nano /etc/network/rules.sh
ovs-ofctl add-flow br0
in_port=2,priority=5,eth_type=0x86dd,ipv6_dst=2007::20,udp_dst=12345,actions=3
ovs-ofctl add-flow br0
in_port=2,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=135,actions=3
ovs-ofctl add-flow br0
in_port=3,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=135,actions=2
ovs-ofctl add-flow br0
in_port=2,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=136,actions=3
ovs-ofctl add-flow br0
in_port=3,priority=10,eth_type=0x86dd,ip_proto=58,icmp_type=136,actions=2
```

```
ovs-ofctl add-flow br0 in_port=2,priority=5,eth_type=0x86dd,ip_proto=58,actions=3
ovs-ofctl add-flow br0 in_port=3,priority=5,eth_type=0x86dd,ip_proto=58,actions=2
nano /etc/network/dec.sh
ovs-ofctl add-flow br0
in_port=2,ipv6_src=2006::20,ipv6_dst=2006::10,priority=210,eth_type=0x86dd,actions=dec_
ttl,3
ovs-ofctl add-flow br0
in_port=3,ipv6_src=2006::10,ipv6_dst=2006::20,priority=210,eth_type=0x86dd,actions=dec_
ttl,2
```

Appendix D

Development of Python Program for Parameters Measurement in SDN-based Backbone Network

#This program is written by May Pyone Han from Chulalongkorn University.

#This program is written at OVS 1 to measure three parameters (delay, packet loss, and the number of hops) of the three paths in the SDN-based backbone network and to send the measurement result to the RYU controller.

```
import subprocess
import os
import socket
from subprocess import Popen, PIPE
import re
import time
import shlex
import datetime

while True:
    command_line = "ping6 -c 1 -I 2004::20 2004::10"
    args = shlex.split(command_line)
    try:
        subprocess.check_call(args,stdout=subprocess.PIPE,stderr=subprocess.PIPE)
        print "\nUpper_Path is available."
        hostname = '2004::10'
        process = subprocess.Popen(['ping6','-c','5',hostname],stdout=PIPE, stderr=PIPE)
        stdout, stderr = process.communicate()
        packetloss = float([x for x in stdout.decode('utf-8').split('\n') if x.find('packet loss')
                            != -1][0].split('%')[0].split(' ')[-1])
        if packetloss < 10.0:
            print("\nPacket_Loss_of_Upper_Path is %s percent" % packetloss)
            loss_u = packetloss
            avg_time = float([x for x in stdout.decode('utf-8').split('\n')
                              if x.startswith('round-trip')][0].split('=')[-1].split('/')[1])
```

```
    print("Average_RTT_of_Upper_Path is %s s" %avg_time)
    RTT_u = avg_time
    res=stdout
    if process.returncode > 0:
        print('server error')
    else:
        pattern = re.compile('ttl=\d*')
        pattern = re.search(pattern,stdout)
        ttl=re.split(r'=',pattern.group(0))
        hops_u=64-int(ttl[1])
        print("NumberofHops_in_Upper_Path is %s" %hops_u)

    HOST = '192.168.1.5'
    PORT = 10000
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect((HOST, PORT))
    s.sendall("\n".join([bytes(loss_u), bytes(RTT_u), bytes(hops_u)]))

except subprocess.CalledProcessError:
    print "\nUpper_Path is not available."
    value =bytes(0)
    HOST = '192.168.1.5'
    PORT = 10000
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect((HOST, PORT))
    s.send(value)

command_line = "ping6 -c 1 -I 2005::20 2005::10"
args = shlex.split(command_line)
try:
    subprocess.check_call(args,stdout=subprocess.PIPE,stderr=subprocess.PIPE)
    print "\nMiddle_Path is available."
    hostname = '2005::10'
    process = subprocess.Popen(['ping6','-c','5',hostname],stdout=PIPE, stderr=PIPE)
    stdout, stderr = process.communicate()
```

```
    packetloss = float([x for x in stdout.decode('utf-8').split('\n') if x.find('packet loss')
                                != -1][0].split('%')[0].split(' ')[-1])
    if packetloss < 10.0:
        print("\nPacket_Loss_of_Middle_Path is %s percent" % packetloss)
        loss_m = packetloss
        avg_time = float([x for x in stdout.decode('utf-8').split('\n')
                                if x.startswith('round-trip')][0].split('=')[-1].split('/')[1])
    print("Average_RTT_of_Middle_Path is %s s" %avg_time)
    RTT_m = avg_time
    res=stdout
    if process.returncode > 0:
        print('server error')
    else:
        pattern = re.compile('ttl=\d*')
        pattern = re.search(pattern,stdout)
        ttl=re.split(r'=',pattern.group(0))
        hops_m=64-int(ttl[1])
        print("NumberofHops_in_Middle_Path is %s" %hops_m)

    HOST = '192.168.1.5'
    PORT = 10000
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect((HOST, PORT))
    s.sendall("\n".join([bytes(loss_m), bytes(RTT_m), bytes(hops_m)]))

except subprocess.CalledProcessError:
    print "\nMiddle_Path is not available."
    value =bytes(0)
    HOST = '192.168.1.5'
    PORT = 10000
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect((HOST, PORT))
    s.send(value)

command_line = "ping6 -c 1 -I 2006::20 2006::10"
```

```
args = shlex.split(command_line)
try:
    subprocess.check_call(args,stdout=subprocess.PIPE,stderr=subprocess.PIPE)
    print "\nLower_Path is available."
    hostname = '2006::10'
    process = subprocess.Popen(['ping6','-c','5',hostname],stdout=PIPE, stderr=PIPE)
    stdout, stderr = process.communicate()
    packetloss = float([x for x in stdout.decode('utf-8').split('\n') if x.find('packet loss')
                        != -1][0].split('%')[0].split(' ')[-1])
    if packetloss < 10.0:
        print("\nPacket_Loss_of_Lower_Path is %s percent" % packetloss)
        loss_l = packetloss
        avg_time = float([x for x in stdout.decode('utf-8').split('\n')
                          if x.startswith('round-trip')][0].split('=')[-1].split('/')[1])
    print("Average_RTT_of_Lower_Path is %s s" %avg_time)
    RTT_l = avg_time
    res=stdout
    if process.returncode > 0:
        print('server error')
    else:
        pattern = re.compile('ttl=\d*')
        pattern = re.search(pattern,stdout)
        ttl=re.split(r'=',pattern.group(0))
        hops_l=64-int(ttl[1])
        print("NumberofHops_in_Lower_Path is %s" %hops_l)

    HOST = '192.168.1.5'
    PORT = 10000
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect((HOST, PORT))
    s.sendall("\n".join([bytes(loss_l), bytes(RTT_l), bytes(hops_l)]))

except subprocess.CalledProcessError:
    print "\nLower_Path is not available."
    value =bytes(0)
```

```
    HOST = '192.168.1.5'
    PORT = 10000
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect((HOST, PORT))
    s.send(value)
print("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
time.sleep(5)
```

Appendix E

Development of Routing Program in RYU Controller (Path Failure Detection, Rerouting)

#This program is written by May Pyone Han from Chulalongkorn University.
#This program is written at the RYU controller to reroute the best path in the SDN-based backbone network.

```
import time
import datetime
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER,
DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib import hub
import core
from core import best
import node_detect
from node_detect import node

# MAC ADDRESS for ethernet ports of OVS1
A1="7a:62:12:e0:1b:55" #ethernet_1(port_no.2)
A2="2a:37:11:e8:3e:c7" #ethernet_2(port_no.3)
A3="c2:b9:18:96:99:34" #ethernet_5(port_no.6)

# MAC ADDRESS for ethernet ports of OVS5
B1="6a:56:0d:47:0a:13" #ethernet_1(port_no.2)
B2="62:aa:51:f8:c0:d8" #ethernet_2(port_no.3)
B3="6a:56:0d:47:0a:13" #ethernet_5(port_no.6)

# MAC ADDRESS for Server
S1="7e:ef:37:56:13:88" #ethernet_0
```

```
# MAC ADDRESS for Mininet_WiFi
MN ="00:0C:29:6D:B2:12"

# MAC ADDRESS for Mininet_WiFi(1)
MN1="00:0C:29:3E:F8:3B"

class SimpleMonitor13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleMonitor13, self).__init__(*args, **kwargs)
        self.switches = {}
        self.datapaths = {}
        self.monitor_thread = hub.spawn(self._monitor)

    def add_flow(self,datapath,match,actions,hard):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        mod = parser.OFPFlowMod(datapath=datapath, command = ofproto.OFPFC_ADD,
                                match=match,    instructions=inst, hard_timeout=hard)
        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 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())

    def _monitor(self):
        x = datetime.datetime.now()
        print("This log is recorded from rerouting of RYU Controller at %s"%x)
        while True:
            node_list = node()
            print("The available node list is %s"%node_list)
            global result, sensor_delay
            result = None
            result, delay = best()
            if result == 0:
                print("Upper Path is chosen.")
            elif result == 1:
                print("Middle Path is chosen.")
```

```
        elif result == 2:
          print("Lower Path is chosen.")

print("XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXX")
  for datapath in self.datapaths.values():
          self.send_get_config_request(datapath)
        hub.sleep(5)

  def send_get_config_request(self, datapath):
    ofp = datapath.ofproto
    ofp_parser = datapath.ofproto_parser
    req = ofp_parser.OFPGetConfigRequest(datapath)
    datapath.send_msg(req)

#Define the function to add flow rules with configuration request messag
  @set_ev_cls(ofp_event.EventOFPGetConfigReply, MAIN_DISPATCHER)
  def get_config_reply_handler(self,ev):
    datapath = ev.msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    if (datapath.id == 29176192297550) and result == 0:
      match = parser.OFPMatch(in_port=4)
      actions = [parser.OFPActionSetField(eth_dst=B1),parser.OFPActionOutput(2)]
      self.add_flow(datapath,match,actions,0)

      match = parser.OFPMatch(in_port=2,eth_type=0x86dd,ipv6_dst='2003::/64')
      actions = [parser.OFPActionSetField(eth_dst=MN),parser.OFPActionOutput(4)]
      self.add_flow(datapath,match,actions,0)

      match = parser.OFPMatch(in_port=7)
      actions = [parser.OFPActionSetField(eth_dst=B1),parser.OFPActionOutput(2)]
      self.add_flow(datapath,match,actions,0)
```

```
    match = parser.OFPMatch(in_port=2,eth_type=0x86dd,ipv6_dst='2009::/64')
    actions = [parser.OFPActionSetField(eth_dst=MN1),parser.OFPActionOutput(7)]
    self.add_flow(datapath,match,actions,0)

if (datapath.id == 86675905817152)  and result == 0:
    match = parser.OFPMatch(in_port=4)
    actions = [parser.OFPActionSetField(eth_dst=A1),parser.OFPActionOutput(2)]
    self.add_flow(datapath,match,actions,0)

    match = parser.OFPMatch(in_port=2)
    actions = [parser.OFPActionSetField(eth_dst=S1),parser.OFPActionOutput(4)]
    self.add_flow(datapath,match,actions,0)

if (datapath.id == 29176192297550) and result == 1:
    match = parser.OFPMatch(in_port=4)
    actions = [parser.OFPActionSetField(eth_dst=B2),parser.OFPActionOutput(6)]
    self.add_flow(datapath,match,actions,0)

    match = parser.OFPMatch(in_port=6,eth_type=0x86dd,ipv6_dst='2003::/64')
    actions = [parser.OFPActionSetField(eth_dst=MN),parser.OFPActionOutput(4)]
    self.add_flow(datapath,match,actions,0)

    match = parser.OFPMatch(in_port=7)
    actions = [parser.OFPActionSetField(eth_dst=B2),parser.OFPActionOutput(6)]
    self.add_flow(datapath,match,actions,0)

    match = parser.OFPMatch(in_port=6,eth_type=0x86dd,ipv6_dst='2009::/64')
    actions = [parser.OFPActionSetField(eth_dst=MN1),parser.OFPActionOutput(7)]
    self.add_flow(datapath,match,actions,0)

if (datapath.id == 86675905817152)  and result == 1:
    match = parser.OFPMatch(in_port=4)
    actions = [parser.OFPActionSetField(eth_dst=A2),parser.OFPActionOutput(6)]
    self.add_flow(datapath,match,actions,0)
```

```
        match = parser.OFPMatch(in_port=6)
        actions = [parser.OFPActionSetField(eth_dst=S1),parser.OFPActionOutput(4)]
        self.add_flow(datapath,match,actions,0)

    if (datapath.id == 29176192297550) and result == 2:
        match = parser.OFPMatch(in_port=4)
        actions = [parser.OFPActionSetField(eth_dst=B3),parser.OFPActionOutput(3)]
        self.add_flow(datapath,match,actions,0)

        match = parser.OFPMatch(in_port=3,eth_type=0x86dd,ipv6_dst='2003::/64')
        actions = [parser.OFPActionSetField(eth_dst=MN),parser.OFPActionOutput(4)]
        self.add_flow(datapath,match,actions,0)

        match = parser.OFPMatch(in_port=7)
        actions = [parser.OFPActionSetField(eth_dst=B3),parser.OFPActionOutput(3)]
        self.add_flow(datapath,match,actions,0)

        match = parser.OFPMatch(in_port=3,eth_type=0x86dd,ipv6_dst='2009::/64')
        actions = [parser.OFPActionSetField(eth_dst=MN1),parser.OFPActionOutput(7)]
        self.add_flow(datapath,match,actions,0)

    if (datapath.id == 86675905817152)  and result == 2:
        match = parser.OFPMatch(in_port=4)
        actions = [parser.OFPActionSetField(eth_dst=A3),parser.OFPActionOutput(3)]
        self.add_flow(datapath,match,actions,0)

        match = parser.OFPMatch(in_port=3)
        actions = [parser.OFPActionSetField(eth_dst=S1),parser.OFPActionOutput(4)]
        self.add_flow(datapath,match,actions,0)
```

#This program is written by May Pyone Han from Chulalongkorn University.
#This program is written at the RYU controller to decide the best path as well as to detect path failure in the SDN-based backbone network.

```
import struct
```

```
from struct import *
import socket, sys
import os
import time
IP="192.168.1.5"
PORT=10000
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind((IP,PORT))

def best():
    i=0
    global loss_u, RTT_u, hops_u, loss_m, RTT_m, hops_m, loss_l, RTT_l, hops_l
    global Working_Path
    for i in [0,1,2]:
        packet, address = s.recvfrom(1024)
        value = packet.decode("UTF-8")
        if i == 0:
            upper_value = value
            if upper_value == '0':
                print("\nUpper_Path is not working.")
                Upper_Path = 0
                loss_u = ' '
                RTT_u = ' '
                hops_u = ' '
                Upper_Path = 0
            elif upper_value != '0':
                print("\nUpper_Path is working.")
                loss_u, RTT_u, hops_u = [float(i) for i in upper_value.split('\n')]
                print("Packet Loss: %s"%loss_u)
                print("Delay: %s"%RTT_u)
                print("Hops: %s"%hops_u)
                Upper_Path = 1
        elif i == 1:
            middle_value = value
            if middle_value == '0':
```

```
            print("\nMiddle_Path is not working.")
            loss_m = ' '
            RTT_m = ' '
            hops_m = ' '
            Middle_Path = 0

        elif middle_value != '0':
            print("\nMiddle_Path is working.")
            loss_m, RTT_m, hops_m = [float(i) for i in middle_value.split('\n')]
            print("Packet Loss: %s"%loss_m)
            print("Delay: %s"%RTT_m)
            print("Hops: %s"%hops_m)
            Middle_Path = 1
    elif i == 2:
        lower_value = value
        if lower_value == '0':
            print("\nLower_Path is not working.")
            loss_l = ' '
            RTT_l = ' '
            hops_l = ' '
            Lower_Path = 0
        elif lower_value != '0':
            print("\nLower_Path is working.")
            loss_l, RTT_l, hops_l = [float(i) for i in lower_value.split('\n')]
            print("Packet Loss: %s"%loss_l)
            print("Delay: %s"%RTT_l)
            print("Hops: %s"%hops_l)
            Lower_Path = 1
        i=int(i)+1

Working_Path = [0,0,0]

if (Upper_Path == 1 and Middle_Path == 1 and Lower_Path == 1):
    Working_Path = [1,1,1]
elif (Upper_Path == 0 and Middle_Path == 1 and Lower_Path == 1):
```

```
        Working_Path = [0,1,1]
    elif (Upper_Path == 1 and Middle_Path == 0 and Lower_Path == 1):
        Working_Path = [1,0,1]
    elif (Upper_Path == 1 and Middle_Path == 1 and Lower_Path == 0):
        Working_Path = [1,1,0]
    elif (Upper_Path == 0 and Middle_Path == 0 and Lower_Path == 1):
        Working_Path = [0,0,1]
    elif (Upper_Path == 1 and Middle_Path == 0 and Lower_Path == 0):
        Working_Path = [1,0,0]
    elif (Upper_Path == 0 and Middle_Path == 1 and Lower_Path == 0):
        Working_Path = [0,1,0]
    elif (Upper_Path == 0 and Middle_Path == 0 and Lower_Path == 0):
        Working_Path = [0,0,0]
#   print('\nWorking_Path:%s'%Working_Path)

    P = None
    if Working_Path == [1,1,1]:
        if RTT_u != RTT_m != RTT_l:
            P = min(RTT_u, RTT_m, RTT_l)
        elif loss_u != loss_m != loss_l:
            P = min(loss_u, loss_m, loss_l)
        elif hops_u != hops_m != hops_l:
            P = min(hops_u, hops_m, hops_l)
    elif Working_Path == [0,1,1]:
        if RTT_m != RTT_l:
            P = min(RTT_m, RTT_l)
        elif loss_m != loss_l:
            P = min(loss_m, loss_l)
        elif hops_m != hops_l:
            P = min(hops_m, hops_l)
    elif Working_Path == [1,0,1]:
        if RTT_u != RTT_l:
            P = min(RTT_u, RTT_l)
        elif loss_u != loss_l:
            P = min(loss_u, loss_l)
```

```
        elif hops_u != hops_l:
            P = min(hops_u, hops_l)
    elif Working_Path == [1,1,0]:
        if RTT_u != RTT_m:
            P = min(RTT_u, RTT_m)
        elif loss_u != loss_m:
            P = min(loss_u, loss_m)
        elif hops_u != hops_m:
            P = min(hops_u, hops_m)
    elif Working_Path == [0,0,1]:
        P = RTT_l
    elif Working_Path == [0,1,0]:
        P = RTT_m
    elif Working_Path == [1,0,0]:
        P = RTT_u #Minimum Value:P

    global v
    v = None
    if (P == RTT_u or P == loss_u or P == hops_u):
        v = 0 #Upper Path is chosen
    elif (P == RTT_m or P == loss_m or P == hops_m):
        v = 1 #Middle Path is chosen
    elif (P == RTT_l or P == loss_l or P == hops_l):
        v = 2 #Lower Path is chosen
    return v,P
```


#This program is written by May Pyone Han from Chulalongkorn University.

#This program is written at the RYU controller to detect the node failure in the SDN-based backbone network.

```
import subprocess
import os
from subprocess import Popen, PIPE
import time
import shlex
```

```
import datetime
def node():
   while True:
      command_line = "ping -c 2 -I 192.168.1.5 192.168.1.1"
      args = shlex.split(command_line)
      available_nodes=[]
      try:
         subprocess.check_call(args,stdout=subprocess.PIPE,stderr=subprocess.PIPE)
         available_nodes.append("OvS1")
      except subprocess.CalledProcessError:
         print "\nOvS1 is not available."

      command_line = "ping -c 2 -I 192.168.1.5 192.168.1.2"
      args = shlex.split(command_line)
      try:
         subprocess.check_call(args,stdout=subprocess.PIPE,stderr=subprocess.PIPE)
         available_nodes.append("OvS2")
      except subprocess.CalledProcessError:
         print "\nOvS2 is not available."

      command_line = "ping -c 2 -I 192.168.1.5 192.168.1.3"
      args = shlex.split(command_line)
      try:
         subprocess.check_call(args,stdout=subprocess.PIPE,stderr=subprocess.PIPE)
         available_nodes.append("OvS3")

      except subprocess.CalledProcessError:
         print "\nOvS3 is not available."

      command_line = "ping -c 2 -I 192.168.1.5 192.168.1.7"
      args = shlex.split(command_line)
      try:
         subprocess.check_call(args,stdout=subprocess.PIPE,stderr=subprocess.PIPE)
         available_nodes.append("OvS4")
      except subprocess.CalledProcessError:
```

```
    print "\nOvS4 is not available."

command_line = "ping -c 2 -I 192.168.1.5 192.168.1.4"
args = shlex.split(command_line)
try:
    subprocess.check_call(args,stdout=subprocess.PIPE,stderr=subprocess.PIPE)
   available_nodes.append("OvS5")
except subprocess.CalledProcessError:
    print "\nOvS5 is not available."

command_line = "ping -c 1 -I 192.168.1.5 192.168.1.8"
args = shlex.split(command_line)
try:
    subprocess.check_call(args,stdout=subprocess.PIPE,stderr=subprocess.PIPE)
    available_nodes.append("OvS6")
except subprocess.CalledProcessError:
    print "\nOvS6 is not available."

command_line = "ping -c 2 -I 192.168.1.5 192.168.1.9"
args = shlex.split(command_line)
try:
    subprocess.check_call(args,stdout=subprocess.PIPE,stderr=subprocess.PIPE)
    available_nodes.append("OvS7")
except subprocess.CalledProcessError:
    print "\nOvS7 is not available."

command_line = "ping -c 2 -I 192.168.1.5 192.168.1.6"
args = shlex.split(command_line)
try:
    subprocess.check_call(args,stdout=subprocess.PIPE,stderr=subprocess.PIPE)
    available_nodes.append("OvS8")
except subprocess.CalledProcessError:
    print "\nOvS8 is not available."
time.sleep(5)
return available_nodes
```

Appendix F

Development of Python Programs in Mininet-WiFi 1 and 2 for Sensor Networks 1 and 2

```
#This program is written by May Pyone Han from Chulalongkorn University.
#This program creates 6LoWPAN-based IoT sensor network 1 in Mininet-WiFi 1.

import os
from mininet.log import setLogLevel, info
from mininet.node import RemoteController, Controller, UserSwitch
from mn_wifi.cli import CLI
from mn_wifi.net import Mininet_wifi, MinietWithControlWNet
from mn_wifi.sixLoWPAN.node import UserSensor, OVSSensor

def topology():
    "Create a network"
    net = Mininet_wifi(iot_module='fakelb', apsensor=OVSSensor,
                       disable_tcp_checksum=False, controller=Controller)
    info("Creating nodes.\n")
    sensor1 = net.addSensor('sensor1', ipv6='2003::1/64', panid='0xbeef')
    sensor2 = net.addSensor('sensor2', ipv6='2003::2/64', panid='0xbeef')
    sensor3 = net.addSensor('sensor3', ipv6='2003::3/64', panid='0xbeef')
    sensor4 = net.addSensor('sensor4', ipv6='2003::4/64', panid='0xbeef')
    sensor5 = net.addSensor('sensor5', ipv6='2003::5/64', panid='0xbeef')
    sensor6 = net.addSensor('sensor6', ipv6='2003::6/64', panid='0xbeef')
    sensor7 = net.addSensor('sensor7', ipv6='2003::7/64', panid='0xbeef')
    sensor8 = net.addSensor('sensor8', ipv6='2003::8/64', panid='0xbeef')
    sensor9 = net.addSensor('sensor9', ipv6='2003::9/64', panid='0xbeef')
    sensor10 = net.addSensor('sensor10', ipv6='2003::10/64', panid='0xbeef')

    ap1 = net.addAPSensor('ap1', panid = '0xbeef', datapath = 'user')
    c1 = net.addController('c1')
    info("Configuring nodes.\n")
```

```
    net.configureWiFiNodes()
    info("Starting Network.\n")
    net.build()
    c1.start()
    ap1.start([c1])
    ap1.cmd('sysctl net.ipv6.conf.all.forwarding=1')
    ap1.cmd('sysctl net.ipv6.conf.all.proxy_ndp=1')
    sensor1.cmd('route add -A inet6 default gw 2003::60')
    sensor2.cmd('route add -A inet6 default gw 2003::60')
    sensor3.cmd('route add -A inet6 default gw 2003::60')
    sensor4.cmd('route add -A inet6 default gw 2003::60')
    sensor5.cmd('route add -A inet6 default gw 2003::60')
    sensor6.cmd('route add -A inet6 default gw 2003::60')
    sensor7.cmd('route add -A inet6 default gw 2003::60')
    sensor8.cmd('route add -A inet6 default gw 2003::60')
    sensor9.cmd('route add -A inet6 default gw 2003::60')
    sensor10.cmd('route add -A inet6 default gw 2003::60')

    ap1.cmd('ip -6 addr add 2003::60/64 dev ap1-pan0')
    info("Running CLI\n")
    CLI(net)
    info("Stoppingnetwork\n")
    net.stop()

if __name__=='__main__':
    setLOgLevel('info')
    topology()

#This program is written by May Pyone Han from Chulalongkorn University.
#This program sends the UDP sensor message from one sensor to the server.
#This program is run by each sensor to send the individual UDP messages to the server.

import sys
import socket
import time
```

```
import os
def client():
   while True():
      output = “36 *C”
      print(“Temperature:”+output)
      host = "2003::60"
      port =12346
      s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
      s.connect((host,port))
      s.send(str.encode(output + “1”)) # 2,3,4,…,10 for sensor 2, sensor 3, sensor 4,…,sensor
                                                                  10
      s.close()
      time.sleep(3)
if __name__ =='__main__':
   client()

#This program is written by May Pyone Han from Chulalongkorn University.
#This program measures the end-to-end delay from each sensor to the server.
#This program is run by each sensor to measure the individual end-to-end delay.

import sys
import socket
import time
import os

def client():
   while True():
      output =os.popen("ping6 -c 4 -I 2003::1 2001::20 | tail -1| awk '{print $4}' | cut -d '/' -f
            2").readline()
      print('Delay:'+output)
      host = "2003::60"
      port =12346
      s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
      s.connect((host,port))
      s.send(str.encode(output +"1")) #2, or 3 is used according to sensor number
```

```
        s.close()
        time.sleep(23)
if __name__ =='__main__':
    client()
```

#This program is written by May Pyone Han from Chulalongkorn University.
#This program is run by the ap node to send UDP sensor messages of all sensors to the server.

```
import socket
import time
import sys

while True:
    PORT = 12346
    IP = ": :"
    s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
    s.bind(IP, PORT)

    packet,address = s.recvfrom(1024)
    m = packet.decode("UTF-8")
    if (len(m)==7):
        SensorMsg="Temperature Sensor "+m[len(m)-2]+m[len(m)-1]+": "+m[0:len(m)-2]
    if (len(m)==6):
        SensorMsg="Temperature sensor "+m[len(m)-1]+": "+m[0:len(m)-2]
    print(SensorMsg)

    PORT = 12347
    HOST = "2007::20"
    s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
    s.connect((HOST,PORT))
    s.sendall(str.encode(m + "1"))
```

```
#This program is written by May Pyone Han from Chulalongkorn University.
#This program is run by the ap node to send the end-to-end delay value to the OVS 1.

import socket
import time
import sys
while True:
    PORT = 123456
    IP = ": :"
    s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
    s.bind(IP, PORT)
    packet,address = s.recvfrom(1024)
    m = packet.decode("UTF-8")
    if (len(m)==8):
        SensorMsg="Sensor "+m[len(m)-2]+m[len(m)-1]+": "+m[0:len(m)-2]
    if (len(m)==7):
        SensorMsg="Sensor "+m[len(m)-1]+": "+m[0:len(m)-1]
    print(SensorMsg)

    PORT = 12345
    HOST = "2001::20"
    s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
    s.connect((HOST,PORT))
    s.sendall(str.encode(m+"1"))

#This program is written by May Pyone Han from Chulalongkorn University.
#This program creates 6LoWPAN-based IoT sensor network 2 in Mininet-WiFi 2.

import os
from mininet.log import setLogLevel, info
from mininet.node import RemoteController, Controller, UserSwitch
from mn_wifi.cli import CLI
from mn_wifi.net import Mininet_wifi, MinietWithControlWNet
from mn_wifi.sixLoWPAN.node import UserSensor, OVSSensor
```

```
def topology():
    "Create a network"

    net = Mininet_wifi(iot_module='fakelb', apsensor=OVSSensor,
                       disable_tcp_checksum=False, controller=Controller)
    info("Creating nodes.\n")
    sensor1 = net.addSensor('sensor1', ipv6='2009::1/64', panid='0xbeef')
    sensor2 = net.addSensor('sensor2', ipv6='2009::2/64', panid='0xbeef')
    sensor3 = net.addSensor('sensor3', ipv6='2009::3/64', panid='0xbeef')
    sensor4 = net.addSensor('sensor4', ipv6='2009::4/64', panid='0xbeef')
    sensor5 = net.addSensor('sensor5', ipv6='2009::5/64', panid='0xbeef')
    sensor6 = net.addSensor('sensor6', ipv6='2009::6/64', panid='0xbeef')
    sensor7 = net.addSensor('sensor7', ipv6='2009::7/64', panid='0xbeef')
    sensor8 = net.addSensor('sensor8', ipv6='2009::8/64', panid='0xbeef')
    sensor9 = net.addSensor('sensor9', ipv6='2009::9/64', panid='0xbeef')
    sensor10 = net.addSensor('sensor10', ipv6='2009::10/64', panid='0xbeef')

    ap1 = net.addAPSensor('ap1', panid = '0xbeef', datapath = 'user')
    c1 = net.addController('c1')
    info("Configuring nodes.\n")
    net.configureWiFiNodes()
    info("Starting Network.\n")
    net.build()
    c1.start()
    ap1.start([c1])
    ap1.cmd('sysctl net.ipv6.conf.all.forwarding=1')
    ap1.cmd('sysctl net.ipv6.conf.all.proxy_ndp=1')
    sensor1.cmd('route add -A inet6 default gw 2009::60')
    sensor2.cmd('route add -A inet6 default gw 2009::60')
    sensor3.cmd('route add -A inet6 default gw 2009::60')
    sensor4.cmd('route add -A inet6 default gw 2009::60')
    sensor5.cmd('route add -A inet6 default gw 2009::60')
    sensor6.cmd('route add -A inet6 default gw 2009::60')
    sensor7.cmd('route add -A inet6 default gw 2009::60')
    sensor8.cmd('route add -A inet6 default gw 2009::60')
```

```
    sensor9.cmd('route add -A inet6 default gw 2009::60')
    sensor10.cmd('route add -A inet6 default gw 2009::60')
    ap1.cmd('ip -6 addr add 2003::60/64 dev ap1-pan0')
    info("Running CLI\n")
    CLI(net)
    info("Stoppingnetwork\n")
    net.stop()
if __name__=='__main__':
    setLOgLevel('info')
    topology()

#This program is written by May Pyone Han from Chulalongkorn University.
#This program sends the UDP sensor message from one sensor to the server.
#This program is run by each sensor to send the individual UDP messages to the server.

import sys
import socket
import time
import os
def client():
    while True():
        output = “36 *C”
        print(“Temperature:”+output)
        host = "2003::60"
        port =12346
        s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
        s.connect((host,port))
        s.send(str.encode(output + “1”)) # 2,3,4,…,10 for sensor 2, sensor 3, sensor 4,…,sensor
10
        s.close()
        time.sleep(3)
if __name__ =='__main__':
    client()

#This program is written by May Pyone Han from Chulalongkorn University.
```

```
#This program measures the end-to-end delay from each sensor to the server.
#This program is run by each sensor to measure the individual end-to-end delay.

import sys
import socket
import time
import os

def client():
   while True():
      output =os.popen("ping6 -c 4 -I 2003::1 2001::20 | tail -1| awk '{print $4}' | cut -d '/' -f
            2").readline()
      print('Delay:'+output)
      host = "2003::60"
      port =12346
      s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
      s.connect((host,port))
      s.send(str.encode(output +"1")) #2, or 3 is used according to sensor number
      s.close()
      time.sleep(23)
if __name__ =='__main__':
   client()
```



```
#This program is written by May Pyone Han from Chulalongkorn University.
#This program is run by the ap node to send UDP sensor messages of all sensors to the server.

import socket
import time
import sys

while True:
   PORT = 12346
   IP = ": :"
   s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
   s.bind(IP, PORT)
```

```
    packet,address = s.recvfrom(1024)
    m = packet.decode("UTF-8")
    if (len(m)==7):
        SensorMsg="Temperature Sensor "+m[len(m)-2]+m[len(m)-1]+": "+m[0:len(m)-2]
    if (len(m)==6):
        SensorMsg="Temperature sensor "+m[len(m)-1]+": "+m[0:len(m)-2]
    print(SensorMsg)

    PORT = 12347
    HOST = "2007::20"
    s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
    s.connect((HOST,PORT))
    s.sendall(str.encode(m + "1"))

#This program is written by May Pyone Han from Chulalongkorn University.
#This program is run by the ap node to send the end-to-end delay value to the OVS 1.

import socket
import time
import sys
while True:
    PORT = 123456
    IP = ": :"
    s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
    s.bind(IP, PORT)
    packet,address = s.recvfrom(1024)
    m = packet.decode("UTF-8")
    if (len(m)==8):
        SensorMsg="Sensor "+m[len(m)-2]+m[len(m)-1]+": "+m[0:len(m)-2]
    if (len(m)==7):
        SensorMsg="Sensor "+m[len(m)-1]+": "+m[0:len(m)-1]
    print(SensorMsg)

    PORT = 12345
```

```
HOST = "2001::20"
s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
s.connect((HOST,PORT))
s.sendall(str.encode(m+"1"))
```

VITA

NAME	May Pyone Han
DATE OF BIRTH	25 July 1996
PLACE OF BIRTH	Myanmar
INSTITUTIONS ATTENDED	Chulalongkorn University (Thailand) University of Technology (Yatanarpon Cyber City, Myanmar)