2018

MPLS (Multi-Protocol Label Switching) assisted routing procedure in Software Defined Networking (SDN)

Humphrey O. Otieno
Faculty of Information Technology (FIT)
Strathmore University

MPLS (Multi-Protocol Label Switching) Assisted Routing Procedure in Software Defined Networking (SDN)

Humphrey Owuor Otieno

A thesis submitted in partial fulfilment of the requirements of the Degree of Master of Science in Information Technology at Strathmore University

Faculty of Information Technology
Strathmore University
Nairobi, Kenya.

April 2018

Declaration

I declare that this thesis on **MPLS (Multi-Protocol Label Switching) Assisted Routing Procedure in Software Defined Networking (SDN)** is our own work. All the sources that have been used and indicated in this project are acknowledged appropriately and referenced. Furthermore, we can attest that this work has not been submitted to any other university for any other degree or examination.

Student Name: Humphrey Owuor Otieno Admission Number: 071949

…… [Signature]

…… [Date]

Approval

This research thesis belonging to Humphrey Owuor Otieno was reviewed and approved (*for Examination*) by the following:

Dr. Vitalis Gavole Ozianyi,
Senior Lecturer, Faculty of Information Technology,
Strathmore University.

…… [Signature]

..... [Date]

Dr. Joseph Orero,
Dean, Faculty of Information Technology,
Strathmore University.

Prof. Ruth Kiraka
Dean, School of Graduate Studies
Strathmore University.

Acknowledgment

I pass my gratitude to God for His grace and mercies that I was of good health and sound mind to reach this far. I pass my gratitude to Dr. Vitalis Ozianyi for the conducive environment, informative consultations, guidance and pointers that were given throughout this journey. I am indebted to my colleagues at the attachment office, Mr. Edmond Menya, Mr. Dickson Owuor for the criticism and support that always put a new perspective to this work whenever I was stuck

Finally, I pass my sincere appreciation to my family for their support, encouragement and prayers.

Dedication

I dedicate this work to God for the sound mind and good health to allow me to pursue this journey and my parents Mrs. Jackline Akinyi Otieno and Mr. Edward Otieno Sure. To these people, I am grateful, as I would not have completed my dissertation without their input.

Abstract

Multi-protocol label switching has been incorporated into provider networks to provide quality of service. Owing to the design of the protocol, its ability to push and pop labels in packets, independent of their underlying protocol makes it popular in interconnecting multiple networks in to one transport pipeline. At the same time, multi-protocol label switching has proven to be a very fast procedure for forwarding devices because the central processing unit cycles required in making a forwarding decision is far less compared to traditional forwarding decision-making metrics like analyzing the internet protocol header. However, current multi-protocol label switching implementation is a complex configuration procedure and does not provide a central bird's eye view of the network topology to network engineers. Logging in to every label switching router and loading multi-protocol label switching configurations to allow it to connect to neighboring label switching routers in the label switching path is required.

Allowing network engineers to have a central view and control of the network topology while still providing multi-protocol label switching services in a simplistic approach will make them achieve adaptive routing and traffic engineering seamlessly. This will improve quality of service and quality of experience in transport networks.

Software defined networking is the approach this research takes towards providing central control because of the flexibility, programmability, and adaptability of the technology. This work proposed the design of a routing procedure that will implement multi-protocol label switching on a software defined network via OpenFlow.

Experimental synthesis and prototyping approach was used to achieve the research objectives. A simulated environment called Mininet provided the implementation test bed. Internet control message packets were the test data to show how multi-protocol label switching labels are added and stripped. An illustration of the packet capture information from the experiment was presented and analyzed.

Keywords: Multi-Protocol Label Switching (MPLS), Software Defined Networking (SDN), OpenFlow, Internet Control Message Protocol (ICMP), Quality of Service (QoS).

Table of Contents

List of figures

List of tables

List of abbreviations/acronyms

API -	Application Programming Interfaces
ATM -	Asynchronous Transfer Mode
BE -	Best Effort
CSP -	Constrained Shortest Path
DCLC -	Delay Constrained Least Cost
DFS -	Depth First Search
DiffServ -	Differentiated Services
FEC -	Forward Equivalence Class
FIB -	Forwarding Information Base
ForCES -	Forwarding and Control Element Separation
GB -	Gigabyte
GHz -	Gigahertz
ICMP -	Internet Control Message
IntServ -	Integrated Services
IP -	Internet Protocol
L2 -	Layer 2
L3 -	Layer 3
LARAC -	Lagrange Relaxation based Aggregate Cost
LER -	Label Edge Router
LLDP -	Link Layer Discovery Protocol
LSR -	Label Switching Routers.
MCP -	Multi Constrained Path
MPLS -	Multiprotocol Label Switching

NETCONF - Network Configuration Protocol

OVS - Open vSwitch

QoE - Quality of Experience

QoS - Quality of Service

RAM - Random Access Memory

RSVP - Resource Reservation Protocol

RX - Receiver

SDN - Software Defined Networks/ Software Defined Networking

SONET - Synchronous Optical Networks

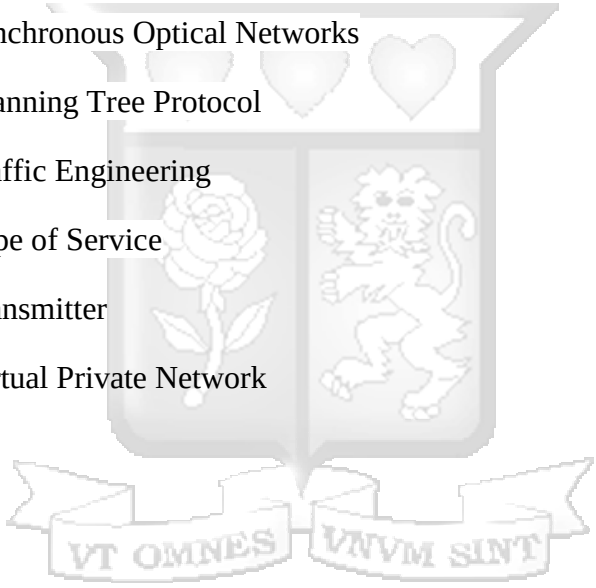
STP - Spanning Tree Protocol

TE - Traffic Engineering

ToS - Type of Service

TX - Transmitter

VPN - Virtual Private Network



Definition of Terms

Forwarding Device - A virtual or physical networking device that can forward packets or frames at either Layer 3 or Layer 2 respectively (Nadeau & Gray, 2013). This research uses routers or switches interchangeably to mean a forwarding device.

Layer 2 (L2) – Layer 2 of the Open Systems Interconnection (OSI) model- Data Link Layer (Curran et al., 2016)

Layer 3 (L3) - Layer 3 of the Open Systems Interconnection (OSI) model- Network Layer (Curran et al., 2016).

NOX - NOX is an open source development platform for C++-based software-defined networking (**SDN**) control applications. It is used to define forwarding rules at the forwarding devices at the data plane ("Installing POX — POX Manual Current documentation", 2018).

OpenFlow - OpenFlow is an open communications protocol that acts on Layer 2 of the OSI model and provides access to the forwarding plane of a router or switch over the network. OpenFlow simply allows the path of data packets within the network of switches to be determined by software that is running on at least two routers (Nadeau & Gray, 2013).

Open vSwitch (OVS) - Open vSwitch is a production quality, multilayer virtual switch licensed under the open source **Apache 2.0** license ("Open vSwitch Documentation — Open vSwitch 2.9.90 documentation", 2018). Open vSwitch is an open-source virtual switch software designed for virtual servers. The role of this software is to forward traffic between different virtual machines (VM) within the same host and even traffic between a VM and a physical network.

POX - POX is an open source development platform for Python based software-defined networking (SDN) control applications. It is used to define forwarding rules at the forwarding devices at the data plane ("Installing POX — POX Manual Current documentation", 2018).

Chapter 1 : INTRODUCTION

1.1 Background of the study

The standard internet design has been based on Best Effort (BE) delivery of packets (Curran, Fenton & Freedman 2016). With the emergence of new applications like video streaming, video on demand and voice over the internet protocol (VoIP), there was the need to make sure these applications perform better if not at their best while their packets traverse the internet.

The growth of multimedia applications, voice, video and data, in the world has led to the increase of traffic generated on the internet. As these applications grow, the user experience and exposure to these applications need to be addressed. One way to ensure that these applications provide the acceptable user experience today is the presence of high-speed links for backhaul connections or even high speed links at the access layer of the network (Trestian et al., 2013). However, the assumption in such a case is that the increase or availability of bandwidth and a routing protocol that calculates the shortest path ensures that relative Quality of Service (QoS) is met. This is not always the case. At times, for most applications, bandwidth is not the problem.

Owing to this, the term QoS on the internet was derived. According to Odom and Cavanaugh (2004), QoS is the capability of a network to provide better service to selected network traffic over various technologies, including Frame Relay, Asynchronous Transfer Mode (ATM), Ethernet and 802.1 networks, Synchronous Optical Networks (SONET), and Internet Protocol (IP)-routed networks that may use any or all of these underlying technologies.

According to Xiao (2008), quite a number of technologies have been developed to ensure networks achieve QoS. They range from Integrated Services (IntServ), Differentiated Services (DiffServ), Spanning Tree Protocol (STP) and Multiprotocol Label Switching (MPLS). The QoS metrics that most of these technologies keep in consideration are jitter, error rate, throughput, redundancy and packet loss rate.

IntServ was designed to create logical end-to-end circuits and create flows based on classes of services that are marked on the Type of Service (ToS) field in an Internet Protocol (IP) packet. The creation of this end-to-end logical circuit was based on negotiations of resources on the internet among nodes that is done by the Resource Reservation Protocol (RSVP) (Mammeri, 2005). The complexity that comes with IntServ and little guarantee that all flows can be

accommodated as much as very few classes of service could be captured by the ToS field made this technology less popular.

Mammeri (2005) further opines that DiffServ seeks to improve where IntServ fails and tries to simplify this process of QoS provision on packetized networks like IP. DiffServ however does not create logical end-to-end circuits and resources are not negotiated in advance. DiffServ works on a resource provision mechanism that is based on classification of packets, metering of the packets, marking and shaping (policing, mapping and dropping) to ensure QoS is achieved in IP networks.

MPLS on the other hand, as discussed by Ahn and Chun (2001) leverages on the fast delivery of packets by ensuring packets spend the shortest time as possible at the MPLS enabled nodes which are called Label Switching Routers (LSRs). The labels added on packets at the nodes are used to define the next hop of the packet. Hence, proper design in the network based on the advantages of MPLS can guarantee some level of QoS. On the other hand, STP has been designed to deal with broadcast storms that emerge from prevalence of these services in the network by ignoring some links in the network (Elder & Harrison, 2015)

A research conducted by García-Dorado et al (2012) on multimedia bandwidth demand states that video traffic is on a steady rise. By 2020 it is estimated that 2/3 of traffic on the internet will be video traffic. Half of that will account for real time video applications and other real time heavy applications.

Therefore, getting a highly efficient network that meets the needs of all the users is desired. In addition, the simplicity and manageability of this network will highly transform data transportation in networking. This research notes that the techniques mentioned above to provide QoS on packetized networks shaped the proposed solution and highly depended on them.

Therefore, having a network that can adapt to changes in its environment is a way to increase the efficiency in terms of service delivery and the maximum utilization of the resources. This network should also be easy to configure, manage and monitor. This gives insight on whether the network is able to meet QoS demands and Quality of Experience (QoE) demands.

To achieve this, the paper borrows a lot from the operation of traditional routing in networks and tradition QoS provisioning mechanisms. A lot of emphasis is laid on MPLS in transport networks. However, where this paper departs from the status quo is the implementation of our proposed

solution based on SDN. SDN is the suitable approach to make sure this research is able to achieve a MPLS assisted routing procedure that is simple to setup and manage.

SDN is a technology that separates the control plane, the data plane and the management plane in a networking device (Akyildiz et al., 2014). The control plane is usually the interface that does the logical decision-making of how and when packets will be sent and through which interface on the device. The data plane on the other hand does explicit forwarding of packets through its outlets based on the instructions that it receives from the control plane. It is safe to say that the control plane controls the data plane of any networking device. The management plane on the other hand is responsible for the monitoring of the network and providing of network statistics of the whole network to the control plane (Hu, 2014).

The mere fact that current networking devices do not separate these interfaces means that, for every *n* device, we have a total of *n* control planes that need to be configured whenever forwarding instructions needs to be discharged to the data plane. That is cumbersome and can result to unforeseen errors in configuration (Hu, 2014).

SDN is leveraging on the separation of control plane from the data plane. This then brings the concept of centralized network control where data planes will be left on the different nodes in a network and a control plane can be hosted on a server. This typically means that 1 control plane can control *n* forwarding devices' data planes.

It is important to note that, for this separation of control plane and data plane to work in a centralized environment, SDN works with a number of protocols to ensure that control instructions are sent to the data plane and it is implemented. Proposed protocols include, Forwarding and Control Element Separation (ForCES), Network Configuration Protocol (NETCONF) and OpenFlow Protocol that define how different flows will traverse the network (Akyildiz et al., 2014).

Finally, to ensure we have the flexibility, programmability, adaptability and robustness that comes with SDN, Akyildiz et al (2014) insist that the openness of the Northbound and Southbound Application Programming Interfaces (API) will play a major role. This comes with the separation of the control plane and data plane in SDN via OpenFlow. The Northbound API allows applications to communicate and share information as they traverse the network, on the data plane, with the control plane. On the other hand, Southbound APIs allows the control plane to communicate directly with the data plane and share information.

1.2 Problem Statement

MPLS has been used to provide QoS in networking based on its fast switching technique that goes against checking of long header prefixes to only checking small labels attached to packets from multiple protocols. However, as Black (2002) explains, MPLS configuration is a very complex task between the service provider and the client. The complexity of MPLS deployment comes from the non-centralized view of the network nodes by the service providers. Especially if it is being done in a Wide Area Network (WAN) implementation. This therefore means configuration of MPLS switches is prone to probable errors as the network grows.

In addition, MPLS enabled forwarding devices are not cheap to acquire. Every MPLS service provider has a function model that vary as opined by Davie and Rekhter (2000) when comparing and merging the operation of MPLS Traffic Engineering (TE) on Huawei and Cisco switches. Fang et al., (2005) also note that it is important to understand that MPLS as a routing protocol is dependent on hardware devices. Some networking nodes do not support MPLS and this is always challenge.

Unlike the traditional way of doing MPLS on vendor specific devices, SDN opens up the arena because it is open source. All one needs to understand is how to invoke OpenFlow messages that will make forwarding devices act like LSR. In addition, SDN provides the centralized view of the network because it connects multiple forwarding devices to one or more control planes but not on a one to one relationship between forwarding devices to control planes (Nadeau & Gray, 2013).

1.3 General Objective

The general objective that this research seeks to achieve is to design and develop a routing procedure based on MPLS operation is an SDN environment.

1.3.1 Specific Objectives

i. To review the operation of MPLS and its variants in current internetworking strategies,
ii. To design a routing procedure for network applications in SDN,
iii. To implement MPLS technique in SDN,
iv. To test the MPLS based routing technique for network applications in SDN.

1.4 Research Questions

i. How does MPLS and its variants operate in current internetworking strategies?
ii. How can a routing procedure for network applications in SDN be designed?

iii. How can MPLS technique be implemented in SDN?

iv. How will the MPLS based routing technique for network applications in SDN be tested?

1.5 Justification

This research is beneficial in reshaping how network administrators and engineers will approach the design of transport data networks with key emphasis put on MPLS implementation. Network administrators and engineers will be able to have a global end-to-end centralized view of the network architecture. In addition, they will not be worried whether their device of choice is MPLS enabled or not. The research will demonstrate how simple it will be for network engineers to setup MPLS forwarding mechanism in their network topology. This will be influenced by the power of SDN and OpenFlow in providing a centralized and easy configuration mechanism of networking devices.

From this, they can be able to come up with a routing procedure. This procedure will assign paths to flows based on the fast packet delivery mechanism implemented by MPLS routing. This is a surety that network will be very flexible and robust, adapting and changing to suit the design of the network engineer.

1.6 Scope

This research is limited to demonstrating how a routing procedure based on MPLS can be achieved in SDN. The study focused on ICMP packets, both request and replies sent from different hosts in the network. This study used an SDN based simulation tool called Mininet working with OpenFlow protocol because of the acceptability and wide interoperability of the two technologies as far as programmable networks are concerned.

Chapter 2 : LITERARTURE REVIEW

2.1 Overview

The chapter exhaustively looks at the previous work in the field of MPLS implementation and routing in SDN. The chapter starts by discussing MPLS operation and its variants in current internetworking strategies. It looks at the environment to support network customization in SDN. The proceeds to scrutinize adaptive routing procedures for network applications in SDN. This is followed by a discourse on what are the techniques that multimedia applications use to communicate to the control plane of a SDN though the Northbound interface. Finally, to conclude, the chapter looks at the possible strategies of implementing a label like switching technique in SDN emulating what happens in MPLS but making it compatible with OpenFlow.

2.2 MPLS operation and its variants in current internetworking strategies

MPLS was designed to solve the problems that were being experienced in IP routing. Black (2002) discusses that routing protocols in traditional IP networks were designed to be invoked on all the devices that were to be used for forwarding functionality. Besides that, as Davie and Rekhter (2000) notes, regardless of the routing protocol that is used, these routers could only forward traffic based on the destination addresses only. The traditional IP routing lookups, as Ould-Brahim (2007) extends this discourse, were performed on each router. This means that each router will make an independent decision when forwarding the packets.

MPLS comes into play to reduce the number of routing lookups and eliminate the need to run a particular routing protocol an all devices involved in the forwarding of MPLS data. MPLS is protocol independent (Le Faucheur, 1998). MPLS uses labels, that are numbers, to identify the packets and enable forwarding decisions to be made based on those numbers. These labels are 32 bit long and are placed between the Layer 2 (L2) and Layer 3 (L3) headers. The labels as explained by Le Faucheur (1998) usually correspond to layer 3 destination addresses or other QoS parameters.

A network that runs MPLS usually comprises of two categories of devices. Label Edge Routers (LERs) are devices placed at the entry points and the exit points of the MPLS network. These are devices that admit user traffic to an MPLS network and then forwards the same user data to non-MPLS networks. In other nomenclatures, these devices are referred to as the Ingress and the Egress

routers respectively. Black (2002) annotates that the edge routers are usually responsible for the route look up activity and assigning of labels to packets. The other category devices in a network that runs MPLS is the LSR that are responsible for swapping different labels and forwarding the packets based on the received labels. The path that the data will use at the entry point of an MPLS network to the exit point is called the Label Switch Path (LSP) (Le Faucheur, 1998).

The MPLS architecture comprises of two main components, the control plane and the data plane (Le Faucheur, 1998). The control plane is responsible for the exchange of L3 routing information and the labels and creating the end to end path for the traffic to follow. The data plane on the other hand is responsible of forwarding data. It simply acts as a forwarding engine.

Based on the functionality of MPLS, other services can be derived from its operation. Traffic Engineering (TE) in MPLS as captured from the discussion by Lee et al (2000) can be provided by MPLS being able to create tunnels to a particular destination. These tunnels can be created based on analysis of the network traffic. These services of TE can further be enhanced to provide load balancing on an MPLS network as opined by (Lee et al, 2000).

MPLS- Virtual Private Network (VPN) is another application that has rode on the capabilities of label switching. This MPLS-VPN as explained by Lee et al. (2000) adds an additional label to determine the VPN and the destination network. VPN routing information and the labels are then propagated to the MPLS domain with the assistance of Boarder Gateway Protocol (BGP) (Lee et al, 2000). The two labels that are used to achieve MPLS-VPN are the top label that points to the egress router and a second label, which points at the exit interface on the egress router.

2.3. Environment to support network customization in Software Defined Networking (SDN)

The routing procedures and network modification in SDN are done in a Linux based environment. Specifically, the environment is Mininet configured virtual machine that runs on Ubuntu server 14.04 LTS. This environment provides the simulation capabilities of setting up a virtual network with virtual switches virtual links and a virtual controller to communicate and control traffic in the network as envisioned by SDN. The figure below captures the architecture of an SDN network emulated by Mininet.

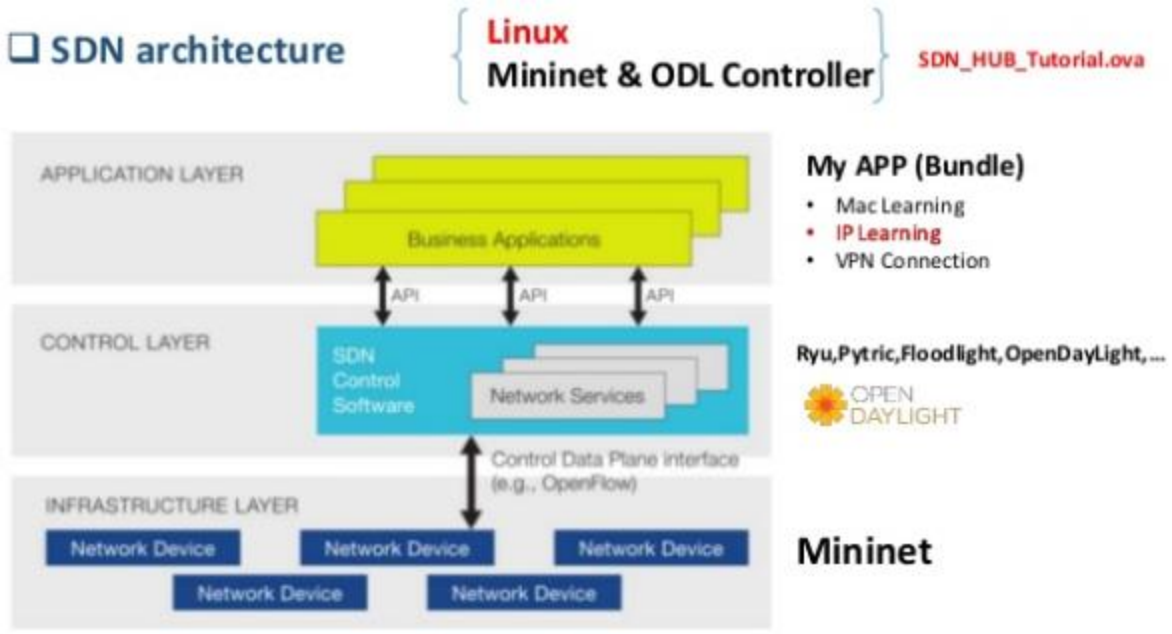


Figure 2-1: SDN Architecture (Vahid Sadri, 22:54:13 UTC)

2.3.1 Analysis of Mininet environment

The Mininet environment concisely is an emulation tool that allows you to create an SDN based network ("Mininet/OpenFlow-tutorial", 2018). It holds up to the groundbreaking advancements that SDN has achieved by breaking up the control plane and data plane in communication forwarding devices.

A number of controllers usually implements the control plane on Mininet. NOX an open source development platform that is based on C++ to control software defined networks and POX a variant of NOX, but purely runs on Python development platform for SDN networks are controllers available for Mininet SDN simulation. Floodlight controller is another SDN controller that is based on Java development platform (Gupta, Sommers, & Barford, 2013). This dissertation opted to settle on POX controller for the implementation.

Besides the controllers, Mininet also comes with preconfigured out of the box Open vSwitch (OVS) that emulates layer 3 and layer 2 forwarding devices that will be controlled by the controllers described above. The mode of control is usually by the use of a communication protocol, OpenFlow that supports SDN by sending flow table instructions to the forwarding devices.

2.3.2 POX overview and architecture

POX is an open source controller used for developing SDN applications ("Installing POX — POX Manual Current documentation", 2018). POX is a python based controller that inherited its operation from its predecessor, NOX controller. POX allows the switches in the SDN network to communicate to the controller via an OpenFlow protocol. OpenFlow enables switches like Open vSwitch, to forward traffic based on flow table modification. As opined by (Csoma et al, 2014) , these SDN switches are dump devices that cannot forward any traffic in an SDN network unless the controller invokes a forwarding procedure based on OpenFlow messages sent from the controller to the switches .

When a switch is powered on in SDN, the first procedure is to attach itself to a controller or controllers depending on the Mininet network topology design. At this moment, the flow tables of the switches are usually empty. Packets arriving at the switch, at this moment, cannot be forwarded to any output interface because there is no match at the switch's flow table. So a packet-in function is invoked in POX and a message is sent to controller. POX then inserts the flow entries to the packet that will be sent to the switches via OpenFlow to modify the flow tables.

Flow entries are inserted based on three parts, the rule match, action and counter. There are existing components in POX that have been designed to emulate different forwarding scenarios in networks. POX defines components in a folder called forwarding. From POX, a flow table modification message can be sent to make the switch act as a hub, Layer 2 learning switch or discover a network topology on its own via Link Layer Discovery Protocol (LLDP) and create flow entries ("Installing POX — POX Manual Current documentation", 2018). A listing of the components have been capture on the image below.

Besides the already existing components is POX, the controller allows network applications programmers and network engineers to define their own components to run in POX. The components are written in Python. The Mininet environment allows the POX controller to either run on the same machine the SDN emulation tool runs or it can be run on another machine and be referenced from there by Mininet by invoking a Mininet process and referring to the controllers IP address and port number.

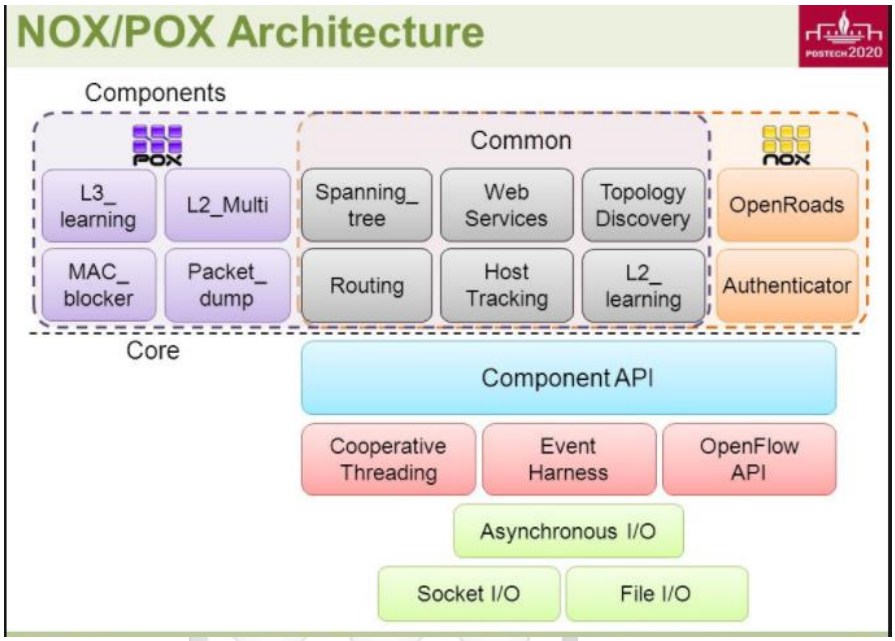


Figure 2-2: POX Architecture ("Software Defined Networking: OpenFlow Switches & Controllers - ppt download," n.d.)

2.3.3 Open vSwitch overview and architecture

Open vSwitch (OVS) is an Open Source Apace 2 licensed multilayer virtual switch that has opened the forwarding function to programming and extension control ("Open vSwitch Documentation — Open vSwitch 2.9.90 documentation", 2018). OVS uses the virtual bridges or interfaces defined by the network topology designed in Mininet to add flow rules and forward packets accordingly. The behavior of the OVS is like a physical switch but only emulated in a virtual machine with virtual interfaces and virtual links (Pfaff et al., 2015). OVS once invoked can run in two modes. The de facto mode is the standalone mode that allows OVS act as a learning switch. However, from the findings of this research, the topology of the network can make it harder for OVS to perform self-learning capabilities. Therefore, this pushes OVS to run in secure mode that relies on the controller to receive flow table modifications messages or flow rules that will determine how inbound and outbound traffic will be treated.

The ovs-switchd module and the kernel do forwarding of flows in Open vSwitch. The first packet that hits the switch goes to the user space ovs-switchd. If it is a packet miss on the OVS flow table, a packet in message is sent to the controller via OpenFlow for flow table instructions. If there are

flow table instructions in the controller, they will pushed down to the switches via OpenFlow ("Open vSwitch Documentation — Open vSwitch 2.9.90 documentation", 2018).

However, if the packet has a match entry, the actions defined should be invoked on that particular flow. Subsequent packets of the same flow do not hit ovs-switchd, but the kernel's cache which already has stored the flow entries on the flow table. Other Command Line (CLI) tools that make OVS perform its forwarding capabilities include the ovs-vsctl that manages the state of the ovsdb-server. The ovsdb-server is a management protocol interface that was designed for SDN networks by OVS. It is able to run legacy management protocols like SNMP. The ovs-dpctl is a kernel module configuration tool as ("Open vSwitch Documentation — Open vSwitch 2.9.90 documentation", 2018) explains while the ovs-ofctl is a kernel module configuration tool that specifically works with OpenFlow protocol.

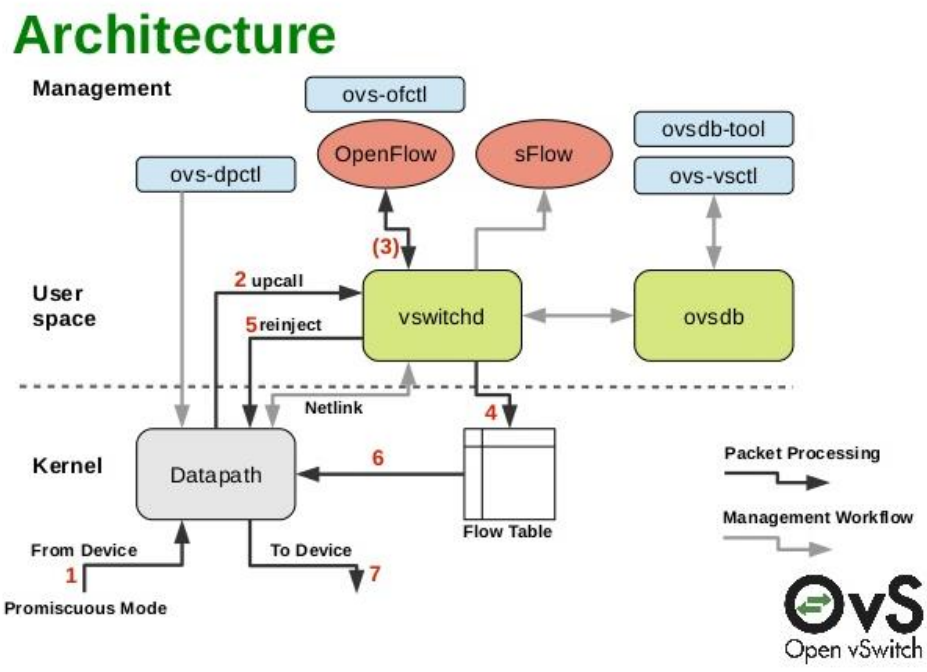


Figure 2-3: OVS Architecture ("The introduction to OVS architecture," n.d.)

OVS, depending on the version, is able to run different OpenFlow version. This research, to be able to do MPLS implementation on SDN had to update the OVS kernel module from the default version of 1.10 that comes with Mininet 2.2 to version 2.5.4 that is able to support OpenFlow version 1.3. The image below summarizes the relation between OVS version and OpenFlow.

Open vSwitch	OF1.0	OF1.1	OF1.2	OF1.3	OF1.4	OF1.5	OF1.6
1.9 and earlier	yes	—	—	—	—	—	—
1.10, 1.11	yes	—	(*)	(*)	—	—	—
2.0, 2.1	yes	(*)	(*)	(*)	—	—	—
2.2	yes	(*)	(*)	(*)	(%)	(*)	—
2.3, 2.4	yes	yes	yes	yes	(*)	(*)	—
2.5, 2.6, 2.7	yes	yes	yes	yes	(*)	(*)	(*)
2.8	yes	yes	yes	yes	yes	(*)	(*)

Figure 2-4: Relation of OVS and OpenFlow Version Numbers ("Using OpenFlow — Open vSwitch 2.9.90 documentation," n.d.)

2.3.4 OpenFlow overview and architecture

As SDN rides on its ability to separate of abstract the control plane from the data plane, the de facto standard that has allowed the sharing of messages between the controller and the forwarding devices is OpenFlow. This open standard for communication as ("Mininet/OpenFlow-tutorial", 2018) explains operates by allowing the controller to communicate with the multiple switches from a central point. OpenFlow allows SDN to achieve the abstraction of the underlying layer without worrying about the hardware. This means that application developers will be able to issue commands, via and API, to the forwarding devices in a standard format without worrying the kind of hardware that is beneath them. OpenFlow is responsible for issuing these flow entries, without changing the configurations of a forwarding device, in a flow table that the forwarding devices call the Forwarding Information Base (FIB) (Shalimov et al., 2013).

OpenFlow allows the feeding of information to the OVS using their existing flow tables. It is imperative to note that each of the OVS has separate flow tables and OpenFlow can interact with each of them by either adding, updating or deleting flow entries. OpenFlow allows symmetric message exchanges to occur between the switch and the controller. These control messages include; Hello message, which is an introductory message that is first sent when a switch and controller connect for the first time or they are trying to keep-alive. Echo message is sent to check for the availability of the switch or the controller. Experimenter message is the default message

that OpenFlow switches use to offer any other value added services to the OpenFlow message type space.

2.4 Adaptive routing procedures for network applications in SDN.

A lot of work has gone into the field of QoS provision in legacy networks as well as contemporary networks like SDN. As Akella and Xiong (2014) opine, the QoS metrics like latency, jitter, buffer fill up rate, throughput and packet drop rates normally affect data networks at the distribution layer and the core layers. That is practically true because as we speak, the access layer technology that is normally applied in most organizations and homes is Ethernet. With Ethernet, there are standards of Ethernet that support 10 Gigabit Ethernet. However, the most common application are 100Mbps Ethernet and 1Gbps Ethernet (Frazier & Johnson, 1999). Usually, these speeds are good enough to provide the quality of service constraints that most multimedia applications request to be met when handling their flows.

Traditional routing algorithms tend to consider only one metric when making forwarding decisions. To provide QoS, an adaptive routing algorithm must consider more than one metric, shortest path, as witnessed in traditional routing algorithms. This becomes very important when dealing with multimedia applications because it not always a guarantee that shortest path will always meet delay constraints a video application requires on a network.

As Zhao et al. (2016) discusses, Constrained Shortest Path (CSP) and Multi Constrained Path (MCP) routing algorithms have been developed to consider more than one metric when defining paths for different flows in a network topology. From this algorithm, Yu et al. (2016) looked at the shortcomings of MCP and CSP and designed Delay Constrained Least Cost (DCLC) with the help of Lagrange Relaxation based Aggregate Cost (LARAC). These advancements in routing algorithm technologies faced two shortcomings. The runtime was slower as the algorithm got more complicated. Besides that, Chen and Sahni (2008) add that there was no dynamic change of networking routing based on network information and statistics.

However, leveraging on SDN, the networking route is able to change dynamically by the use of triggers from the controller. Therefore, routing rules are defined for specific network flows keeping in mind that network state and topology is always changing. Works from Egilmez et al. (2012) looked at defining flows into three types, QoS Level 1, QoS Level 2 and Best Effort flows. In

addition, each flow is assigned different routing rules. In order to meet the QoS metrics that are defined by these flow types, there was rerouting of traffic among queues on the forwarding devices to make sure QoS metrics are adhered to at all times.

Yu et al. (2016) argue that as much Egilmez et al. (2012) define traffic into three flows, when it comes to multimedia traffic like video, there is a huge degradation in QoS whenever there is a lot of traffic in the network. This is because video traffic in general will require more than the three paths or two defined by previous work of (Egilmez et al., 2013). Yu et al. (2016) further poke gaps in the previously proposed algorithms that rely on shortest path calculations and reassignment of paths.

A new approach discussed by Yu et al. (2015) is presented where the base layer packets and enhancement layer packets in the case for video traffic is considered when making initial routing and rerouting decisions. Base layer packets are the most essential packets in a video stream that cannot be subjected to delay variation and packet loss. They usually kick start the video stream then they are followed by enhancement layer packets. These enhancement layer packets on the other hand should complement the base layer packets and withstand a little delay variation and packet loss to some degree. According to Yu et al. (2015), in what they called a dynamic and adaptive routing algorithm, in case of congestion, the base layer packets should always be given first priority for rerouting then followed by the enhancement layer packets in QoS enabled topologies in SDN.

How the dynamic and adaptive routing algorithm was designed to handle and maintain QoS is through leveraging on OpenQoS mechanism that separates flows into three, QoS Level 1, QoS Level 2 and Best Effort flows. Dynamic and adaptive routing algorithm then calculates the least cost path based on CSP using the maximum delay variation as the constraint that is passed to the algorithm. Using this information, then controller then assigns QoS Level 1 flows and QoS Level 2 flows to the shortest path calculated. If at all, the shortest path does not satisfy the delay variation provided by CSP, LARAC algorithm then looks for a feasible path that can still meet the maximum delay variations defined by these two QoS Levels (Yu et al., 2015).

Dynamic and adaptive routing algorithm defines base layer packets as QoS Level 1 flows and enhancement layer packets as QoS Level 2. This means that rerouting QoS Level 1 packets to a

feasible path that does not meet the bandwidth requirement to guarantee maximum delay variation specified will not solve the problem. Hence, the algorithm, before rerouting to the feasible path, checks if the feasible path has enough bandwidth to handle both BE traffic and QoS level 1 traffic. If that is possible, base layer packets are rerouted to this feasible path. If that is not the case, QoS Level 2 traffic is the one that is rerouted to this feasible path while QoS Level 1 traffic stays on the shortest path calculated by CSP. This decongests the network on these flows and guarantees QoS.

This Dynamic and adaptive routing algorithm that leverages on SDN capabilities to enhance OpenQoS in providing QoS services to multimedia applications, especially video traffic has been quoted to reduce packet loss rate for base layer packets by 77.3% and enhance 51.4% coverage (Yu et al., 2015). It does this under various network loads of the shortest path and feasible paths that looks at shortest path and maximum delay constraints as the QoS constraints when making forwarding and rerouting decisions (Yu et al., 2015).

Dynamic and adaptive routing algorithm as designed and proposed by Yu et al. (2015) considered the shortest path and delay constraints by the use of CSP and LARAC algorithm. However, for real time applications like safety critical Real Time Systems (RTS) and video streaming which falls under multimedia applications, getting to estimate and guarantee the end to end delay of this traffic goes a long way to ensure QoS is achieved.

On the other hand, works by Kumar et al. (2017) appreciates that end-to-end delay analysis and guarantees cannot be applied in old legacy networks. As much as SDN can allow network administrators get a global view of the network architecture through the management plane and perform flow modification, still SDN cannot reason out delays, what SDN reasons about when doing path allocations for multimedia traffic is bandwidth. Class I flows as described by Kumar et al. (2017), which is characteristic for RTS and Real Time Multimedia applications traffic need the SDN to think of delay guarantees to be met on top bandwidth considerations that it makes.

The end goal that Kumar et al. (2017) envisioned was to make sure network engineers do not get accustomed to only finding the shortest path to from the Transmitter (TX) to Receiver (RX). They do acknowledge that path layout is significant problem in SDN when doing adaptive and dynamic routing in SDN. However, when it is well thought out, Class I flows can arrive just in time through

a custom end to end delay guarantees procedure and bandwidth utilization measures taken by Kumar et al. (2017) in their discussion.

Kumar et al. (2017) expects that before path allocation procedure gets underway, the Class I real time flows have to specify their delay and bandwidth requirements and let the controller via OpenFlow assign them the correct path at the data plane that meets their end to end delay requirements. The management plane aides in giving a global view of this SDN topology.

Their proposed algorithm aims at solving one major problem, laying out the path for each flow such that it satisfies the delays and bandwidth constraints set by the flow. This is done by using the MCP formula and passing bandwidth and to estimated delay as parameters. Output from the MCP formulation is then provided to their second algorithm, Algorithm 2 that either relaxes the bandwidth or delay constraints, to each queue that a flow has been assigned to, in order, so that to make sure the path allocated guarantees end-to-end delay of real time traffic.

The novel approach that this work of Kumar et al. (2017) makes is that bandwidth utilization can be easily calculated in SDN. However, for delay estimation, there is a need to consider delay causes by the link and the forwarding devices on a path a flow follows. Therefore, processing delays, queueing delay, transmission delays and propagation delays needed to be estimated correctly and passed as an overall summation of estimated delays to the MCP formulation for a path allocation to be made in comparison with the Class 1 flow demands.

Something to note in their work is estimation of propagation and transmission delays are as a result of a function of the physical properties of the link. Where, transmission delay is calculated as packet length/ bandwidth allocated while propagation delay depends on the physical length of the medium and the propagation speed which falls in between 0.59c and 0.77c on any physical medium. C in this case is the speed of light. Processing delay from their experimental data as recorded by Kumar et al. (2017) was between 3.2µs to 4.1µs.Queing delay in this case was set to negligible because the work of Kumar et al. (2017) was overprovisioning the bandwidth on all queues each flow took; hence, they did not face such delays.

2.4.1 Techniques multimedia applications use to communicate directly with the network control plane in SDN

Communication or sending feedback to SDN through the Northbound API supports the idea of building an application aware network. An application aware network in the end goes beyond provision of QoS as it is, but also tracks the user experience to guarantee some level of QoE. The challenge, as explained by Jarschel et al. (2013) is to understand what kind of information should be shared to the network by the application.

The Northbound API supported in SDN will play a vital role to ensure information is exchanged between the application and the network control plane. Previous works from Curtis et al. (2011) have shown that using this interface, a datacenter flow scheduling was optimized by notifying it of elephant flows and mice flows on the Hosts socket buffers beforehand.

Leveraging on this capability, Jarschel et al. (2013) found out that video application could be a good example to show how multimedia applications can communicate directly to the control plane. Jarschel et al. (2013) used video streaming applications, for example YouTube in their work. The kind of application data that they needed was the buffer filling level and the occurrence of playback stalling. A YouTube quality-monitoring tool called YoMo could capture all this. YoMo when installed on browsers like Mozilla Firefox is able to identify Transmission Control Protocol (TCP) flows used in transmission as well as track buffered and current playtime in YouTube player.

This information provides network administrators with information whether their network architecture is able to provided QoE to video application as expected by their customers. The information from YoMo is fed into the controller as input. When buffer level goes beyond a specified threshold as contemplated by Jarschel et al. (2013), the controller is notified of degrading QoE and it goes through an optimum path selection and redirecting that flow to a less loaded link to retain QoE levels for YouTube video streaming. This application-state-aware approach works well as shown by Jarschel et al. (2013), but there is extreme overhead that is introduced in the network control channel. It can get worse if the application states change constantly.

2.5 Implementation of MPLS routing technique in SDN

The goal of implementing MPLS or Label switching in SDN as Bellessa (2015) explains in his work is to make sure there is very minimal complexity at the networks core but a very intelligent

egress and ingress switching nodes. The normal mode of operation of the core network switches in SDN is based on rule matching. According to Bellessa (2015), this de facto operation is very memory intensive and makes it costly to adapt. In addition, the normal SDN operation couples up the host requirements that are fed to the control layer with the network core behavior propagated down to the forwarding devices. Such minor inflexibility when one wants to achieve less memory intensive and simple core layer network operations motivated the work of Bellessa (2015) to implement MPLS kind Label Switching on OpenFlow.

MPLS is a kind of label swapping standard that uses labels that are attached to the packet header as they traverse the network. The work of these labels is to identify a corresponding Forward Equivalence Class (FEC) that encapsulates the packets and forwards it in the same direction regardless of their destination addresses (Ahn & Chun, 2001). This approach assures network engineers of the flexibility of the network because they can determine which paths are being taken by traffic in the network.

Naïve implementation of MPLS on OpenFlow in SDN is not viable. This will mean adding a functionality to OpenFlow to perform label switching. This will require a complete hardware overhaul of network devices that run the new OpenFlow version. Fabric networks is a solution that was proposed by Bellessa (2015) to separate the addressing and control of the traffic from the core network to the edge switches.

Implementing Fabric networks becomes a challenge because of the compatibility of the commodity hardware required to make MPLS run on OpenFlow. Shadow MACs is a project done by IBM to provide a solution to this by using up the larger memory spaces that Layer 2 (L2) address space is assigned to commodity switches to attach (Agarwal et al., 2014). Besides that, the commodity switches can do quick L2 address matching compared to OpenFlow header matching which makes it fast and less memory intensive to the forwarding devices. However, the only limitation to this is only one label information can be attached to this L2 header space. However, MPLS does label stacking for it to allow switching within the core network through label pushing, swapping and popping that directs packets over a specific path.

The Shadow MACs approach envisioned by IBM plays a key role in implementation of MPLS kind of Label switching in SDN. Leveraging on the idea that the label on the packet's L2 address

space will determine forwarding decisions, label switching was achieved with a few tweaks to the implementation. The tweak that Bellessa (2015) proposed was implementing MPLS on OpenFlow but without the use of Label Stacking but they introduced a technique called Label Flattening. This technique appreciates the fact that Label Stacking to determine forwarding decisions at each node is not supported by SDN at the time of his work. What he propose is a mechanism of creating a path on a network that is defined by a single label and not a stack of labels.

Label flattening operation was achieved with a Depth First Search (DFS) based algorithm. The algorithm known for path discovery and tree generation. Through this methodology, Bellessa (2015) discusses that for the network to generate one flattened Label to identify a path; the source devices must broadcast its path discovery message that contains its own ID to all the neighbors. The node that receives the message goes through its corresponding label match and actions. For add and swap actions the node appends the corresponding MPLS label to the discover message and forwards the message to the corresponding node. For a remove action, the node appends the remove MPLS label and forwards to the corresponding node. This in the end create a path that has a flattened unique label Identifier (ID). The DFS algorithm guarantees no repeated path. Therefore, no loops can occur within this network path.

2.6 Proposed System Architecture

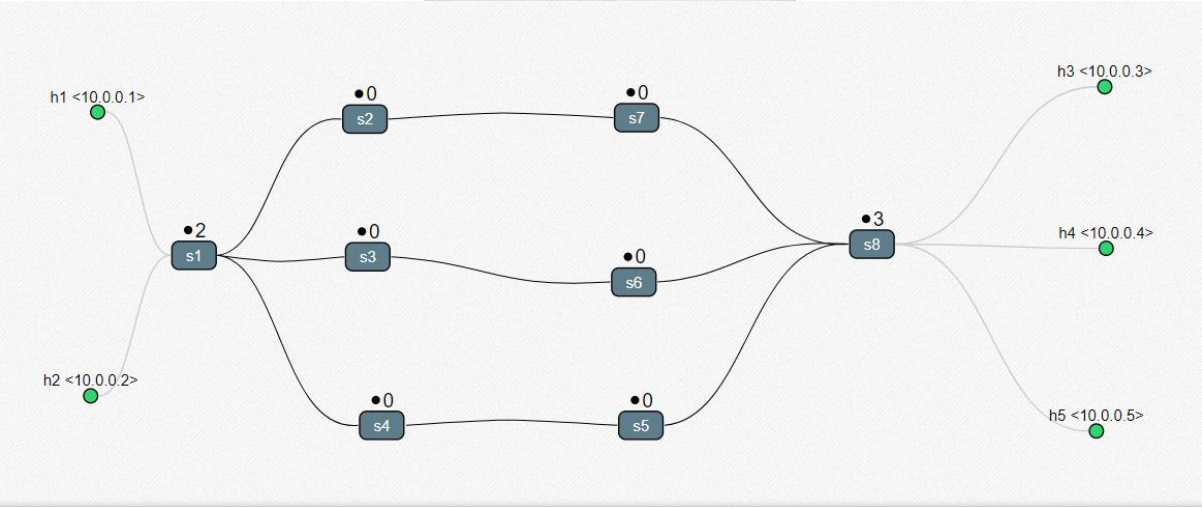


Figure 2-5: System Architecture

The figure 2.6 represents the proposed system architecture this research based its implementation on. The researcher presents, in figure 2.6, a collection of eight switches S1-S8 that will be

running OVS on the Mininet VM discussed above. The different hosts were configured and connected to the switches. The researcher used the switches available to be able to create the LSP that would connect the hosts on S1 entry interface to S8 entry interface. The researcher focused on ICMP traffic to demonstrate this.

Chapter 3 : RESEARCH METHODOLOGY

3.1 Overview

This chapter thoroughly examines the approach that the research took. The actionable approach to ensure the study yielded a reputable and very reliant SDN network based on a routing that is based on MPLS for ICMP traffic. The chapter begins by looking at the research design this study used, that design is be supported target population the research will be working with.

Data collection methodology that has been used for the research's sample space will be covered in this chapter. The chapter then moves forwards to see how that data collected was analyzed to suit the needs and answer the questions presented by the study. Finally, the chapter justifies the quality of the research and looks at the ethical considerations that were made when conducting the research.

3.2 Research Design

A mixed approach was used in this research. This research sought to use both experimental synthesis and prototyping approach to achieve the objectives. This approach allowed the researcher to be flexible in their data collection method and analysis.

The goal of this research was to demonstrate the operation of MPLS assisted routing procedure in SDN. Qualitative approach was used to evaluate different literature that implement MPLS on SDN. This was useful in giving insights on how to come up with an emulation to achieve the objectives set out by the research.

In addition, this research was also a pure research that took the scientific. Experiments were used to test if the set objectives were in tandem with the scientific theories in predicting the outcomes. The simulations done in the research were to show how application separation was done to achieve MPLS assisted routing and show how flows traverse the SDN network architecture.

This study used extreme programming as the system development methodology coupled with Rapid Application development. This is due to the expected always changing parameters of the systems, prototyping and testing and finally due to the reuse of already existing software components that the system will be tweaking to suit the needs of the proposed system.

3.3 Target population and sampling frame

The research adopted probability-sampling method. To be precise, convenience sampling was favorable where the selected samples were easily available and convenient. According to Singh (2006), this kind of sample techniques makes information-gathering process to be quick. This method was applied in choosing the SDN emulator, Mininet with the supporting components like Putty, Xming X server and Wireshark.

Cluster sampling was important in data collection because topology designs on Mininet does not resist the number of switches, hosts and controllers that the SDN emulator can accommodate at a particular instance. The findings of Kothari (2004) supports the sampling technique where he opines that cluster-sampling technique is usually appropriate when compiling the list samples space composing a population.

3.4 Data Collection

This study relied heavily on both primary and secondary data collection methods. Primary data collected was to enable the researcher report on how MPLS assisted in implementing a routing procedure in SDN. The first-hand information that forms primary data was be collected by conducting experiments and observation since the research was pure, taking an experimental approach. The method as backed up by Choy (2014), allows the researcher to check on the validity of the data based on supporting scientific theories already in place.

Primary data was supported by secondary data that equally played a major role in this study. Secondary data sought to explain how adaptive routing techniques could be achieved in SDN. This secondary data also looked at the technologies needed to ensure the researcher developed an application aware network. A thorough understanding of SDN and OpenFlow concepts and MPLS technique was vital to ensure the research succeeded. Secondary data collection, through reviewing of existing literature played a fundamental role in this success.

3.5 Data analysis and presentation

Data from the different Mininet SDN consoles and Wireshark packet capture screens was gathered and analyzed. Inferences and deductions models were applied to draw conclusions from the experiments. The results were represented images with accompanied analysis.

3.6 System Development Methodology

The development and simulation methodology that this research applied was prototyping. Prototyping required the creation of a product that goes through several iterations that span building, testing and reworking until an acceptable prototype that meets the research's objective is arrived at. The researcher chose this approach because it works best for scientific research that does emulation experiments.

This approach also works best in scenarios where the project requirements are unknown and undefined well ahead of time. In this case, developing the MPLS assisted routing procedure in SDN presented the researcher with such challenges. This pushed the researcher to conduct trial and error experiments iteratively until the routing procedure; assisted by MPLS on SDN environment was achieved.

The researcher was involved in the following processes while prototyping:

i. **Basic Requirement Identification:** This process requires the basic understanding of the system requirements. The researcher in this case was out to look for the rudimentary details of the internal and external designs of an OpenFlow based MPLS on SDN and routing techniques in SDN.

ii. **Quick Design:** A quick design of the system based on the data collected by the researcher from the first step formed the activities on this process. In order to realize the objectives that researcher set, paper drawings and computer assisted network diagrams from SDN tools like Spear by Narmox was used at this stage (Demo.spear.narmox.com, n.d.).

iii. **Developing the initial Prototype:** The initial prototype was developed at this stage. After conducting a few tests, some components to enable the researcher design an OpenFlow based MPLS switching on SDN and routing techniques on SDN did not respond as expected. In some instances, some of the LSRs were not implementing the MPLS labels as desired by the researcher. At the same time, the POX component, SDN controller that the researcher developed at this stage was not able to do traffic separation on different switches as envisioned.

iv. **Review of the Prototype:** The prototype had to be presented to relevant stakeholders in the project for review and testing. Feedback collected at this point brought the researcher closer to achieving their objective and enhancing the functionality of the product.

v. **Revise and enhance the Prototype:** Feedback gathered by the researcher was reviewed at this stage. Enhancement and redesigning of the product took place. Considerations that were made by the researcher at this point was the scope of the project as well as the time constraints. The researcher had to revise the custom topology on Mininet and review the best packet forwarding technique to ensure connectivity a number of times before deploying the MPLS technique on each of the LSR. The LSRs were decided at this stage.

vi. **Product Engineering:** The considerations made in the previous stages led the researcher to determine the final prototype to be used in the rest of the research. The dominant design enabled the researcher to meet the objectives was reached at in this stage. The final prototype that was able to do routing assisted by MPLS was finalized in the SDN environment called Mininet.

The type of prototyping applied in this research study is incremental prototyping. Incremental prototyping refers to building multiple functional prototypes of the various sub systems and then integrating all the available prototypes to form a complete system (Peffers et al., 2007).

3.7 Research Quality

The research thoroughly explored the implications of the findings and examined where old and new knowledge are in harmony and where they are not. The research sought to ensure routing protocols are not changed in terms of reinventing the wheel but offering major improvement through precise application separation and alterations of factors that affect routing through knowledge harmonization. Moreover, the research also recommended appropriate course of actions from the findings that were observed

3.8 Ethical considerations

This study used the IT infrastructure provided by Strathmore University. Ethical issues that this research took into consideration were the adherence to the Strathmore IT infrastructure usage policy. The researcher did not pass any traffic to the Strathmore LAN that is considered unlawful by their security policy document. The researcher also took keen consideration to make sure the

software installed and run by this research's system were installed on approval by the Information Technology (IT) department and were deemed safe to deploy, interface and run on the network.

Chapter 4 : SYSTEM DESIGN AND ANALYSIS

4.1 Overview

This chapter focuses on requirement analysis that the proposed routing procedure assisted by MPLS in SDN should meet. It outlines the functional and non-functional requirements. It proceeds to define the environment the researcher used to achieve the implementation of the system. This is coupled with a description of the components the researcher utilized to meet the research goals outlined in the previous sections (Chapter 1, 2 and 3). The chapter then proceeds to present the analysis and the diagrammatic designs based on the required parameters to implement the proposed system.

4.2 Requirement Analysis

In order to achieve the objectives that were set out in chapter 1, this section of the dissertation looks at the various requirements that need to be met by the system.

4.2.1 Functional Requirements

i. The routing procedure should be able to ensure, first, connectivity between the all the nodes in the network topology that the dissertation outlines.
ii. The routing procedure should be able to detect ICMP traffic from nodes in the network and forward them to appropriate switches that will handle such connections.
iii. The routing procedure should allow the various Label Switching Routers in the Label Switching Path to push and pop labels are required by MPLS protocol on ICMP traffic
iv. The routing procedure's output should be captured on Wireshark to indeed prove MPLS was applied on the specified packets.

4.2.2 Non-functional requirements

The proposed system should be able to do traffic separation and employing MPLS to push specified traffic. Therefore, the system should show some integrality by being able to integrate the different components in an SDN to achieve the proposed routing procedure. The system should also demonstrate compatibility to the set of communication software standards in running such routing procedures in SDN environment.

4.3 Diagrammatic Representation of the system

4.3.1 General System Architecture

The figure below shows how Host 1, Host h3 and the OpenFlow controller will be able to communicate with the OVS forwarding devices that are configured to do MPLS traffic forwarding. The controller is the most important component configures flow table manipulation messages to the switches to allow them to do MPLS packet forwarding. The Host devices, Host 1 and Host 2 are the initiators of the ICMP packets to the labeled by the switches to match MPLS flow specifications defined by the OpenFlow controller messages delivered via OpenFlow protocol to the switches.

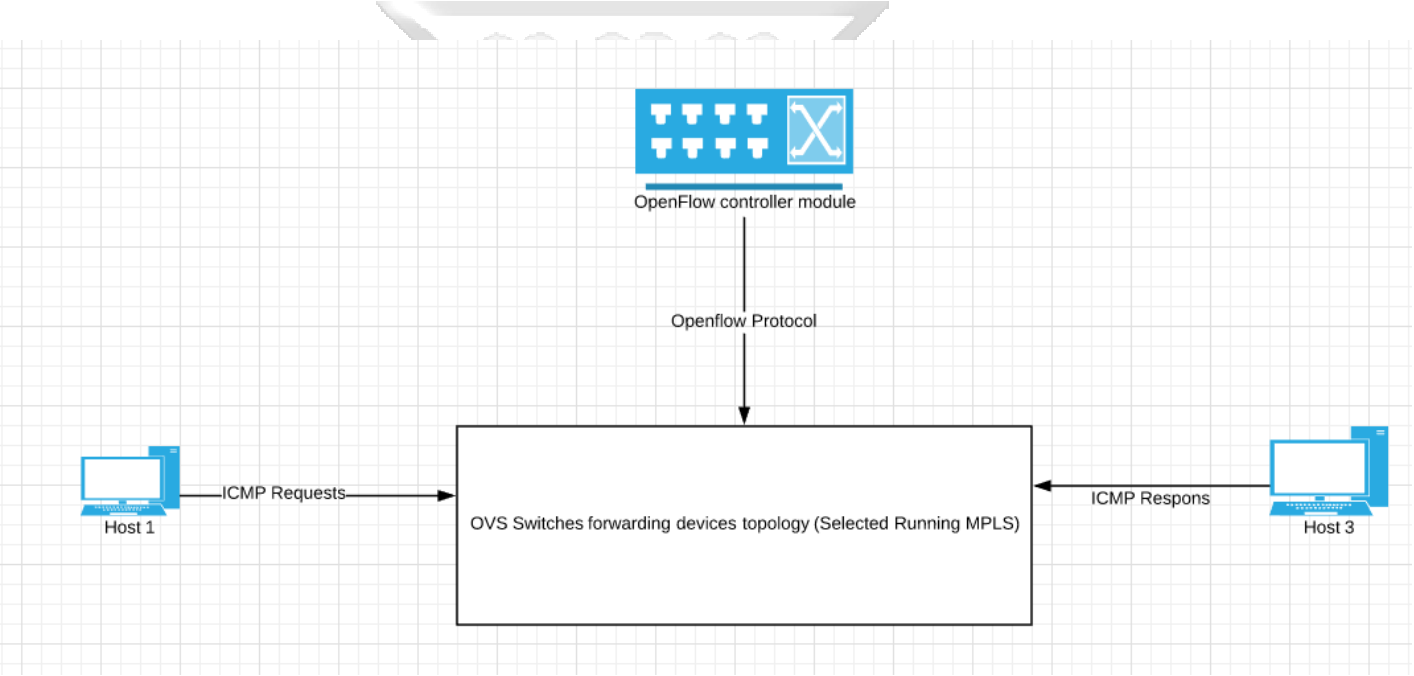


Figure 4-1: General System Architecture

4.3.2 Use Case Diagram

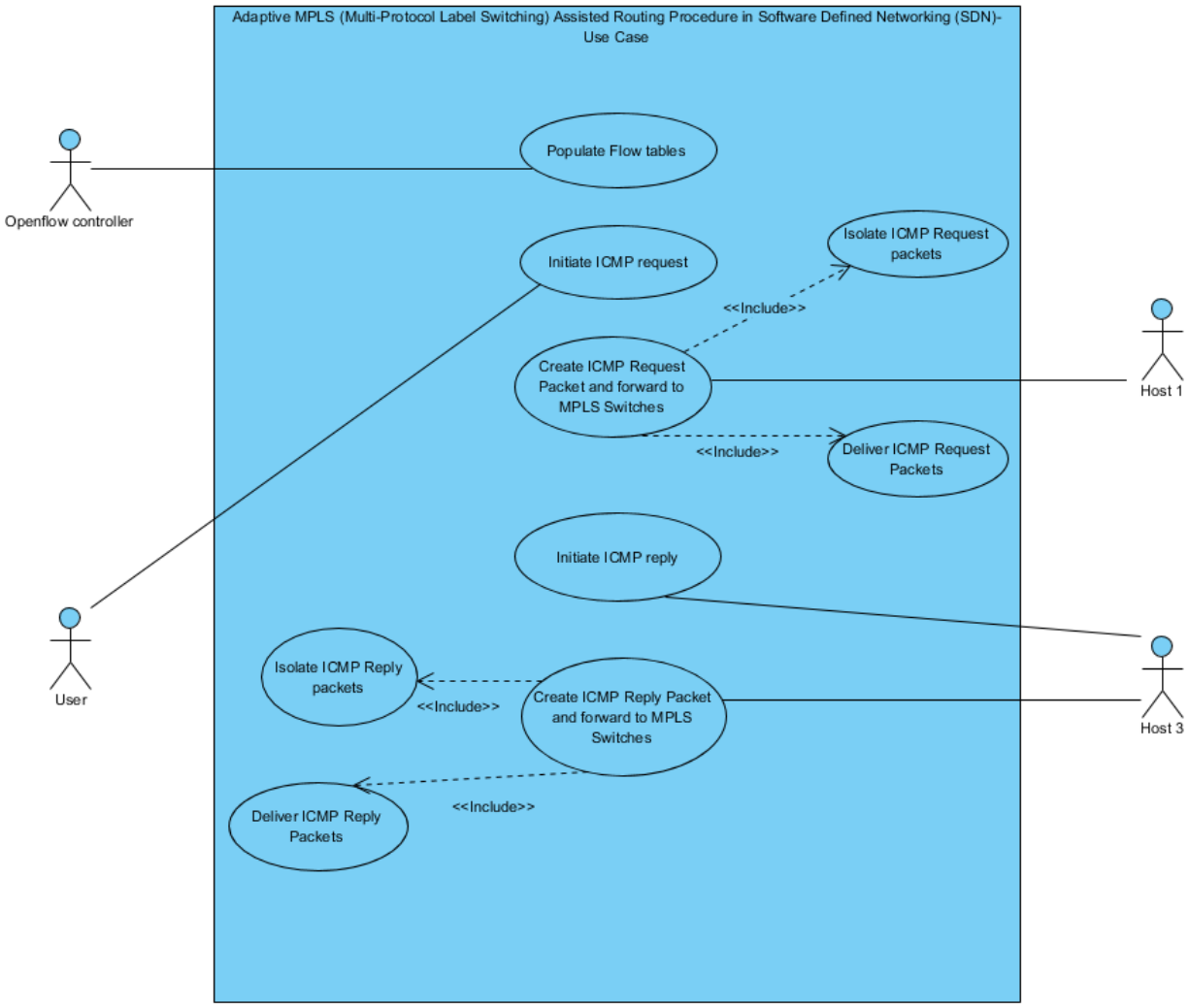


Figure 4-2: Use case Diagram

The use case depicted above consists of four possible users that will interact with the system. The OpenFlow controller will be responsible for feeding the OpenFlow messages that will emulate an MPLS assisted routing procedure in SDN system. The User in the system will be responsible of initiating the ICMP requests that will target Host 3. Host 1 will be responsible in packetizing that request in an ICMP request message and add the necessary information to allow the switches in the SDN setup to act on it accordingly. At this point, the system will receive this information, inspect the packet whether it is an ICMP packet or not based on the OpenFlow rules given to it by the controller.

Afterwards, the system will isolate the ICMP request packets and forward them to the switches predefined by the controller to handle MPLS request in the network. This process will ensure the ICMP request packet is delivered to Host 3.

Host 3 is responsible of initiating an ICMP reply procedure and constructing an ICMP reply packet, adding the appropriate content and forwarding the message to the system for manipulation. The system will repeat the process of inspecting the packet whether it is an ICMP reply or not. All ICMP replies will be submitted to the flow pipelines defined by the system to the appropriate switches that will then deliver this reply to Host 1.

The assumption made in the use case diagram is, the topology of the network was configured and launched correctly without any errors.

4.3.3 Sequence Diagram

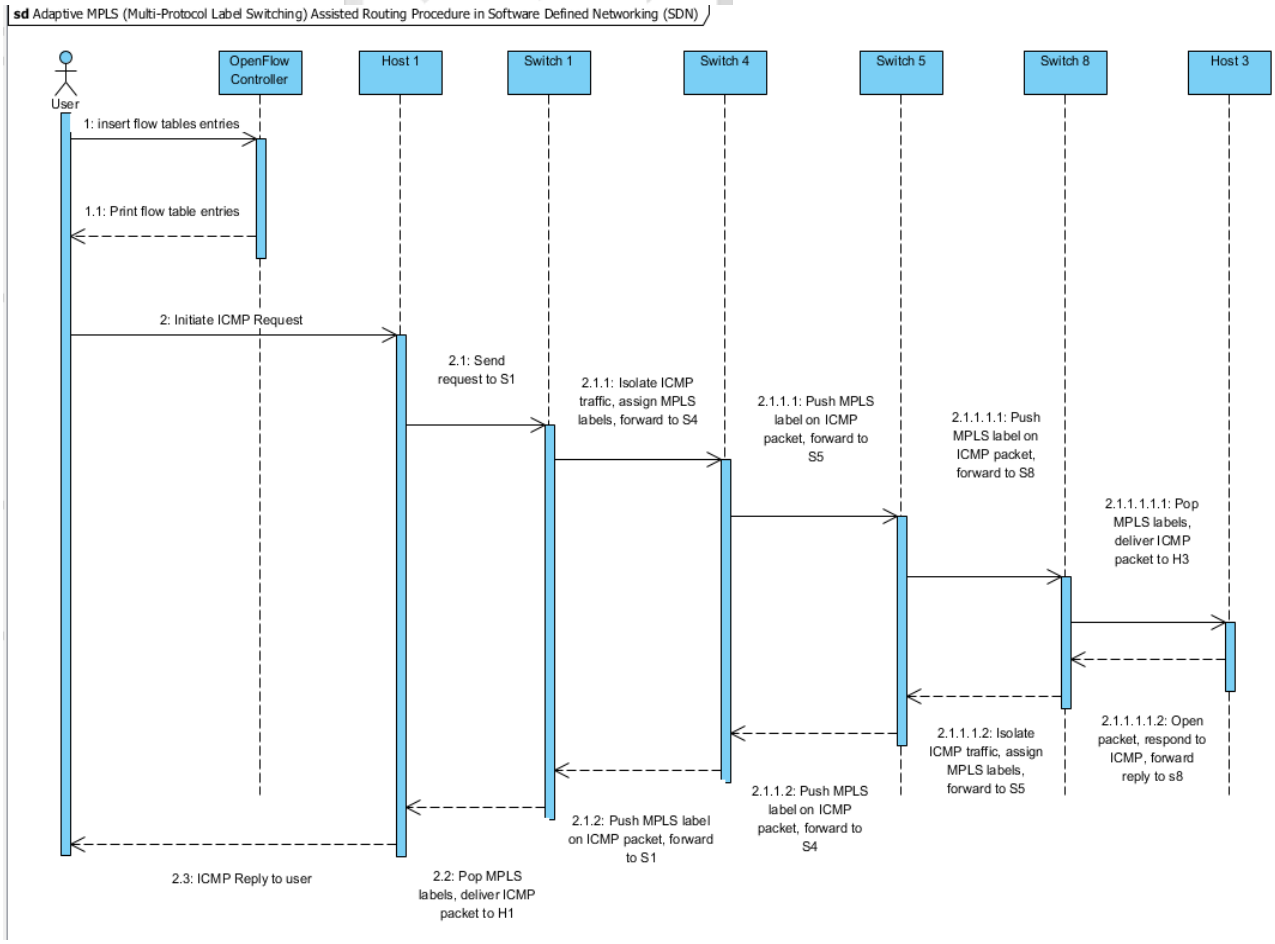


Figure 4-3: Sequence Diagram

The sequences diagram illustration above outline the series of events the actors and the system will be engaged in to bring MPLS routing procedure in SDN to fruition. The user who will double up as the network administrator will send configuration messages via OpenFlow protocol to the OpenFlow controller module that will feed this configuration to all the switches. The flow table entries will be printed back to the user. The user will initiate an ICMP request message. This message will run ARP in the background as the Layer 2 protocol. The switch 1, which is the ingress switch as captured in the network topology, will identify the ICMP packets sent to it and isolate them. This switch will check these packets across its pipeline and try to match the flows with the flow table entries in the flow table. A successful match will allow the switch push an MPLS label and forward this ICMP traffic through its output port that connects it to switch 4 as defined in the FIB that was constructed by the OpenFlow controller on each switch.

Switch 4 and switch 5 will receive these packets, tagged to be forwarded to them through their respective input ports, and push their own MPLS labels to the ICMP packet and forward them out through their output ports sequentially. Switch 5 will forward directly to switch 8.

Switch 8 as the egress switch, for traffic coming from Host 1, will be responsible in popping the MPLS labels attached to the ICMP packets and forwarding this packets through its output port that connects it to Host 3. Host 3 will interpret the packet, ICMP request, and construct an ICMP reply forwarded to switch 8 again.

Switch 8 in the instance will act as the ingress switch for the ICMP reply coming back from Host 3. Switch 8 will identify the ICMP reply packets sent to it and isolate them. Switch 8, will check these packets across its pipeline and try to match the flows with the flow table entries in its flow table. A successful match will allow the switch to push an MPLS label and forward this ICMP traffic through its output port that connects it to switch 5 as defined in the FIB constructed by the OpenFlow controller on each switch.

Switch 5 and Switch 4 will receive these packets tagged to be forwarded to them through their respective input ports, push their own MPLS labels to the ICMP packet and forward

them out through their output ports sequentially. Switch 4 will forward directly to switch 1.

Switch 1 will acts as an egress switch for the ICMP reply and do the popping of MPLS labels attached to the ICMP reply and forward this packet through its output port that connects it to Host 1. Host 1 will interpret the packet and mark the end of the activities involved in the system.

Chapter 5 : SYSTEM IMPLEMENTATION AND TESTING

5.1. Overview

This chapter focuses on how the proposed system design in the previous chapter was achieved. The chapter discussed the model components and the test bed setup that the researcher assembled during the implementation and testing phase. The chapter then moves to look at the actual implementation where topology design, link discover implementation and embedding of MPLS labels to ICMP packets takes center stage. The chapter proceeds to report on the test results that the researcher collected using multiple terminal windows and Wireshark capture results screenshots. Finally, the chapter concludes by looking at the challenges the researcher faced during the implementation phase.

5.2 Model Components

The software and hardware components that were used for the experiment to be realized in this research were informed by the nature of the tests to be conducted to achieve the objectives set out by the research.

1. Hardware components, A Personal Computer
    i. 4 Gigabyte (GB) Random Access Memory (RAM)
    ii. 500GB hard drive disk storage
    iii. Intel core i5-443OS at 2.7 GHz of processing speed with DirectX 11 graphical capabilities.
2. Software Components
    i. Windows 10 Host Operating System
    ii. VitualBox Hypervisor
    iii. Ubuntu Server 14.04 Custom Mininet Version 2.2 image (Guest OS)
    iv. Putty terminal emulator, serial console and network file transfer application.
    v. Xming X server X11 display server for Windows Operating System

5.3 Test bed Setup

The emulation of an SDN network to support an MPLS assisted routing was done primary on the Mininet custom image running on Ubuntu 14.04 virtual machine. This emulation will be dependent on other components to make it work efficiently. It was imperative for the researcher to confirm

that POX, OVS and OpenFlow Protocol is running on the Mininet Virtual Machine (VM). To test for the correct functionality of the three SDN components of choice by the researcher, a simple Mininet network topology was invoked and tested.

The versions of these components played an important role in the implementation of the research objectives. Other components required to be intergraded by the researcher is connecting Putty and Xming X server to the Mininet VM. The researcher opted to load and save configurations on Putty that connects to the Mininet VM and enables X11 forwarding for Xming display capabilities to be realized.

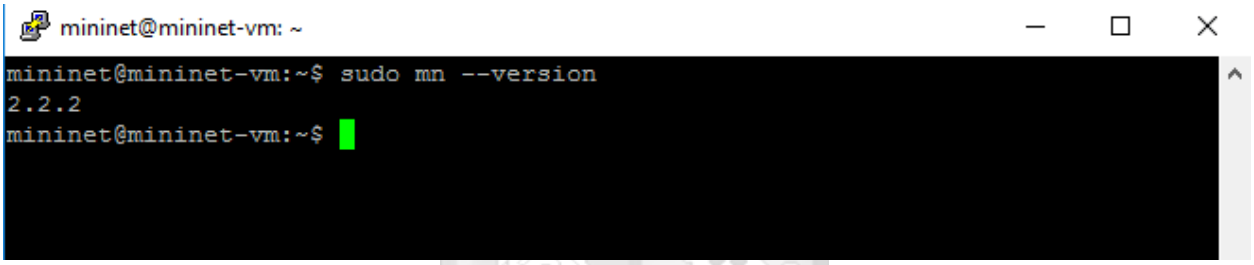


Figure 5-1: Verification of Mininet Version 2.2 Installation

Figure 5-1 shows the results obtained by the researcher when confirming that Mininet was installed which further shows the version Mininet is running post installation.

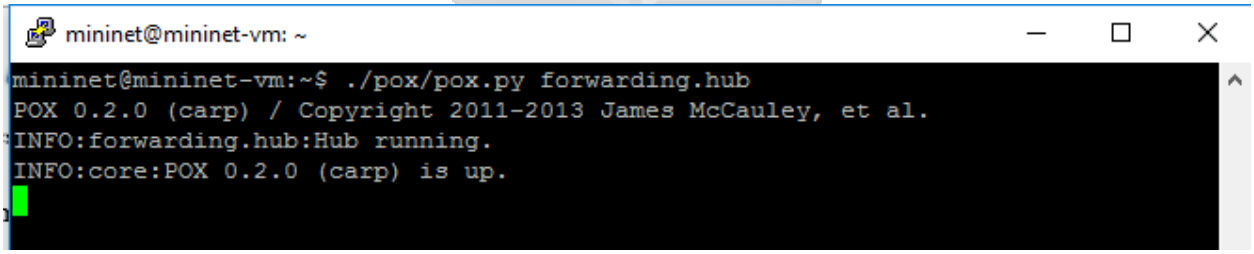


Figure 5-2: Verification of POX installation, version and correct functionality

Figure 5-2 shows the results obtained by the researcher when running POX controller for the first time to confirm its correct operation. The researcher opted to run a test hub operation from the controller by referring to a file that emulates a hub operation in POX called forwarding.hub as shown the figure. The results show that controller was up and running and when queried, it will inform the switches to forward traffic based on how a hub would operate.

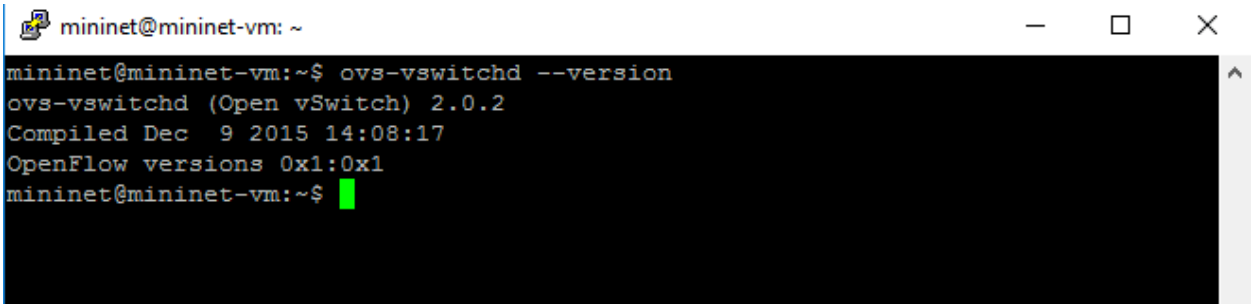


Figure 5-3: Verification of Open vSwitch installation and OpenFlow version

Figure 5-3 shows the results obtained by the researcher when confirming that OVS was installed which further shows the version Openflow and OVS are running post installation.

5.4 System Implementation

The system implementation was achieved through three phases; topology design, link discovery to ensure connectivity between the nodes and finally matching ICMP traffic and embedding MPLS labels on ICMP packets.

5.4.1 Topology design

The first phase was to design a network topology that matches the diagrammatic representation of the model shown in figure 4-1. This was to be done on Mininet using custom topology definitions class that Mininet provides. This topology was written in python to emulate the 8 OVSs, 14 Ethernet links and 1 POX controller. The python file to create the network topology designed by the researcher was named *thesis.py* Details of the topology from the Mininet console are shown in the figure 5-4 below.

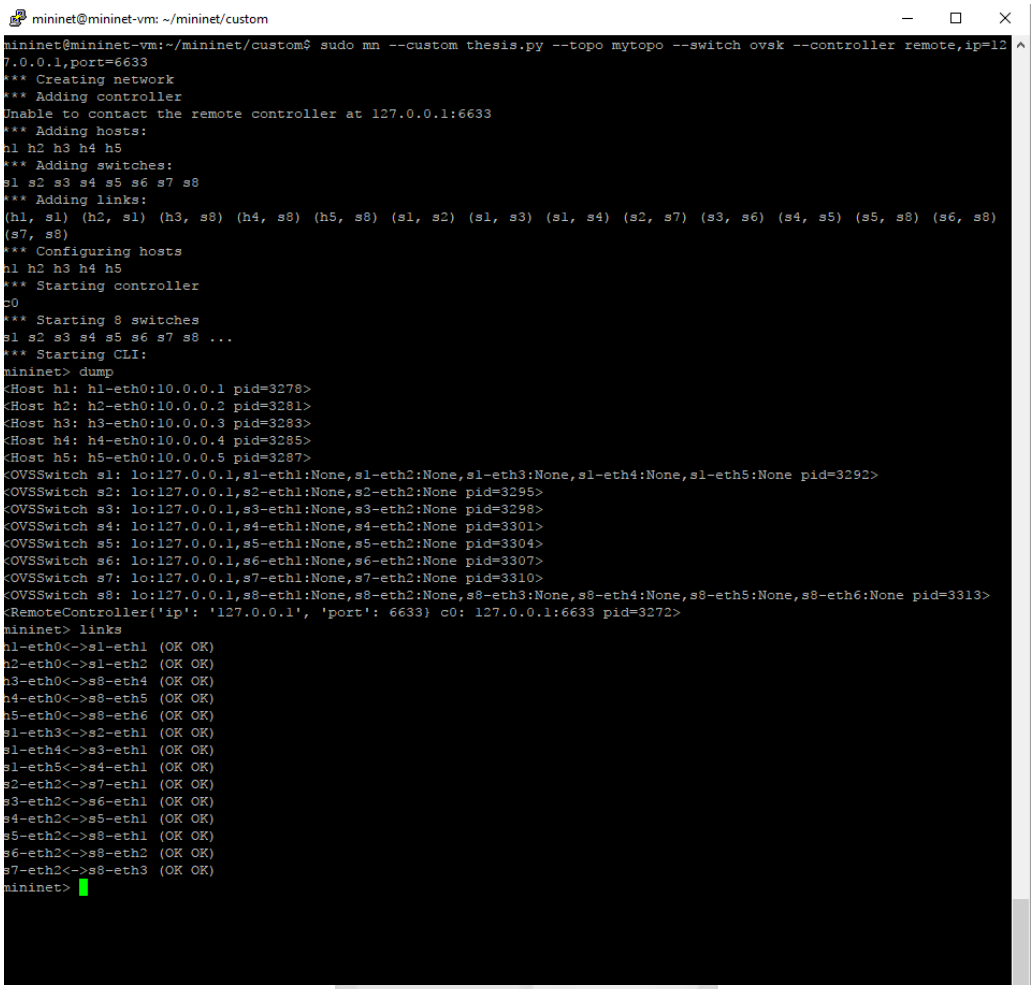


Figure 5-4: Mininet console outlining topology

5.4.2 Link discovery to ensure connectivity between the nodes

Since the researcher opted for a custom topology connected to a remote controller invoked by this command on Mininet, "***sudo mn --custom thesis.py --topo mytopo --switch ovsk --controller remote,ip=127.0.0.1,port=6633***" as shown the figure 5-4 above. It was important for the researcher to make sure there is connectivity between the nodes in the topology. This was achieved by looking for a component on OVS that does Link Layer Discovery Protocol (LLDP) and allows devices with in a network to advertise their identity. The researcher coupled this vendor neutral link layer protocol with OpenFlow link layer discovery mechanism to ensure the nodes could connect to one another. The figure 5-5 below shows invocation of this link discovery procedure carried out by running the command, ***./pox/pox.py forwarding.l2_multi OpenFlow.discovery***.

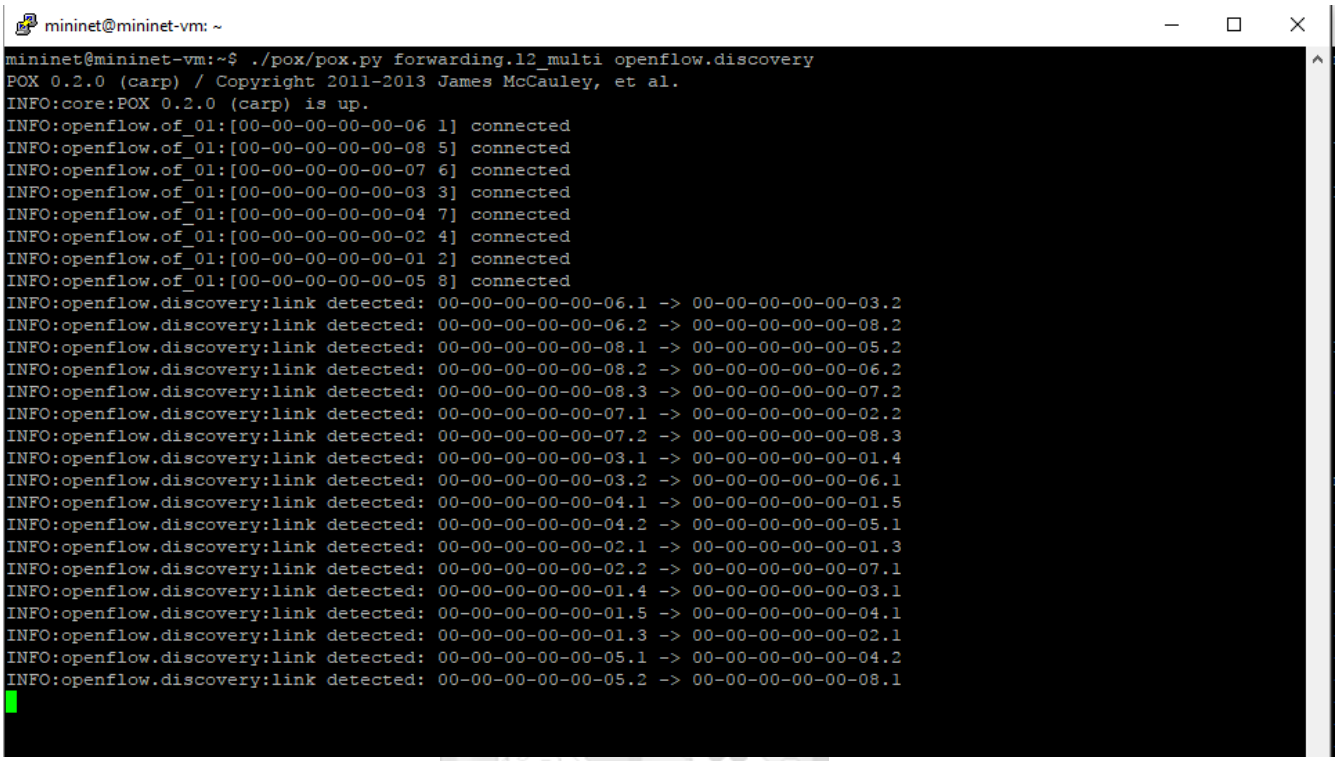


Figure 5-5: Link discovery results

The figure 5-6 that follows shows the ping results that were obtained by the researcher after running the ***pingall*** command from the Mininet console.

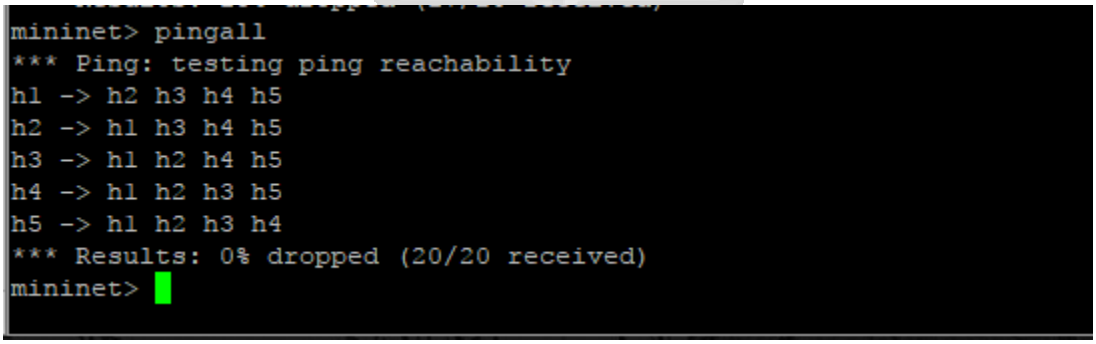


Figure 5-6: Connectivity test results

5.4.3 Embedding MPLS labels on ICMP packets.

In order for MPLS labels to be embedded to the ICMP packets being sent on the topology defined by this research in section 5.4.1, there were a number of modifications that the researcher needed to in the test bed environment. As of release date of the Long Term Support Version (LTS) of the Mininet VM, it only supported version 2.0.2 of OVS and OpenFlow version 1.0 as shown in figure

5-1 and 5-3 respectively. However, to implement MPLS labels on ICMP packets required the upgrade of OVS to allow it run version 2.5.4 that in turn supported OpenFlow up to version 1.3 that would push and pop MPLS labels with ease.

After the upgrade of OVS on Mininet VM, the researcher picked on switch 1, switch 4, switch 5 and Switch 8 to form the LSP that MPLS labeled ICMP traffic would follow. Switch 1 and Switch 8 were designed to do both MPLS Penultimate Hop Popping (PHP) and Label pushing since they were the edge switches, ingress and egress switches.

On the other hand, the switch 3 and switch 5 were configured to perform label pushing of the ICMP traffic that passes through them in both directions. More information to illustrate how MPLS labels were being pushed and popped by the LSRs in the LSP is captured in the next section 5.5.

The researcher configured these LSR by sending OpenFlow messages that would allow the ingress and egress switch to specifically forward ICMP requests and responses via the path provided and at the same time attach MPLS labels by pushing and popping them via the OpenFlow messages sent to the switch.

For switches 1 and 8, the researcher opted to use two flow tables. The first table was for identification of ICMP traffic and pushing Address Resolution Protocol (ARP) requests and the second flow table was for pushing and popping MPLS labels. The tables were identified as table 0 and table 1 respectively. Table 0 was resubmitting traffic to table 1 for MPLS labelling as envisioned by the researcher.

Switches 4 and 5 FIB operated by the use of only one flow table that was designed to do label pushing in both directions. The flow tables in switch 4 and 5 were identified by their default value which is 0. The figures below illustrate the different flow table entries for the switches from switch 1 to switch 8.

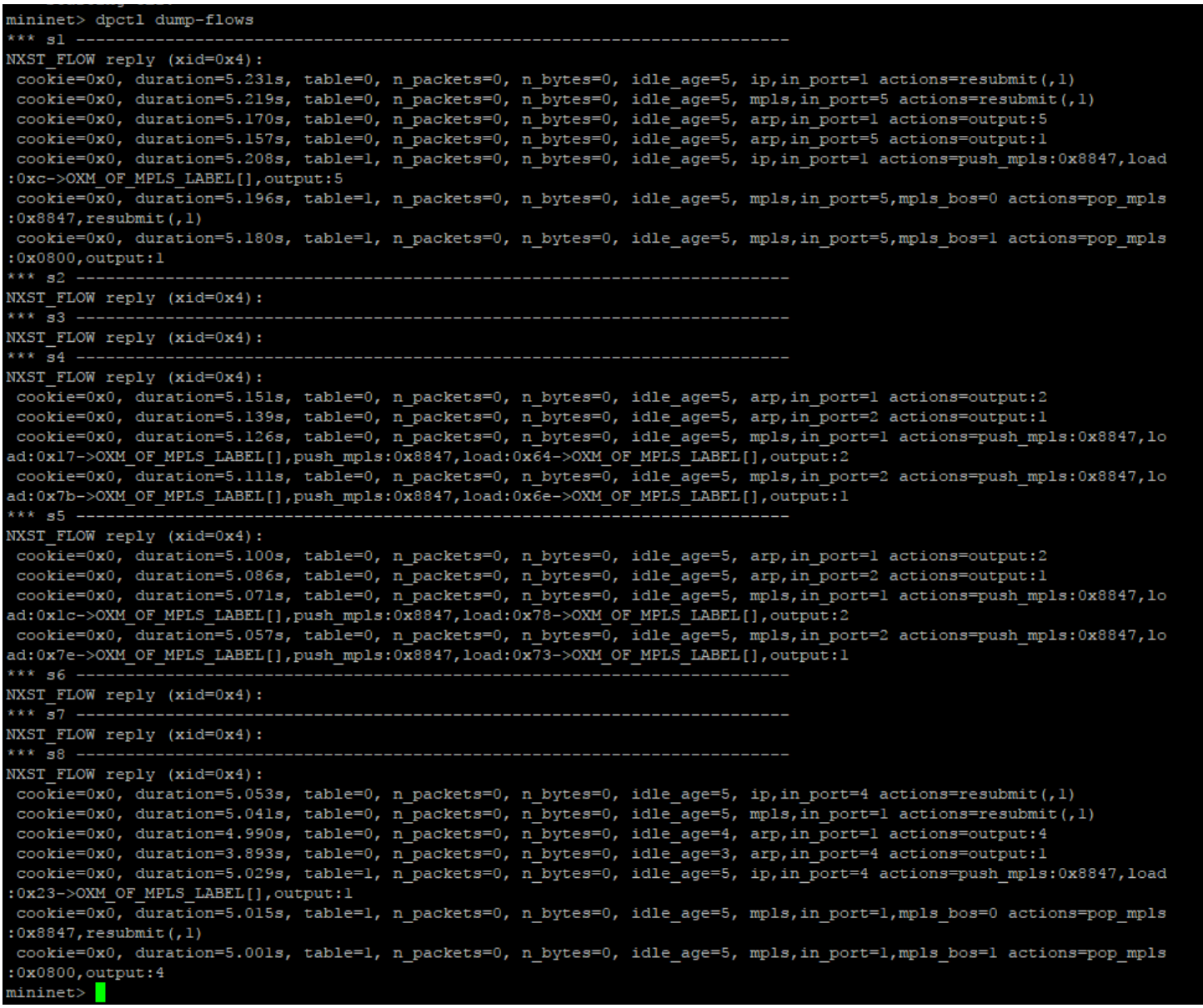


Figure 5-7: Flow table entries on S1, S4, S5 and S8

The figure 5-7 shows the flow table results that were obtained by running dptcl-dump flows command on the Mininet VM terminal. From the flow table results above, it can be seen that S1 and S8 were configured to have two flow tables, table=0 and table=1. The first table=0 was defined to isolate IP traffic, in this case ICMP traffic and forward it to table=1 for the addition on MPLS labels. Table 0 at the same time was used to forward ARP requests. The other switches in the LSP, S4 S5 and S8 was shown figure 5-7 had only one flow table table=0 that was used to only push MPLS labels.

5.5 System Testing

The experiments that this research conducted by this research were aimed at testing whether the ingress and egress switches, the edge switches were able to single out ICMP traffic and the same time pop and push MPLS labels. In addition, the other switches in the LSP were tested whether they were able to push MPLS labels in both directions of traffic flow. The terminal emulators (Xterm) illustrated the rest results from the research for Switch 1, Switch 4, Switch 5 and Switch 8 coupled with Wireshark packet capture results.

5.5.1 Xterm results for the ingress and egress switches

The results were obtained by doing a *tcpdump* on the Ethernet interface 5 in switch 1 and Ethernet interface 1 in switch 8 as shown below.

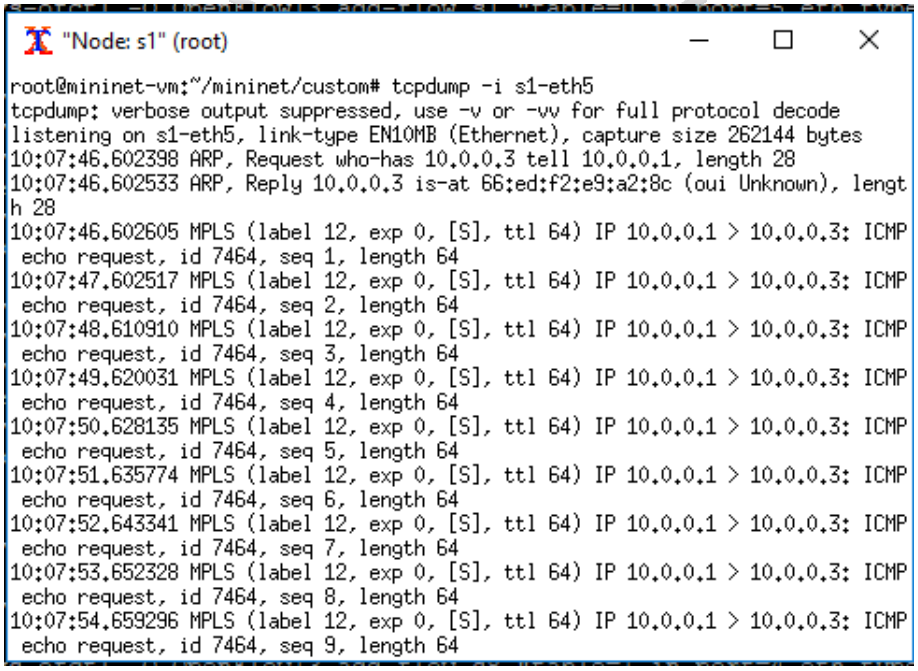


Figure 5-8: Xterm Results for S1

Figure 5-8 shows how S1 on interface 5 was able to add label 12 to the traffic passing through it as it goes to H3 which is identified by IP address 10.0.0.3. The figure also demostrates that the packet which MPLS label 12 was added was an ICMP packet.

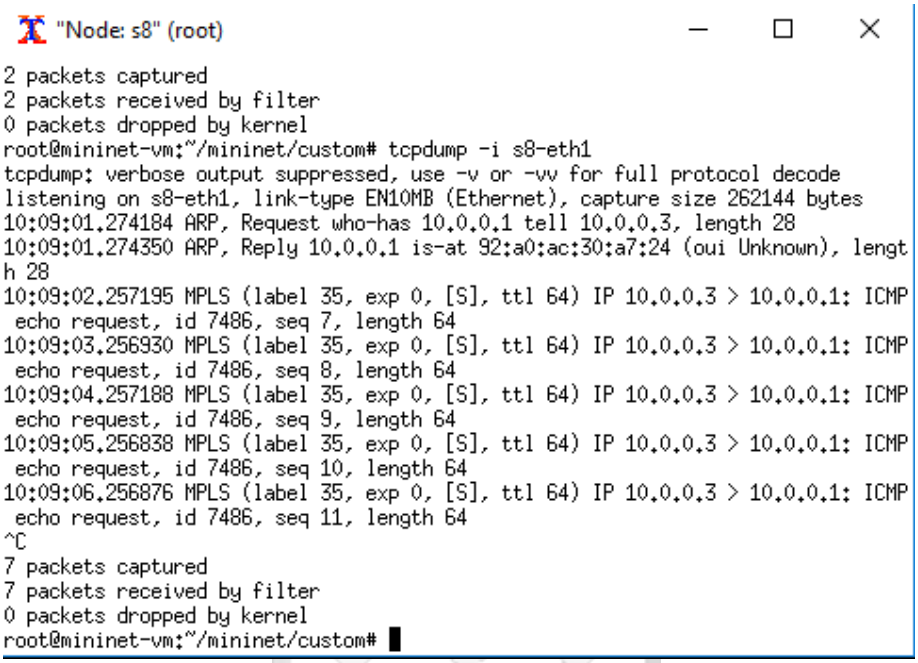


Figure 5-9: Xterm Results for S8

Figure 5-9 shows how S8 on interface 1 was able to add label 35 to the traffic passing through it as it goes to H1 which is identified by IP address 10.0.0.1. The figure also demostrates that the packet which MPLS label 35 was added was an ICMP packet.

5.5.2 Xterm results for the other switches in the LSP

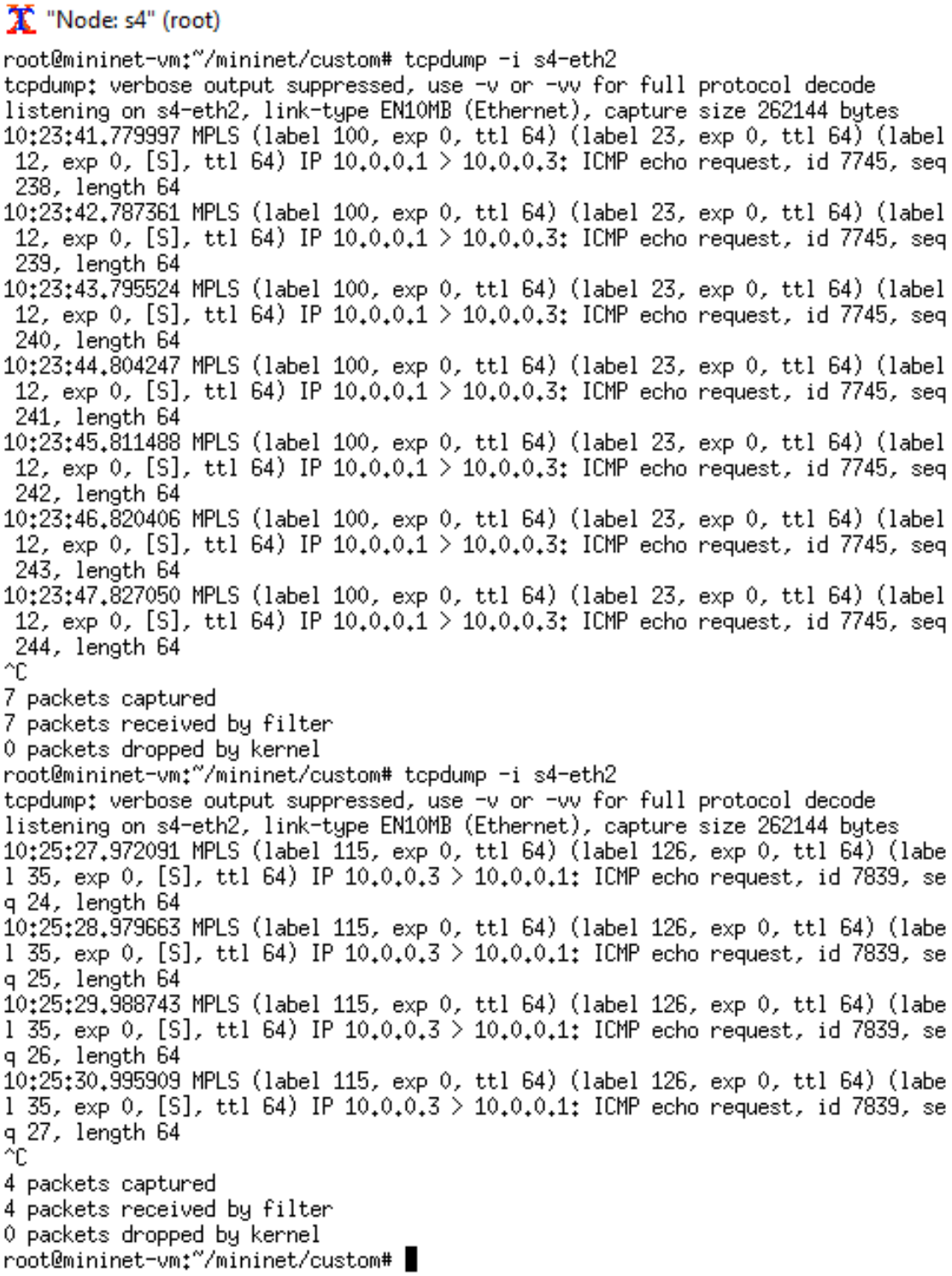


Figure 5-10: Xterm Results for S4

Figure 5-10 shows how S4 on interface 2 was able to receive traffic coming from H1 which had passed through S1 and label 12 was added. S4 additionally adds label 23 and label 100 to the traffic passing through it as it goes to H3 which is identified by IP address 10.0.0.3. The figure also demostrates that the packet which MPLS label 35 was added was an ICMP packet. The same figure 5-10 on the lower half, demonstares how reverse traffic coming from H3 to H1 is labeld anf forwarded. From the figure, traffic that was labelled by S8 and label 35 added has been given two additional labels, 126 and 115 as it moves from H3 to H1 via S4.

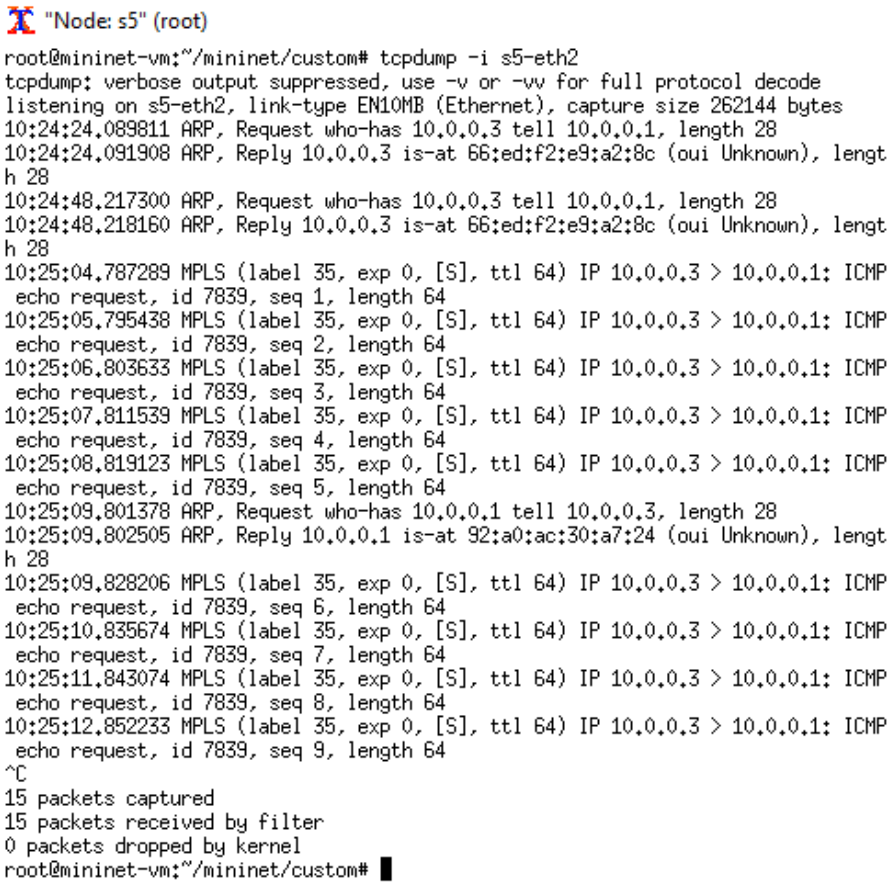


Figure 5-11: Xterm Results for S5

Figure 5-11 on the other hand shows how S5 pushed MPLS label 35 that was added to ICMP packet moving from H3 to H1 via S5 from S8.

5.5.3 Wireshark capture results from Host 1 to Host 3

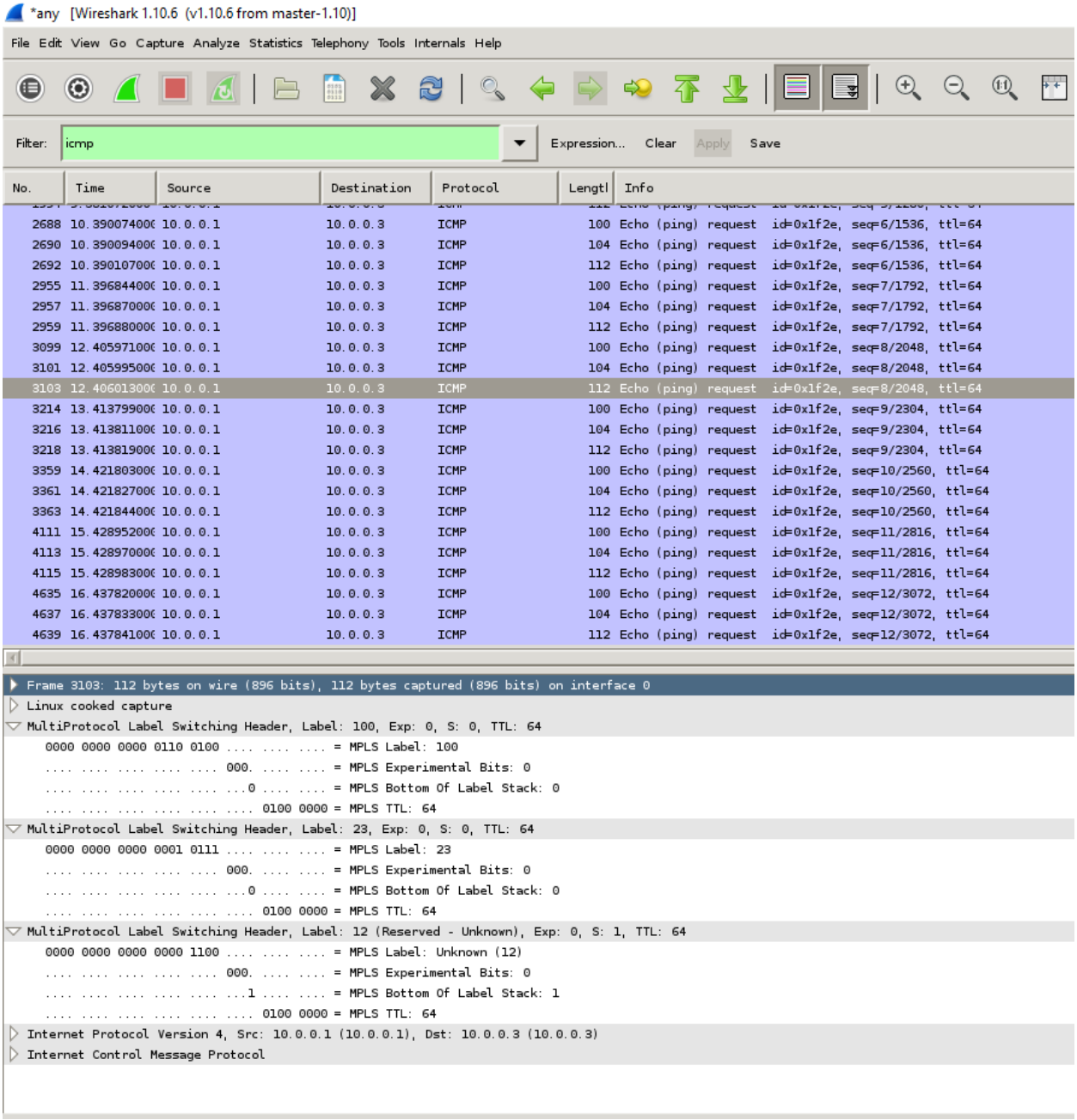


Figure 5-12: Wireshark capture results from H1 to H3

5.5.4 Wireshark capture results from Host 3 to Host 1

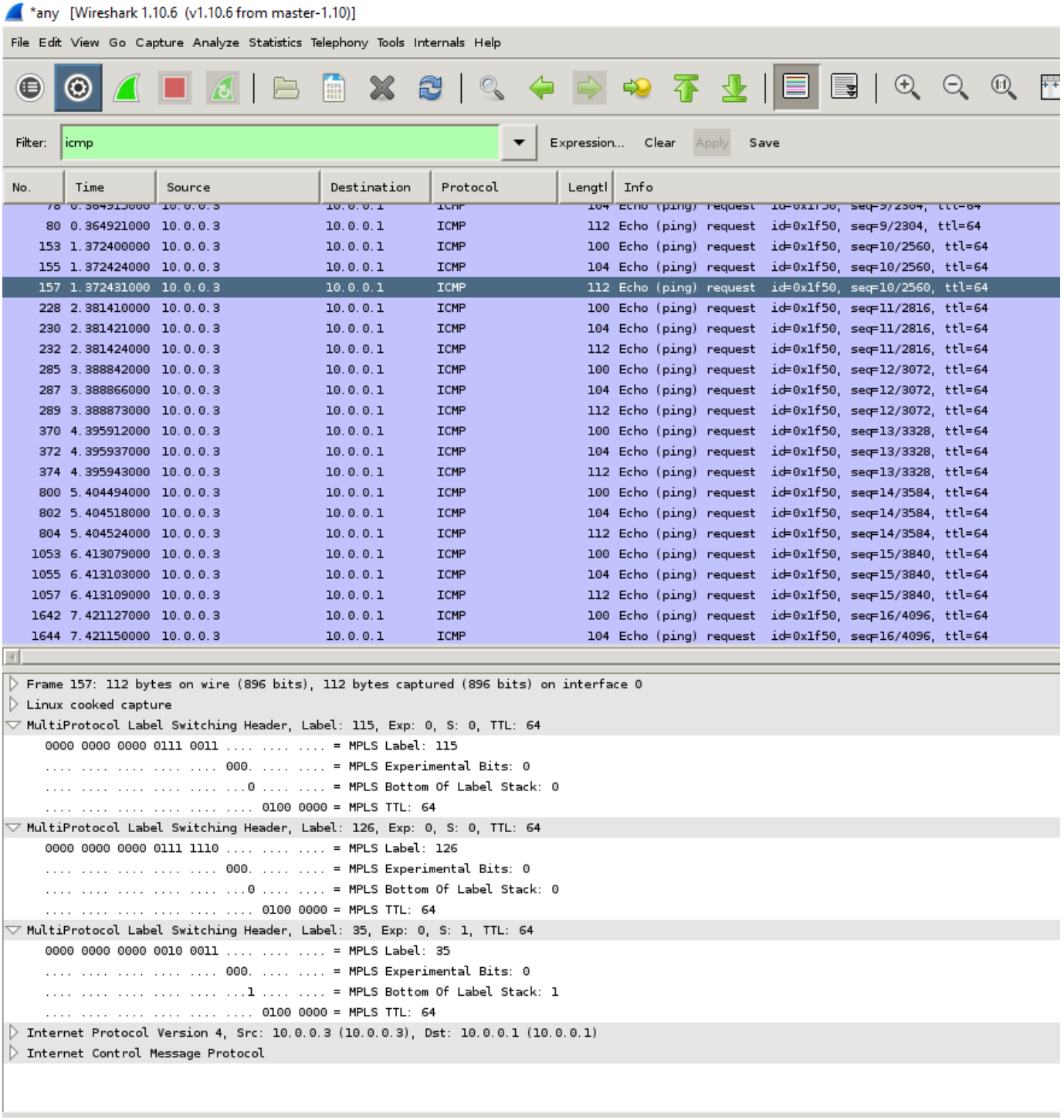


Figure 5-13: Wireshark Capture Results from H3 to H1

Figure 5-12 and 5-13 bring out the concept of label stacking. These figures demonstrate, from the Wireshark capture screens how traffic moving from H1 to H3 via the defined LSP had a label stack

of 12, 23 and 100 as shown in figure 5-12. On the other hand, traffic moving from H3 to H1 had a label stack of 35,126 and 115 as shown in the Wireshark capture on figure 5-13. These two figures also show that these labels were attached on IP traffic, specifically, ICMP traffic.

5.5.5 System testing classes

Table 1: System testing classes

Test Class	**Inspection Check**	**Priority**
Functional	Are there labels being popped and pushed by Label Switching Routers in the Label Switching Path as required by MPLS protocol on ICMP traffic by the routing procedure?	High
Functional	Does the routing procedure ensure, first, connectivity between the all the nodes in the network topology that the dissertation outlines?	Medium
Functional	Does the routing procedure detect ICMP traffic from nodes in the network and forward them to appropriate switches that will handle such connections?	High
Functional	Is there output being captured on Wireshark to indeed prove MPLS was applied on the specified packets?	High
No functional	Is there integrality and compatibility to software standards being demonstrated by the system implementation?	Medium

5.5.6 System testing results

Table 2: System testing results

Test Class	Test Results	Comment
Functional	Pass	MPLS labels were being pushed and popped by the relevant LSRs on the LSP while ICMP traffic was singled out and rerouted via a specified path
Non Functional	Pass	There was integrality during the implementation of the system and adherence to software standards was demonstrated.

5.6 Challenges faced in implementation

To implement the research's objective stumbled upon challenges that affected the functionality of the system. The complexities this work came across are discussed on the next section 5.6.1

5.6.1 Complexity

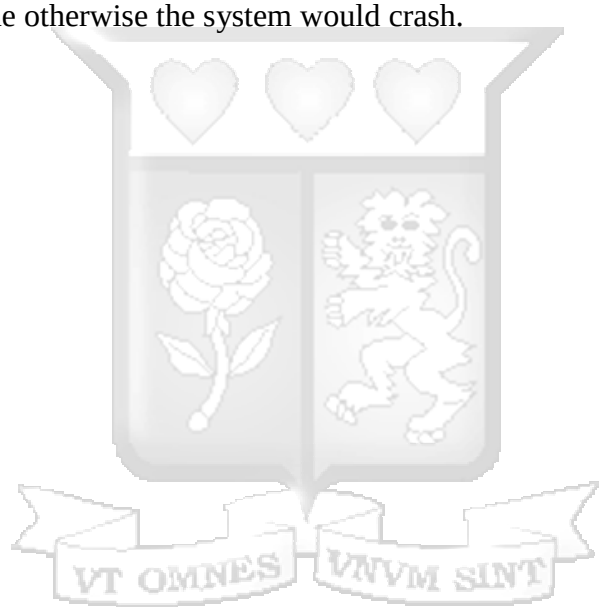
Understanding SDN, since it is an emerging topic in computer networking was a complex task. In addition, understanding the operation and manipulation of the different components like OVS, OpenFlow message structure and POX controller operation was a challenge the researcher faced. The researcher was pushed to study thoroughly the documentation of these software components in order to make sure they can relate and work to achieve the objectives set out by the research. The researcher was also required to understand the underpinning concepts of SDN before the implementation of the system.

5.6.2 Conflicting version of software components

Implementing MPLS in SDN was an enhancement on what SDN with OpenFlow could do on an emulated testbed. As of the time of the implementation of this research, Mininet.org, released the Mininet VM image that runs OVS version 2.0.2 and OpenFlow version 1.0. Painstaking research revealed that MPLS pushing and popping of labels via OpenFlow is easily achieved by running OpenFlow version 1.3.0 or higher. This pushed the researcher to upgrade OVS to a version that could support OpenFlow 1.3.0, which was OVS 2.5.4.

Upgrading OVS from the current Mininet 2.2 release was a critical procedure that required a lot of tenacity in how it was done. The researcher, in multiple occasions, damaged the kernel components of the VM (Mininet VM) that required the whole process of implementation to be started from scratch.

Another conflict that the researcher encountered was the interconnectivity of the devices in this research's topology and allowing them to communicate. In the first instance when running LLDP on OpenFlow, the devices would be able to ping each other. However, upon running MPLS that required OpenFlow version 1.3, it would end up crashing the OpenFlow LLDP that is designed to run on OpenFlow version 1.0. This meant that the researcher could not run both procedures on the controller at the same time otherwise the system would crash.



Chapter 6 : DISCUSSION

6.1 Overview

This chapter brings the test results gathered in the previous chapter into perspective. The discourse in this chapter looks at how Mininet was able to implement an MPLS assisted routing procedure on SDN with the help of OVS and OpenFlow. This chapter also sheds light on the test results that were obtained by this research. This chapter concludes the research and offers recommendation and insights to future studies that can build up from this.

6.2 Discussion of results

The design of an operational topology as outlined by the diagram and the terminal results captured in figure 5-4 sets precedence to achieving the research's objective. The topology consisted of five Host named H1, H2, H3, H4 and H5. Only H1 and H3 were chose to participate in the forwarding of ICMP encapsulated MPLS packets. However, it is important to note that any device could be chosen and can as well forward traffic in the network. Nonetheless, the decision to choose the two devices was based on the discretion of the researcher and to be better illustrate the concept that this research advocates for.

When it comes to the participating switches in the LSP, again, it was out the researcher's preference to choose S1, S4, S5 and S8. S1 and S8 were inevitable because they are the ingress and the egress switches in the topology.

From the results capture by the different Xterm screens, Mininet console and Wireshark, the research has illustrated that S1, S4, S5 and S8 were configured to do MPLS based on the flow table entries shown in the chapter above. These flow entries were very specific to each switch in the instance that there were predefined labels that each of the switches were supposed to push to the next LSR in the LSP.

When it comes to label pushing, the tcpdump results captured from S1 shows it pushing label 12 as the traffic is moving from H1 to H3. S4 on the other hand is shown to push different labels depending on the direction that the traffic is going. As it can be seen from the terminal screen capture of S4, label 23 and 100 is pushed when the traffic is moving from H1 to H3 and label 35, 126 and 115 received from S5 and S8 are used to forward and push traffic that comes from H3 to H1. The same explanation applies to the screen capture from terminal of S5. Label 35 is pushed to

S4 when traffic is coming from H3 to H1 as the test results on the image suggests. The researcher opted to display only the terminals of OVS switches S1, S4, S5 and S8 because they were the only configured to participate in MPLS.

The Wireshark capture screen used to report and analyze the test results in the previous chapter brings out the concept of label stacking in to perspective. The figure 5.3.3 shows the Wireshark capture of traffic of traffic coming from H1 to H3. Each of the OVS switches has achieved label stacking in that traffic's LSP by pushing an MPLS label on top of the packet being sent. As the figure illustrates, S1 pushed label 12 followed by label 23 and label 100 that formed the label stack of the packet that traversed that LSP. The same results are shown by the figure 5.3.4 where traffic moves from H3 to H1. The label stack that is realized by the LSRs involved in the LSP is made up of labels 35, 126 and 115.

Finally, the research has demonstrated that only ICMP packets were able to be captured and assigned labels to kick start MPLS operation. The Wireshark captures discussed above were set to filter only ICMP packets on the filter bar to demonstrate MPLS effectively.

Chapter 7 : CONCLUSION AND RECOMMENDATION

7.1 Conclusion

In order to have networks that are more customized, SDN is the current trend that allows this flexibility and customization. With these capabilities, network engineers can no longer be limited in terms of how creative they can get when it comes to network design. As highlighted by the research, Mininet provides a suitable environment for any kind of creativity to be emulated, tested and probably ported to the real world scenario.

The openness of networking as SDN decouples the control plane and data plane and allowing them to be run by components that are purely open source enhances understanding into how networks operate and how they can be manipulated effectively. This research proposed and implemented a routing procedure that is based on MPLS to send data traffic on an SDN environment. As demonstrated by the research, this once complex task of configuring MPLS has been achieved in the least cost, financially with minimal vendor specific requirement as possible. The research has highlighted and emphasized from the implementation that allowing central connectivity, manageability and control of the network makes design and troubleshooting of the network an achievable task. Leveraging on these capabilities demonstrated by the research, QoS and QoE could easily be achieved with minimal effort but proper intuition and great traffic engineering skills.

7.2 Recommendations

SDN, coming with the benefits of customization, easy manageability and open source component building can be used to convert otherwise complex, proprietary of hardware specific network operations and procedures into every day devices both hardware and software to achieve the same goals of networking. Therefore, this research recommends the following:

i. Network engineers and administrators can learn how traditional routing is done, based on different metrics like latency, number of hops from the traditional networking equipment operations and transform the same knowledge into SDN and build the same applications in SDN.

ii. Routing of forwarding decisions on switches can be made more granular in SDN by looking at the subtle details applications and networks require in order to operate effectively and optimally. This has been enhanced by the tiptop customization of OVS

flow tables using OpenFlow messages that can be customized to the last degree because of the numerous options available in their command structure. This not only leads to immense customization of packet forwarding rules but also moving towards achieving adaptability, QoS and QoE in networks easily.

7.3 Suggestions for Future Research

The researcher managed to demonstrate how traditional network forwarding and traffic engineering techniques like MPLS could be achieved easily and cheaply in SDN without the use of any specialized equipment, but open source technologies. The researcher focused on only ICMP packets for the demonstration. However, this research has the following recommendations for future work:

i. The researcher recommends building of flow table messages on POX controller that can directly be fed in to different switches dynamically but running one command in the POX components folder to make it easier to manipulate the switches flow tables dynamically.

ii. The researcher also recommends that future work should study and implement a scenario where more than one packet type is isolated and given different routes at the same time MPLS being attached to these packets. The researcher was on course trying to achieve this but was short on time due to the complexity of understanding POX controller and OpenFlow advanced message structure

iii. In future, the researcher recommends that the TE process that requires analysis from network engineers to select the appropriate LSP for MPLS traffic should be done by software components. The researcher hopes that software should be developed to automatically select paths based on network statistics that is provided by the network nodes and use this data to create an appropriate LSP dynamically and populate flow tables as envisioned in recommendation (i).

7.4 Contributions

This research has made contribution to the field of SDN by coming up with a simple and effective way to implement a way of creating LSP and forwarding specific type of traffic using MPLS in SDN. The researcher was able to add to the literature already existing by creating a custom

topology and demonstrating that this routing procedure can be scaled regardless of the number of network nodes. The research has also illustrated that hardware components should no longer be a problem to network administrators and engineers whenever they want to come up with very complex and custom network applications because SDN has broken the glass ceiling and leveled the playing to allow for creativity and innovation.

References

Agarwal, K., Dixon, C., Rozner, E., & Carter, J. (2014). Shadow macs: Scalable label-switching for commodity ethernet. In *Proceedings of the third workshop on Hot topics in software defined networking* (pp. 157–162). ACM.

Ahn, G., & Chun, W. (2000). Design and implementation of MPLS network simulator supporting LDP and CR-LDP. In *Networks, 2000.(ICON 2000). Proceedings. IEEE International Conference on* (pp. 441–446). IEEE.

Akella, A. V., & Xiong, K. (2014). Quality of service (QoS)-guaranteed network resource allocation via software defined networking (SDN). In *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on* (pp. 7–13). IEEE.

Akyildiz, I. F., Lee, A., Wang, P., Luo, M., & Chou, W. (2014). A roadmap for traffic engineering in SDN-OpenFlow networks. *Computer Networks*, *71*, 1–30.

Bellessa, J. C. (2015). *Implementing MPLS with label switching in software-defined networks* (PhD Thesis).

Black, U. D. (2002). *MPLS and label switching networks*. Prentice Hall PTR.

Chen, S., Song, M., & Sahni, S. (2008). Two techniques for fast computation of constrained shortest paths. *IEEE/ACM Transactions on Networking (TON)*, *16*(1), 105–115.

Choy, L. T. (2014). The strengths and weaknesses of research methodology: Comparison and complimentary between qualitative and quantitative approaches. *IOSR Journal of Humanities and Social Science*, *19*(4), 99–104.

Csoma, A., Sonkoly, B., Csikor, L., Németh, F., Gulyas, A., Tavernier, W., & Sahhaf, S. (2014). ESCAPE: Extensible service chain prototyping environment using mininet, click, netconf and pox. In *ACM SIGCOMM Computer Communication Review* (Vol. 44, pp. 125–126). ACM.

Curran, J., Fenton, N., & Freedman, D. (2016). *Misunderstanding the internet*. Routledge.

Curtis, A. R., Kim, W., & Yalagandula, P. (2011). Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM, 2011 Proceedings IEEE* (pp. 1629–1637). IEEE.

Davie, B. S., & Rekhter, Y. (2000). *MPLS: technology and applications*. Morgan Kaufmann Publishers Inc.

Egilmez, H. E., Civanlar, S., & Tekalp, A. M. (2013). An optimization framework for QoS-enabled adaptive video streaming over OpenFlow networks. *IEEE Transactions on Multimedia*, *15*(3), 710–715.

Egilmez, H. E., Dane, S. T., Bagci, K. T., & Tekalp, A. M. (2012). OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks. In *Signal & Information processing association annual summit and conference (APSIPA ASC), 2012 Asia-Pacific* (pp. 1–8). IEEE.

Fang, L., Bita, N., Le Roux, J.-L., & Miles, J. (2005). Interprovider IP-MPLS services: requirements, implementations, and challenges. *IEEE Communications Magazine*, *43*(6), 119–128.

Frazier, H., & Johnson, H. (1999). Gigabit ethernet: From 100 to 1,000 mbps. *IEEE Internet Computing*, *3*(1), 24–31.

García-Dorado, J. L., Finamore, A., Mellia, M., Meo, M., & Munafo, M. (2012). Characterization of isp traffic: Trends, user habits, and access technology impact. *IEEE Transactions on Network and Service Management*, *9*(2), 142–155.

Gupta, M., Sommers, J., & Barford, P. (2013). Fast, accurate simulation for SDN prototyping. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* (pp. 31–36). ACM.

Hu, F. (2014). *Network Innovation through OpenFlow and SDN: Principles and Design*. CRC Press.

Installing POX — POX Manual Current documentation. (n.d.). Retrieved June 6, 2018, from https://noxrepo.github.io/pox-doc/html/

Jarschel, M., Wamser, F., Hohn, T., Zinner, T., & Tran-Gia, P. (2013). SDN-based application-aware networking on the example of youtube video streaming. In *Software Defined Networks (EWSDN), 2013 Second European Workshop on* (pp. 87–92). IEEE.

Kempf, J., Whyte, S., Ellithorpe, J., Kazemian, P., Haitjema, M., Beheshti, N., … Green, H. (2011). OpenFlow MPLS and the open source label switched router. In *Proceedings of the 23rd International Teletraffic Congress* (pp. 8–14). International Teletraffic Congress.

Kothari, C. R. (2004). *Research methodology: Methods and techniques*. New Age International.

Kumar, R., Hasan, M., Padhy, S., Evchenko, K., Piramanayagam, L., Mohan, S., & Bobba, R. B. (2017). Dependable end-to-end delay constraints for real-time systems using SDNs. *ArXiv Preprint ArXiv:1703.01641*.

Lara, A. (2015). *Using software-defined networking to improve campus, transport and future internet architectures* (PhD Thesis). The University of Nebraska-Lincoln.

Le Faucheur, F. (1998). IETF multiprotocol label switching (MPLS) architecture. In *ATM, 1998. ICATM-98., 1998 1st IEEE International Conference on* (pp. 6–15). IEEE.

Lee, H., Hwang, J., Kang, B., & Jun, K. (2000). End-to-end QoS architecture for VPNs: MPLS VPN deployment in a backbone network. In *Parallel Processing, 2000. Proceedings. 2000 International Workshops on* (pp. 479–483). IEEE.

Mammeri, Z. (2005). Framework for parameter mapping to provide end-to-end QoS guarantees in IntServ/DiffServ architectures. *Computer Communications*, *28*(9), 1074–1092.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., … Turner, J. (2008). OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review, 38*(2), 69–74.

Nadeau, T. D., & Gray, K. (2013). *SDN: Software Defined Networks: An Authoritative Review of Network Programmability Technologies*. O'Reilly Media, Inc.

Odom, W., & Cavanaugh, M. J. (2004). *Cisco QOS Exam Certification Guide (IP Telephony Self-Study)*. Pearson Education.

Open vSwitch Documentation — Open vSwitch 2.9.90 documentation. (n.d.). Retrieved June 6, 2018, from http://docs.openvswitch.org/en/latest/

openflow-tutorial: Openflow Tutorial on Mininet. (2018). Mininet. Retrieved from https://github.com/mininet/openflow-tutorial (Original work published 2013)

Ould-Brahim, H. (2012, February). Border gateway protocol procedures for multi-protocol label switching and layer-2 virtual private networks using Ethernet-based tunnels.

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems, 24*(3), 45–77.

Pfaff, B., Pettit, J., Koponen, T., Jackson, E. J., Zhou, A., Rajahalme, J., … Shelar, P. (2015). The Design and Implementation of Open vSwitch. In *NSDI* (pp. 117–130).

Psenak, P., Pillay-Esnault, P., & Rosen, E. C. (2006). OSPF as the Provider/Customer Edge Protocol for BGP/MPLS IP Virtual Private Networks (VPNs).

SDN Narmox Spear. (n.d.). Retrieved June 6, 2018, from http://demo.spear.narmox.com/app/?apiurl=demo#!/dashboard

Shalimov, A., Zuikov, D., Zimarina, D., Pashkov, V., & Smeliansky, R. (2013). Advanced study of SDN/OpenFlow controllers. In *Proceedings of the 9th central & eastern european software engineering conference in russia* (p. 1). ACM.

Sharafat, A. R., Das, S., Parulkar, G., & McKeown, N. (2011). Mpls-te and mpls vpns with openflow. *ACM SIGCOMM Computer Communication Review*, *41*(4), 452–453.

Singh, Y. K. (2006). *Fundamental of research methodology and statistics*. New Age International.

Software Defined Networking: OpenFlow Switches & Controllers - ppt download. (n.d.). Retrieved June 6, 2018, from http://slideplayer.com/slide/6204491/

The introduction to OVS architecture. (n.d.). Retrieved June 6, 2018, from http://hustcat.github.io/an-introduction-to-ovs-architecture/

Trestian, R., Muntean, G.-M., & Katrinis, K. (2013). MiceTrap: Scalable traffic engineering of datacenter mice flows using OpenFlow. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on* (pp. 904–907). IEEE.

Uchupala, P. (2015). Overseer: SDN-Assisted Bandwidth and Latency Aware Route Optimization based on Application Requirement.

Using OpenFlow — Open vSwitch 2.9.90 documentation. (n.d.). Retrieved June 6, 2018, from http://docs.openvswitch.org/en/latest/faq/openflow/

Vahid Sadri. (22:54:13 UTC). *OpenDayLight (ODL) Project*. Engineering. Retrieved from https://www.slideshare.net/VahidSadri/odl-44143211

Velrajan, S. (2013). Application-aware routing in software-defined networks. *Aricent Networks*.

Xiao, X. (2008). *Technical, commercial and regulatory challenges of QoS: An internet service model perspective*. Morgan Kaufmann.

Yu, T.-F., Wang, K., & Hsu, Y.-H. (2015). Adaptive routing for video streaming with QoS support over SDN networks. In *Information Networking (ICOIN), 2015 International Conference on* (pp. 318–323). IEEE.

Zhao, J., Hammad, E., Farraj, A., & Kundur, D. (2016). Network-aware QoS routing for smart grids using software defined networks. In *Smart City 360°* (pp. 384–394). Springer.

Appendix A: OpenFlow MPLS Messages Sent to S1, S4, S5 and S8

thesis_flows - Notepad

File Edit Format View Help

```
sudo ovs-ofctl -O OpenFlow13 add-flow s1 "table=0,in_port=1,eth_type=0x800,actions=goto_table:1"
sudo ovs-ofctl -O OpenFlow13 add-flow s1 "table=0,in_port=5,eth_type=0x8847,actions=goto_table:1"
sudo ovs-ofctl -O OpenFlow13 add-flow s1 "table=1,in_port=1,eth_type=0x800,actions=push_mpls:0x8847,set_field:20->mpls_label,output:5"
sudo ovs-ofctl -O OpenFlow13 add-flow s1 "table=1,in_port=5,eth_type=0x8847,mpls_bos=0,actions=pop_mpls:0x8847,resubmit(,1)"
sudo ovs-ofctl -O OpenFlow13 add-flow s1 "table=1,in_port=5,eth_type=0x8847,mpls_bos=1,actions=pop_mpls:0x800,output:1"

sudo ovs-ofctl -O OpenFlow13 add-flow s1 "table=0,in_port=1,eth_type=0x806,actions=output:5"
sudo ovs-ofctl -O OpenFlow13 add-flow s1 "table=0,in_port=5,eth_type=0x806,actions=output:1"


sudo ovs-ofctl -O OpenFlow13 add-flow s4 "table=0,in_port=1,eth_type=0x806,actions=output:2"
sudo ovs-ofctl -O OpenFlow13 add-flow s4 "table=0,in_port=2,eth_type=0x806,actions=output:1"
sudo ovs-ofctl -O OpenFlow13 add-flow s4 "table=0,in_port=1,eth_type=0x8847,actions=push_mpls:0x8847,set_field:23->mpls_label,push_mpls:0x8847,set_field:100->mpls_label,output:2"
sudo ovs-ofctl -O OpenFlow13 add-flow s4 "table=0,in_port=2,eth_type=0x8847,actions=push_mpls:0x8847,set_field:123->mpls_label,push_mpls:0x8847,set_field:110->mpls_label,output:1"

sudo ovs-ofctl -O OpenFlow13 add-flow s5 "table=0,in_port=1,eth_type=0x806,actions=output:2"
sudo ovs-ofctl -O OpenFlow13 add-flow s5 "table=0,in_port=2,eth_type=0x806,actions=output:1"
sudo ovs-ofctl -O OpenFlow13 add-flow s5 "table=0,in_port=1,eth_type=0x8847,actions=push_mpls:0x8847,set_field:28->mpls_label,push_mpls:0x8847,set_field:120->mpls_label,output:2"
sudo ovs-ofctl -O OpenFlow13 add-flow s5 "table=0,in_port=2,eth_type=0x8847,actions=push_mpls:0x8847,set_field:126->mpls_label,push_mpls:0x8847,set_field:115->mpls_label,output:1"


sudo ovs-ofctl -O OpenFlow13 add-flow s8 "table=0,in_port=4,eth_type=0x800,actions=goto_table:1"
sudo ovs-ofctl -O OpenFlow13 add-flow s8 "table=0,in_port=1,eth_type=0x8847,actions=goto_table:1"
sudo ovs-ofctl -O OpenFlow13 add-flow s8 "table=1,in_port=4,eth_type=0x800,actions=push_mpls:0x8847,set_field:35->mpls_label,output:1"
sudo ovs-ofctl -O OpenFlow13 add-flow s8 "table=1,in_port=1,eth_type=0x8847,mpls_bos=0,actions=pop_mpls:0x8847,resubmit(,1)"
sudo ovs-ofctl -O OpenFlow13 add-flow s8 "table=1,in_port=1,eth_type=0x8847,mpls_bos=1,actions=pop_mpls:0x800,output:4"

sudo ovs-ofctl -O OpenFlow13 add-flow s8 "table=0,in_port=1,eth_type=0x806,actions=output:4"
sudo ovs-ofctl -O OpenFlow13 add-flow s8 "table=0,in_port=4,eth_type=0x806,actions=output:1"


sudo ovs-vsctl set bridge s1  protocols=OpenFlow13
sudo ovs-vsctl set bridge s4  protocols=OpenFlow13
sudo ovs-vsctl set bridge s5  protocols=OpenFlow13
sudo ovs-vsctl set bridge s8  protocols=OpenFlow13
```

Appendix A shows the Openflow messages structured and sent to the different switches to MPLS.

Appendix B: Topology design python code

mininet@mininet-vm: ~/mininet/custom

```
"""Custom topology example

Two directly connected switches plus a host for each switch:

   host --- switch --- switch --- host

Adding the 'topos' dict with a key/value pair to generate our newly defined
topology enables one to pass in '--topo=mytopo' from the command line.
"""

from mininet.topo import Topo

class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self ):
        "Create custom topo."

        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        host1 = self.addHost( 'h1' )
        host2 = self.addHost( 'h2' )
        host3 = self.addHost( 'h3' )
        host4 = self.addHost( 'h4' )
        host5 = self.addHost( 'h5' )
        ingress_switch = self.addSwitch( 's1' )
        egress_switch  = self.addSwitch( 's8' )
        switch2 = self.addSwitch( 's2' )
        switch3 = self.addSwitch( 's3' )
        switch4 = self.addSwitch( 's4' )
        switch5 = self.addSwitch( 's5' )
        switch6 = self.addSwitch( 's6' )
        switch7 = self.addSwitch( 's7' )
        # Add links
        self.addLink( host1, ingress_switch )
        self.addLink( host2, ingress_switch )
        self.addLink( ingress_switch, switch2 )
        self.addLink( ingress_switch, switch3 )
        self.addLink( ingress_switch, switch4 )
        self.addLink( switch4, switch5 )
        self.addLink( switch3, switch6 )
        self.addLink( switch2, switch7 )
        self.addLink( switch5, egress_switch )
        self.addLink( switch6, egress_switch )
        self.addLink( switch7, egress_switch )
        self.addLink( http_service, egress_switch )
        self.addLink( udp_service, egress_switch )
        self.addLink( ftp_service, egress_switch )
topos = { 'mytopo': ( lambda: MyTopo() ) }

~
~
~
~
~
~
~
~
```

Appendix B shows the python code written by the researche to define the topology in Mininet VM.

Appenidx C: 12_Multi python code snipet adopted from Mininet inbuilt components to perfrom link disocvery

mininet@mininet-vm: ~/pox/pox/forwarding

```
# Copyright 2012-2013 James McCauley
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
A shortest-path forwarding application.

This is a standalone L2 switch that learns ethernet addresses
across the entire network and picks short paths between them.

You shouldn't really write an application this way -- you should
keep more state in the controller (that is, your flow tables),
and/or you should make your topology more static.  However, this
does (mostly) work. :)

Depends on openflow.discovery
Works with openflow.spanning_tree
"""

from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.revent import *
from pox.lib.recoco import Timer
from collections import defaultdict
from pox.openflow.discovery import Discovery
from pox.lib.util import dpid_to_str
import time

log = core.getLogger()

# Adjacency map.  [sw1][sw2] -> port from sw1 to sw2
adjacency = defaultdict(lambda:defaultdict(lambda:None))

# Switches we know of.  [dpid] -> Switch
switches = {}

# ethaddr -> (switch, port)
mac_map = {}

# [sw1][sw2] -> (distance, intermediate)
path_map = defaultdict(lambda:defaultdict(lambda:(None,None)))

# Waiting path.  (dpid,xid)->WaitingPath
waiting_paths = {}

# Time to not flood in seconds
FLOOD_HOLDDOWN = 5

# Flow timeouts
FLOW_IDLE_TIMEOUT = 10
FLOW_HARD_TIMEOUT = 30
```

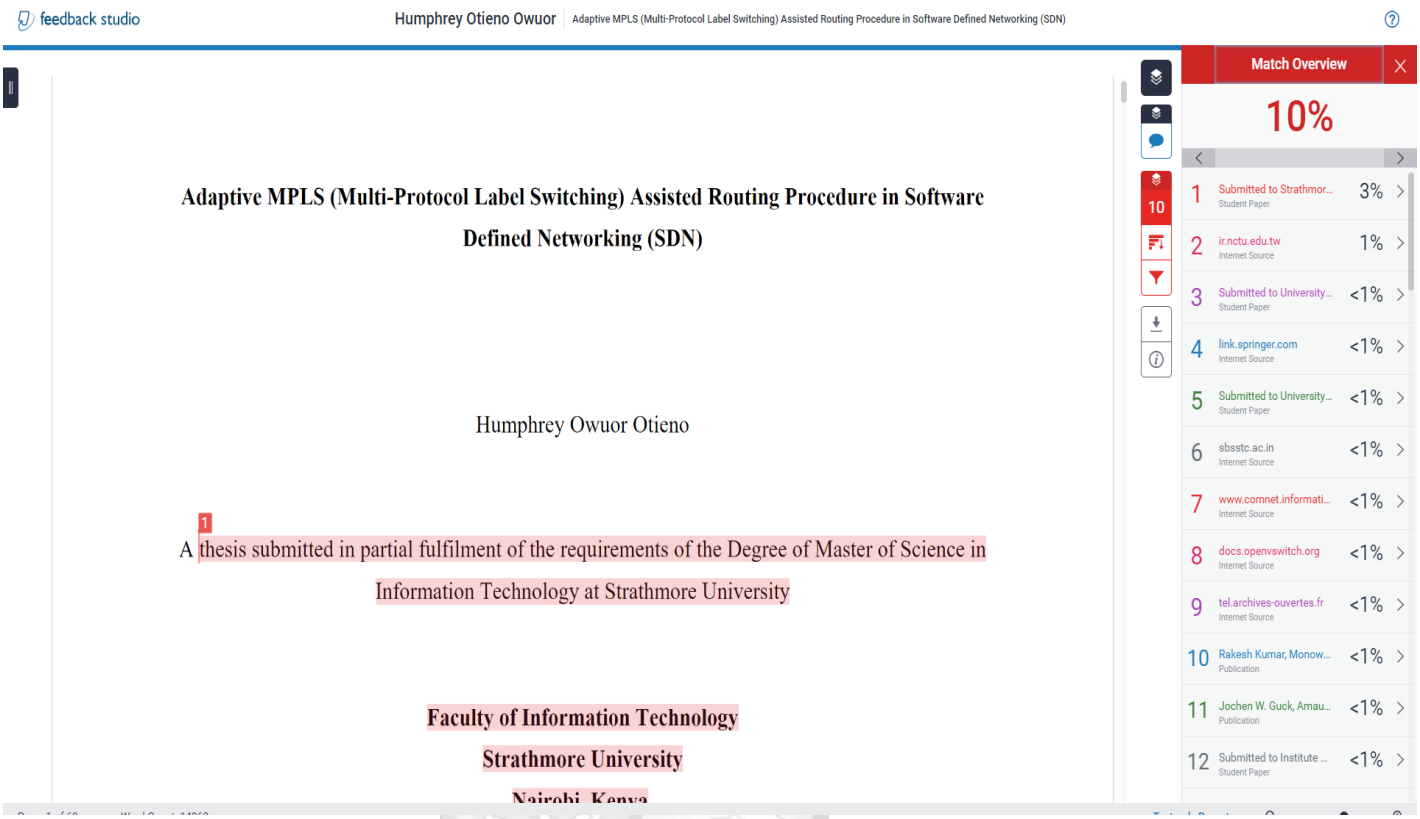
Appenidx D: 12_Multi python code snipet adopted from Mininet inbuilt components to perfrom link disocvery

mininet@mininet-vm: ~/pox/pox/forwarding

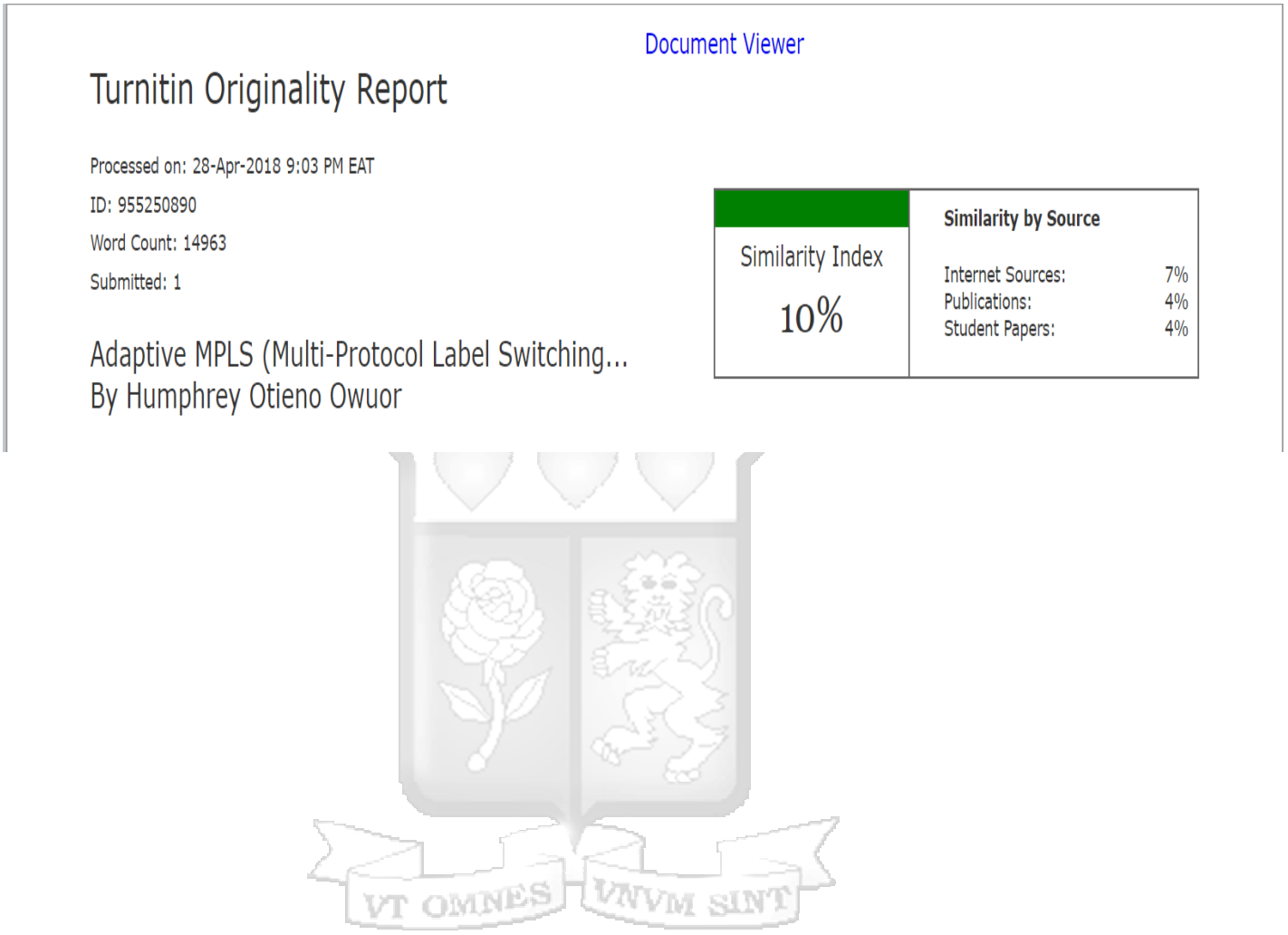
```
def _calc_paths ():
  """
  Essentially Floyd-Warshall algorithm
  """

  def dump ():
    for i in sws:
      for j in sws:
        a = path_map[i][j][0]
        #a = adjacency[i][j]
        if a is None: a = "*"
        print a,
      print

  sws = switches.values()
  path_map.clear()
  for k in sws:
    for j,port in adjacency[k].iteritems():
      if port is None: continue
      path_map[k][j] = (1,None)
    path_map[k][k] = (0,None) # distance, intermediate

  #dump()

  for k in sws:
    for i in sws:
      for j in sws:
        if path_map[i][k][0] is not None:
          if path_map[k][j][0] is not None:
            # i -> k -> j exists
            ikj_dist = path_map[i][k][0]+path_map[k][j][0]
            if path_map[i][j][0] is None or ikj_dist < path_map[i][j][0]:
              # i -> k -> j is better than existing
              path_map[i][j] = (ikj_dist, k)

  #print "--------------------"
  #dump()


def _get_raw_path (src, dst):
  """
  Get a raw path (just a list of nodes to traverse)
  """
  if len(path_map) == 0: _calc_paths()
  if src is dst:
    # We're here!
    return []
  if path_map[src][dst][0] is None:
    return None
  intermediate = path_map[src][dst][1]
  if intermediate is None:
    # Directly connected
    return []
  return _get_raw_path(src, intermediate) + [intermediate] + \
         _get_raw_path(intermediate, dst)


def _check_path (p):
  """
  Make sure that a path is actually a string of nodes with connected ports
```

Appendix C and D outline a snippet of the Mininet inbuilt python code that does LLDP

Appenix D: Originality Report with name



Appendix E: Originality Report with General Percentage



Document Viewer

Turnitin Originality Report

Processed on: 28-Apr-2018 9:03 PM EAT

ID: 955250890

Word Count: 14963

Submitted: 1

Adaptive MPLS (Multi-Protocol Label Switching...
By Humphrey Otieno Owuor

Similarity Index	Similarity by Source	
10%	Internet Sources:	7%
	Publications:	4%
	Student Papers:	4%