Building a Flexible and Inexpensive Multi-layer Switch for Software-Defined Networks

Submitted in fulfilment
of the requirements for the degree of

Master of Science

of Rhodes University

Tinashe Magwenzi

Grahamstown, South Africa
February 2019

Abstract

Software-Defined Networking (SDN) is a paradigm which enables the realisation of programmable network through the separation of the control logic from the forwarding functions. This separation is a departure from the traditional architecture. Much of the work done in SDN enabled devices has concentrated on higher end, high speed networks (10s GBit/s 100s GBit/s), rather than the relatively low bandwidth links (10s MBit/s to a few GBit/s) which are seen, for example, in South Africa.

As SDN is increasingly becoming more accepted, due to its advantages over the traditional networks, it has been adopted for industrial purposes such as networking in data centres and network providers. The demand for programmable networks is increasing but is limited by the ability of providers to upgrade their infrastructure. In addition, as access to the Internet has become less expensive, the use of Internet is increasing in academic institutions, NGOs, and small to medium enterprises.

This thesis details a means of building and managing a small scale Software-Defined Network using commodity hardware and open source tools. Core to the SDN Network illustrated in this thesis is the prototype of a multi-layer SDN switch. The proposed device is targeted to serve lower bandwidth communication (in relation to commercially produced high speed SDN-enabled devices). The performance of the prototype multi-layer switch had shown to achieve: data-rates of up to 99.998%, average latencies that are under 40μs during forwarding/switching and under 100μs during routing while using packet sizes between 64 bytes and 1518 bytes, and a jitter of less than 15μs during all tests.

This research explores in detail the design, development, and management of a multi-layer switch and its placement and integration in small scale SDN network. This includes testing of Layer 2 forwarding and Layer 3 routing, OpenFlow compliance testing, the management of the switch using created SDN applications, and real life network functionality such as forwarding, routing and VLAN networking to demonstrate its real world applicability.

Acknowledgements

I would like to give my deepest thanks the following: My parents, Mr and Mrs Magwenzi, my Aunt and Uncle, Mr and Mrs Mkwakwami for the support during my studies. I would like to take this time to also thank and appreciate my supervisors, Alfredo Terzoli and Mosiuoa Tsietsi for their continuous patience and guidance when I needed it. Most of all I would like to thank God for giving me this opportunity and making it possible for me to complete my studies. Lastly, to all who played a role in aiding me during this journey may God bless you all.

This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Technology and Human Resources for Industry Programme (THRIP) and CORIANT. The authors acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author(s) and that none of the above mentioned sponsors accept liability whatsoever in this regard.

Contents

1 Introduction **1**

1.1 Problem Statement 1

1.2 Objectives 2

1.3 Organisation of Thesis 3

2 Literature Review **5**

2.1 Introduction 5

2.2 Traditional Architecture 6

2.2.1 Overview 6

2.2.2 Architecture 8

2.2.3 Benefits 10

2.2.4 Limitations 11

2.2.5 The Need for Programmability in Networking 12

2.3 Software-Defined Networking 13

2.3.1 Overview 14

2.3.2 Architecture 14

2.3.3 Northbound APIs 16

2.3.3.1 REST API 16
2.3.3.2 OSGi Framework 16
2.3.4 Southbound APIs 17
2.3.4.1 ForCES Protocol 17
2.3.4.2 Network Configuration (NETCONF) Protocol 18
2.3.4.3 OpenFlow Protocol 19
2.3.5 Benefits of SDN 20
2.3.6 Limitations 21
2.4 A Case for OpenFlow 22
2.5 Summary 23

3 Open Source Software Tools 25

3.1 OpenFlow 25
3.1.1 Version Timelines 26
3.1.2 OpenFlow Switch 27
3.1.3 OpenFlow Pipeline 27
3.1.3.1 Flow Table 29
3.1.3.2 Matching 29
3.1.3.3 Table-Miss Entry 30
3.1.4 OpenFlow Channel and Control Channel 30
3.1.5 Group Table 31
3.2 OpenFlow Switch for Data Path 31
3.2.1 Table Lookup Mechanisms 32

3.2.2 OpenFlow Pipeline Processing Architecture . . . 33
3.2.2.1 General-purpose Processors . . . 33
3.2.2.2 Network Flow Processors . . . 34
3.2.2.3 Field Programmable Gate Arrays . . . 34
3.2.2.4 Application Specific Integrated Circuits . . . 34
3.2.3 Hardware, Software or Hybrid Table . . . 34
3.2.4 Evaluating Implementation Models . . . 35
3.3 Open Source Virtual Switches . . . 36
3.4 Open vSwitch . . . 37
3.4.1 Open vSwitch Architecture . . . 38
3.4.2 Open vSwitch Data Paths . . . 40
3.5 Open vSwitch Alternate Data Path . . . 40
3.5.1 Data Plane Development Kit . . . 41
3.5.2 Core Components . . . 41
3.5.3 Poll Mode Driver (PMD) . . . 43
3.6 OVS-DPDK . . . 44
3.7 SDN Controllers for the Control Plane . . . 45
3.7.1 OpenDaylight Framework . . . 45
3.7.2 Open Network Operating System (ONOS) . . . 46
3.7.3 Ryu Controller . . . 47
3.7.4 Faucet SDN Controller . . . 48
3.7.5 Overview . . . 49
3.8 Performance Metrics . . . 50

3.8.1 RFC 2455 Benchmarking Methodology 50
3.8.1.1 Test Conditions 50
3.8.1.2 Traffic Used 51
3.8.1.3 The Tests 51
3.8.2 iPerf Network Benchmark Tool 52
3.9 Summary 53

4 Designing the Overall SDN Network 54

4.1 Design Considerations 54
4.2 Design Objectives 55
4.3 Architecture of the SDN Network 56
4.3.1 Controller Selection 57
4.3.2 Configuration of the Controller 58
4.3.3 Basic Operating Principle 58
4.4 Network Flow Pipeline 59
4.4.1 Base Application 61
4.4.2 Forwarding Application 62
4.4.3 Routing Application 63
4.4.3.1 IP Addressing 63
4.4.3.2 Subnet Mask and Default Gateway 63
4.4.3.3 Address Resolution 63
4.4.3.4 Implementing the Router Logic 64
4.4.4 DHCP Service 65

4.4.4.1 Overview of DHCP Features 65
4.4.4.2 Address Assignment and Allocation Mechanisms 65
4.4.4.3 DHCP Process 66
4.4.4.4 Implementing DHCP in SDN 67
4.4.5 WebAPI Application 68
4.5 Summary 68

5 Designing the SDN Switch 70

5.1 Design Guidelines 70
5.2 Hardware Architecture 71
5.2.1 Network Card 72
5.2.2 General-purpose CPU 72
5.2.3 Main Memory 73
5.2.4 Motherboard 73
5.3 Software Architecture 74
5.4 Tools to Evaluate the Performance of the Switch 75
5.4.1 Benchmarking 76
5.4.2 Network Benchmarking 78
5.4.3 Compliance Testing 78
5.5 Summary 78

6 Setting the Stage for Testing **80**

6.1 Overview . . . 80

6.2 Multi-layer Switch . . . 80

6.3 Controller . . . 82

6.4 OpenFlow Protocol Version . . . 82

6.5 Controller and Switch Installation . . . 83

6.5.1 Controller Installation . . . 83

6.5.2 SDN Switch Installation . . . 83

6.5.2.1 Prerequisites . . . 83

6.5.2.2 Installation . . . 84

6.5.2.3 Configuration . . . 86

6.5.2.4 Validation . . . 87

6.5.3 Installation Overview . . . 88

6.5.4 OpenFlow Switch Performance Setup . . . 89

6.5.4.1 Tester Specification . . . 89

6.5.4.2 IO Test . . . 89

6.5.4.3 Layer 2 Switching . . . 90

6.5.4.4 Layer 3 Routing . . . 92

6.6 Ryu Controller and Applications . . . 94

6.6.1 Base Application . . . 95

6.6.2 Web File Server Application . . . 99

6.6.3 L2Switch Application . . . 101

6.6.4 Router Application . . . 102

6.6.5 DHCP Application 104
6.6.6 Running the Controller Software and Applications 105
6.7 Summary 105

7 Switch Benchmark Tests and Results 107
7.1 Tests General Structure 107
7.2 IO Forwarding Test 108
7.2.1 Results 110
7.2.2 IO Summary 111
7.3 Layer 2 Benchmarking 111
7.3.1 Results – Single Core 112
7.3.2 Analysis – Single Core 112
7.3.3 Results – Three Cores 113
7.3.4 Analysis – Three Cores 114
7.3.5 Layer 2 Benchmarking Summary 115
7.4 Layer 3 Benchmarking 116
7.4.1 Results 116
7.4.2 Layer 3 Benchmarking Summary 117
7.5 OpenFlow Compliance Testing 117
7.5.1 Compliance Results 117
7.5.2 Compliance Summary 118
7.6 Summary 119

8 Real Life Testing **120**

8.1 Overview 120

8.1.1 Policy Configuration 121

8.1.2 Specification of Hosts Used 121

8.2 Hub and Learning Switch Use Case 122

8.2.1 Configuration 123

8.2.1.1 Verifying Controller Settings 123

8.2.1.2 Verifying Hosts Settings 123

8.2.2 Results 124

8.2.3 Throughput Tests 127

8.2.4 L2Switch Summary 127

8.3 Routing Use Case 128

8.3.1 Configuration 128

8.3.1.1 Verifying Controller Settings 129

8.3.1.2 Verifying Hosts Settings 129

8.3.2 Result 130

8.3.3 Throughput Tests 134

8.3.4 Routing Summary 135

8.4 VLAN Use Case 137

8.4.1 Configuration 137

8.4.1.1 Verifying Controller Settings 138

8.4.1.2 Verifying Hosts Settings 140

8.4.2 Results 140

8.4.3 Throughput iPerf . . . 142
8.4.4 VLAN Summary . . . 142
8.5 Summary . . . 144

9 Conclusion **146**
9.1 Achieved Objectives . . . 146
9.1.1 Tools to Use for Developing an SDN Switch . . . 147
9.1.2 Performance of the Switch . . . 147
9.1.3 Simplified Structure for Network Applications . . . 148
9.1.4 Simplified the Process of Network Management . . . 148
9.2 Limitations . . . 148
9.3 Future work . . . 149
9.4 Summary . . . 149

Appendices **161**

A TestPMD Info **162**
A.1 OpenFlow Table Description . . . 163

B Faucet Compliance Tests **164**
B.1 Test Results . . . 164

C Web UI **170**
C.1 Full View of the web UI . . . 170
C.2 Configuration Toolbar . . . 172

D Use Case: Layer 2 **173**

D.1 IP Addresses 173

D.2 DHCP Logs During IP Allocation 174

E Use Case: Layer 3 **176**

E.1 Host Network Configurations 176

E.2 DHCP Logs 177

F Use Case: VLAN **180**

F.1 Host Port Configuration 180

F.2 VLAN Network Configurations for Host 2, Host 3 and Host 4. 181

F.3 Settled OpenFlow Table 181

List of Figures

2.1 A Traditional Computer Network. 7

2.2 Traditional Networking Planes. Source [1]. 8

2.3 Basic Scheme of an Network Device. Source [2]. 9

2.4 Components that makeup the SDN Architecture. Source [3]. 15

2.5 ForCES Architectural Diagram. Adapted from: [4]. 18

2.6 Router Configuration with Separate Blades. Adapted from: [4]. 19

2.7 Router Configuration with Separate Boxes. Adapted from: [4]. 19

2.8 NETCONF Protocol Layers. Source [5]. 20

3.1 Main Components of an OpenFlow Switch. Source: [6]. 26

3.2 OpenFlow Version Timeline. Source: [7]. 27

3.3 Packet Flow Through the Processing Pipeline. Source: [6]. 28

3.4 Performance and Programmability. Source: [8]. 32

3.5 Open vSwitch Architecture. Adapted from: [3]. 38

3.6 Open vSwitch Cache Hierarchy. Source: [9]. 40

3.7 Native OVS vs OVS with DPDK. Adapted from: [10]. 42

3.8 Core Components Architecture. Source: [11]. 43

3.9 OVS-DPDK three-tire cache architecture. Source: [12]. 44
3.10 Ryu Application Programming Model. Source: [13]. 48
3.11 Faucet Architecture. Source: [14]. 49

4.1 Generic Architecture Overview. 56
4.2 Pipeline for the Implementation of an SDN Network. 60
4.3 Configuring of SDN Switches Via the Web UI. 61
4.4 Layer 3/Routing Lookup. 64
4.5 DHCP Process. Source: [15]. 67
4.6 DHCP Flow Process in Proposed SDN Architecture. 68

5.1 SDN Prototype Switch Block Diagram. 71
5.2 Architecture with DPDK. Source: [16]. 74
5.3 OpenFlow Switch Benchmark Setup. 76
5.4 Hardware Switch Testing Setup. 79

6.1 Multi-layer Switch Interconnection Design. 81
6.2 CPU Partitioning (a), and Internal Structure (b) of the Prototype. 88
6.3 Switch Benchmark Setup. 89
6.4 Verifying L2 Behaviour Using Trace Tool. 91
6.5 Logical Connections Between Switch and TRex Traffic Generator. 92
6.6 Verifying L3 Behaviour Using Trace Tool. 94
6.7 Proposed SDN Network Design. 95
6.8 The Full View of The Web UI. 97
6.9 Configuration Toolbar Showing Settings for an OpenFlow Switch. 98

7.1 Multi-layer Switch Benchmark Setup. 108
7.2 Representation of an IPv4 packet. Source [17] 109
7.3 DPDK IO Benchmark Setup. 109
7.4 The IO Forwarding Rate of the Multi-layer Switch. Source [18]. 110
7.5 PMD Thread Utilisation on a Single Core for 64 Byte Frames. 112
7.6 PMD Thread Utilisation on Three Cores for 64 byte Frames. 113
7.7 Average Latencies for Single Core and Three Cores (Lower is better). 114
7.8 Maximum CPU Utilisation Per Core (lower is better). 115
7.9 Thread Utilisation for 64 Byte Frame. 116
7.10 Comparison Between Layer 2 and Layer 3 CPU utilisation. 117

8.1 SDN Network Architecture. 122
8.2 Routing Use Case Topology. 128
8.3 sFlow Monitoring Switch During iPerf Benchmarks for VLAN. 135
8.4 VLAN Use Case Topology. 137
8.5 sFlow Monitoring Source-Destination traffic for VLAN. 143
8.6 sFlow Monitoring Source VLANs. 144

C.1 The Full View of The Web UI. 171
C.2 Configuration Toolbar Showing Settings for an OpenFlow Switch. 172

List of Tables

3.1 Summary of Performance Characteristics 35

3.2 Overview of SDN Controllers 49

5.1 Exanple MAC and IP addresses to evaluate an OpenFlow switch. 76

5.2 Test Frames and OpenFlow Match Fields 77

5.3 OpenFlow Switch flow entry 77

6.1 Hardware & Software specifications for the Multi-layer Switch. 81

6.2 Hardware & Software Specifications for the Controller. 82

6.3 Software & Hardware components for TRex 90

6.4 Web URLs for REST Messages. 96

6.5 Events for the Web UI. 99

7.1 IO Forwarding Rate for the Multi-layer Switch. 110

7.2 IO Latency for the Multi-layer Switch. Source [18] 111

7.3 Layer 2 Forwarding Rate. 112

7.4 L2 Switching Latency, Jitter and CPU Utilisation for 3 Cores 114

7.5 Layer 3 Switching Latency, Jitter and CPU Utilisation for 3 Cores. 116

7.6 Summary Results for Compliance Testing. 118

8.1 Hosts Hardware & Software Specifications. 121

8.2 Cookie Grouping for VLAN 2 and 110 141

Chapter 1

Introduction

Software-Defined Networking (SDN) technology is an approach to network management that has gained popularity in recent years. The shift towards SDN has been brought by the need to configure network devices more flexibly and efficiently. The management of these network devices is made easier through APIs in software. SDN achieves this by separating the decision-making functions from the forwarding functions. In SDN, the decision function exists externally as a logically centralised controller in software, hence making changes to the software results in the manipulation of network behaviour. In an enterprise, network management is generally complex. SDN decouples the control and forwarding plane and abstracts the internals of a networking device from the administrator. The benefits include a simplified management, greater flexibility, and a reduction in operational costs. Within SDN, the control plane now operates on a general-purpose machine existing externally from the infrastructure.

1.1 Problem Statement

The architecture of SDN differs from the traditional network architecture by separating the control from the forwarding plane [19]. The result brings benefits to network applications such as load balancers and policy engines that reside at the top of the framework. SDN enables the control and management of tens, hundreds or even thousands of network devices from a locally centralised controller, manipulating how the traffic is forwarded [19]. This control software simplifies network management by fully inheriting the control functionality, that is to say that the central control of these devices realises the benefit of

making intelligent decisions based on a more global perspective, thus reducing down-time in the event of faults in individual devices [20].

There are a number of open protocols which enable SDN. Examples are OpenFlow, NETCONF and ForCES. These enable the development of highly interoperable networks. SDN-enabled devices are gradually becoming accepted for commercial networking and are mainly targeted for enterprise networks. However, these solutions are still extremely expensive (due to the level of packet switching that reach tens to hundreds Gigabit/s) and unavailable to most researchers in spite of being based on open standards such as OpenFlow [21, 22]. Furthermore, the openness of the standards coupled with availability of commodity/inexpensive computing hardware opens the opportunity for researchers to prototype cost effective SDN solutions on a reduced budget.

Similar existing solutions for developing inexpensive SDN networks are Mininet – which creates virtualised networks widely used in research institutions [23] and NetFPGAs – which can implement Ethernet switches or routers with high speed packet processing [24]. However, NetFPGAs are costly and require extensive knowledge in programming [25]. Other cost-effective implementations for small-scale architectures include the Raspberry-Pi [23] and Zodiac FX [26], but they offer speeds in order of hundreds of Megabits/s. However, there are new solutions that have switching speeds of a few Gigabits/s as seen by the Zodiac GX, which shows that there is a market for such devices [26].

This thesis explores the development of an inexpensive SDN solution using free and open software, utilising commodity hardware to come up with a system that is comparable to existing commercial solutions targeted to small-to-medium scale deployments.

1.2 Objectives

The focus behind this dissertation was to develop flexible inexpensive, small SDN network comprising both software and hardware tools to operate and monitor the network as seen in commercially deployed SDN networks.

The primary goals of this were to develop a device that is capable of forwarding or routing network packets between modules of the underlying network through the manipulation of software. This device should be able to receive commands from a software entity so that generic modifications to the network may be performed.

Secondary goals include the development of applications for this system which are intended to be simplified and to abstract low-level control functions of the underlying hardware while maintaining these functions to allow flexibility.

With the above goals considered, the method of investigation was set as follows:

1. Investigate potential architectures, for both hardware and software, which may be used as the underlying hardware of the system. This also includes best practices on components placement and inter-component signalling to confirm proper system design with optimal performance.
2. Determine the performance of the system by conducting a series of conformance test to evaluate system performance and functions. These tests would determine the behaviour of the system at wire speeds of 1 GBit/s on each port.
3. Determine a suitable structure for developing network applications that take advantage of the feature set of the selected hardware. To simplify development, the design approach including abstractions such as API calls to allow integration of generic languages such as Python or Java which would allow flexibility and aid in the development of new applications.
4. Develop a simplified interface for easy network management.

1.3 Organisation of Thesis

The work done for this dissertation is structured in nine chapters and a list of appendices with supporting material. The presented structure is as follows:

- **Chapter 2** introduces the necessary background information related to the technologies used. This includes the introduction and discussion to the architectures of traditional networks and Software-Defined Networks (SDN). This chapter provides an overview of networking functions such as routing and switching.
- **Chapter 3** describes open source software tools used in the development of the proposed system. This includes the architecture of the OpenFlow protocol and the details of the OpenFlow switch components (i.e. OpenFlow channel, OpenFlow controller, flow tables and the group table). It also describes technologies used

to implement the OpenFlow pipeline. The chapter also provides an overview of OpenFlow controllers and a brief review of their differences.

- **Chapter 4** describes the design of the multi-layer SDN switch including the network functionality and system behaviour.
- **Chapter 5** describes the implementation of the switch. Section 5.2 provides the commodity devices used in building the switch. Section 5.3 provides its internal software structure.
- **Chapter 6** sets the stage for testing the SDN switch and its integration into small SDN networks.
- **Chapter 7** presents the results of performance and compliance tests of the multi-layer switch.
- **Chapter 8** presents real scenarios for the deployment of that SDN switch. This includes core network functionality such as layer 2 switching, layer 3 routing, network isolating (VLANs) and network device monitoring.
- **Chapter 9** concludes, highlighting the results from previous chapters; limitations and suggestions for future work.

Chapter 2

Literature Review

2.1 Introduction

The idea of simplifying network management enables better operation and control over data travelling in a network. The manipulation and control of data travelling over a packet switched network has become the subject of much academic enquiry [27, 28]. Network devices that couple the control plane (functions that manage the behaviour of network components) and the forwarding plane (functions that forward the actual packets) have inherent difficulties. Networks that make use of these devices have become difficult to manage due to complexity [29]. The management of these traditional networks requires manual configuration of individual devices, thus introducing difficulties when adapting to dynamic conditions such as faults or changes in traffic behaviour. Modern network applications must adapt to these conditions while maximising efficiency and maintaining quality and performance.

SDN is a networking paradigm adopted by industry for the provisioning and maintenance of network infrastructure [30, 31]. SDN brings the flexibility of managing and operating networks through software and it delivers an alternate architectural design from conventional computer network architecture [32]. Moreover, SDN proposes an open form of network management, control and forwarding. SDN brings about the decoupling of control and forwarding functions, where the network control functions or decision making is not embedded in each network device. Furthermore, the control of these network devices is logically centralised, allowing the orchestration of network operations over a span of network devices.

SDN promises to improve on most of the limitations that are currently faced in the traditional architecture [33]. Computer networks today consist of heterogeneous devices. These devices (e.g. switches, routers, middle-boxes) from multiple vendors and are controlled by sets of distributed and refined protocols or protocol suites to ensure that information is successfully and efficiently conveyed within a network. Traditionally, these devices form networks that are 'static' because of the tight coupling of the control logic and the forwarding functions within the network device. The firmware of these devices consists of proprietary software and has made it difficult for operators to innovate and specify high-level policies [34].

This chapter reviews the traditional network architecture and introduces the architecture of SDN and the key aspects behind its functionality. This chapter will also present different protocols that enable SDN, focusing on its enablement through open protocols.

2.2 Traditional Architecture

Within a traditional computer network, data pass between two or more entities or network nodes such that information can be exchanged. This data, as individual packets, travels from node to node. A network consists of connected nodes. Consequently each node performs packet buffering, packet scheduling, header modification, and forwarding. Information is forwarded until it has reached its destination. Devices that form these networks consists mainly of routers and switches which perform switching and routing. These functions are defined and explained in detail next.

2.2.1 Overview

A computer network usually consists of at least two connected computer devices to allow the exchange of information or the sharing of resources such as disk storage, printing services and so forth. In general, computer networks comprise a collection of devices often called nodes. These nodes connect with each other via different forms of media such as copper wires, wireless, or optical fibres. Modern networks transmit information through networks in packet-switched mode. Packet-switching is where data packets are individually routed and forwarded between nodes within the network. Nodes send digital information as a group of packets where each packet carries information in the header that allows appropriate processing of these packets. Nodes can be classified as end-hosts and

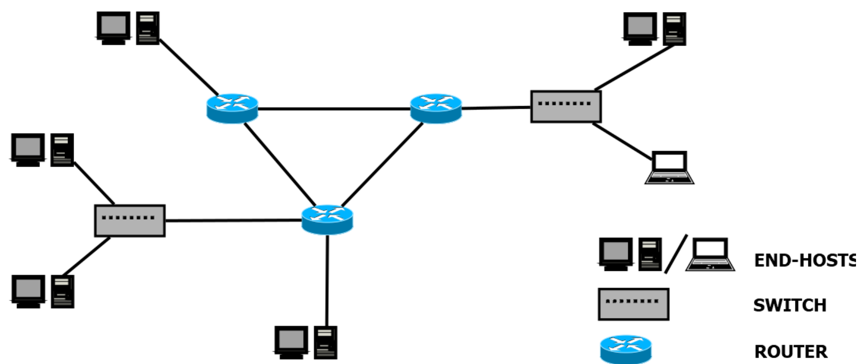


Figure 2.1: A Traditional Computer Network.

intermediary hosts. End-hosts communicate information with each other by transmitting it as packets over a network while intermediary hosts make up the network infrastructure. Furthermore, intermediary hosts offer network resources and network services to end-hosts. Figure 2.1 shows an example network with end-hosts that share information and intermediary hosts (routers and switches) that mediate data in the network.

Information which is sent as packets is sent from the sender (source) to the receiver (destination). The task of transmitting information within a network is typically broken down into two functions which are routing and forwarding. The task of routing decides, with respect to other connected intermediary hosts within the network, based on two criteria. (1) the state of the interconnections between these nodes, which may also include other factors such as the link connection, and (2) to decide within each network device the route/path in which a packet should traverse to achieve the goal of forwarding packets. The task of routing within the nodes happens continuously as each node continually updates the state of the network which could change at any instant due to the node failures, failures in the connections between the nodes, or high congestion due to unforeseen traffic patterns. Generally, routing involves complex algorithms which determine paths that packets take from source to destination.

The second function of forwarding differs from routing since it is much simpler. Within each node, each packet will have a defined rule that contains labels and information that is used to interpret the packet. The link between the routing and forwarding performed within a node is realised as a collection of data structures called the Forwarding Information Base (FIB) which is in each intermediary host. The routing process maintains the state of the FIB while the forwarding process makes use of the FIB to determine how to forward each packet.

The traditional network architecture contains three planes [1] which are: the management

plane, the control plane, and the data plane (forwarding logic). The management plane is responsible for the control and management of the node. Here network administrators are able to monitor and configure the device through common mechanisms such as the command-line interface (CLI), making use of protocols to manage and monitor network devices such as SNMP (Simple Network Management Protocol), and more. The control plane is responsible for learning and building awareness of the network. In addition, the gathered information is used by the node to determine the egress port to forward traffic. The forwarding logic or data plane handles the majority of these packets. The data plane comprises of ports which receive and transmit packets between ingress and egress ports. The data plane is responsible for the forwarding of packets from the ingress port to the egress port. It will forward these packets based on sets of rules built by the control plane for packets travelling through the network. These rules are saved or are laid out in the FIB. When a packet arrives at the device's ingress port, the data plane forwards the packet to the next node until it is received at the next end-host. Figure 2.2 shows the relationship between architectural layers among multiple network nodes.

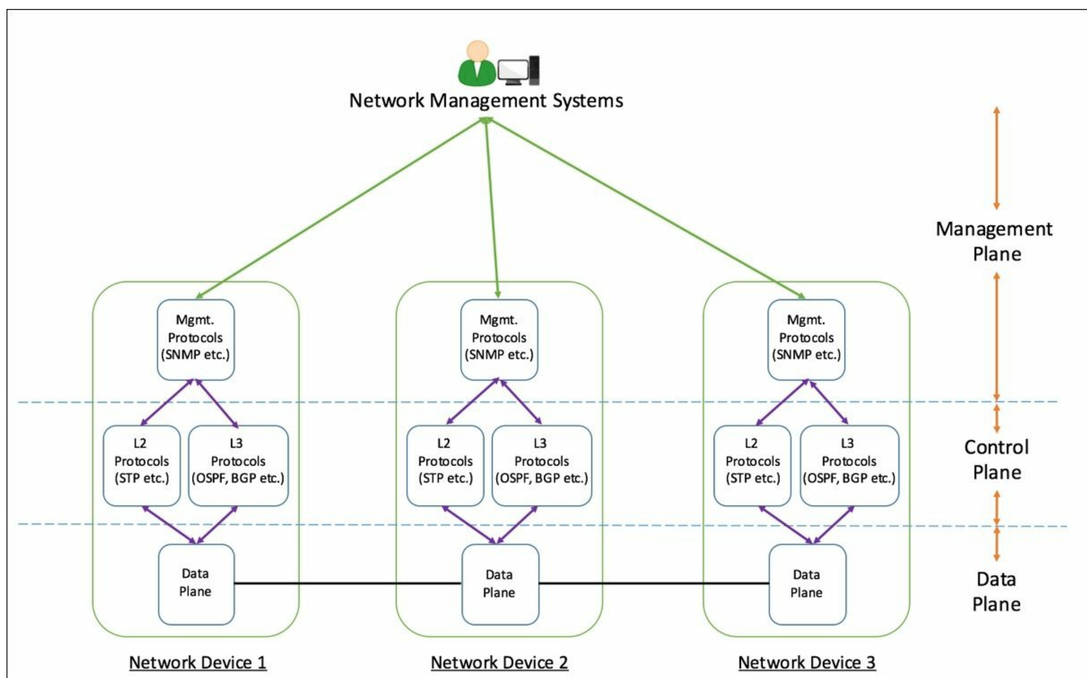


Figure 2.2: Traditional Networking Planes. Source [1].

2.2.2 Architecture

Not all packets are processed in the data plane or forwarding logic. In contrast, information on handling each packet has not been set or a rule is not found in the FIB for

processing a particular type of packet. When, for example, the destination of an arriving packet is unknown, the data plane forwards this packet to the control plane where the details of the packet are learned, processed, and later forwarded. Similarly, control traffic such as routing protocol messages are also forwarded to the control plane. The control plane makes use of control protocols to determine how packets are to be processed. Hence, the control plane is responsible for making the decision of how a packet is to be treated based on the information found in the packet header. Within the control plane, a node stores the structure of the network topology as a data set called the Routing Information Base (RIB). The RIB is maintained by the exchange of information between other control planes within the network. This control logic will then update the FIB, once the RIB is stable (i.e. once the node has developed a view of the network topology), with the rules to follow such that similar packets are processed the same way in future. Nevertheless, the data plane alone handles a high volume of packets. Figure 2.3 illustrates the basic scheme inside a network device.

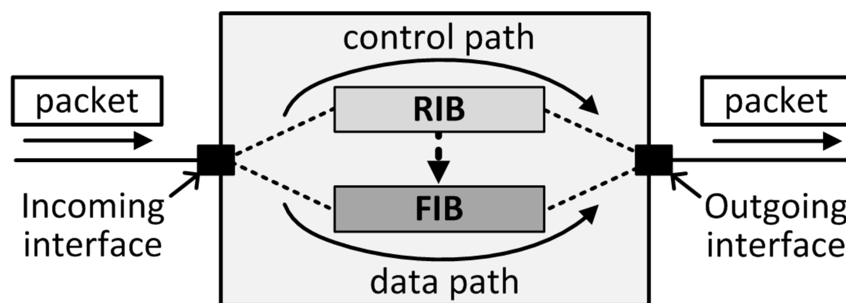


Figure 2.3: Basic Scheme of an Network Device. Source [2].

A network can be of any size, from two devices to thousand or even more. When networks become large, they are usually subdivided into smaller portions called subnetworks (subnets). Usually, these subnets consist of machines that are linked at layer 2 of the OSI Model using technologies like Ethernet, while the linking between these subnets is at layer 3 forming the larger network [35]. The OSI model consists of seven stacked layers that implement protocols to define the networking framework [35]. Depending on either layer 2 (communication between nodes in the same subnet) or layer 3 (communication between nodes in the different subnets) the function of the control layer is dependent on the location of the source and destination nodes. The control plane functions include either a layer 2 switch or layer 3 router.

The function of routing and forwarding is separated within each node to make optimisation possible between each tasks. The demand for high-speed packet processing is achieved through special techniques for forwarding packets. As an example, the requirement for

forwarding each packet on a 10 Gbps port requires on average, a packet processing time of 10 nanoseconds. The forwarding table(s) is normally maintained in a Dynamic Random Access Memory (DRAM memory) [36]. The DRAM lookup of information is expensive and as such, it is generally excluded. The result is the use of special hardware memories and data structures within the forwarding engines are used to optimise the number of memory accesses required to look for an action for a packet.

The control plane applies the logic defined in control protocols (e.g. routing protocols like STP, RIP, OSPF, BGP) that then updates the FIB. Since each networking node internally has three planes, each device is isolated creating distributed architecture. These networking nodes, i.e. switches, routers and middleware, perform layer 2 – layer 7 switching, where the control plane of one node communicates with control planes of neighbouring nodes to maintain its RIB (for routing) and the FIB. The exchange of this information occurs when these nodes broadcast data about connected/neighbouring hosts to the rest of the network. The control plane will use this information to decide which path to follow. The FIB may also be interacted with via the management plane where the administrator may define the other rules like VLANs, ACLs or forwarding rules.

Examples of information that is announced are: broadcasting the state of the node to all network nodes that are directly connected, or broadcasting the addresses of all directly connected end-hosts. The information is used by each network node to calculate individually the desired routes. The state often can change; hence, network nodes periodically announce their updated local state and receive updates from other nodes.

2.2.3 Benefits

The architecture of a traditional network consists of control and data plane. The advantages seen in this architecture are as follows:

- **Resilience:** There is one instance of the control plane per device and this makes it highly resilient to failure, i.e. no single point of failure.
- **No round-trip latency:** Since each device hosts a control plane, there is a minimal delay when the data plane request policy directions from the control plane on how each arriving should be handled.

- **Less software maintenance:** traditional devices require less software maintenance as vendors maintain the firmware that run these devices with the aim to provide superior user experience. Hence, network administrators have less to focus on.
- **Effective decision-making process:** Each network node uses routing and switching built in hardware by vendors that deliver high bandwidth end to end.

2.2.4 Limitations

Traditional network devices can be described as plug-and-play which is designed to deliver packets from end to end if possible [37]. However, this architectural approach has limitations. The networking devices consist of a layered architecture where each device decides how each packet is to be forwarded. The nature of these traditional network devices introduces limitations and complexity that are listed next:

- **Tight Coupling:** Not only are the control and data planes distributed but they are also developed and managed using hardware and software that is tightly integrated.
- **Lack of Openness:** The internal components are proprietary, which results in network devices whose internal hardware and software include proprietary firmware from vendors. The network devices developed are often based on the same hardware family [36]. The interdependencies created by the lack of openness creates limitations such as limited innovation, complex management, cost of maintenance, stability issues, and scalability.
- **Management:** The environments in which network technologies operate require them to meet the demands of high availability, security and efficient delivery of information [38]. Protocols have been designed to solve specific issues and have been included as part of the functionality of these network technologies. This design approach has led to complex network management and has become a burden where several parts of the network require reconfiguration. The addition or removal of devices may require, for example, reconfiguration of access lists, VLAN, quality of service policies, routing protocols and as such, network administrators rather keep networks static. Large networks generally have devices from multiple vendors and the management using APIs (Application Program Interfaces) or CLI which makes the management of multi-vendor networks very cumbersome.

- **Virtualisation (Slicing / Traffic Isolation):** Business regulations and corporate policies sometimes require the isolation of traffic in the network. Traditional solutions use MPLS or Virtual Routing and Forwarding (VRF) – VRF-Lite to create logic slices and managing and deploying of these technologies is cumbersome and time-consuming [39].

- **Scalability:** In reality, these networks are complex to manage because of multiple devices that require manual configuration. Limitations arise when scaling these networks due to the difficulty in being able to predict traffic behaviour and provisioning capacity to meet variable demand.

- **Innovation:** The internal infrastructure, as mentioned, is proprietary and researchers have no way of accessing the data forwarding functionality in the data plane. Innovation is limited by the functions provided by the vendors and may not meet the need particularly in the features developed.

- **Cost:** Cost has two components of capital expenditure (CAPEX) and operational expenditure (OPEX). The OPEX of managing a large network is of concern due to the required resources. Example of cost includes the required labour to locate issues within an already complex environment, which requires specialised and experienced personnel. The CAPEX is normally high due to the proprietary software included by vendors.

2.2.5 The Need for Programmability in Networking

Traditional networks are static and innovation is limited to the vendor's product release cycles that may take years and may not meet the demand for easier management for the networks. Multiple devices from multiple vendors implement their own standard way of interfacing to the management plane of the device. Having a standardised interface, i.e. abstracting the forwarding plane from the control, will realise programmability that will bring about new innovative ideas.

The virtualisation through viewing the global view of the network can benefit better use of network resources and allow the easy manipulation of how data flows through *logical topologies* regardless of the physical design. The ability to control and manipulate the behaviour of the network through centralising (physically or logically) the control will enable easier management and centralisation allows changing the behaviour of the entire network with just one command.

The reduction of operating costs requires less time to implement the policies. A standard API will enable compatibility between devices from different vendors. This will allow monitoring of the infrastructure by calling these APIs to get the status of the network device. Modulation enables coherence between policies and allows the dynamic application of policies that will enable dynamic resource allocation.

2.3 Software-Defined Networking

The need for programmable networks has influenced the growth of SDN [34, 40]. The definition from the Open Network Foundation states that SDN is the separation of the control plane from the data plane, where the control of infrastructure is centralised through the management of one or more network devices [41]. SDN is largely based on open standards [41] which is a major characteristic difference from the traditional network architecture. SDN also differs from traditional networks that control is directly programmable. The programmability of the network is due to the decoupling of the network control logic from the forwarding functions. This separates the control of the network from the application perspective. Because of this abstracted control, network administrators have the ability to control network traffic dynamically.

The term Software-Defined in this context means that the high-level processes that make use of network resources are able to interact with the network nodes. These high-level processes are able to request information from the network. Furthermore, this would also encourage new network management methods (e.g. routing algorithms) to be developed and deployed more rapidly without the limitations of expense and slow standardisation often seen in traditional networks [40]. Hence, the SDN standard will speed up the rate of innovation within modern computer networks.

SDN is also able to support a variety of functions beyond layer 2 forwarding or layer 3 routing. For example, access control and traffic monitoring. SDN will also give the possibility of simplifying network management and improve network utilisation. Thus, decrease operating costs. SDN enables the control of network devices to be directly programmable. It promises to achieve dynamic, cost-effective, adaptable management to enable visualisation, monitoring and debugging of the network resources, and much more [20, 21, 42, 43].

2.3.1 Overview

The intention behind the SDN paradigm is to bring about the adoption of a standard that enables the 'forwarding abstraction' for the FIB. This is to allow the configuration of the FIB and the interaction with the forwarding node from higher-level processes. Through the forwarding abstraction, the added flexibility of network control enables easier implementation of a variety of algorithms. Moreover, the exposing of the functionality of the devices via open interfaces will not only allow the forwarding of packets based on layer 2 destination address but will also allow multi-layer forwarding. Furthermore, applications and services are able to forward packets as source addresses as well.

2.3.2 Architecture

The architecture of SDN consists of three planes: management, control, and data [39]. A software controller (SDN controller) abstracts the behaviour of the network device through open API interfaces. The control logic is removed from the infrastructure whereas the end devices only perform data forwarding. The SDN controller is thus, responsible for maintenance of the infrastructure. It interacts with the infrastructure through the southbound interface. The southbound interface, explained in greater detail later in Section 2.3.4, separates the control plane from the data plane as a result of, the control plane becomes an independent software platform. This allows software development without the need to engage low-level details of independent devices. This allows developers to focus on the behaviour of the network as means of controlling this software. Consequently, the software can be developed independently of hardware.

The level of control will fulfil the requirements or expectations of the changing requirements mentioned earlier in Section 2.2.4. The components that form the architecture of SDN include the SDN controller, network nodes and SDN applications, as shown in Figure 2.4.

The data plane comprises network nodes that take the role of forwarding the traffic i.e. carries the traffic and transmits packets to the next node.

The control plane consists of an SDN controller – a system that makes the decision as on where to and how the network traffic is directed within the data plane. The model of SDN would centralise the control plane as a software entity i.e. the SDN controller. Within the control plane, the controller receives the complex high-level policies from the

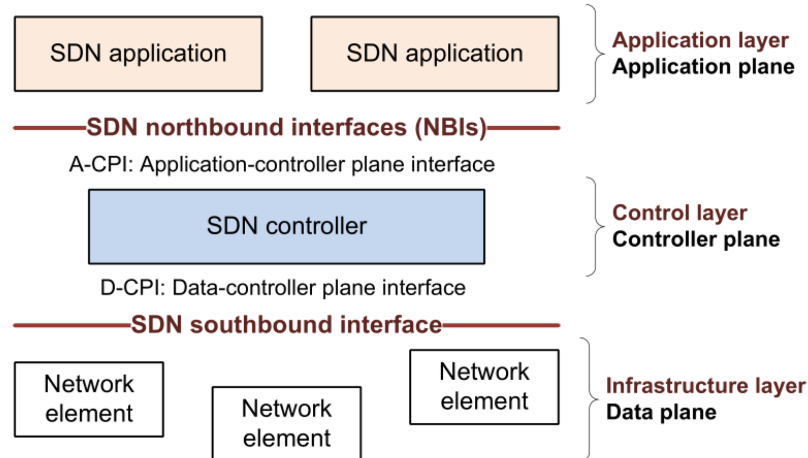


Figure 2.4: Components that makeup the SDN Architecture. Source [3].

management plane, translates them to low-level rules and then transmits them to the data plane. The controller will dynamically update these rules according to new policies, network events or alerts.

The management plane interacts with the control plane which supports direct communication with the controller. SDN applications communicate their requirements and desired network behaviour to the SDN controller. These applications may also access the abstracted view of the network for internal decision-making purposes or to perform policies such as monitoring, quality of service enforcement, load balancing, etc.

The three planes in the SDN architecture are divided by two interfaces: Northbound Interface (NBI) and Southbound Interface (SBI). The control plane, separated from the hardware, is implemented in software as the controller. Thus, communication between the three planes is done through the two interfaces. The SDN controller has access to the data plane through the SBI where it can manage multiple devices. The SDN controller also exposes the data plane configurations to the management plane via the NBI allowing SDN applications to configure the SDN controller and monitor the data plane.

SDN purports to institute an industry-wide interface standard that may realise "forwarding abstraction" for the data plane. This architecture enables the control or management of the network to be automated and programmed on standard x86 server machines using a standard interface that controls the network nodes. Standardisation also brings forth innovation in networks. SDN allows access to the forwarding of the networking node, hierarchical arrangement of arbitrary planes, and the increase of the control which also has the advantages of security. SDN actively reduces the congestion within the network

through load balancing which effectively improves the management of network resources within the network [20].

2.3.3 Northbound APIs

A Northbound API provides the configuration and management services for SDN applications. These then use this information to specify the required network control and are represented in the form of policies that support device abstractions. This API is implemented as an open, vendor-neutral interface.

Commonly used NBIs are the REST API and the OSGi model [44].

2.3.3.1 REST API

The REST API is a software architecture that was first introduced by Roy Fielding in his PhD thesis [45]. It uses a Client-Server Model where the client or server can be developed separately without depending on the other. A fundamental concept of REST is a resource. This can be information that can manipulated or accessed. The information or states is represented using common formats such as Extensible Markup Language (XML) or JavaScript Object Notation (JSON). Servers share the state of an application with one or more clients. The server does not maintain the state of the application. As such, the client will retrieve representations (views) of resources via a URL where each resource will have a unique URI to identify them. REST makes use of 'create', 'read', 'update', 'delete' (CRUD) operations to manage the resources [46]. Within an SDN environment, resources are managed as data types. Examples are controller node, firewall rules, the configuration of the network, switch, port, link, flow entry, VLAN, etc.

2.3.3.2 OSGi Framework

The OSGi framework [47] includes a set of specifications for dynamic application composition that make use of reusable Java components called bundles. Each bundle publishes its services and applications use services provided by other bundles. Bundles may be installed, started, stopped, updated, or removed on the fly without the need for restarting the OSGi services registry, i.e. will not have the need to reset the controller after updating applications. This framework allows applications to be able to use resources from other

applications. Furthermore, it allows applications to load during runtime in a dynamic fashion.

2.3.4 Southbound APIs

SDN can be implemented through various protocols. Examples of open source protocols include ForCES, Netconfig and OpenFlow. One of the goals for SDN is to bring about the standardisation of southbound protocols. This is to enable consistency of functionality between devices from multiple vendors within an SDN environment.

2.3.4.1 ForCES Protocol

Forwarding and Control Element Separation (ForCES) [4] is defined by a framework and related protocols to standardise information exchange between the control plane and forwarding plane. The framework defined in [48] presents an architecture of logical components. These logical components include Control (CE), Forwarding Elements (FE), Control Element Manager (CEM) and Forwarding Element Manager (FEM) [48]. Figure 2.5 shows the architecture.

- **Control Elements** is a logical entity that uses ForCES protocol to interact with one or more FEs. Its functions include execution of control and signalling protocols. Each CE may interact with one or more FE.
- **Forwarding Elements** these provide per-packet processing and handling as instructed by one or more CEs. The CE control the FEs using the ForCES protocol.
- **Control Element Manager** is responsible for managing generic tasks for CE. It particularly determines which FE(s) should communicate with a CE through a process called FE discovery and may involve the manager to learn the capabilities of the FEs.
- **Forwarding Element Manager** is responsible for managing generic tasks for FE. It determines which CE(s) and an FE should communicate.

There are two ways the CE and FE are physically separated: blade level (shown in Figure 2.6) and box level (shown in Figure 2.7) [48]. In blade level, the proprietary

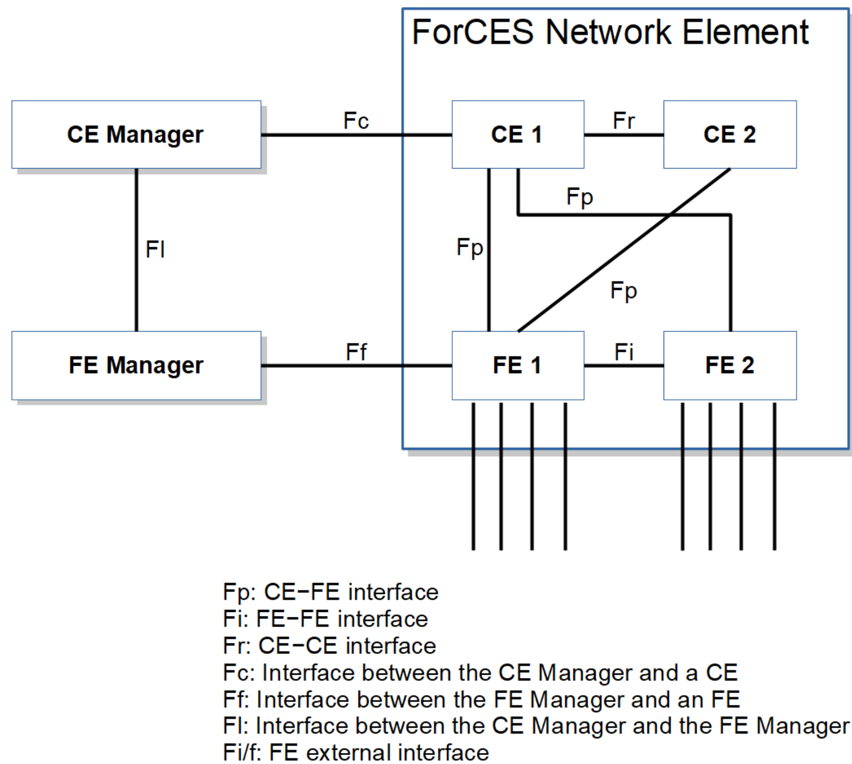


Figure 2.5: ForCES Architectural Diagram. Adapted from: [4].

protocol between the CE and FE within a single network device/element is replaced by the ForCES protocol standard. In the box level, the CE and FE exist as two separate devices and interface through the ForCES standard. ForCES enables SDN through the separation of the control and forwarding data planes of various network devices, such as IP routers, switches and firewalls.

2.3.4.2 Network Configuration (NETCONF) Protocol

The NETCONF protocol [5], defines simple mechanisms for how a network device is managed, how the configuration information is retrieved and how the configuration data is updated or changed. It is a network management protocol standardised by the IETF. NETCONF uses Remote Procedure Calls (RPCs) to establish secure communication between a client (either a script or application) and a server (a network device). The client can send a series RPC messages to the server/network device, and in turn can receive a series of corresponding RPC response messages. The client can discover the capabilities of the network device and is permitted to alter the behaviour and features exposed by the device. The RPC messages are encoded in XML for data and protocol messages. Figure 2.8 shows four layers of the partitioned entities which conceptualise NETCONF.

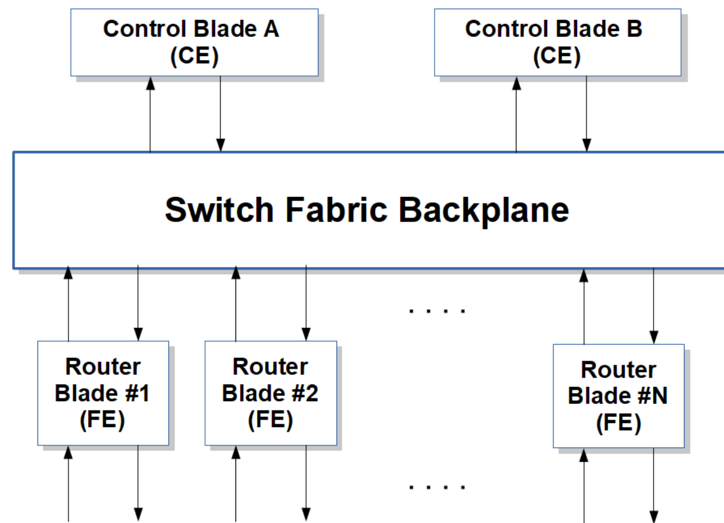


Figure 2.6: Router Configuration with Separate Blades. Adapted from: [4].

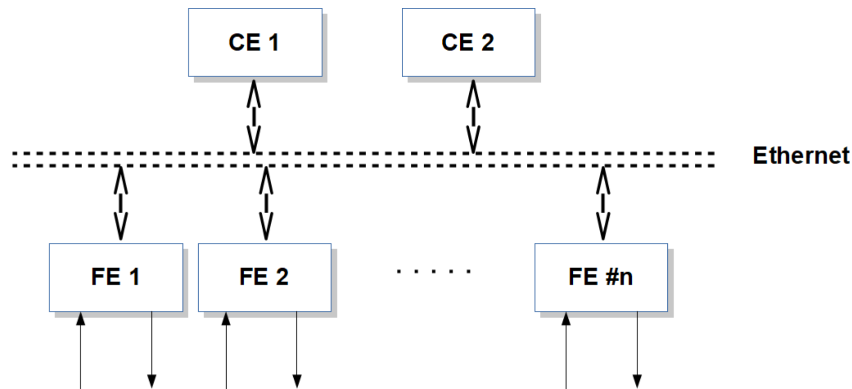


Figure 2.7: Router Configuration with Separate Boxes. Adapted from: [4].

- **Secure Transport** provides secure communication path between client and server.
- **Messages layer** provides simplified framing mechanism for encoding RPCs and notifications.
- **Operations layer** defines a set of protocol operations.
- **Content layer** includes the configuration and notification data. it is expected that standardising of NETCONF data is expected to transpire [5].

2.3.4.3 OpenFlow Protocol

OpenFlow [41] was created at Stanford University and is now maintained by a non-profit consortium, the Open Network Foundation (ONF). ONF's goal is to promote the

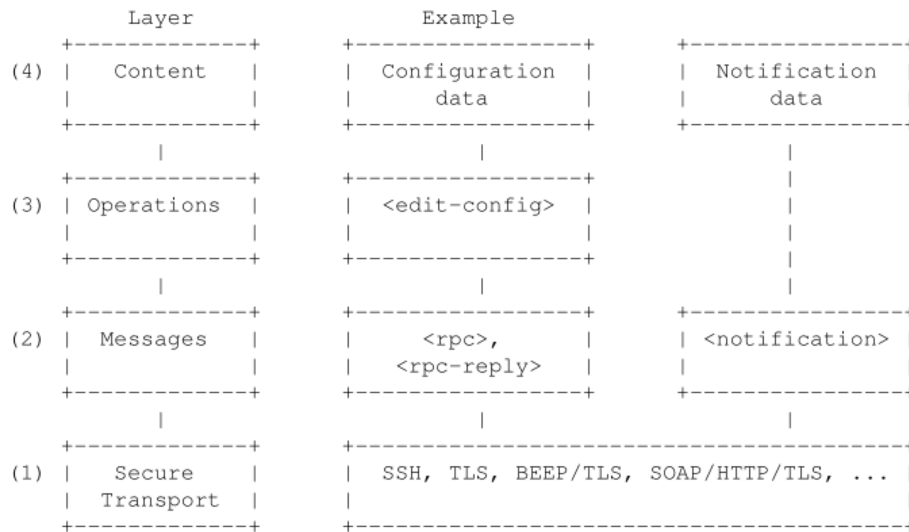


Figure 2.8: NETCONF Protocol Layers. Source [5].

transformation of network infrastructure and carrier business models [49].

OpenFlow is a standardised protocol for SDN to handle SBI communication between an SDN controller and an SDN switch. Using the OpenFlow protocol, a controller can discover the details, status, and the network topology of the switch; while controlling the forwarding behaviour by providing flow rules with defined actions in the switch's flow table(s). A flow table is similar to how traditional network devices use their FIB. The OpenFlow controller is logically centralised. As a result, there is no need to physically configure each OpenFlow switch since the control plane is decoupled from the forwarding plane. OpenFlow uses the OpenFlow channel as the interface between the control plane and the data plane plane. The application plane comprises applications separated from the underlying forwarding infrastructure that handle business policies. The controller consists of centralised intelligence that simplifies, optimises, enables greater control, and enforces policy management across all network devices.

2.3.5 Benefits of SDN

SDN brings forth changes to the way that networks are built and operate, and affects how these networks accomplish business requirements. SDN will allow networks to use non-proprietary open standards. With SDN the programming and management of networks will become easier which will offer its users greater control over networks and operators may tailor network functionality to optimise network utilisation. Thus, reduce the overall cost of operating the network. Other benefits include:

- **Simplify network management** – The SDN model of control centralisation will allow the network to be configured and monitored from a single node. This would allow easier management of complicated default network configuration and better modelling or abstraction of network functionality. The management software now deals only with a centralised controller which will allow more robustness of programmatic interfaces since the complexity has been hidden from the management.

- **Fast service deployment** – The development of new features and applications are deployed faster due to a simplified way of network management.

- **Automated configuration** – The SDN controller allows previously manual configuration tasks such as VLAN assigning and QoS configuration to be automated. The benefits of centralisation allow a single command to run on multiple devices simultaneously.

- **Network Virtualisation** – The SDN NBI interface provides an abstracted view of the network and enable SDN applications to communicate network behaviour and requirements. The controller is able to implement constructs of layer 2 and layer 3 to provide bridging or routing between virtual machines.

- **Reducing operational expenditure** – SDN has the benefit of automating network deployment. As a result, will reduce the cost of operating a network. Automation reduces errors experienced in managing multi-vendor devices and also reduces the time and amount of labour required to configure them.

2.3.6 Limitations

SDN aims to enable easier management of multiple devices by moving the control intelligence from network devices to a centralised control that runs as the SDN controller software. One of the expected values of SDN lies in exposing the capabilities and features of network devices through interfaces implemented as open and vendor-neutral in an interoperable way. However, SDN does not come without criticism, and some of the limitations that this architecture introduces are as follows:

- **Complexity due to redundancy** – In a production network, a single controller that manages the entire network becomes a single point of failure. Therefore, additional controllers are added to have multiple redundant control planes. Hence, these

different clusters are expected to work together to manage a network. This adds complexity.

- **Complexity of interfaces** – Previously, interfaces were proprietary and a single control plane managed a single data plane. In SDN, controllers manage and control multiple networks. The control interface should support controller redundancy, i.e. a single node managed by multiple controllers. This implies that a new control interface adds more complexity to allow interoperability between planes.

- **Security concerns** – Controllers run externally on machines which creates the risk of communications between controller and node. At controller-application level, one concern is the level of access to network resources [21]. Since applications may require different privileges, creating mechanisms for multilevel authorisation may provide protection of these network resources.

- **SDN expertise** – Designing an SDN network requires technical skills and a compatible controller, controller software (applications), and elements that interoperate. This requires considering the requirements of the network that are supported by the controller, applications and switch's capabilities. Hence, to simplify this task, enterprises outsource network operations.

- **Latency** – Traditional switches have dedicated control processor logic with tasked applications that run on real-time operating systems with protection against interrupt loss due to task pre-emption. On the other hand, SDN controllers run on general-purpose computers that have associated overhead due to general solutions. The distance between the control plane and data plane in SDN is further away in comparison to traditional with its own associated latency. Considerations may need to include the time taken by a controller to add or update the configurations of the data plane.

- **SDN controller performance** – Due to centralisation, the health of the SDN controller affects the health of the entire network. Monitoring of the system is not limited to the data plane but also the health of the controller.

2.4 A Case for OpenFlow

There are multiple protocols that exist in the implementation of SDN, however, OpenFlow is one protocol that is particularly noteworthy due to its multiple benefits. This include

benefits such as lowering costs, enabling flexibility, promotes rapid service development, adaptable to business objectives, and more [50]. OpenFlow has attracted many proof of concept or prototype implementations and used in the industry by organisations [23]. To date, there are a number of open-source implementations in the form of OpenFlow controllers, as well as physical and virtual switch implementations which are discussed later in the next chapter.

2.5 Summary

This chapter highlighted the architecture and components of traditional and SDN networks. Traditional networking faces challenges in managing the management plane and control plane design as the network size scales and requires dynamic applications. The architecture of traditional network devices introduces the heterogeneously distributed structure of coupled of control and forwarding planes. This distributed structure has evolved in efforts to satisfy the continual growth of the Internet and also to address issues of network administrators around consistency between and fast convergence [36]. The approach in SDN is to have the control plane software centralised into an SDN controller and allows easier management which was previously centralised through APIs or CLI.

The main challenge with the design of this structure is the level of flexibility and ease of user control, programmability and the interdependencies of the management, control and data planes. The is difficulty in manageability and the restriction on flexibility as the network scales. Traditional devices are generally implemented using ASICs running proprietary software with limited programmability. The centralisation of the control planes logically makes better for ease of management, scaling and flexibility. The SDN architecture proposes a separation of the forwarding and control planes of a network. The separation of these plane is brought about by open APIs that expose the functionality of these networks which enables programmability. This chapter also compared the two technologies, traditional and SDN architectures, and how SDN solves the issues of complex management and limited flexibility.

Finally, this chapter looked at some of the available APIs for northbound and southbound interfacing and saw that protocols are the key enablers of SDN. These protocols help build an end-to-end SDN solution. The designing of SDN solutions requires defining a list of sets of protocols used. The choice of northbound APIs is defined by the APIs that the vendors of SDN controller have made available, while the southbound interfacing favours

OpenFlow. The OpenFlow protocol is a prominent SDN standard protocol for implementing controller and switch interaction and the organisational structure of OpenFlow control planes along with different approaches to programming network controllers. In Section 3.2 shows the different approaches in the implementation of an OpenFlow. The next chapter looks into the data plane aspects, focusing on OpenFlow-based networking and display why OpenFlow is used.

Chapter 3

Open Source Software Tools

In Chapter 2, the control and data planes of both traditional and SDN networks were reviewed. This chapter will focus on OpenFlow, a protocol that enables SDN. This chapter aims to justify the use of OpenFlow as the protocol of choice for the SBI. It will also discuss, in general terms, the versions available, the feature set prominent, the implementations of OpenFlow, and the approach taken in terms of the architecture of a switch and its data paths. The chapter will also look at multiple SDN controllers that implement a network's control plane.

3.1 OpenFlow

The OpenFlow standard [41] is a popular southbound interface for controllers in SDN implementations. It enables SDN by allowing software to modify the behaviour of the underlying network nodes through programmable APIs. The underlying network nodes can be either virtual or physical where each network node hosts a local flow table used in determining the behaviour of how packets are transmitted. The OpenFlow standard was the first standard aligned with SDN [51] and was first proposed by McKeown in 2008 [21] and is now maintained by the Open Network Foundation (ONF)[41].

OpenFlow devices are fully controlled by an OpenFlow controller and the capability of the flow table not only includes IP, but also other protocols. The makeup of OpenFlow across its versions requires at least one or more flow tables. The ONF releases specifications for each released version of OpenFlow. OpenFlow defines the following components: an OpenFlow switch, a controller, and the OpenFlow protocol. Figure 3.1 shows the basic composition of an OpenFlow switch.

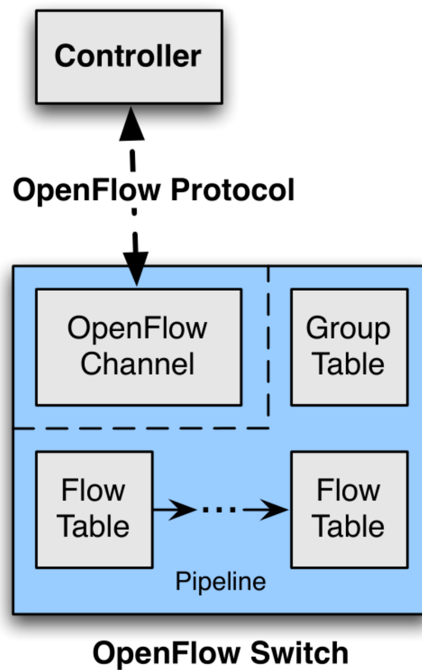


Figure 3.1: Main Components of an OpenFlow Switch. Source: [6].

3.1.1 Version Timelines

The OpenFlow protocol is gradually evolving and has undergone a number of changes since its inception. Its progression is shown as a timeline in Figure 3.2. OpenFlow is available in multiple versions and an important considerarion is the incompatibility between them. The determining factor for selecting a specific OpenFlow version was influenced by the need for compatibility with popular systems. The initial release of OpenFlow 1.0 had a limited subset of functions and ONF quickly released in version 1.1 and 1.2. Vendors, however, requested that the OpenFlow development cycle be slowed down [52]. This is because OpenFlow hardware implementations became increasingly incompatible with each other, and vendors would need to re-implement specialised hardware to keep up. In version 1.3, vendors were able to use this stable version before the ONF started releasing new versions again. This thesis focuses on OpenFlow version 1.3 and its exploration of the OpenFlow protocol because of the large support amongst hardware and software vendors [53, 54].

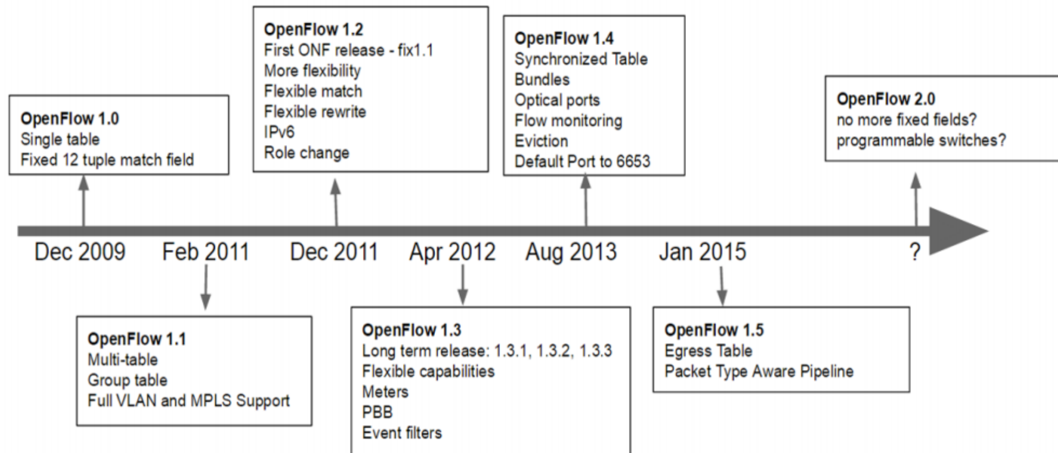


Figure 3.2: OpenFlow Version Timeline. Source: [7].

3.1.2 OpenFlow Switch

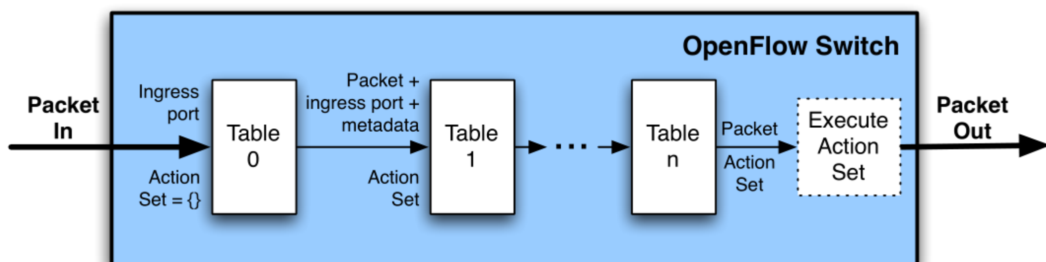
An OpenFlow switch includes an *OpenFlow channel*, one or more *flow tables* and a *group table* [6] (See Figure 3.1). The OpenFlow controller communicates with and manages the switch via this *channel*. The OpenFlow controller also pushes flows onto the switch's flow table through this channel. The flow table holds information used by the OpenFlow switch to forward packets within a network. The flow table is similar to the FIB found in traditional contexts (see Section 2.2.2). The difference is that the sets of rules defined are called flow entries and these are stored in the flow table. Using the OpenFlow protocol, the controller can add, delete and modify flow entries by means of reactive (as a response to arriving packets) or proactive (predefined rules for known paths and routes). An OpenFlow switch processes arriving packets through its pipeline. The pipeline shows how the switch interfaces with the controller and other network devices.

3.1.3 OpenFlow Pipeline

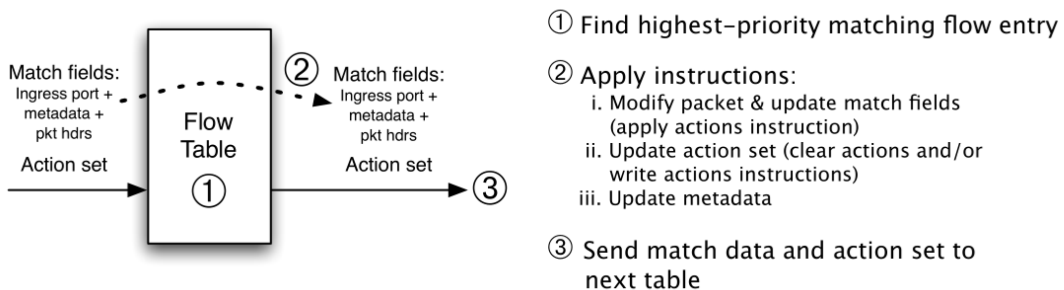
OpenFlow switches either support the OpenFlow-only or OpenFlow-hybrid pipelines [55]. OpenFlow-only switches only support packet processing through the OpenFlow pipeline whereas OpenFlow-hybrid switches are capable of processing packets through the OpenFlow pipeline as well as the normal processing that exists in traditional network devices. Normal processing may include layer 2 Ethernet switching or layer 3 routing. In the

OpenFlow-hybrid pipeline, the switch may also allow packets to move from the OpenFlow pipeline to the normal pipeline by outputting packets through the '*normal*' action.

An OpenFlow switch contains a minimum of one flow table where each flow table holds sets of rules called flow entries which determine how a packet is handled upon arrival. The OpenFlow-pipeline process packets using the flow entries as shown in Figure 3.3. Packets are matched against the flow entries beginning with the default flow table, *Table 0* [6]. If a packet matches a flow entry, the instruction set in the flow entry is then applied. A flow entry may also include instructions to direct packets to other flow tables. Each entry can only point to table numbers higher than the one where the instruction originated. Therefore, processing only goes forward and not backward. This is a precaution to prevent cycles in the pipeline [6]. After matching all flow entries within these tables, the pipeline processing stops and the packet is processed with the associated instructions.



(a) Packets are matched against multiple tables in the pipeline



(b) Per-table packet processing

Figure 3.3: Packet Flow Through the Processing Pipeline. Source: [6].

In the event that none of the flow entries match the packet i.e. a table-miss, the OpenFlow switch will execute a predefined table-miss flow entry. This entry will instruct the switch on how to handle packets that do not have any matches in the flow table. Details about the pipeline components are explained in more detail in the following subsections.

3.1.3.1 Flow Table

A flow table consists of flow entries. Entries are uniquely identified by taking the match fields and priority together [6]. A flow entry with all fields omitted defines the table-miss and has a priority of zero. Below shows fields of a flow entry.

- **Match fields** these are used to match against packets arriving at the switch and contain packet headers.
- **Priority** used to define the order of matching flow entries.
- **Counters** used for monitoring packet statistics, they are updated whenever a packet matches an entry.
- **Instructions** contain a set of actions to apply to the packet when it has a matching packet header.
- **Timeouts** define the maximum timeout (hard timeout) or idle time before discarding a flow entry from the flow table.
- **Cookie** its main use is for managing flow entries. They are used by the controller to filter flow entries for purposes such as flow statistics, flow modification, and flow deletion requests [6]. The controller chooses the value of the cookie.
- **Flags** these alter the management of flow entries. An example flag is `OFPFF_SEND_FLOW_REM` which triggers the 'flow removed' message when a flow expires or is deleted.

3.1.3.2 Matching

The OpenFlow specifications defines the processing that should be executed upon receiving a packet at a port on the forwarding device. Upon receiving a packet, the switch performs a lookup in the table, beginning with Table 0. Depending on the requirements of the pipeline, the switch may perform table lookups in other flow tables. Information extracted from the packet headers defines match fields to be used for table lookups. The packet type determines the process to be followed in the pipeline. In addition to the packet headers, the ingress port, meta data fields and other pipeline fields also form part of the matching [6]. When all match fields (header fields and pipeline fields) match a flow

entry, it is selected. If the table contains multiple matching flow entries, then the one with the highest priority is selected. Upon selection, the counters corresponding to the flow entry are updated and the instruction set included in the flow entry is then applied. Each table will update the counters, execute, and form the packet matching fields in the subsequent tables.

3.1.3.3 Table-Miss Entry

Each table must define a table-miss entry. This is meant to continue the processing of a packet in the event that there are no matches found. The table-miss contains no matching fields and the priority is set to the lowest value of zero. The table miss entries are defined by the user and enable better operation of the switch [6]. The behaviour is like any other flow entry and may be used to discard a packet, forward it to the OpenFlow controller for further processing, or forward it to the normal pipeline if supported by the switch. The table-miss entry behaves like any other flow entry and it can be installed or removed by the controller by the controller. In absence of a table-miss entry, by default the packet is dropped/discarded [6].

3.1.4 OpenFlow Channel and Control Channel

The OpenFlow channel [6] is the interface that is used in the connection between an OpenFlow switch and OpenFlow controllers. The controller manages the switch, obtains events and sends packets to the switch using this interface. The switch can support a single or multiple OpenFlow channels with a single or multiple OpenFlow controllers. The OpenFlow channel connection uses the Transmission Control Protocol (TCP), which can be encrypted using Transport Layer Security (TLS). The *channel* is sometimes referred to as a *secure channel* indicating the use of an encryption of the connection.

There are three types of messages that OpenFlow supports [6], *controller-to-switch*, *asynchronous*, and *symmetric*. The OpenFlow controller manages or retrieves the state of the switch through controller-to-switch messages. Asynchronous messages are initiated by the switch to inform the controller about network events or changes to the switch state. Lastly, symmetric messages are initiated by either the controller or the switch. Symmetric message functions include connection startup by using '*Hello*' messages, verifying the liveness of the connection using '*echo*' messages, or used for testing experimental functions of the switch.

3.1.5 Group Table

A group table represents additional methods of packet processing. It is the last table to implement multiple actions. It enables OpenFlow to represent a set of ports as a single entity. There are different group types to represent abstractions such as multicasting or multi-pathing. A group entry is composed of a set of group buckets where each bucket contains the set of actions to be applied before forwarding to the port. Groups buckets may also be chained to other groups. Below shows components of a group table entry.

The entries are defined as follows:

- **Group identifier** is used to match against packets arriving at the switch and contain packet headers.
- **Group type** is used to define the group's behaviour based on how the sets of actions are to be executed.
- **Counters** are updated whenever a packet is processed by a group.
- **Action buckets** comprise of a list of action buckets. Each bucket contains a set of actions that are applied as a set.

The action/behaviour is executed based on the type. The type may be defined as below:

- **all:** The behaviour is to execute each bucket. Application: broadcast and multicast.
- **select:** Will execute a select bucket. Application: port mirroring.
- **indirect:** Executes one defined bucket in this group.
- **fast failover:** Execute the first live bucket, if there are no live buckets then packets are dropped.

3.2 OpenFlow Switch for Data Path

OpenFlow switches are either physical or virtual. The physical (hardware) implementation includes platforms like application specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs), while virtualised switches like Open vSwitch can

run on general-purpose CPUs. The architecture of the switch and the type of platform used impact the performance. An OpenFlow switch, in practice, has a limited amount of resources available [56]. The resource constraints include memory, the CPU and bandwidth. The OpenFlow pipeline will match arriving packets to the flow entries and execute the set of instructions associated with the matching flow entries. The mechanisms used in the implementation of the OpenFlow switch may limit the performance of the hardware switch, hence may impose bottlenecks of the network.

A computer system comprises four basic components. These are: the Logic Unit (processing unit), Memory, I/O modules and System interconnection (System bus). The performance in general is dependent on these components. The performance metrics for logic unit, Memory, I/O modules and System bus are generally measured in terms of throughput and latency. These four components are used in the implementation of the system platforms that form general-purpose Processors (CPU), NetFPGA, FPGA and ASIC [57]. One fundamental key for the implementation of efficient high packet processing of flows involves two elements, performance and programmability. Figure 3.4 shows the relationship between performance and programmability, the trade-off being that delivering high performance limits the level of programmability of that hardware and vice-versa [21].

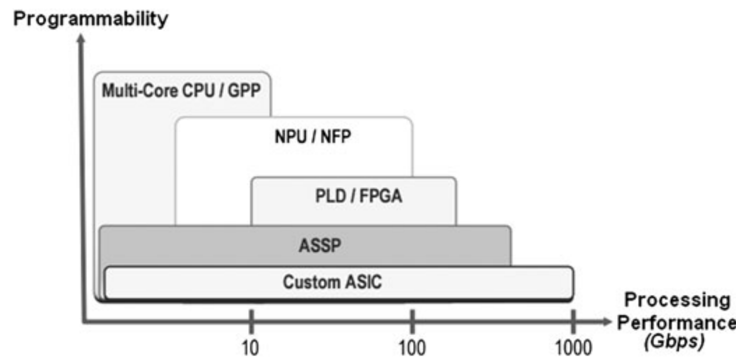


Figure 3.4: Performance and Programmability. Source: [8].

3.2.1 Table Lookup Mechanisms

OpenFlow table match lookup involves accessing memory. Different memory addressing techniques are used in the implementation of OpenFlow table. However, these achieve different speeds in terms of access times. The memory speed (in Hz) and lookup time (number of cycles use to fetch data) affects the overall latency and data rate of the switch. The different memory types include dynamic random access memory (DRAM), static random access memory (SRAM), and advanced memory addressing such as a content

addressable memory (CAM) or Ternary CAM (TCAM). The design of each memory type factor into the characteristics. These characteristics include access speed, bit density, cost per byte of memory, memory space, circuitry space on board, and power consumption.

DRAM is the cheapest way of memory addressing and the highest bit density per integrated circuit. SRAM is useful for small amounts of high-speed memory [58]. while CAM and TCAM have the best lookup performance at a higher cost, power consumption, and area required for circuitry on the motherboard [59, 60]. CAM and TCAM allow the lookup based on content rather than the address.

3.2.2 OpenFlow Pipeline Processing Architecture

The type of memory used for the lookup table determines the mechanism used to access the OpenFlow table implemented. During packet processing, the OpenFlow table is accessed to retrieve necessary information for processing packets. The packet processing logic (CPU, FPGA or ASIC) coupled with these memory technologies form the core hardware architecture of the OpenFlow pipeline. The following sections provide a brief commentary on the processing logic.

3.2.2.1 General-purpose Processors

General-purpose CPUs provide the highest flexibility with respect to programming. They support rapid development of complex packet processing implemented through high-level languages and design tools [21]. The logic unit of CPUs includes multi-core processors, which generally use DRAM and provide the cheapest practical application of OpenFlow devices. Software switches are known to be used for virtualised networks as the underlying network device [61]. Running in user-space, software switches have their OpenFlow tables in DRAM. The OpenFlow switch runs as a program in main memory with a multi-level cache memory system. The I/O module includes a hierarchy design to support multiple devices. The physical ports are implemented as network card interfaces (NICs). The system bus on these computer systems interfaces with the CPUs through the Peripheral Component Interconnect (PCI) interface. The limitation is due to the memory access times of DRAM. Multi-core system such the Intel® Xeon processors can achieve several tens of Gigabits/s [62].

3.2.2.2 Network Flow Processors

NFPs are less programmable in comparison to CPUs, i.e. limited to network flow applications. They implement SRAMs for the lookup table [63]. SRAM in comparison to DRAM for a general-purpose environment, provide faster access, but at a greater financial cost. OpenFlow Table lookup with SRAM and NFP can provide 200Gbps line-rate with over 100 million packets per second (Mpps) [21].

3.2.2.3 Field Programmable Gate Arrays

FPGA implement their Lookup tables in SRAM. FPGAs, unlike general-purpose CPUs, have less flexibility in terms of programmability [64]. However, unlike general-purpose CPUs, they require programming expertise. They may achieve line speeds of 200 Gbps per device and switching of 200 Mpps [21].

3.2.2.4 Application Specific Integrated Circuits

ASICs use CAM and TCAM [65]. These are optimised to have the fastest packet processing. They achieve high speed by searching the entire memory in just a single clock cycle [66]. CAM and TCAM memory searches are faster compared to SRAM and DRAM. With TCAMs, bandwidths of 800 Gbps are possible [67].

3.2.3 Hardware, Software or Hybrid Table

The performance of an SDN switch depends on the implementation of the SDN pipeline, where the data plane and the control plane are separated. The data plane, in this example, is an OpenFlow-enabled switch which performs the forwarding of packets, whereas the control plane makes decisions (rules) regarding how to deal with the packets. An OpenFlow-enabled switch then stores the rules as flow table entries within one or more flow tables. These rules are either exact-matches (with all match fields specified) or wildcard-matches (some fields have any value). During data plane forwarding, flows in the tables are stored and accessed from memory (DRAM, SRAM or TCAM). Depending on the memory technology used, the penalties for searching memory (latency) for a flow entry negatively affects the overall data rates and forwarding latency.

The TCAM memory implemented in ASICs usually has very limited capacity, few capabilities and is power hungry and expensive [68]. TCAMs consume large circuit areas within switches where the circuit area is a key design constraint [60]. Because they are limited by space and cost, the memory is usually extended to SRAM to implement a software flow table and have a hybrid of both software and hardware tables. TCAM hardware flow tables offer the best performance achieving near line rate switching speed.

This heterogeneity within the switch influences the performance of a software-defined network. The performance characteristics and key features of OpenFlow-enabled devices are as follows:

1. OpenFlow switches from different vendors have their own way of implementing the OpenFlow pipeline. Hence, flow handling, processing logic, bus, and memory requirements differ.

2. The capacity and capabilities also differ across switch models.

3. Hardware tables are fixed and are incompatible between versions of OpenFlow.

4. The cost for performance in the table lookups is at the expense of memory.

5. The complexity within the switch does not necessarily hinder performance [69]. For example, TCAMs are more complex than DRAMs but offer better performance.

6. The switch's packet processing i.e. the data path performance, looks at the processing speed of the OpenFlow pipeline.

3.2.4 Evaluating Implementation Models

Table 3.1: Summary of Performance Characteristics

	CPUs	FPGAs	ASICs
Table Capacity	High	Moderate	Limited
Programming Flexibility	Very Flexible	Flexible	Least Flexible
Memory Type	DRAM	SRAM	TCAM(and SRAM)
Switching Capacity	Lowest	High	Very High

Table 3.1 shows a summary of the switching capacity, flexibility and table size based on the type of memory access used in the implementation of an OpenFlow pipeline. Looking

at the table capacity, i.e. the size of the OpenFlow Tables, it is less complex to implement larger table sizes in virtualised environments than physical hardware. The flexibility in terms of programmability makes it easier to implement extensions which ultimately affects the ability to add new features.

CPUs generally implement software switches and features are easy to develop and deploy. FPGAs, as mentioned before, require specialised knowledge in the programming and is limited to how much resources are available. ASICs once developed, the addition of new features achieved through firmware upgrades is limited to what is available on the ASIC hardware.

As mentioned in Section 3.2.3, ASICs tend to have a hybrid software-hardware tables but are generally the most expensive to develop. To conclude, the switching capacity increases from CPUs to FPGAs to ASICs [57].

This thesis aims to meet the objective of building an inexpensive flexible multi-layer switch. The implementation of an OpenFlow switch using the CPU model as the processing logic of the switch is most favourable due to its high programmability at a low cost. Though the switching capacity is lesser than either FPGAs or ASICs, the implementation of a switch using a CPU offers switching capacity that meet the requirements of small-medium scale networks. In addition, a software (virtual) switch is suitable for an OpenFlow switch if using the CPU architecture as its processing logic.

The following section looks at candidate virtual OpenFlow-enabled switches that are open source.

3.3 Open Source Virtual Switches

There are multiple open source programs that are native to OpenFlow or compatible with OpenFlow – to mention a few:

- **Indigo** – Developed by Big Switch, aims to enable support for OpenFlow on both physical hardware and hypervisors [70].
- **Open vSwitch** – A project under the Linux Foundation, this is a multi-layer switch that supports OpenFlow 1.0 – 1.5 [71].

- **Lagopus** – An open source high-performance virtual switch with DPDK-powered software data plane [72].
- **LINC** – Written in Erlang, the OpenFlow-pipeline is strictly OpenFlow-only. The switch supports OpenFlow 1.2, 1.3 and 1.4 [73].
- **Of13softswitch** – Produced by Ericsson, implements a OpenFlow 1.1 softswitch [74].
- **BOFUSS** – This is a user-space switch based on the Of13softswitch. It currently only supports OpenFlow 1.3 [75].

Open vSwitch is used extensively by numerous companies in their solutions due to its ease of adaptation in hardware switches networking stack [26, 76, 77]. Open vSwitch is also designed to be adopted in hardware stacks used in the industry for host-based applications [61]. The majority of the code found in Open vSwitch is written in platform-independent C and may be ported into multiple environments including switching chipsets [78]. As a result, Open vSwitch will form an important role in the focus of this thesis.

3.4 Open vSwitch

Open vSwitch (OVS) [71] is an open source software switch that runs on general-purpose architecture. It is designed to run in a virtualised environment and is also designed to be distributed across multiple physical servers that use Linux-based virtualisation including Xen/XenServer, KVM, and VirtualBox [79]. OVS is OpenFlow enabled, hence is accessible remotely through an OpenFlow controller that can configure the OpenFlow settings and behaviour of OVS [71]. OVS is supported by many Linux distributions, such as Ubuntu and even Windows platforms. The majority of OVS, as mentioned before, is written in C programming language. This enables the support for multiple distributions. The code in OVS is written to be platform-independent, hence the porting of OVS into multiple environments including switching chipsets is possible [79]. OVS has also been used to form part of the networking stack in hardware silicon [76, 77] and offloading packet processing of software switches to hardware [80]. Through OpenFlow, OVS exposes itself to management and control. OVS is a multi-layer switch that enables the programming of the forwarding behaviour and control through software.

3.4.1 Open vSwitch Architecture

The architecture of OVS comprises components that aid in the processing of packets. These components, each with a different function, form a modular structure. As a virtualised software switch, OVS makes use of the resources on the host machine for packet processing including a multi-level cache for performance. Figure 3.5 shows the archit ecture of OVS.

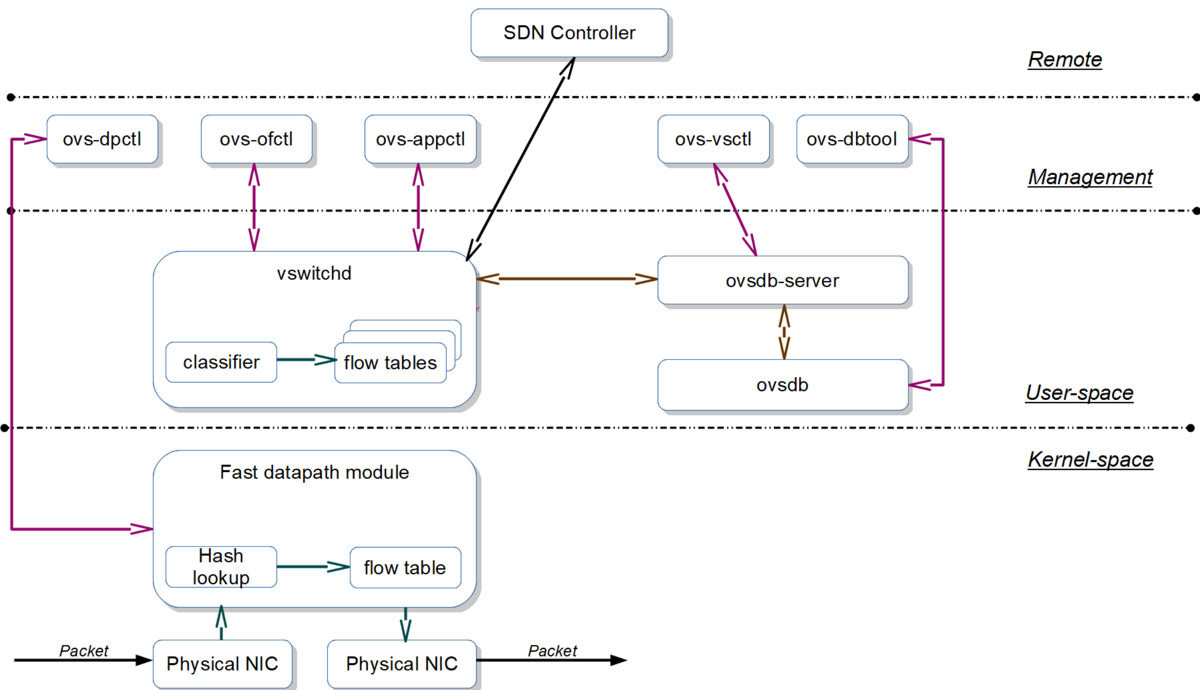


Figure 3.5: Open vSwitch Architecture. Adapted from: [3].

There are two major components included in its internal architecture which are: `ovs-vswitchd` and `kernel module`. The first component, `ovs-vswitchd`, is a daemon program that implements the core switching functions of OVS. This program runs in user-space on the host machine and maintains the flow table. The daemon determines how the packets are to be handled via flow entries. Flow entries include *actions* that list the physical port or tunnels that the ingress packets should be transmitted in. The packet classifier supports the matching of flow entries based on layer 2 to layer 4 implementing switching features for forwarding, dropping, or modify packets. It also implements the management of the data-path flow counters and handles flow expiration [1].

The second component, the `kernel module` resides in kernel space. This module is unaware of OpenFlow and processes packets based on instructions called *actions* set by the

`ovs-vswitchd` daemon. Information from packet headers is extracted and hashed to use as indices to find flows. When a match is found, the packet is processed further by the data-path based on the actions that will forward, modify, encapsulate/de-capsulate and increment packet counters. If a packet has no set action in the data-path module, the kernel module will request instructions on handling the packet from `ovs-vswitchd`. The associated actions are then returned to the data-path and these are normally cached in the kernel module [81]. Future packets that have similar destination will be processed using the actions. The daemon communicates with the kernel module and also communicates with the system through an abstract interface.

Apart from the two major components, OVS includes another component named OVSDB server. The function of this server is to store the configurations represented as an OVSDB table. The communication between `ovs-vswitchd` and OVSDB server makes use of the OVSDB protocol. OVS also includes a command line interface (CLI) where the configuration and behaviour may be configured. `ovs-vswitchd` communicates with remote OpenFlow controller(s) using OpenFlow. The configurations are managed through an OpenFlow channel with an OpenFlow controller. The controller may also configure the OpenFlow switch via the OVSDB protocol or the CLI. The CLI in OVS provides a set of utilities that support the following features to configure, monitor, and debug Open vSwitch.

- `ovs-vsctl` – A CLI tool to configure OVS to interact with the configuration database. Using this tool, a user may communicate with *ovsdb server* which maintains an Open vSwitch configuration database. The tool can be used to initialise Open vSwitch bridges, configure ports, interfaces, and setting up the OpenFlow controller address.
- `ovs-ofctl` – A tool for monitoring and managing OpenFlow switches. It may be used to request information such as current state, features, configuration, and OpenFlow table entries.
- `ovs-appctl` – This may be used to invoke commands supported by `ovs-vswitchd` program and display the response on a standard output (CLI).
- `ovs-dpctl` – A tool to create, modify and delete Open vSwitch data paths, for example, the *kernel module* found in Linux.

3.4.2 Open vSwitch Data Paths

The transaction between the `kernel module` and `ovs-vswitchd` daemon, i.e. when the kernel module queries the daemon for actions, is a costly process which is why those actions are cached in the kernel module. The `kernel module` and the `ovs-vswitchd` are referred to as the 'fast path' and 'slow path' respectively [79]. The performance of the switch is heavily reliant on the caching of the data-path.

Figure 3.6 shows the cache hierarchy within the `kernel module`. The 'slow path' includes the OpenFlow flow table and its flow entries. Because of the sizes of the tables, performance is slow due to the fact that the passing of packets to the user-space daemon requires context switching. Context switching between the kernel space and user-space requires the computing process to store and restore the state of a CPU such that multiple processes can share CPU resources. 'Fast path' (data-path) caching is further divided into two levels, named megaflow and microflow [81]. The performance ranges from high to low following microflow, megaflow and flow table classifiers.

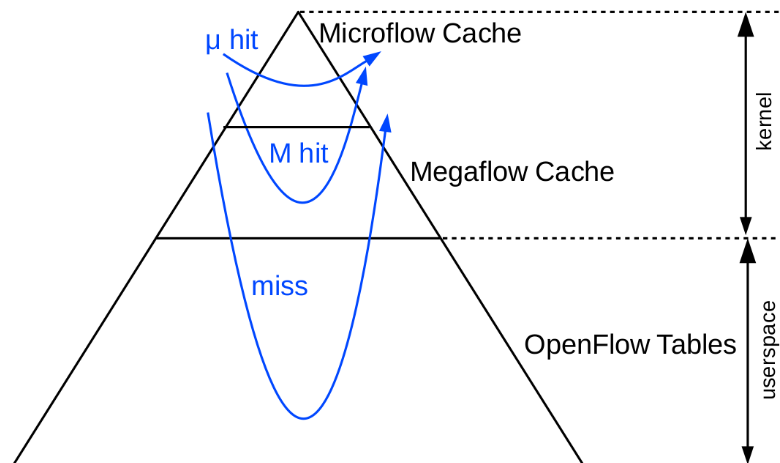


Figure 3.6: Open vSwitch Cache Hierarchy. Source: [9].

3.5 Open vSwitch Alternate Data Path

The packet processing performance of OpenFlow, where the data-path implemented as the kernel module, the data rate is limited by the delay penalty incurred during context switching. The transfer of data between kernel and user-space involves system calls and interrupts. In an environment where resources are limited, the performance of the switch

is greatly diminished when under high load. This is due to the increase in overhead. Alternatively, OVS may bypass the kernel and implement its data path in user-space using an alternate driver set which eliminates any requirements for interrupts, system calls and context switching between user-space programs and kernel space programs. This approach offers a more desirable performance compared to the performance of a kernel data-path. This method is discussed in subsequence sections which include the structure and how it operates in the user-space.

3.5.1 Data Plane Development Kit

The Data Plane Development Kit (DPDK) [11] is a set of user-space drivers that enable OVS to provide high-performance packet processing through accelerated user-space data-paths.

DPDK implements a run-to-completion model which means that DPDK performs packet processing in a chain of functional stages [82]. The result is an enhanced network packet data rate with a much lower latency. The user-space drivers of DPDK implement threads that poll the ingress ports of OVS. This bypasses the kernel and avoids the need for interrupt processing. DPDK further optimises its performance by using techniques such as hugepages, multi-core processing, processor affinity, no copy from Kernel, lockless ring design with readers, and writers running on separate cores.

DPDK supports the development of applications that can take advantage of high-speed data packet processing that DPDK offers, which means that applications can process packets much faster without the need for implementing kernel modules. DPDK fast path avoids context switching and packets are made available in user-space directly (as raw packets). Figure 3.7 shows the difference between the two data paths, i.e the kernel module and DPDK.

3.5.2 Core Components

DPDK provides a set of libraries that are needed for high-performance packet processing. These components enable DPDK's fast path mechanism for packet processing. Listed below are these components, with the global overview is shown in Figure 3.8 [82].

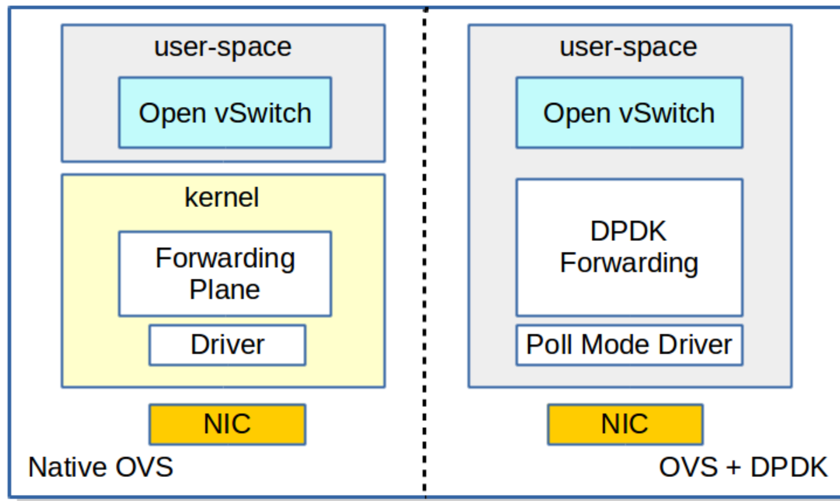


Figure 3.7: Native OVS vs OVS with DPDK. Adapted from: [10].

- **Environment Abstraction Layer, EAL (rte_eal+libc)** functions include obtaining low-level resources such as hardware and memory space and then provide a generic interface that abstracts the environment specifics from the applications and libraries. During initialisation, it decides how to allocate these resources.

- **Memory Pool Manager (rte_mempool)** is responsible for allocating pools of objects in memory. A pool is created in hugepages as memory chunks of 4KB, 16KB, etc, and uses a ring for storing free objects. The memory manager provides an alignment helper to ensure that objects are allocated in contiguous blocks equally on all DRAM channels.

- **Network Packet Buffer Manager (rte_mbuf)** reduces, by a significant amount, the time that the operating system spends allocating and de-allocating buffers using advanced techniques such as Bulk Allocation, Buffer Chains, Per Core Buffer Caches, etc. Fixed size buffers are pre-allocated and stored in memory pools. This manager provides an API to allocate or free buffers, manage control messages, and packet buffers to carry network packets.

- **Ring Manager (rte_ring)** uses a ring structure providing a lockless multi-producer, multi-consumer FIFO API, instead of lockless queues. The advantages compared to lockless queues are easier to implement, adapted to bulk operations and generally faster. This ring is used by `rte_mempool` manager and may provide a mechanism for communication between cores and/or execution blocks.

- **Timer Manager (rte_timer)** provides a timer service to execution units allowing

functions to be executed asynchronously. Its uses included periodic calls such as garbage collectors or some state machines (ARP, bridging, etc).

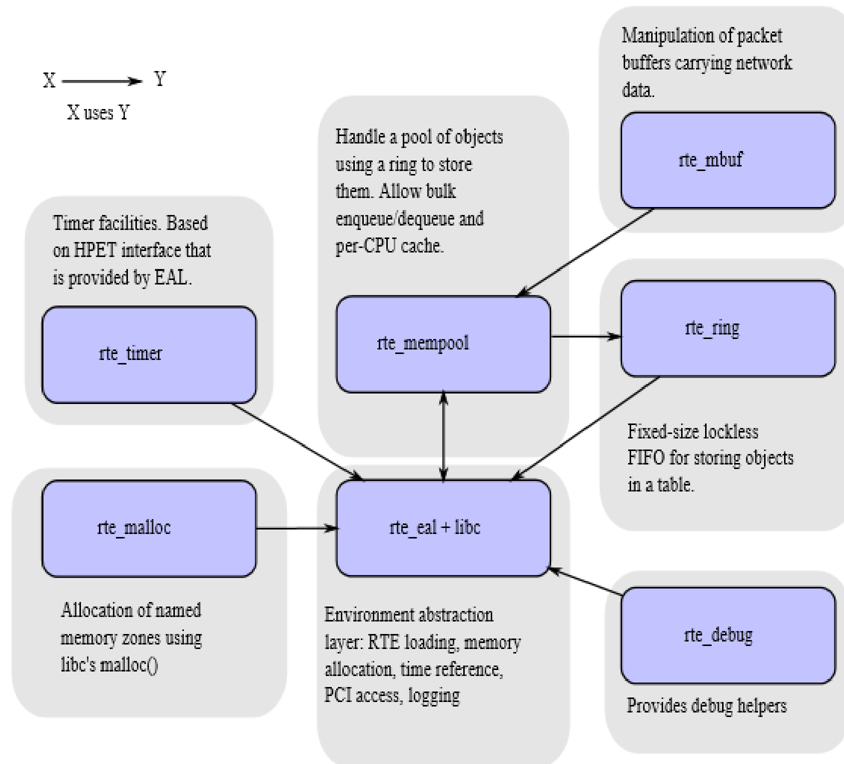


Figure 3.8: Core Components Architecture. Source: [11].

3.5.3 Poll Mode Driver (PMD)

The PMD provides threads that run in user-space to enable fast packet processing. The EAL (Environment Abstraction Layer) creates these threads, i.e. the poll mode driver, to poll file descriptors of a device. PMD is designed to work without interrupts, avoiding context switching between the kernel and user-space. A PMD is made up of APIs to configure network interface controllers (NICs) and their respective queues. Additionally, PMDs access the descriptors of ingress and egress ports directly to quickly receive, process and deliver packets in the user's application. The DPDK library provides a set of drivers for a defined set of commodity NICs e.g. Intel® ixgbe [1], Netronome nfp [2] and Mellanox mlx5 [3], and so on.

[1]Intel® Network Adapter Driver for PCIe Intel® 10 Gigabit Ethernet Network Connections
[2]Netronome NFP-4xxx and NFP-6xxx flow processor families
[3]Mellanox Network controllers: ConnectX-4, ConnectX-4 Lx, ConnectX-5, Bluefield

3.6 OVS-DPDK

OVS (Section 3.4) and DPDK (Section 3.5.1) can be integrated to form OVS-DPDK (OVS switch using DPDK as the data-path). The architecture of OVS-DPDK involves process chaining that requires the movement of packets from the physical interface to user-space, the processing of the packets, and then finally sending it back to the appropriate physical interface for forwarding. Since this processing involves multiple stages of chaining, each incurs a latency overhead. This chaining will consume CPU cycles at each stage during packet processing.

OVS makes use of DPDK to implement the data-path in user-space for 'fast path' processing. The PMDs allows fast packet processing at line rate speeds by polling the NIC for the communication of packets. The OVS-DPDK uses three-tier look-up tables/caches for processing which includes an Exact Match Cache (EMC), data-path classifier and the `ofproto` classifier table. The EMC only caches exact matching flow entries while the data-path classifier works as a wildcard matching table. Finally, the third-level table is the `ofproto` classifier table whose contents are managed by an SDN OpenFlow-compliant controller. Figure 3.9 depicts the three-tier cache architecture in OVS-DPDK.

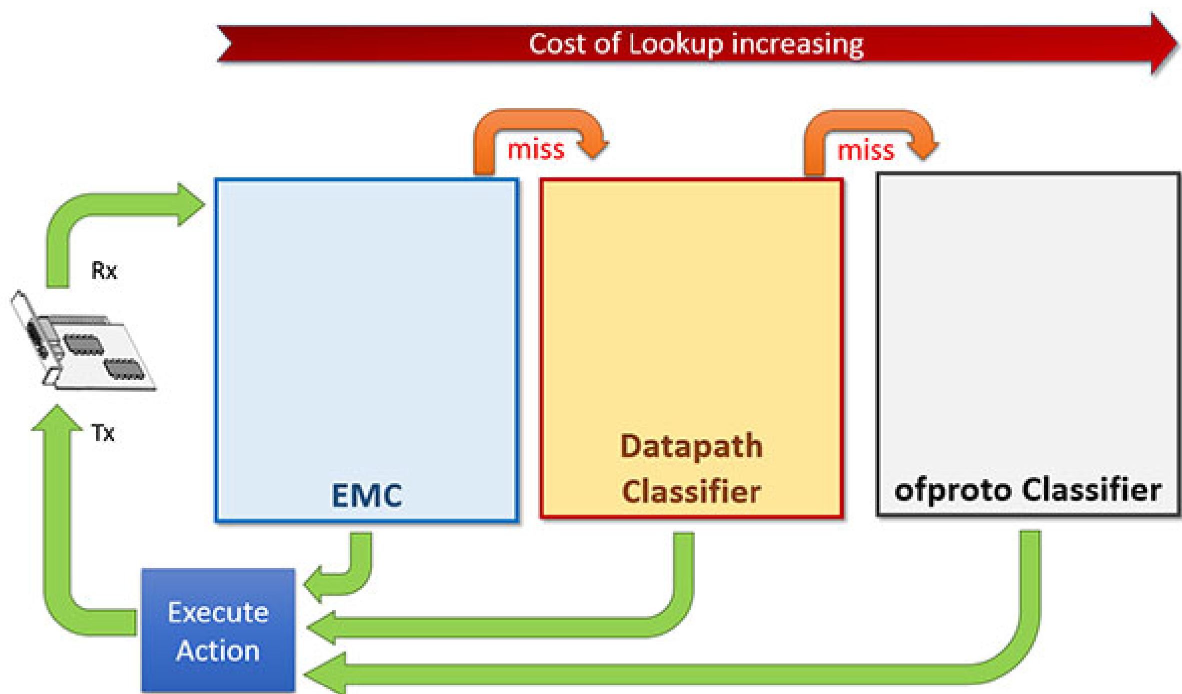


Figure 3.9: OVS-DPDK three-tire cache architecture. Source: [12].

When packets arrive, they traverse through multiple tables beginning with flow table

0. If a match is found, the corresponding forwarding action is executed. Real world deployments handle thousands of flows and this quickly fills up the EMC cache. The critical aspect for the overall performance depends on the performance of the data-path classifier. The performance of the classifier is affected by the hash calculation used in searching for a match. Leveraging on SIMD (Single Instruction Multiple Data) can speed up hash computation [83].

3.7 SDN Controllers for the Control Plane

An OpenFlow switch is usually managed by a controller. These controllers coupled with OpenFlow switches are used to create SDN networks. There are a variety of OpenFlow controllers that are available to date. Examples of open-source controllers include OpenDaylight[4], Open Network Operating System (ONOS)[5], Ryu[6] and Faucet[7]. These controllers provide APIs that wrap the OpenFlow protocol. By using these APIs, the controllers provide a programmable environment that enable rapid network application development [41]. As mentioned in the previous chapter, the controller forms the control plane within the SDN architecture.

3.7.1 OpenDaylight Framework

OpenDaylight [84], which is part of the Linux Foundation, offers a community-led and industry-supported controller. It is a Java-based SDN controller that can execute on any system that supports Java. OpenDaylight uses the following tools to implement the SDN concepts [84]:

- **Maven:** OpenDaylight uses Maven[8], a tool to build and manage Java-based projects, to define the bundles to load and start and also the scripting of the dependencies between bundles.
- **OSGi:** A back-end framework of OpenDaylight to allow the dynamic loading and binding of bundles for the communication between them.

[4]OpenDaylight: Home, https://www.opendaylight.org/
[5]ONOS - A new carrier-grade SDN network operating system, https://onosproject.org/
[6]Ryu SDN Framework, https://osrg.github.io/ryu/
[7]Faucet SDN Controller, https://faucet.nz/
[8]Maven – introduction, https://maven.apache.org/what-is-maven.html

- **Java interface:** This is the main way how bundles receive information such as events, specifications and forming patterns.
- **REST API:** This northbound API exposes network behaviour and functionality, this include functions such as topology management, host discovery, flow programming, etc.

OpenDaylight uses the OSGi framework and REST as the northbound APIs. Applications can be loaded onto the controller during runtime using the OSGi framework while the REST API is used for applications that execute outside the controller's address space that may run on a separate system. The applications implement the business logic, which makes use of the controller to gather network intelligence. The applications may run algorithms to analyse and orchestrate rules on the network. OpenDaylight implements multiple protocols for its southbound interface which are supported as plugins [84]. Examples of supported plugins are OpenFlow 1.0, OpenFlow 1.3, BGP-LS, and more.

The OpenDaylight controller can run multiple plugins by adding them to the controller code that links dynamically into a Service Abstraction Layer (SAL). The SAL function is to intercept the requests from the plugins using the underlying protocol that is between the controller and the network devices. Hence, the details of the southbound protocol are abstracted from the application's perspective. OpenDaylight uses the Topology Manager to store and manage information about the devices that the controller is managing. OpenDaylight also uses other components like ARP handler, Host Tracker, Device Manager, and Switch Manager to generate the topology database for the Topology Manager.

3.7.2 Open Network Operating System (ONOS)

Open Network Operating System (ONOS) [85] was the first open source SDN network operating system and is also a project under the Linux Foundation (just like DPDK and OVS) [71]. ONOS was targeted at network operators to deliver high availably scaling and performance [86]. In recent years, ONOS has gained popularity among service providers and network operators.

The architecture of ONOS is designed for service providers and defines its architecture as follows [86]:

- **Distributed Cores** provide scaling, high availability and performance. ONOS runs as a service on a cluster of servers enabling rapid recovery in the event of server failures. ONOS instances work together to bring web style agility and scale.
- **Northbound abstraction/APIs** enables the control, management and configuration services through an abstraction.
- **Southbound abstraction/APIs** enables the management of both OpenFlow and traditional devices using pluggable southbound protocols.
- **Software Modularity** enables rapid development, debugging and maintenance of ONOS by a community of developers.

ONOS runs as a service with the same ONOS software running on clusters of servers. Each instance works together and creates what appears to be a single platform, where the instances are hidden from the applications. This allows ONOS to be scalable because instances behave as a single logical entity. As a result, the distributed core is the key feature of ONOS.

3.7.3 Ryu Controller

The Ryu controller [87] is an open-source, component-based SDN framework where developers can create network applications as software components using the defined APIs provided by Ryu. Ryu fully supports OpenFlow versions 1.0, 1.2, 1.3, 1.4, 1.5 and Nicira Extensions. These extensions were proposed by the vendor, Nicira, to address the lack of generic and/or vendor-specific error codes in OpenFlow version 1.0 [88]. It also supports other protocols such as Netconf, OF-Config, and so forth. The Ryu platform allows rapid development and prototyping of network control software using Python programming language [89]. Figure 3.10 shows the programming model used in Ryu.

Ryu applications run as a single thread that processes events. The series of actions that an application takes, occurs when an event is sent by the controller (`ryu-manager process`) which comes from the `data path thread`. Actions such as OpenFlow messages between the controller and switch, trigger events in the `data path thread`. All attached applications will load the `event loop` and call the corresponding `event handler`. The set of actions such as routing or switching are then defined as a `handler`. The events are received by the application in a form of a `FIFO queue`. The application is responsible for draining the *queue* and calling the appropriate *event handler* for each received *event type*.

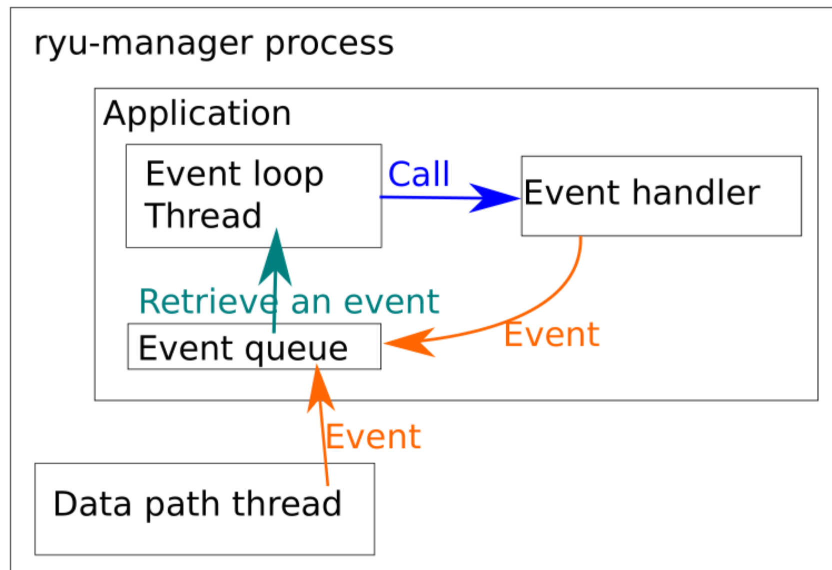


Figure 3.10: Ryu Application Programming Model. Source: [13].

3.7.4 Faucet SDN Controller

Faucet [90] is a compact open source OpenFlow controller based on the Ryu architecture. Faucet only supports OpenFlow version 1.3 and can manage physical hardware switches to deliver high performance. Its architecture is shown in Figure 3.11. Faucet has two OpenFlow controller components, `Faucet` itself, and `Gauge`. Faucet manages all the forwarding and the state of the switch. It also makes the internal information about a switch to be available through a monitoring system called `Prometheus` that can be visualised via `Grafana`, a monitoring dashboard. Gauge also establishes a connection with the switch and monitor port information and flow statistics. However, Gauge does not modify the switch state, so that the monitoring functions can be restarted or upgraded without affecting the forwarding of the switch.

Faucet implements multiple tables to implement the network flow pipeline. Its features include VLAN switching, IPv4 and IPv6 routing (static or Border Gate Protocol routing), access control lists (ACLs), port mirroring, and policy-based forwarding [54]. Faucet uses a two-system deployment scheme which consists of a controller and an OpenFlow-enabled switch that provides a 'drop-in' replacement for a traditional network device such as a switch or router. The controller is typically deployed as a Linux machine running Ubuntu, although it may run on other systems such as Windows. Faucet only supports switches with OpenFlow-only pipeline and is configured using configuration files (`faucet.yaml` and `gauge.yaml` shown in Figure 3.11), which resembles a traditional network configuration file.

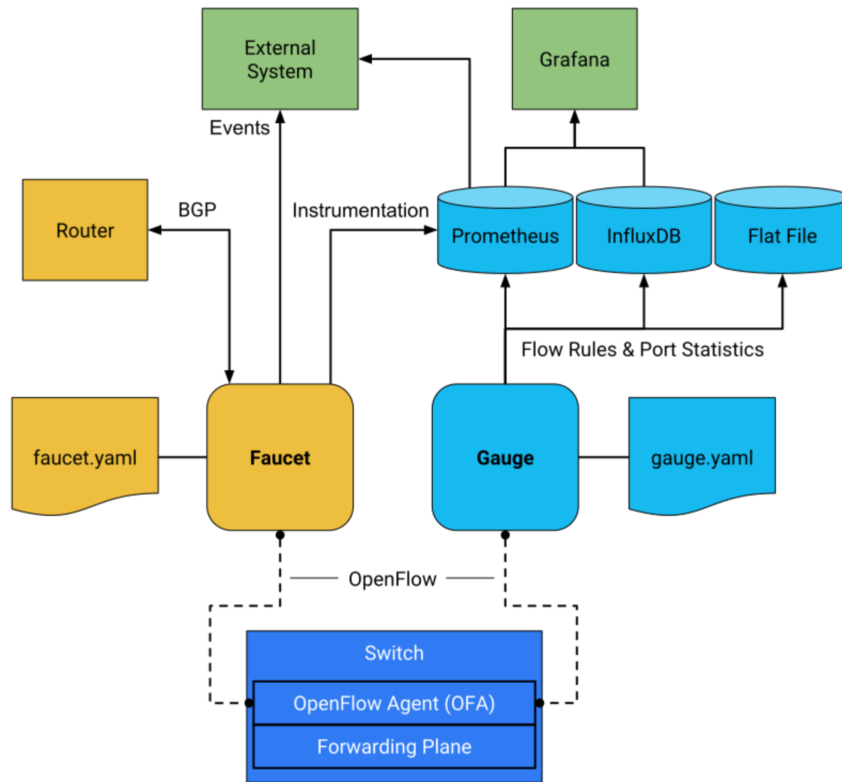


Figure 3.11: Faucet Architecture. Source: [14].

3.7.5 Overview

There are many variations of OpenFlow controllers, but they share the same objective of controlling and configuring data plane switches [91]. Table 3.2 shows a comparison of the SDN controllers mentioned in previous sections.

Table 3.2: Overview of SDN Controllers

OpenFlow Controller	OpenDaylight	Open Network OS (ONOS)	Ryu	Faucet
Northbound Protocols	OSGi, REST	OSGi, REST	Python, REST	Python
Southbound Protocols	BGP, Netconf, OpenFlow, OVSDB	BGP, Netconf, OpenFlow, TL1, OVSDB	BGP, Netconf, OpenFlow, OF-config,	BGP, OpenFlow
OpenFlow Version	1.0, 1.3, 1.4	1.0, 1.3, 1.4	1.0, 1.2, 1.3, 1.4, 1.5	1.3
Primary Language	Java	Java	Python	Python

Each SDN controller given in Table 3.2 supports OpenFlow version 1.3. This corresponds

to the version of OpenFlow selected for this thesis, OpenFlow version 1.3, during the development of the milti-layered SDN switch. This was found to be appropriate in enabling compatibility between the multi-layer SDN switch and any SDN controller that supports OpenFlow version 1.3.

3.8 Performance Metrics

This chapter has identified numerous factors that impinge on switch performance. The thesis also has, as an objective, performance in mind, therefore a methodology was used to provide methods for benchmarking. The methods used in the benchmarking of a device includes: (1) using a benchmark methodology to observe the performance behaviour, and (2) benchmarking the device within a network.

The benchmark methodology used to analyse the multi-layer SDN switch was the RFC 2455 Benchmarking Methodology, while the tool selected for network benchmarking was the iPerf Network Benchmark Tool.

3.8.1 RFC 2455 Benchmarking Methodology

RFC 2544 [92] describes a benchmarking methodology developed by the Internet Engineering Task Force (IETF). It defines a number of tests that can be used to quantify the performance characteristics of a network device. These tests are aimed at providing data for which devices from different vendors can be evaluated. The results produced by each set of tests apply to the evaluation for a selected circumstance. For example, the evaluation of VLAN tagged frames will provide the vendor with the behaviour of the device when performing in a network where VLAN tagging is used. RFC 2544 also defines the setup for testing single or a group of devices. The RFC 2544 document defines the parameters such as frame formats, frame sizes, and the expected stream formats. It also describes specific formats for reporting the results of these tests.

3.8.1.1 Test Conditions

The tests defined must run consistently without changes in configuration or running a specific protocol or feature. This is to avoid biased results that do not reflect actual

performance of the system. The ideal tester recommended includes both transmitting and receiving ports. This will allow the verifying of the sequencing of sent frames with the packets received. RFC 2544 supports layer 2 and layer 3 benchmarks. Layer 3 tests include the IP packets configuration, parameters such as network mask and subnets, while layer 2 frames include parameters such as frame size or bit rate.

3.8.1.2 Traffic Used

RFC 2544 defines test frame formats to be used in the benchmarking of a network device. Below are some of the frame formats that are covered by the RFC.

- **Traffic pattern:** Typically traffic in a network is not constant but occurs in bursts. However, the RFC proposes that the tests use constant traffic and with repeated bursts of frames with the minimum inter-frame gap.
- **Protocol addresses:** Real-world traffic involve multiple streams of data. The RFC addresses this by suggesting that tests are re-run using a random destination address. For layer 3, a distributed range of 256 networks for routers and layer 2 tests uniformly distributed over the full MAC range.
- **Maximum frame rate:** LAN testing to use maximum frame rate with a defined frame size. WAN testing to use rate greater than the maximum theoretical rate.
- **Frame sizes:** The range of frame sizes recommended are: 64, 128, 256, 512, 1024, 1280, 1518 bytes. This covers the range of frame sizes transmitted.
- **Frame formats:** The format of the layer 3 frames of TCP/IP for routing and UDP Echo frame for layer 2.

3.8.1.3 The Tests

Vendors can use these tests to ensure that the Service Level Agreement (SLA) between the consumer and a service provider are met. The benchmarking tests defined in RFC 2544 document include the six tests of throughput, latency, back-to-back frames, frame loss rate, system recovery and reset. The details of these tests are:

- **Throughput:** Is given as the maximum rate at which frames are transmitted by the test equipment without any frame loss. The performance report must include the maximum frame rate, the frame size used, the theoretic line rate for that frame size, and the type of protocol used.
- **Latency:** The time taken by the networking equipment upon receiving a frame on the input port to the time the same frame is seen on the output port of the device.
- **Back-to-back:** This will measure the buffering capacity of a device. The test checks the speed at which a device is able to recover from an overload condition.
- **Frame loss:** The percentage of frames lost under a steady load. The frame loss rate calculated as shown in Equation 3.1.

$$Frame\ loss = \frac{(inputcount - outputcount) \times 100}{inputcount} \tag{3.1}$$

- **System recovery:** The test checks the speed at which a device is able to recover from an overload condition.
- **Reset:** The test checks the speed at which a device is able to recover from a device or software reset.

3.8.2 iPerf Network Benchmark Tool

iPerf [93] is a network benchmarking tool that actively measures network bandwidth on IP networks. The goal is to determine the maximum achievable throughput in the IP network. iPerf can also perform other measurements such as latency and packet loss. It can generate customised UDP and TCP packets of different frame sizes at specified rates and intervals. The tool collects statistics based on the time interval of the number of packets sent. The transmission of TCP or UDP traffic involves a client and a server where the client generates traffic and is sent to the server. At the server, the traffic is then analysed and measured. iPerf can also measure throughput at the application level (Layer 5) of the TCP/IP model. Additionally, iPerf supports bidirectional transmission between clients and servers.

3.9 Summary

In this chapter, the discussion explored the components of an OpenFlow-enabled network. OpenFlow is an open standard that enables software controllers to remotely manage OpenFlow switches. However, the standard does not define a northbound API. The implementation of the northbound API is left to the design of the OpenFlow controller. As seen in Section 2.3.3 and Section 3.7.5, there are a number of software controllers which implement different NBI protocols such as REST API or the OSGI model and the implemention also differ in programming languages which gives flexibility to choose a controller based on application. The chapter introduced multiple network processing technologies such as ASICs, FPGAs and general-purpose CPU used in implementing an OpenFlow-enabled switch. Each technology affects the manner in which the OpenFlow pipeline is implemented, which also affects the cost of building an OpenFlow switch. The architecture of general-purpose CPUs offers the least cost and greater flexibility when it comes to implementing an OpenFlow pipeline for switches.

This chapter also introduced Open vSwitch (OVS), a popular virtual switch in SDN applications. OVS is a multi-layer software switch that runs on general-purpose CPUs and it implements a multi-level caching for its packet processing. OVS in conjunction with Data Plane Development Kit (DPDK) provides accelerated packet processing. DPDK enhances packet data rates by reducing the overhead of executing processing in the kernel.

The chapter briefly examined SDN controllers and the high-level interaction between controllers and the management plane. Understanding the functionality of SDN controllers forms the basis for the knowledge required in building an SDN network.

Chapter 4

Designing the Overall SDN Network

The previous chapter provided an overview of different network processing technologies implemented in OpenFlow devices. From these technologies, general-purpose CPUs offer the most flexibility in terms of programmability and is the cheapest solution for implementing OpenFlow devices (Section 3.2.2). This dissertation aims to develop a multi-layer SDN switch that makes use of open source tools and commodity hardware as a result, creating the underlying hardware for SDN networks. At the time of starting this thesis, there were several high-speed SDN devices, but no implementations that offered cheaper solutions for small networks.

This chapter gives a description of the overall architecture that was implemented. The implemented switch was then later used in several use cases. In Section 4.1 outlines the overall architectural design considerations for this thesis. Section 4.2 lists the objectives drawn from these considerations. Section 4.3 introduces the overall architecture and designhosted by the prototype multi-layer switch.

4.1 Design Considerations

Prior to the development of programmable networks and the implementation of several use cases of SDN networks, it is important to define the goals of implementing both the multi-layer switch and the embedded SDN network used to test the set of use cases. SDN offers numerous possibilities, however, it may not be possible to cover all possibilities as these are outside the scope of the thesis. The objectives presented in this dissertation were motivated by the following:

1. Develop an inexpensive multi-layer switch.
2. Evaluate this device capabilities and features.
3. Use this prototype device in an SDN network.
4. Demonstrate the simplicity of management brought by manipulating the behaviour of the programmable network device within a programmable network.

The task of creating an SDN system is divided into two sub-tasks which are: (1) the selection and development of the control software and (2) the design and implementation of the multi-layer switch. The control software relates to the logic required to create and manage a network. This includes defining network addresses and the logic of the control plane. The multi-layer switch includes the design of the hardware and software of the data plane.

4.2 Design Objectives

To meet the specific challenges of designing inexpensive SDN Networks (discussed in Chapter 1), the following are the requirements:

- **Use of commodity hardware and open source tools:** This fits well because it provides an affordable means to build a multi-layer switch.
- **Simplified programming model:** As outlined in Chapter 1, the advantage of SDN over traditional networking enable the association of centralised decisions and policy making. This means that the users or applications may operate or change the behaviour of the network. This grants users and SDN applications to reactively handle the state of the network changes. The framework should provide the programmer and or application with primitives to express complex network services as applications.
- **Performance:** One important aspect of hardware implementation is performance, achieving realistic performance for the proposed prototype provides evidence that an inexpensive implementation is feasible.
- **Extensibility:** This system should be extensible in order to account for the variety of different SDN deployments.

To satisfy the design objectives mentioned above, the structure of the overall system will include key components and modules that combine to form each functional entity as seen in each of the three layers of the SDN Architecture (discussed in Section 2.3.2). These entities include an SDN enabled prototype device (data plane device), an SDN controller software (control plane) and SDN Applications (application layer). The SDN Applications will then set network policies and convey them to the SDN controller via the northbound interface (NBI) using OpenFlow messages. The SDN controller then translates these policies to OpenFlow messages for the data plane switches that define the behaviour of the network based on the policy set in the SDN Applications.

4.3 Architecture of the SDN Network

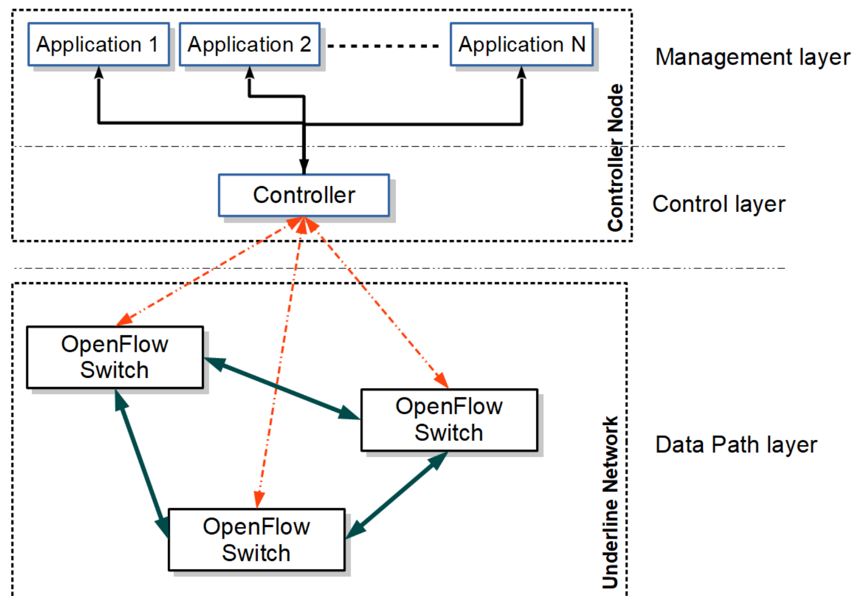


Figure 4.1: Generic Architecture Overview.

Figure 4.1 shows the diagram of the proposed SDN network which consists of a physical controller, one or more switches, and a set of SDN applications. The applications and services are located at the top layer. The controller connects to OpenFlow switches via a traditional TCP/IP network. The controller has access to the switch's configurations through the OpenFlow protocol. From the controller, the logical view and control access over the network topology at the data plane allows the applications to request information about the network. Additionally, the OpenFlow protocol enables the switch to communicate with the controller via the OpenFlow channel that exposes the switch's functionality and the handling of the controller-switch connectivity. The controller uses a high-level

specific protocol to speak with the applications. It can request and collect standard OpenFlow specific statistics (e.g. number of packets matched per flow entry) from the switches in the network. These collected statistics, along with some primitive operations as interfaces, are then made available to applications for their use. Finally, SDN applications programmatically carry out network services that will send policies through the interfaces to the OpenFlow controller.

The underline network, for this proposed platform, is configured as an Ethernet standard network (IEEE 802.3) [94]. The IEEE 802.3 is the protocol standard and frame format used for communication for Local and Metropolitan Area Networks (LANs and MANs).

When an Ethernet frame is received at the SDN switch/node, header fields are extracted from the frame. Table lookups check for matching entries in the OpenFlow table. If available, the corresponding Ethernet frame is processed and forwarded as per the 802.3 frame format. As traffic flows through the network, each switch will receive commands from the controller and deal with the traffic accordingly (Section 3.1). The remainder of the chapter covers the methodology and design of the management layer and the applications that are part of the control software.

4.3.1 Controller Selection

The controller is a functional component that enables the access to APIs that manipulate the behaviour of the network. The prototype hardware switch will need to communicate with the controller. In a previous chapter, Section 3.7.5 provided an overview of several OpenFlow controllers. This information was used in deciding which SDN controller to use. The criteria used for selection was based on the following: ease of management, use, learning curve, and availability of documentation.

For this dissertation, the controller of choice was the Ryu SDN controller. OpenDayLight was not considered due to its steep learning curve. ONOS and Faucet on the other hand lack documentation. Ryu was selected since it has good documentation with an active community keeping it updated and a moderate learning curve.

Ryu controller also exhibits other advantages. It supports multiple OpenFlow protocol versions (1.0, 1.2, 1.3, 1.4 and 1.5) [87]. It includes built-in applications and examples which are well documented. It also includes a documented book with examples showing how Ryu applications are developed and how they manipulate network behaviour while

other examples in the book show how to provide functionalities that can be consumed by other Ryu applications [13]. The proposed solution employs a single general-purpose machine (compute node) running the Ryu controller and several applications. These applications implement network policies and manipulate the underline network.

4.3.2 Configuration of the Controller

Section 3.7.3 illustrates the structure of Ryu and the interaction between an application and the controller (`ryu-manager`). The decorator application class method `ryu.controller.handler.set_ev_cls` is used to trigger *event handlers* within the application's code. These handlers implement the logic behind the control software.

4.3.3 Basic Operating Principle

The OpenFlow switch depends on the controller for directing of traffic (Figure 4.1). Unknown packets that arrive at the OpenFlow switch are sent to the Ryu controller that forwards the packet to one or more Ryu applications. Within a Ryu application, an event called `ofp_event.EventOFPPacketIn` is triggered in the *data path thread*. The event is then sent to each application's *event queue* in the `ryu-manager` process. The event loop thread created for each Ryu application then loads the event and calls the corresponding event handler.

After the event handler is called, in this case the handler for the `EventOFPPacketIn` event, the application can extract information which is used for management, discovery and monitoring of the network. Applications are able to obtain information about the network while deciding how the packet is to be processed and return this information (possibly by installing rules as flow entries in the OpenFlow table) to the OpenFlow switch. The decision process includes switching, routing and render services to creating and maintaining the network.

To achieve the objective of simplifying network management, a centralised control interface is employed. To grant the administrator easy access and control over the network, the test bed includes a web user interface or web UI that allows the administrator to manage the network remotely. The administrator is able to submit new rules through the web UI. The controller is accessed from the web UI through an API that implements the REST calls. A Ryu application then translates these REST messages (formatted in JSON) into

network policies that are sent to the controller and finally translated into OpenFlow flow table entries. These flow entries are then pushed onto the switch and the defined policy is applied to the unknown packet as well as similar packets that arrive in future.

4.4 Network Flow Pipeline

The operating of the proposed multi-layer switch is defined by network functionalities. These functionalities are used to represent the behaviour of the network. The first function is forwarding. It is defined as a layer 2 (L2) functionality where all connected host machines are within the same broadcast domain. Each network will have a subnet that defines the range of IP addresses that all host machines will be associated with. When configuring for networks with multiple subnets, routing between subnets is required, hence the second function is a layer 3 (L3) router. The third function, if required, is to automate the allocation of IP addresses for each end-host. Other non-core functions include simplify access to the configurations for the above three functions and providing that access via an interface. This is achieved using a user interface (web UI).

These mentioned functionalities are implemented as applications within the Ryu controller. A brief description of each Ryu application is given below:

1. **Base** – Provides services for the web UI and the initialisation of flow entries within the OpenFlow tables for each managed OpenFlow switch to allow the controller to capture events.
2. **L2Switch** – Provide for the L2 function. The frame format defined in the 802.3 will allow the manipulating of Ethernet traffic based on the L2 source and destination addresses (MAC addresses).
3. **Router** – Provide for L3 routing for traffic moving between different subnets.
4. **SimpleDHCPServer** – Provide for dynamic assignment of IP addresses to hosts using the Dynamic Host Configuration Protocol (DHCP).
5. **WebAPI** – A file server that provides web resources (such as HTML pages, script files and images) used in the web UI.

The pipeline of the OpenFlow table for all managed multi-layer switches is given in Figure 4.2. This pipeline shows the overall process within each manged switch. The initial

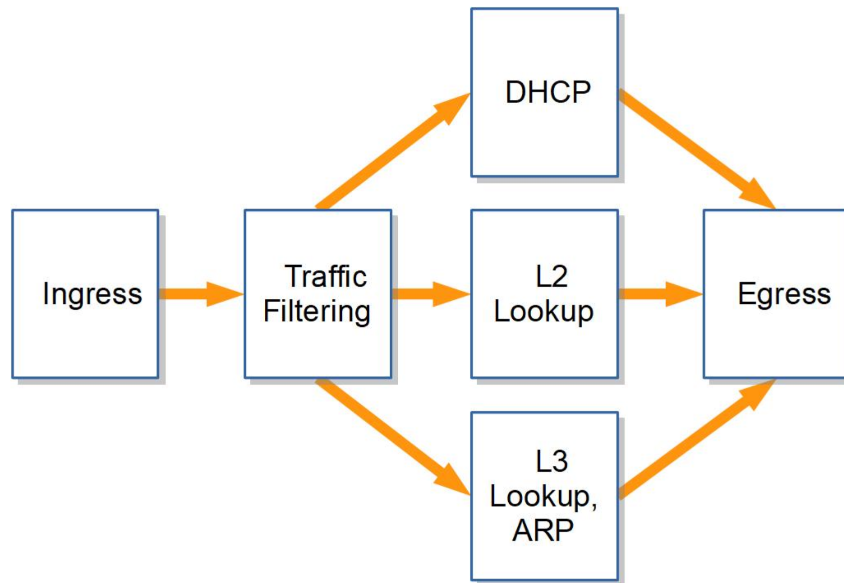


Figure 4.2: Pipeline for the Implementation of an SDN Network.

flows are OpenFlow table entries entered by the controller during its initial connection with the switch. These flows instruct the switch on how to filter traffic (filter by L2, L3 or DHCP traffic). The pipeline process implemented within the multi-layer switch can be represented as six process blocks. The pipeline process starts with the `Ingress` block. This block represents any ingress port on the OpenFlow switch. All arriving packets are filtered by the `Traffic Filtering` process block which represents the installed flow entries. These packets are filtered according to whether the network is configured with a single subnet or multiple subnets (i.e. L2 or L3 functionality) and whether the dynamic allocation of IP addresses for connected end-hosts has been enabled (i.e. the DHCP service is enabled). The filtering process performed by the `Traffic Filtering` block is achieved by classifying the traffic base on Ethernet type (EtherType) or IP protocol number. The EtherType [95] is a two octet (16 bit) field in an Ethernet frame that indicates the size or protocols encapsulated within the payload of the Ethernet frame and it defines the type of packet received. If, for example, a packet has the EtherType of IPv4 or ARP, the IP protocol number will provide more information used to further classify the traffic. The classification of IPv4 traffic can show us if the packet is of type Transmission Control (TCP) or User Datagram (UDP) (It is worth noting that DHCP traffic are encapsulated within a UDP datagram). The information gathered is then used to process the traffic accordingly (`DHCP`, `L2 Lookup` and `L3/ARP Lookup`). Once filtered, the packet may processed by either the “DHCP”, “L2 Lookup” or “L3/ARP Lookup” which are processed by the *SimpleDHCPServer*, *L2Switch* and *Router* applications respectively. These three blocks heavily depend on the involvement of the controller. After each application has

completed processing, the packet is then sent out the egress port represented by `Egress` block.

The following sections show the design of the applications that control the network node, that is essentially the multi-layer switch.

4.4.1 Base Application

The first Ryu application implemented in the proposed solution is the **Base** application. The group of functions provided by the **Base** application are: (a) populate the OpenFlow table with initial OpenFlow rules for all managed OpenFlow switches and (b) provide a web interface to configure the functions of the managed switch. As shown in the pipeline (see Figure 4.2), this requires the switch to categorise traffic according to function. The categorisation of traffic that travels within the network is done by packet classification based on a set of rules defined as match fields and actions.

The other function performed by the Base application is to provide a web UI that allows the user to view and configure the network. The user is able to select and add the desired rules for forwarding (L2), routing (L3) and assigning of IP addresses. The user is also able to view the OpenFlow table entries within each managed switch from the web browser. The topology and table display functionalities are adopted and modified from the built-in application *TopologyAPI* found in the application `rest_topology.py`. Figure 4.3 shows the structure of the proposed solution. The configuration of an SDN switch can be done from the web UI accessible from a web browser.

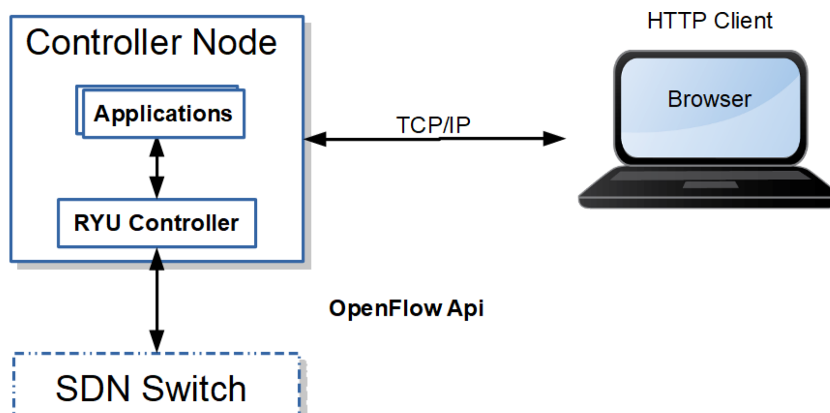


Figure 4.3: Configuring of SDN Switches Via the Web UI.

4.4.2 Forwarding Application

The second Ryu application implemented is the **L2Switch**. This application is concerned with the communication of devices within the same layer 2 broadcast domain (same subnet). To understand the function of the **L2Switch** application, taking a look at a traditional layer 2 switch will explain the behaviour of this application. A traditional layer 2 switch performs the following tasks:

- Learns the MAC addresses of connected hosts on each port and saves it in a MAC table.
- Packets addressed to a known host are forwarded to the port connected to the host.
- Packets addressed to an unknown host are flooded to other ports.

Layer 2 Forwarding in Ryu

OpenFlow switches and the Ryu controller implement a layer 2 switching functionality by having the OpenFlow switch performing the following instructions:

- Modify the address of the received packet or send the packet from a specified port.
- Send the unknown packet to the controller (Packet-In).
- Send the packet from the controller through the specified port (Packet-Out).

The performing of these tasks and instructions achieves L2 switching. The Packet-In function is important for the learning of MAC addresses. The controller uses Packet-In to receive packets from the switch. Information about the host and connected port is used in the learning of the MAC table. Once learnt, the switch is able to correctly send the received packets. The destination MAC address of the packets is then used to determine the next action. The action taken is dependent on if the host is known or unknown defined as follows:

- If the host is known or has been learnt, the Packet-Out function is used to transfer the packets to the port to which the target host is connected.
- If the host is unknown, the Packet-Out function performs flooding.

4.4.3 Routing Application

The **L2Switch** application can forward traffic for a single network. Inter-networks are created by joining two or more networks at layer 3. The communication between these networks would require a device with routing capabilities [96]. The implementation of these capabilities are managed by the application **Router**.

The IP protocol of the TCP/IP model [35] is designed to facilitate the routing of information over multiple networks. Within the TCP/IP model, a router is able to transmit data from one network to the next as routers exchange critical information necessary for deciding the next path to send the data to.

4.4.3.1 IP Addressing

The primary function of Internet Protocol (IP) is to deliver data across an internetwork [35]. IP addresses have two different functions: (1) provide a unique identification of a network interface and (2) provide a system to route data.

The second function facilitates routing. IP addresses can be used by routers to figure out what to do with a packet based on the IP address. Related information such as subnet mask and gateways are used in routing. A device may have at least one IP address (one per network interface). End devices like hosts such as computers and network printers usually have one IP address whereas routers have more than one IP address (one IP per each port).

4.4.3.2 Subnet Mask and Default Gateway

In subnetting or classless addressing, the subnet mask is required to qualify the address. The mask is then used to identify the network ID and host ID. The default gateway generally is the IP address of the router that provides default routing functions. This router is used when a router is unable to see the destination IP in its local network. The default gateway will then take care of routing functions.

4.4.3.3 Address Resolution

Address Resolution Protocol (ARP) [35] is used by devices to request/reply IP addresses to find out which hardware interface (MAC address) corresponds to a specific IP address.

ARP is used to find a node's MAC address when its IP address is known. A broadcast of an ARP packet is sent by the sender. This ARP packet contains the IP address of the node with the unkown MAC address. Once sent, the sender waits for a response containing the MAC address. This is then stored in a cache for later use.

4.4.3.4 Implementing the Router Logic

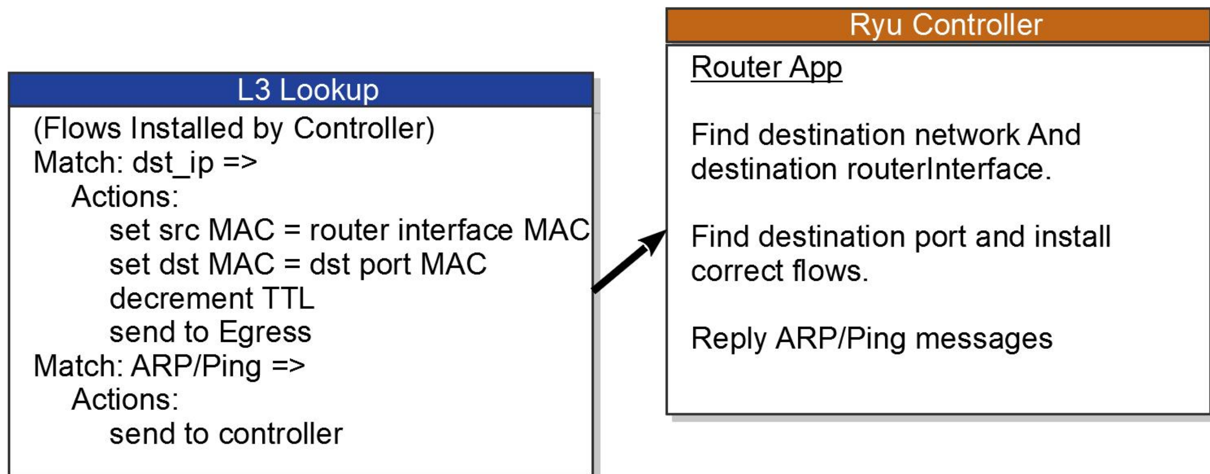


Figure 4.4: Layer 3/Routing Lookup.

The logic behind the router is adapted from the example Ryu application *RestRouterAPI* found in the `rest_router.py` file. In the proposed solution, the functionalities mentioned above (subnet masking, gateways and address resolution) form part of the application **Router**. Figure 4.4 shows the L3 lookup within an OpenFlow switch during the routing of IP addresses. Each router uses a forwarding table that resides and maintained by the module/class *RouterAPI* within the application. A router, by right, should forward packets by examining the destination IP (`dst_ip`) extracted from each packet's header. The `dst_ip` value is used to index into the router's forwarding table. Other values stored in the forwarding table are the *'netmask'* and *'gateway IP'*. These values are used to indicate the router's outgoing interface that a packet is forwarded.

VLAN support

The virtual local area network (VLAN) functionality was adapted from Ryu's example application and forms part of the *RouterAPI* implemented in the application **Router**.

VLAN [96] enables the configuring of multiple virtual local area networks to be defined over a single physical infrastructure. Hosts within a VLAN are able to communicate with only hosts within the same VLAN. VLANs have advantages of traffic isolation, efficient use of network devices and the management of tenants.

4.4.4 DHCP Service

SDN allow users to coordinate and manage a network by automating the behaviour of the network. A key aspect in automating TCP/IP networks is the allocating of IP addresses. IP addresses enable devices within the network to be able to identify each other. As part of the process of automating IP allocation, dynamic IP allocation is used. The configuring of IP addressing is done using the Dynamic Host Configuration Protocol (DHCP protocol) [15]. The application **SimpleDHCPServer** serves this purpose. The workings of the DHCP protocol is elaborated next.

4.4.4.1 Overview of DHCP Features

The allocation of addresses for hosts through manual or automated methods may be used in creating a network. The manual allocation of IP addresses is achieved by configuring the IP address for each the host by manually adding the IPs within the operating system settings. This can become a cumbersome task especially when updating or reassigning IP addresses for hosts in a large network. The second method is the assigning of IP through the support for a dedicated server that dynamically appoints addresses for hosts. This server, known as the DHCP Server, uses the hardware addresses (MACs) to assign IP addresses. The DHCP makes use of a pool of IP addresses for the allocation of addresses. The DHCP server still supports static mapping of addresses where it is required.

4.4.4.2 Address Assignment and Allocation Mechanisms

As mentioned before, the DHCP protocol allows assigning of IP addresses from a shared pool managed by the DHCP server. The time which an IP address is assigned is either chosen by the server or until the client informs the DHCP server that it no longer requires the address. The administrator sets a range or set of ranges of IP addresses that are available in the pool which is managed by the DHCP server. Clients configured to use DHCP will contact the server to request an IP address. The server then decides the time

for leasing an IP address and offers the leased free address from the pool to the client. Upon expiry, clients will either renew the lease or is assigned a new one. Below are the are some of the benefits of using DHCP:

- **Automatic Assignment:** IP addresses are assigned without administrator intervention.
- **Centralized Management:** All the IP addresses are managed by the DHCP server using a shared pool. The DHCP server can ensure that the pool of IP addresses is updated when an IP address is leased.
- **Conflict Avoidance:** IP addresses managed by the DHCP server are selected from a pool and conflicting addresses are avoided.

There are two choices that are available in the implementing of the DHCP server with SDN. The first choice is to use a dedicated DHCP host server and the other is a DHCP service running as an application from the SDN controller. In the first option, the DHCP server is connected as an end-host with all DHCP messages being forwarded to the DHCP server. Switches and routers are to forward DHCP traffic to this server. This option is seen in large networks such as data centres. The second option is where the DHCP service runs as an SDN application. This is similar to the DHCP services seen in smaller network implementation such as a home or office router. This solution would allow administrators to have access to DHCP settings along with other network configurations. The test bed utilises the second option and is illustrated over the next few sections.

4.4.4.3 DHCP Process

The process behind the DHCP protocol illustrated in Figure 4.5 shows the timing relationships for a typical client-server interaction. The allocation of a network address which is initiated by the host sending a broadcast DHCP_DISCOVER message on its local subnet. The message is captured and the DHCP server responds with a DHCP_OFFER which includes configuration parameters. The host then sends a DHCP_REQUEST, requesting the offered configuration. Finally, the DHCP server acknowledges with DHCP_ACK, including the committed network address.

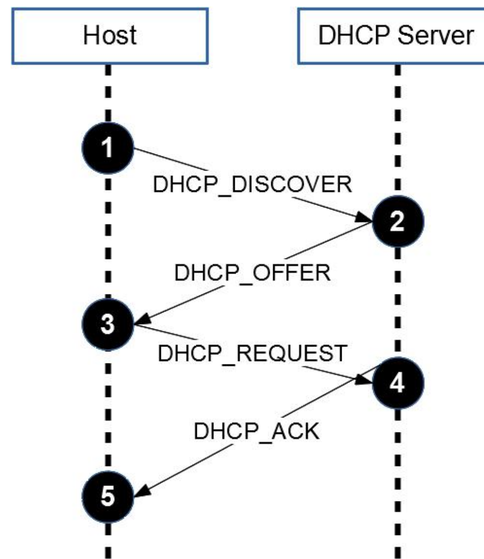


Figure 4.5: DHCP Process. Source: [15].

4.4.4.4 Implementing DHCP in SDN

Upon connecting to the controller, the **Base** application (Section 4.4.1) installs an OpenFlow rules to capture and send DHCP traffic to the controller. These rules allow all matching DHCP traffic to be handled within the controller by the **SimpleDHCPServer** application. The DHCP application, when enabled, will then handle the received DHCP packets and reply accordingly (DHCP process shown in Section 4.4.4.3).

Figure 4.6 illustrates the flow of DHCP traffic within the proposed architecture. The process is implemented as follows:

1. A host sends a DHCP_DISCOVER message.
2. The message is then classified as DHCP and forwarded to the controller.
3. The host will then sends a DHCP_REQUEST message.
4. The DHCP application then sends a DHCP_OFFER message back to the host.
5. Again at the OpenFlow switch, it is classified as a DHCP packet and forwarded to the controller.
6. The DHCP application then acknowledges the request and sends a DHCP_ACK message.
7. The host now has a configured IP.

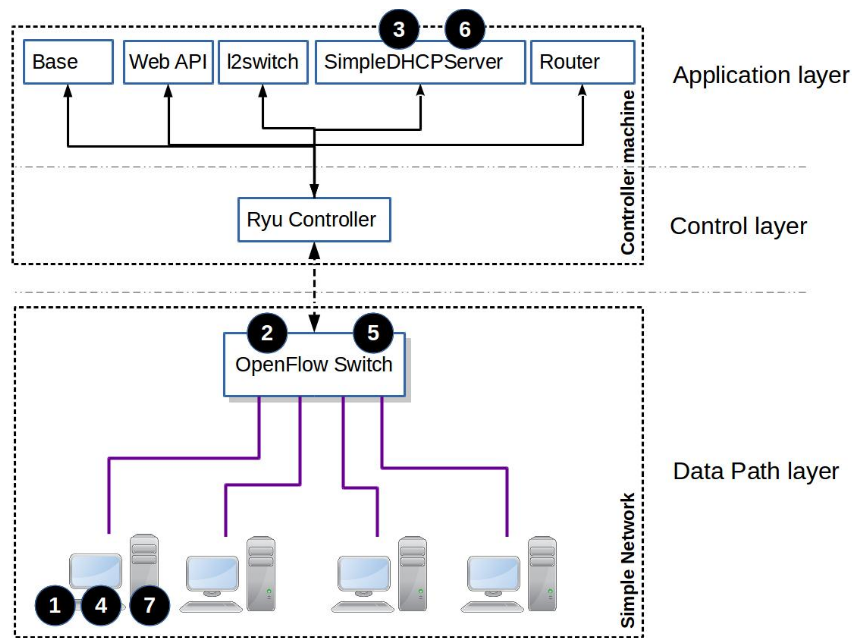


Figure 4.6: DHCP Flow Process in Proposed SDN Architecture.

4.4.5 WebAPI Application

The final Ryu application implemented is the **WebAPI**. The purpose of this application is to provide web resources that are used in the displaying of the web UI. This application works in conjunction with the **Base** application. Each time the web UI webpage is requested, the HTML web files along with the required scripts and image resources are sent back. The web UI contains a form to allow the entering of policies. Hence the sole purpose of this application is to serve up a file server to allow access to web resources to the users of the web UI

4.5 Summary

This chapter introduced the structure of the proposed architecture. The architecture consists of a controller, network applications and SDN switches. The controller is the core component in this implementation. The applications communicate with the switch via the controller. The controller functions were divided into five applications which focus on certain specific roles.

The chapter also introduced the network flow pipeline for SDN switches. The flow pipeline described is implemented by the applications in the controller. The logic of the pipeline

is then implemented as flow entries. These flows define the process that the switch would follow, governing the behaviour of the switch. The behaviour of the switch would, as first approximation, boils down to three network functions: DHCP service, L2 and L3.

Chapter 5

Designing the SDN Switch

The previous chapter gave the overall architecture introducing the design of the control software which includes the Ryu controller and applications, i.e. the control plane and management plane of the proposed architecture. This chapter describes the design of the data plane hardware switch and the components used in developing the hardware. One of the main objective for this thesis is to provide an inexpensive way of integrating an OpenFlow device targeted for small-to-medium scale scenarios. The development of such a device required a well thought structured model. The architecture of SDN presents functional layers, which offer a guideline for the approach of developing physical implementations.

This chapter is structured as follows. The chapter begins by covering a list of guidelines that were considered in designing the hardware prototype. Section 5.1 lists the design guidelines. Section 5.2 covers the hardware architecture for the OpenFlow switch. Thereafter, Section 5.3 describes the software components used to build the switch and covers the interfaces between them to meet the objectives defined. Finally, to verify that the developed prototype is able to accomplish realistic performance, Section 5.4.1 discusses the tools involved to evaluate the performance and capabilities, and how they are implemented.

5.1 Design Guidelines

The guidelines that underpin the design of the data plane node were derived from the objectives presented in a previous chapter. The guidelines are as follows:

1. Inexpensive.

2. Reasonable performance and compatibility within SDN network environment, i.e. does it meet OpenFlow specifications?.

3. The level of ease of reproduction.

The level of performance of a system is affected by the quality of hardware used in the development of the prototype: higher performing hardware has a higher cost. In effort to keep the cost reasonable, commodity hardware was used. To evaluation of performance and compatibility, the resulting prototype underwent a series of tests. Finally, the use of commodity hardware and open source components provide the means for reproducing the prototype allowing interested users a chance to create their own network equipment.

5.2 Hardware Architecture

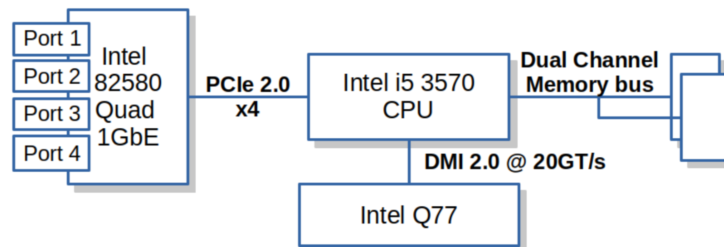


Figure 5.1: SDN Prototype Switch Block Diagram.

The hardware architecture of the SDN prototype switch is shown in Figure 5.1. The architecture shows a four port network card interconnected to a commodity desktop machine via the PCIe bus. The figure also shows a general-purpose CPU used as the processing hub of the SDN prototype switch. Other components that used includes a Motherboard and Main Memory.

To ensure that the hardware is optimised, by closely examining each component for any bottlenecks is crucial in the creation of a performant switch. Discussed later in the next section, the requirements of OVS-DPDK includes the transferring of packets between the PMD threads and the network card. Therefore, the data bus speeds between the network card and main memory will have an effect on the overall performance of the switch.

The Peripheral Component Interconnect Express (PCIe) v2.0 bus supports a speed of 5 GT/s (Giga-Transfers per sec) with an effective speed of 500 MBytes/s (4 Gbits/s) per

lane [97, 98]. The network card uses a total of four PCIe lanes. Packets transmitted between memory and network card pass via the PCIe and CPU. The CPU also can affect the performance where several features found in its architecture may boost networking application [99]. The functionality and requirements of each component in Figure 5.1 is discussed in more detail the next four sections.

5.2.1 Network Card

The Intel® 82580 Quad 1GbE network card interface (NIC) has four 1 Gbps, full-duplex Ethernet ports. It links to the system via the PCIe bus over four lanes for the full bandwidth. This NIC was selected because it supports the DPDK drivers [100] which are used to implement the data path in user space. To accommodate the maximum port speeds on all ports, the network card was connected to a four lane PCIe bus to avoid bottlenecks between the NIC and the main memory. The required bandwidth for four full-duplex ports is a data rate of 8 Gbps calculated as shown in Equation 5.1, where 4 is the number of ports and 2 represents bi-direction traffic at 1 Gbps. The NIC uses the bus to also send descriptors of packets [101].

$$4\ (ports) \times 2\ (bi - direction) \times 1\ Gbps = 8\ Gbps \tag{5.1}$$

In this implementation of the switch, the number of ports on the Intel® 82580 NIC are used as the number of ports available for use on the SDN switch. An extra port is required for the management of the switch is provided by the NIC that is available on the motherboard. This is the network interface that the switch uses to communicate with the controller to receive instructions.

5.2.2 General-purpose CPU

The purpose of this component is to provide the processing logic. The DPDK library includes drivers called Poll Mode Drivers (PMDs) which process packets as they arrive. Packets arriving at the Intel® 82580 NIC are made available to OVS for processing. Each port on the Intel® 82580 NIC has 8 ingress queues and 8 egress queues. Each Enabled ingress queue creates a PMD thread that shares workload where each thread manages a separate ingress/egress port [82, 101]. The performance of OVS is improved further

by pinning PMD threads to free cores. The CPU's multi-threaded architecture allow faster packet processing as multiple packets are processed simultaneously on each thread. The DPDK library also supports hugepages which boosts the switch's performance by improving the memory hit rate. Hugepages [102] allow the CPU to efficiently allocate memory chunks for programs. These chunks are called pages. When hugepages are enabled, a CPU can allocate larger pages (chunks) of memory to a program therefore requiring a smaller number of pages to store data of the same size. For example, a process using 1 GB of memory would require 262144 lookup entries for a 4 KB page (1 GB/4 KB), whereas 2 MB hugepage would only require 512 entries (1 GB/2 MB). Fewer page numbers requires less time to search and locate where the memory is mapped.

5.2.3 Main Memory

Memory is another influencing factor on the performance of the system. Memory speed may be measured in terms of how many data transfers are made per second. In newer computer systems, the overall bandwidth of the amount of memory transferred is improved by transmitting data across multiple memory channels. More memory channels offer higher memory throughput to the CPU or reduces memory latency [103]. For example, if a single channel transmits 64 bits at a time, a dual-channel memory would mean that data is transferred in chunks of 128 bits [104, 105]. The number of channels and the overall bandwidth for the memory affects the transfers between the NIC and the memory and how fast the network packets are fed to the CPU.

5.2.4 Motherboard

The motherboard is what connects the NIC, CPU and memory together. By building an OpenFlow switch using a general-purpose CPU, avoiding any bottlenecks by optimising component placement within the system will ensure that this prototype is able to achieve desirable performance. Looking at Figure 5.1, the link between the NIC, CPU and main memory does not pass through the chipset. Therefore minimising the distance between the NIC and main memory, i.e. minimising the path that data travels improves performance. The motherboard also defines the memory channels that are available. The total memory bandwidth for this implementation is given in Equation 5.2.

$$Total\ Memory\ Bandwidth = Memory\ Type \times Max\ Number\ of\ Memory\ Channels \times Memory\ Bus\ Width \tag{5.2}$$

5.3 Software Architecture

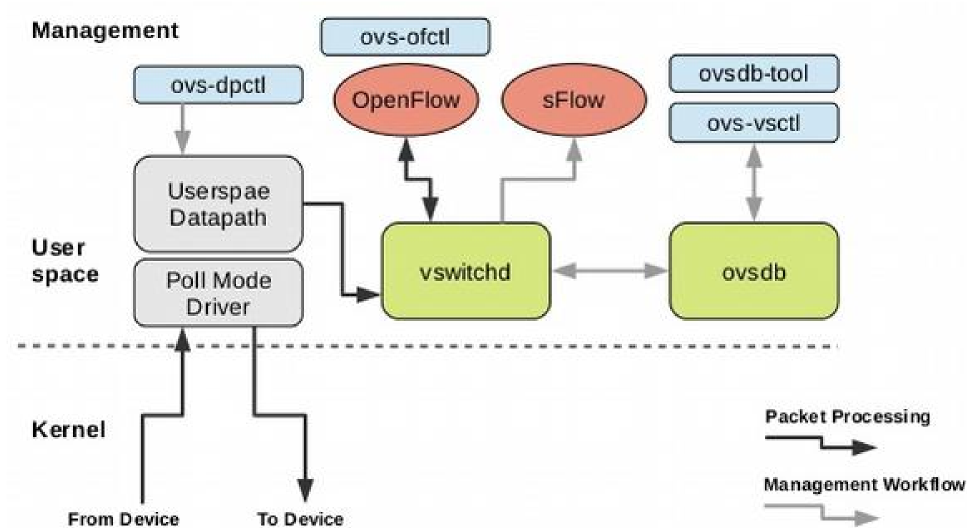


Figure 5.2: Architecture with DPDK. Source: [16].

The software architecture of the prototype SDN switch is shown in Figure 5.2. The software includes OVS-DPDK that run in user-space (Section 3.6). The core function of the switch consists of two main components, `ovs-vswitchd` (vswitchd) and `user-space data-path` are located in the user-space as shown. These two components represent the program that maintains the OpenFlow tables and defines the core functions of the switch (`ovs-vswitchd`) and the program that handles packet processing (`user-space data-path`).

The previous chapter illustrated how SDN applications and the controller may be designed and used for the management of OpenFlow switches. As discussed, the controller is responsible for managing flow control by adding flow entries to the switch. These flow entries are then saved in the OpenFlow table maintained by `ovs-vswitchd`. All arriving packets are then processed by the `user-space data-path` that extracts the information from the packet and creates match fields and applies flow actions. These actions are set by `ovs-vswitchd` program and cached in the `user-space data-path`. To recap from Section 3.4.2, the `user-space data-path` implements the 'fast path' and non-cached actions

would result in queries sent to the `ovs-vswitchd` ('slow-path'). The `ovs-vswitchd` program then searches the flow tables and then sends actions to the `user-space data-path` that then applies the defined action. However, when an entry is not found in the flow table, the table miss entry is executed. In this implementation, the table miss action is set to send these packets to the controller. Packets that are sent to the controller are defined as *Packet-In* events where SDN applications handle these packets accordingly. After processing, the packets are sent back to the multi-layer switch (i.e. *Packet-Out* events). During processing of these packets, SDN applications make use of the packet header information to install OpenFlow entries which are sent to the `ovs-vswitchd` program. The `ovs-vswitchd` program then uses these OpenFlow entries to define actions for future arriving packets which are then processed by `user-space data-path`. This process is repeated for all arriving packets.

The `user-space data-path` maintains cached actions to efficiently process packets without unnecessary overhead (Section 3.5). A delay overhead is seen each time a miss occurs in the `user-space data-path`'s EMC, Megaflow or `ovs-vswitchd` table, where the processing of the packet finally occurs at the control plane (SDN controller) or management plane (SDN applications).

5.4 Tools to Evaluate the Performance of the Switch

This section describes tools used to evaluate the performance, features and characteristics of an OpenFlow-enabled Ethernet switch. (The results of the evaluation tests are shown in a later chapter, Chapter 7). The goal is to report on the performance and level of OpenFlow compliance of the SDN switch as characterised by the OpenFlow 1.3 specification document.

The evaluation of the peak packet transfer rate and layer 2 processing corresponds to the loopback forwarding setup from the DPDK library and layer 2 MAC address flow entries respectively. The evaluation of the layer 3 performance uses the IP address as the match field for the flow entries. The loopback test will show the peak maximum transfer rate that the hardware (this includes the CPU, memory and bus transfers) is able to achieve which is also known as the Input/Output (IO) rate. This will help in identifying any bottlenecks within the interconnection between the NIC, CPU and memory prior to any packet processing. While layer 2 or layer 3 benchmarking evaluates the behaviour during packet processing. Testing the layer 2 and layer 3 packet processing evaluates the

processing capability of the CPU. This will help identify if the CPU is the bottleneck of the system. Ultimately, results for layer 2 and layer 3 will show the behaviour of the switch during layer 2 and layer 3 functionality. The evaluation of packet processing using MAC and IP addresses shown in Table 5.1 will allow the use of OpenFlow flow entries, hence initiate packet processing using the OpenFlow pipeline within the SDN switch.

Table 5.1: Exanple MAC and IP addresses to evaluate an OpenFlow switch.

Port Number	MAC Address	IP Address
Port 1	00:00:00:00:00:01	10.0.0.1
Port 2	00:00:00:00:00:02	10.0.1.1

The MAC or IP addresses for ports 1 and 2 are used as the match fields for the OpenFlow rule. The resulting actions are the forwarding of Ethernet packet based on the destination MAC or IP address.

5.4.1 Benchmarking

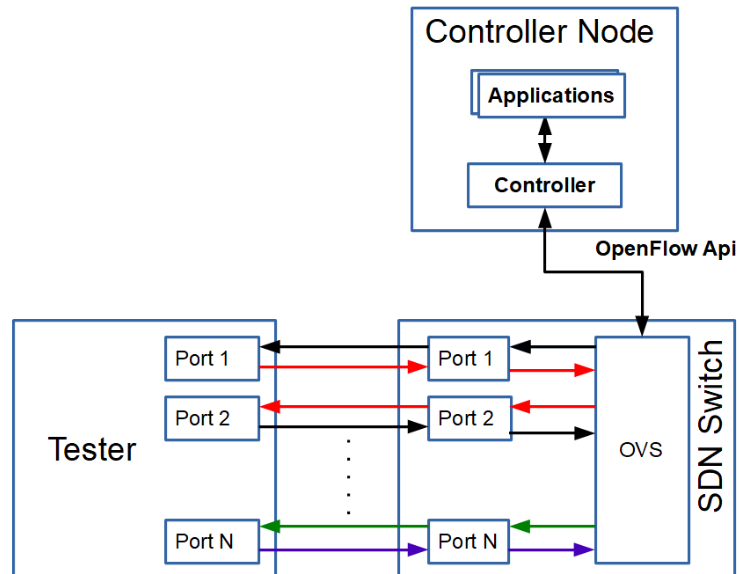


Figure 5.3: OpenFlow Switch Benchmark Setup.

Section 3.8 provided the overview of the methodology developed by the IETF for the evaluation of an Ethernet device [92]. Figure 5.3 shows the setup used to evaluate the switch with four ports. The connections are made from the egress ports of the tester to the ingress ports of the switch and from the egress ports of the switch back to the tester. The tester sends packets to the switch which are processed by the `user-space data-path`

from OVS-DPDK architecture. From the documentation, the performance evaluation of the OpenFlow switch includes using multiple frame sizes to characterise its performance over a range of multiple frame sizes. The range defined for these sets of tests describe frame sizes from the smallest (64 byte) to largest (1518 byte) frame size. Table 5.2 shows the frames type and sizes used for the evaluation. The report includes the measurement of the latency, frame loss and maximum throughput. The latencies measured were the average latency, maximum latency and jitter.

Table 5.2: Test Frames and OpenFlow Match Fields

Test	**Frame Size**	**OpenFlow Match Field**
CPU & IO	64, 128, 256, 512, 1024, 1280, 1518	Port number
Layer 2	64, 128, 256, 512, 1024, 1280, 1518	Src and Dest MAC addr.

Table 5.3 shows match fields used for creating matches for the entries. The EtherType (`dl_type`) is set to 0x0800[1], this applies to filter IPv4 traffic. The destination MAC address (`dl_dst`) field is used to match frames with a specified MAC destination address. Using Table 5.3 as reference, all Ethernet frames with the destination address of `00:00:00:00:00:01` and `00:00:00:00:00:02` are forwarded to port 1 and port 2 of the switch respectively. Likewise, `nw_dst` filters packets using the IP destination. Therefore, for `10.0.0.1` or `10.0.1.1` addresses the frames are forwarded to port 1 or port 2 of the switch. By making use of these flow entries, the tester is able to receive the frames that are sent to the switch.

Table 5.3: OpenFlow Switch flow entry

Test Frame	**Flow Match**	**Action**
Layer 2	dl_type=0x0800,dl_dst=00:00:00:00:00:01	output:1
Layer 2	dl_type=0x0800,dl_dst=00:00:00:00:00:02	output:2
Layer 3	dl_type=0x0800,nw_dst=10.0.0.1	output:1
Layer 3	dl_type=0x0800,nw_dst=10.0.1.1	output:2

The key differences between between layers 2 and layer 3 are: layer 2 frames make use of the MAC address for forwarding and identifying of network nodes within the same network while layer 3 use IP addressing to determine the routing of packets between directly or indirectly connected networks (same or different subnet) – where the IP address identifies interfaces on each node. Layer 3 requires techniques that translate IP addresses to MAC addresses (e.g. ARP protocol), the use of *gateways* and *subnetmasks* for routing and setting of network boundaries. For this reason, the layer 3 benchmarking was also

[1] 0x800 is the hexadecimal EtherType for IPv4, https://www.iana.org/assignments/ieee-802-numbers

extended to measure the bandwidth between end-hosts using a network benchmarking tool.

5.4.2 Network Benchmarking

Performance measurement and validation of OpenFlow Compliance is important to ensure that the prototype switch follows the OpenFlow protocol specifications and that the performance is reasonable. Additionally, these tests reveal general behaviour and tendencies of the prototype switch during traffic forwarding and routing. Therefore, the iPerf tool was used to measure the bandwidth of the multi-layer switch under different network use cases.

iPerf [93] was used to determine the maximum achievable throughput, latency and packet loss in an IP network. Each host machine connected to the a network switch is set up to function both as a client and server, hence it measures the bandwidth for bi-directional traffic. The benchmarks were ran using TCP packets at the maximum rate with the interval set to at least 60 seconds. The collected statistics for the generated traffic was then analysed to evaluate the throughput of the network.

5.4.3 Compliance Testing

The OpenFlow 1.3 specification document [6] defines the requirements for an OpenFlow Logical Switch. This document describes multiple functions that are required, optional or experimental. Functions marked ‘Required’ are mandatory while ‘Optional’ and ‘Experimental’ represent additional functions and future OpenFlow features that a switch developer may implement. For this dissertation, the Faucet SDN controller comprehensive test suite [90] was used to test and verify these features. This testing suite validates OpenFlow features defined by the specification and also runs benchmarking tests such as layer 2 switching and layer 3 routing. The test suite runs as a docker container as shown in Figure 5.4.

5.5 Summary

This chapter focused on the design of both the hardware and software of the prototype device. The chapter began by discussing the design for the selected hardware comprised

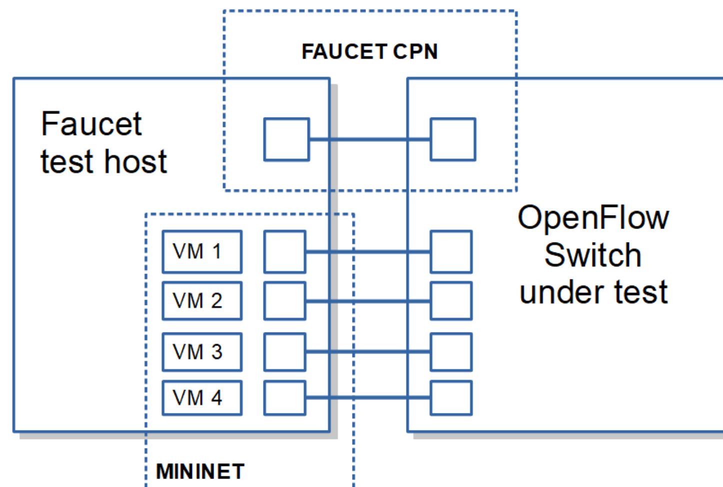


Figure 5.4: Hardware Switch Testing Setup.

of the Intel® 82580 Quad 1GbE network card where the NIC was directly interfaced with the CPU to fully take advantage of the system defined in Section 5.2.

The discussion also covered the behaviour of the software architecture. The architecture consists of two main components, the `ovs-vswitchd` and `user-space data-path`. The program `ovs-vswitchd` maintains the flow table and other core functions while the `user-space data-path` performs the actual processing of traffic.

The remainder of the chapter focused on the design structure for the evaluation of the prototype device. This covered the approach for evaluating the switch. The benchmarks would begin with the IO measurements to evaluate interconnections between the NIC and the system. Then the evaluation of L2 and L3 functionality which measures the performance of the CPU. Lastly, evaluating the OpenFlow functionality of the prototype device evaluates the features supported by the hardware. Results of the actual evaluation of the switch are reported in Chapter 7.

Chapter 6

Setting the Stage for Testing

This chapter details the implementation of the proposed architecture for an SDN network. The aim of the discussion is to demonstrate the implementation of functions that satisfy the flexibility and easy management of networks. Several implementations of SDN use cases were presented that made use of layer 2 switching, routing and network partitioning. The logic behind these use cases was implemented in the controller as a series of applications that are managed from a web interface.

6.1 Overview

The implementation described here begins with the bottom layer, the data plane, and covers the development of a multi-layer SDN switch that was used to create SDN networks. This device was constructed from commodity hardware and open-source software. Once the multi-layer switch was built and configured, it was tested to inspect the performance under load. Afterwards, the switch was then paired with an SDN controller that hosted a set of applications. The rest of the chapter is dedicated to describing in greater detail how each SDN application was developed and to illustrate how the applications inter-operated with each other to fulfil the objective of simplify network management.

6.2 Multi-layer Switch

This device is responsible for forwarding packets based on MAC address as well as IP address. It runs on a general-purpose machine that processes traffic as well as receive

instructions from an SDN controller.

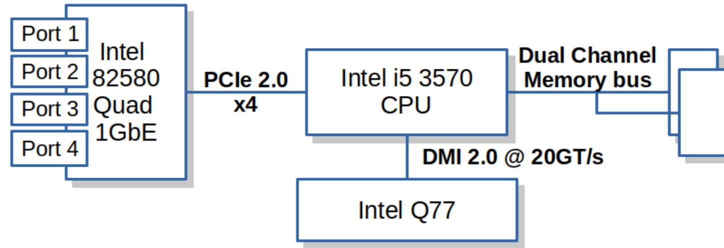


Figure 6.1: Multi-layer Switch Interconnection Design.

Table 6.1 shows the specifications of the multi-layer switch while the Figure 6.1 shows the hardware architecture of the switch. The switch consists of a multi-threaded CPU running Ubuntu 17.10 Server edition. The device has five Ethernet ports, four of which were used for forwarding and routing. The remaining one was used as the management port for the switch. The software on the switch is Open vSwitch 2.9.90 and DPDK 17.11.3 compiled using GCC 7.2.0.

Table 6.1: Hardware & Software specifications for the Multi-layer Switch.

Item	**Description**
Platform	Intel® DQ77MK
Chipset	Intel® Q77 chipset (formerly Panther Point)
CPU	Intel® Core™i5 3570 (6MB Cache, 3.39GHz), 4 core, 4 threads
Memory	2x4GB, Dual Channel @ 1333MHz, Memory Bandwidth @ 170.624 GBits/s
Operating System	Ubuntu Server 17.10 (GNU/Linux 4.13.0-39-generic x86_64)
NICs	Intel® 82580 Quad 1GbE (PCIe 2.0), Intel® 82574L Gigabit Ethernet, Bus Bandwidth @ 16 GBits/s
BIOS	MKQ7710H.86A.0054.2012.1120.1444
Open vSwitch	2.9.90
Intel® Ethernet Drivers	igb - version 5.4.0-k
DPDK version	17.11.3
GCC version	(Ubuntu 7.2.0-8ubuntu3.2) 7.2.0

The bus and memory bandwidth were calculated using equations from Section 5.3. The bus speed is given as 16 GBits/s (shown in Equation 6.1) while the memory bandwidth was calculate to be 170.624 GBits/s where 1333MHz is the Memory speed, 2 represents Dual Channel and 64 is the bus width of the memory module (shown in Equation 6.2).

$$16\ GBits/s = 4\ PCIe\ lanes \times 4GBits/lane \tag{6.1}$$

$$170.624\ GBits/s = 1333MHz \times 2\ Channels \times 64\ Bits \tag{6.2}$$

On the multi-layer switch, the motherboard's NIC (Intel® 82574 NIC) is reserved for communication with the controller while the ports on the added network card (Intel® 82580 NIC) are used as regular data ports.

6.3 Controller

This software allows centralised control of the multi-layer OpenFlow switch, a key characteristic of SDN networks. The Ryu controller was the controller of choice. Discussed later in this chapter (Section 6.6), The Ryu controller uses the Python programming language and its applications are written as Python scripts. Table 6.2 shows the specifications of the machine hosting the controller. The controller runs on a multi-threaded CPU running Ubuntu 16.04 LTS Desktop environment. The management network, i.e the network that links the controller to the switch, is a standalone.

Table 6.2: Hardware & Software Specifications for the Controller.

Item	Description
Platform	H170 PRO GAMING
Chipset	Intel® 760G (780L)
CPU	Intel® Core™i5-6400 CPU (6MB Cache, 2.70GHz), 4 core, 4 threads
Memory	2x4GB, Single Channel @ 2133MHz
Operating System	Ubuntu Desktop 16.04 LTS (GNU/Linux 4.15.0-39-generic x86_64)
NICs	Intel® 82580 Quad 1GbE, PCIe 2.0 x4

6.4 OpenFlow Protocol Version

The Ryu controller supports multiple OpenFlow versions (1.0, 1.2, 1.3, 1.4 and 1.5), and so does the switch (1.0 – 1.5). Naturally, the communication between the switch and the software controller requires the same OpenFlow version. In this thesis, OpenFlow version 1.3 is used because it is supported by most open source and commercial SDN controllers, and most production software and hardware switches [54, 53]. The implementation of OpenFlow version 1.3 will offer seamless transition when upgrading to a

commercial OpenFlow switch. Nevertheless, OpenFlow version 1.5 is supported by both the prototype switch and Ryu controller and is compatible where both the controller and switch make use of a common OpenFlow version.

6.5 Controller and Switch Installation

This section covers the installation of the controller and installation of software on the switch.

6.5.1 Controller Installation

The installation of the Ryu Controller is a simplified process with just a single command. The Python Installation Program, `pip`, was used to install it. The following instruction installs Ryu:

```
1 $ pip install ryu
```

After the installation, running the Ryu controller and a few applications only requires the execution of a single command. For example, running the application script files, `app1.py` and `app2.py` is done as follows:

```
1 $ ryu-manager app1.py app2.py
```

6.5.2 SDN Switch Installation

In this section, the building and installation of Open vSwitch and DPDK is discussed. OVS and DPDK were build from source files using GCC version 7.2.0. At the time of implementing, the latest version of Open vSwitch was version 2.9.90, which recommended DPDK version 17.11.3 [106].

6.5.2.1 Prerequisites

The building of Open vSwitch and DPDK requires the following:

- DPDK version 17.11.3

- A DPDK supported NIC to provide physical Ethernet ports (Intel® 82580 Quad 1GbE) [100].

6.5.2.2 Installation

To configure Open vSwitch to use DPDK, the DPDK library needs to be installed first.

Installing DPDK

Before the installation, the directory where the DPDK would be installed was initially defined. The source files were downloaded and extracted (shown bellow in lines 1-3). The `DPDK_DIR` was assigned to the string path of DPDK library. The variable `$DPDK_DIR` was used to reference this directory. The installation was initiated as follows:

```
1 $ cd /usr/src/
2 $ wget http://fast.dpdk.org/rel/dpdk-17.11.3.tar.xz
3 $ tar xf dpdk-17.11.3.tar.xz
4 $ export DPDK_DIR=/usr/src/dpdk-17.11.3
5 $ cd $DPDK_DIR
```

The target location were the DPDK builds its files was defined and DPDK was installed as follows:

```
6 $ export DPDK_TARGET=x86_64-native-linuxapp-gcc
7 $ export DPDK_BUILD=$DPDK_DIR/$DPDK_TARGET
8 $ make install T=$DPDK_TARGET DESTDIR=install
```

The DPDK library was installed in the folder `/usr/src/`. The `make install` command (line 8) builds DPDK drivers that were later used to enable the OVS-DPDK architecture. The DPDK library contains example applications including an application called `TestPMD` which was used to verify the installation and also test the IO throughput benchmarks mentioned in the previous chapter.

Installing Open vSwitch

Once DPDK was installed, the installation of Open vSwitch followed. The following steps show its installation. Step 1, downloading of OVS:

```
1 $ cd ~
2 $ git clone https://github.com/openvswitch/ovs.git
3 $ cd ovs
```

Step 2, bootstrap. This creates a configuration script file named `configure`. This script can be used to customise the installation directory or change the location of the database directory.

```
4 $ ./boot.sh
```

Step 3, configure the sources of OVS to enable DPDK function. The `CFLAGS` parameter turns on flags for GCC compiler optimisation [1]. Here the compiler attempts to improve the performance and/or code size at the expense of compile time and/or memory used during compiling. "`-g -O2 -msse4.2`" enables efficient hash computation, where SSE4.2 instruction extension is supported by Intel® i5-3570 [107] which can improve performance when performing same instruction on multiple data. This improves performance by enabling SIMD capability improving hash table lookups within the data-path classifier (see Section 3.6 for caching in DPDK) [83].

```
5 $ ./configure --with-dpdk=$DPDK_BUILD CFLAGS="-g -O2 -msse4.2"
```

Step 4, building of the source files followed by the installation. To enable better performance, the compiler optimisation options are enabled (Ofast), to use special instructions (msse4.2) and multi-threaded compilation (j4) using all available cores in the Intel® i5-3570 CPU.

```
6 $ make 'CFLAGS=-g -Ofast -msse4.2' -j4
7 $ make install 'CFLAGS=-g -Ofast -msse4.2' -j4
```

Step 5, created the OVSDB database, which is used for saving configurations once the OVS-DPDK switch is running.

```
8 $ ovsdb-tool create /usr/local/etc/openvswitch/conf.db vswitchd/vswitch.ovsschema
```

Step 6, configure `ovsdb-server` to use the database created, to listen on a Unix domain socket, to connect to any managers specified in the database and to use the SSL configuration in the database:

```
9 $ ovsdb-server --remote=punix:/usr/local/var/run/openvswitch/db.sock --remote=db:Open_vSwitch,
    ↪ Open_vSwitch,manager_options --private-key=db:Open_vSwitch,SSL,private_key --
    ↪ certificate=db:Open_vSwitch,SSL,certificate --bootstrap-ca-cert=db:Open_vSwitch,SSL,
    ↪ ca_cert --pidfile --detach --log-file
```

[1] Options that control optimisation, https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

6.5.2.3 Configuration

After the installation, the operating system hosted in the switch was configured to reserve memory and CPU cores. This avoids memory fragmentation and restricts the operating system to a single core. The reserved memory is allocated for hugepages while the isolated cores are used for packet processing. This process is described below.

Configuring the Operating System (OS)

Performance is improved by enabling the hugepages which are supported by the CPU (Inte® i5 3570), reserving larger memory chunks that are used by the DPDK user-space program. The Intel® i5 3570 processor has 4 cores and supports hugepages of 2MB [107]. Three of the four cores were reserved for packet processing. To avoid fragmentation of the hugepages, options were added to the kernel bootline. This results in contiguous hugepages in memory. The adding of hugepages is done by appending the parameters shown below to *GRUB_ CMDLINE_ LINUX* in the file located at */etc/default/grub*. Grub is updated and the system is rebooted. *GRUB_ CMDLINE_ LINUX* was updated as follows:

```
1 hugepagesz=2MB hugepages=2048 iommu=pt intel_iommu=on isolcpus=1-3
```

The above parameters, `hugepagesz` and `hugepages`, allocates 4 Gigabytes (2048 * 2 Megabytes) of memory to hugepages of 2MB and `isolcpus` reserves cores 1, 2 and 3. These cores were isolated from the Linux scheduler so DPDK-based applications can “pin” to them. This command is persistent, that is to say that these parameters are set each time the machine was rebooted. This was included to avoid fragmentation of the memory used by OVS, thereby enabling efficient use of the memory.

Once rebooted, the following commands allocate the created hugepages to DPDK shown below (lines 1 – 4). The hugepages are allocated and the user space driver (`igb_uio`) is loaded (lines 6 and 7). This driver (`igb_uio.ko`) was created during the building of DPDK. The `igb_uio` is a driver for the Intel® 82580 NIC.

```
1 $ mkdir -p /mnt/huge
2 $ mkdir -p /mnt/huge_2mb
3 $ mount -t hugetlbfs hugetlbfs /mnt/huge
4 $ mount -t hugetlbfs none /mnt/huge_2mb -o pagesize=2MB
5 #DPDK Load igb_uio
6 $ modprobe uio
7 $ insmod $DPDK_DIR/x86_64-native-linuxapp-gcc/kmod/igb_uio.ko
8 $ cp $DPDK_DIR/tools/dpdk_nic_bind.py /usr/bin/.
9 $ dpdk_nic_bind.py --status
```

```
10 #DPDK bind driver ovs
11 $ dpdk-devbind.py -b igb_uio 01:00.0 01:00.1 01:00.2 01:00.3
```

The `igb_uio` driver is bound to each port on the Intel® 82580 (in line 11). Each port is referenced by the PCI address of 01:00.X where 'X' corresponds to the port number, i.e. `01:00.0` is port 0, `01:00.1` is port 1, `01:00.2` is port 2 and `01:00.3` is port 3. The binding process loads drivers that enables packet processing in the user-space. This finalises the DPDK configuration.

Configuring OVS-DPDK Settings

After the configuring of the OS and NIC, OVS is then configured to use the loaded `igb_uio` driver. The driver represents the `user-space data-path` program which runs on the pinned PMD threads (Section 3.5.1). Thereafter, the hugepages and isolated cores are allocated to OVS and pinned to PMD threads as shown in lines 2 and 3 respectively in the snippet below:

```
1 $ ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
2 $ ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-socket-mem="4096"
3 $ ovs-vsctl --no-wait set Open_vSwitch . other_config:pmd-cpu-mask=0xe
```

6.5.2.4 Validation

The group of commands below added the physical ports of the Intel® 82580 NIC to the virtual switch named `br0`.

```
1 $ ovs-vsctl add-br br0 -- set bridge br0 datapath_type=netdev
2 $ ovs-vsctl add-port br0 ovsport1 -- set Interface ovsport1 type=dpdk options:dpdk-devargs
    ↪ =01:00.0 ofport_request=1
3 $ ovs-vsctl add-port br0 ovsport2 -- set Interface ovsport2 type=dpdk options:dpdk-devargs
    ↪ =01:00.1 ofport_request=2
4 $ ovs-vsctl add-port br0 ovsport3 -- set Interface ovsport3 type=dpdk options:dpdk-devargs
    ↪ =01:00.2 ofport_request=3
5 $ ovs-vsctl add-port br0 ovsport4 -- set Interface ovsport4 type=dpdk options:dpdk-devargs
    ↪ =01:00.3 ofport_request=4
```

The virtual switch `br0` was populated with four physical ports corresponding to the PCI addresses. Port 0 was defined as `ovsport1`, port 1 as as `ovsport2`, port 2 as as `ovsport3` and port 3 as as `ovsport4`.

The code below configures the multi-layer switch to use OpenFlow version 1.3 (line 1). Line 2 requests the data path ID or switch ID. The switch returns its ID,

`0x0000001b21a6da00`. In later chapters references to ID **0000001b21a6da00** refers to the switch.

```
1 $ ovs-vsctl set bridge br0 protocols=OpenFlow13
2 $ ovs-vsctl get bridge br0 datapath_id
```

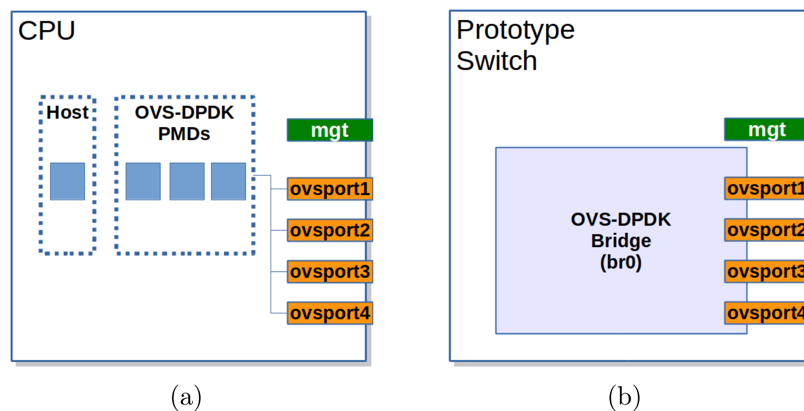
Starting the Switch `ovs-vswitchd`

After configuration, the switch will require the IP of the controller. This will allow the switch to establish communication between the switch and the controller. As discussed earlier, 'Packet-In' messages for unknown packets are forwarded to the controller's IP address. The controller will make decisions and add OpenFlow rules to be applied. Next, the switch is now ready to start. Starting OVS-DPDK switch is done as follows:

```
1 $ ovs-ctl --no-ovsdb-server --db-sock="$DB_SOCK" start
```

Additional features were added to the OVS which included the monitoring of the multi-layer switch from a remote server. OVS supports sFlow, a tool used in the industry to monitor high speed switch networks [108].

6.5.3 Installation Overview



(a) (b)

Figure 6.2: CPU Partitioning (a), and Internal Structure (b) of the Prototype.

Figure 6.2 shows the result of the integration of the switch at this stage. Figure 6.2(a) shows the CPU partitioning and logical topology. The host and OVS-DPDK shared the CPU cores. The OVS-DPDK Poll Mode Drivers (PMDs) require hardware resources such as dedicated cores and hugepages. The available four cores were utilised where one core

was reserved for the host, i.e. operating system and the remaining cores were then made available for OVS-DPDK's use.

Figure 6.2(b) shows the structure of the switch (from the user's perspective). It consists of five physical ports. As mentioned above, four are regular network ports of the switch while the fifth port is the management port for the switch to communicate with the Ryu controller.

6.5.4 OpenFlow Switch Performance Setup

The previous sections demonstrated the operations that were taken to install and configure the multi-layer switch and the controller. This section covers the tools used in the benchmarking and evaluation of the SDN switch. Figure 6.3 shows the setup for the benchmarking of the SDN switch (details covered in Section 5.4). A traffic generator named TRex was used to evaluate the switch.

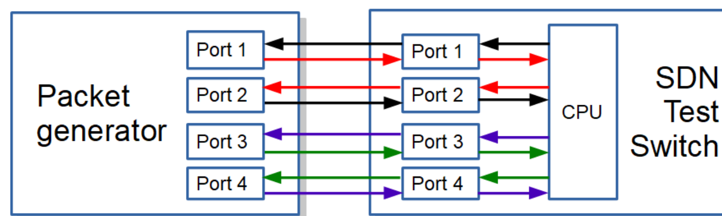


Figure 6.3: Switch Benchmark Setup.

TRex [109] is an open source traffic generator. The TRex stateless function supports multiple data streams, the modification of packet fields, and provide per stream statistics including latency.

6.5.4.1 Tester Specification

Table 6.3 shows a list of hardware and software that make up the TRex traffic generator. The packet generator (tester) is connected as discussed in Section 5.4.

6.5.4.2 IO Test

The SDN switch is configured to execute in IO mode using `TestPMD`. This mode allows a CPU core to forward packets from the ingress (Rx) port and transmit them to the egress

Table 6.3: Software & Hardware components for TRex

Item	Description
Platform	H170 PRO GAMING
Chipset	Intel® 760G (780L)
CPU	Intel® Core™i5-6400 CPU (6MB Cache, 2.70GHz), 4 core, 4 threads
Memory	2x4GB, Single Channel @ 2133MHz
Operating System	Ubuntu Server 16.04 LTS (GNU/Linux 4.15.0-39-generic x86_64)
NICs	Intel® 82580 Quad 1GbE (PCIe 2.0), Intel® Ethernet Connection (2) I219-V
BIOS	MBEC-GMB-0105.11.0.0.1168.31/D1
Intel® Ethernet Drivers	igb - version 5.4.0-k
DPDK version	18.05
GCC version	(Ubuntu 5.4.0-6ubuntu1 16.04.10) 5.4.0 20160609

(Tx) port without any packet processing. This test benchmarked the maximum rate at which the IO is able to forward packets.

6.5.4.3 Layer 2 Switching

The Benchmark for layer 2 behaviour followed a similar set up as the IO test but included L2 packet processing. The traffic was directed based on the destination MAC address of the received Ethernet packet. The MAC addresses for each port on the TRex traffic generator were added to the OpenFlow table. The Layer 2 test will show additional overhead incurred by L2 packet processing when compared to the IO test. Below is a list of the MAC addresses for the TRex traffic generator:

- **Port 1** : 00:1b:21:a6:d3:bc
- **Port 2** : 00:1b:21:a6:d3:bd
- **Port 3** : 00:1b:21:a6:d3:be
- **Port 4** : 00:1b:21:a6:d3:bf

Similar to the setup seen in the IO test, this test includes packet processing. In order for OVS to perform L2 switching, flows were added to direct traffic to connected ports. Below is a set of instructions that add OpenFlow entries to the switch via the command line interface (CLI)(Listing 6.1).

Listing 6.1: Layer 2 Flow Entries to the Switch via the CLI.

```
sudo ovs-ofctl add-flows br0 -O OpenFlow13 - <<'EOF'
  table=0, priority=2 dl_type=0x0800,dl_dst=00:1b:21:a6:d3:bc, actions=output:1
  table=0, priority=2 dl_type=0x0800,dl_dst=00:1b:21:a6:d3:bb, actions=output:2
  table=0, priority=2 dl_type=0x0800,dl_dst=00:1b:21:a6:d3:be, actions=output:3
  table=0, priority=2 dl_type=0x0800,dl_dst=00:1b:21:a6:d3:bf, actions=output:4
EOF
```

The above had added flows to table 0 with a set priority of 2. The match field for these flows uses the MAC destination address (defined by `dl_dst`) and EtherType (defined by `dl_type`). The `dl_dst` parameter instructs the switch to filter by the destination MAC address while the `dl_type` parameter instructs the switch to filter traffic with EtherType of IPv4. For the flow entry in line 2, the rule is defined for all packets with destination MAC address of 00:1b:21:a6:d3:bc and with EtherType of IPv4 are to be forwarded to port 1. The same logic applies to the other three flow entries (lines 3 to 5) where packets with corresponding match of MAC address and EtherType are sent to respective ports.

The switch's behaviour can be analysed further by using the trace tool in OVS. This tool can be used to view what happens to a packet when it goes through the data plane of the multi-layer switch. Figure 6.4 shows the steps that the switch takes when a packet addressed to MAC of `00:1b:21:a6:d3:bc` (i.e. port 1 of TRex connected to port 1 of the switch, Section 6.5.4) is passed to any of the switch's port. The figure shows that the packet matches the flow entry in table 0. The result action is to output the packet out of port 1 as expected. The packet is not changed, hence '`Final flow: unchanged`'.

```
openvswitch@openvswitch:~$ sudo ovs-appctl ofproto/trace br0 dl_typ
e=0x0800,dl_dst=00:1b:21:a6:d3:bc
Flow: ip,in_port=ANY,vlan_tci=0x0000,dl_src=00:00:00:00:00:00,dl_ds
t=00:1b:21:a6:d3:bc,nw_src=0.0.0.0,nw_dst=0.0.0.0,nw_proto=0,nw_tos
=0,nw_ecn=0,nw_ttl=0

bridge("br0")
-------------
 0. ip,dl_dst=00:1b:21:a6:d3:bc, priority 2
    output:1

Final flow: unchanged
Megaflow: recirc_id=0,eth,ip,in_port=ANY,dl_dst=00:1b:21:a6:d3:bc,n
w_frag=no
Datapath actions: 1
openvswitch@openvswitch:~$
```

Figure 6.4: Verifying L2 Behaviour Using Trace Tool.

6.5.4.4 Layer 3 Routing

The multi-layer switch also supports layer 3 routing which was also benchmarked. The sets of tests performed included traffic that was directed based on the destination IP address of the received packet. This function is used when traffic travels from one subnet to another. In this case, the switch must alter the source and destination MAC addresses of a packet each time a packet changes subnets.

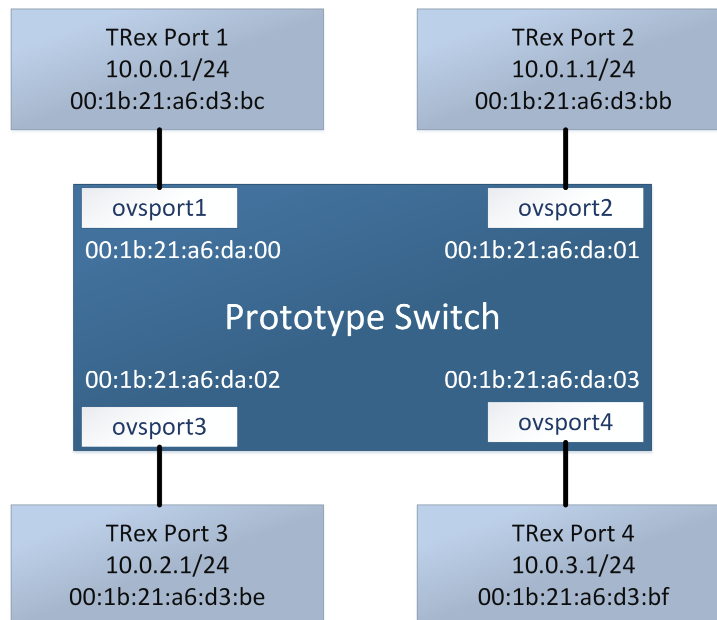


Figure 6.5: Logical Connections Between Switch and TRex Traffic Generator.

Figure 6.5 shows the logical connections between the switch and TRex traffic generator. TRex was configured to generate traffic from different subnets. The figure shows the MAC and IP addresses for each port on the TRex machine. It also shows the port MAC addresses for the switch. To recap on layer 3 functionality from Section 4.4.3, the behaviour of a L3 switch (router) performs the following tasks as shown:

1. Set the source and destination MAC address where the source becomes the switch's MAC address and the destination becomes the MAC address of the machine hosting TRex. In normal situations, protocols like ARP are used to determine the destination MAC address. In this setup, it is assumed that the MAC addresses were already defined in the switch.

2. Decrement the time to live, i.e. the TTL, when a packet changes subnets. The TTL or time to live [35] specifies how long the packet should live in the network.

Routers decrement this value before transmitting packets. When the resulting TTL is reduced to zero, it is assumed to have taken too long to route and the packet is discarded.

3. Output to corresponding port.

To accomplish L3 switching, flows were added to direct traffic based on the destination IP of the packet. Listing 6.2 shows a set of flow entries that where installed via the CLI. Here it is seen that layer 3 operations are generally more complex than layer 2. The above command added flows to the switch. The match field for these flow entries uses the EtherType (defined by `dl_type`) and IP destination address (defined by `nw_dst`). For the flow table entry in line 2, the rule defines that all IPv4 traffic with destination IP address of 10.0.0.1 are to be treated as follows:

1. Set the source and destination MACs. This is done by the action `set_field` which sets the Ethernet source (`eth_src`) to 00:1b:21:a6:da:00 and Ethernet destination (`eth_dst`) to 00:1b:21:a6:d3:bc. These corresponds to the source port (`ovsport1`) and destination port (`TRex port 1`) shown in Figure 6.5.

2. Decrement TTL value given by the action `dec_ttl`.

3. Output to port `ovsport1` defined by the action `output:1`

The same principle applies to the other three flow entries (lines 3 to 5).

Listing 6.2: Adding of Layer 3 Flow Entries to the Switch via the CLI.

```
sudo ovs-ofctl add-flows br0 -O OpenFlow13 - <<'EOF'
  table=0, priority=2 dl_type=0x0800,nw_dst=10.0.0.1/24, actions=set_field:00:1b:21:a6:da:00->
      ↪ eth_src, set_field:00:1b:21:a6:d3:bc->eth_dst, dec_ttl,output:1
  table=0, priority=2 dl_type=0x0800,nw_dst=10.0.1.1/24, actions=set_field:00:1b:21:a6:da:01->
      ↪ eth_src, set_field:00:1b:21:a6:d3:bd->eth_dst, dec_ttl,output:2
  table=0, priority=2 dl_type=0x0800,nw_dst=10.0.2.1/24, actions=set_field:00:1b:21:a6:da:02->
      ↪ eth_src, set_field:00:1b:21:a6:d3:be->eth_dst, dec_ttl,output:3
  table=0, priority=2 dl_type=0x0800,nw_dst=10.0.3.1/24, actions=set_field:00:1b:21:a6:da:03->
      ↪ eth_src, set_field:00:1b:21:a6:d3:bf->eth_dst, dec_ttl,output:4
EOF
```

The trace tool in OVS was used to verify the correctness of the flow entries. For example, consider a packet with IP destination of 10.0.0.1 and a TTL value of 16. As shown in Figure 6.6, the packet matches a flow entry in table 0 where the packet's source MAC is changed to `00:1b:21:a6:da:00`. The destination MAC is changed to `00:1b:21:a6:d3:bc` and the TTL is changed to 15 before outputing it to port 1. This demonstrates the L3 routing capability of the switch.

```
openvswitch@openvswitch:~$ sudo ovs-appctl ofproto/trace br0 dl_typ
e=0x0800,nw_dst=10.0.0.1,nw_ttl=16
Flow: ip,in_port=ANY,vlan_tci=0x0000,dl_src=00:00:00:00:00:00,dl_ds
t=00:00:00:00:00:00,nw_src=0.0.0.0,nw_dst=10.0.0.1,nw_proto=0,nw_to
s=0,nw_ecn=0,nw_ttl=16

bridge("br0")
-------------
 0. ip,nw_dst=10.0.0.0/24, priority 2
    set_field:00:1b:21:a6:da:00->eth_src
    set_field:00:1b:21:a6:d3:bc->eth_dst
    dec_ttl
    output:1

Final flow: ip,in_port=ANY,vlan_tci=0x0000,dl_src=00:1b:21:a6:da:00
,dl_dst=00:1b:21:a6:d3:bc,nw_src=0.0.0.0,nw_dst=10.0.0.1,nw_proto=0
,nw_tos=0,nw_ecn=0,nw_ttl=15
Megaflow: recirc_id=0,eth,ip,in_port=ANY,dl_src=00:00:00:00:00:00,d
l_dst=00:00:00:00:00:00,nw_dst=10.0.0.0/24,nw_ttl=16,nw_frag=no
Datapath actions: set(eth(src=00:1b:21:a6:da:00,dst=00:1b:21:a6:d3:
bc)),1
```

Figure 6.6: Verifying L3 Behaviour Using Trace Tool.

6.6 Ryu Controller and Applications

Chapter 4 introduced the structure of the Ryu controller and supporting applications. Configuring a network using the solution proposed in this thesis makes use of these applications which includes a base application and a few helper applications. This section covers the implementation of these applications. The Ryu controller framework allows the use of the following components [89] in implementing applications:

- **Executable:** The main executable program, `ryu-manager`.
- **Ryu Base:** This component (`ryu.base.app_manager`) is the central management of Ryu applications that loads applications, shares data, and route message between them.
- **Ryu controller:** Includes four components which are: '`ryu.controller.controller`', '`ryu.controller.dpset`', '`ryu.controller.ofp_event`' and '`ryu.controller.ofp_handler`'. The `ryu.controller.controller` is the main component that handles connections from switches and generates and send events to appropriate entities like Ryu applications. The `ryu.controller.dpset` component manages switches, the `ryu.controller.ofp_event` component define OpenFlow events, and `ryu.controller.ofp_handler` includes OpenFlow message handling including the negotiation.

- **Protocol encoder and decoder:** `ryu.ofproto.ofproto_v1_3` defines the OpenFlow 1.3 and `ryu.ofproto.ofproto_v1_3_parser` encodes/decodes these definitions.

The above list is not exhaustive and only includes the components necessary in the application design covered within this thesis. Figure 6.7 shows the overall structure of the proposed SDN network which contains five applications (`Base`, `WebAPI`, `L2Switch`, `RouterAPI` and `SimpleDHCPServer`), the Ryu controller, the prototype switch and connected end-hosts. The applications were implemented as Python scripts which are covered in detail in the following sections

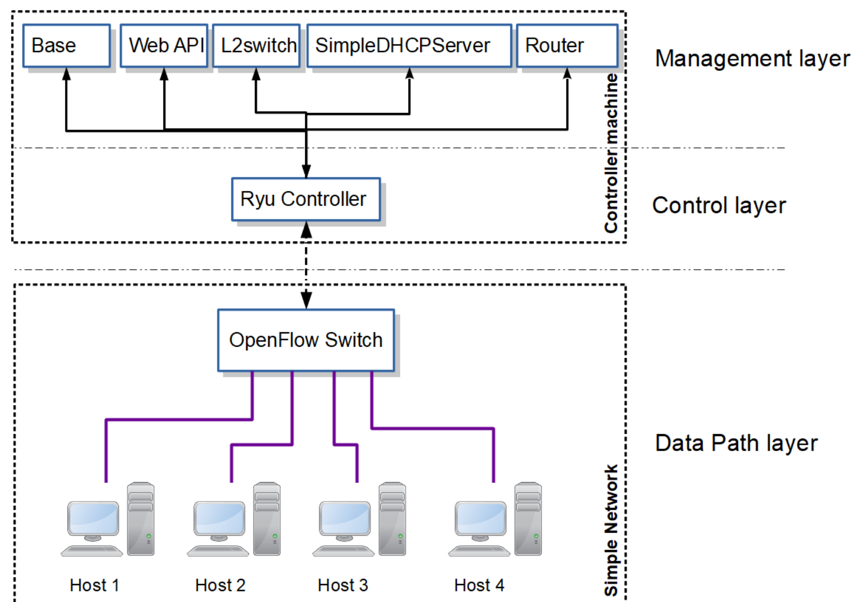


Figure 6.7: Proposed SDN Network Design.

6.6.1 Base Application

The base application in the controller defines the core application responsible for serving the web UI, management of the OpenFlow switches, and the exchange of configuration details between applications. The scripts for this application were saved as `base.py`. The implementation of this application is defined as a Python class `Base(app_manager.RyuApp)`. Within the class, a web server (WebControlApi) is used to serve a web UI. The web UI is defined by mapping server functions (`WebControlApi` functions) to Web Server Gateway Interface (WSGI). The WSGI connects Web applications and Web servers in Python. Table 6.4 shows a list of mapped web URLs and their functions.

Table 6.4: Web URLs for REST Messages.

Function	URL	REST Calls	Description
Configure	<url>*/conf/switch_id	GET, PUT	Used to send/receive switch configurations, where switch_id is the identifier
Initialise	<url>*/init/switch_id	GET	Used to submit the defined configurations, where switch_id is the identifier
Monitor	<url>*/topology	GET	Get topology info
Topology	<url>*/topology/switches	GET	Get switches info
Topology	<url>*/topology/links	GET	Get links info
Topology	<url>*/topology/hosts	GET	Get hosts info
Topology	<url>*/topology/ws	*NONE	Web socket
File server	<url>*/	GET	To serve up HTML files, scripts and image resources managed by the `webapi.py`

The '`<url>`', in the table, is defined by the format *http://WebServerAddress:Port*. Access to the web UI on the controller machine is given by http://localhost:8080 which is accessible from a web browser. Furthermore, the network design and state can be visualised within the UI. Figure 6.8 and Figure 6.9 (in the next few pages) are screenshots taken from the web UI showing the network topology of a single switch with four hosts connected, configuration toolbar (showing the configuration of a switch) and the current state of the OpenFlow table. The UI was adapted from an example given by Ryu. The configuration toolbar and icons for connected hosts (in the network topology) were added to allow the configuring of the switch and to view hosts connected to the switch.

The OpenFlow switches are managed by using OpenFlow controller component `ryu.controller.dpset`. The 'dpset' is used to register the switch with the `WebControlApi` server (lines 3–8 from Listing 6.3). The `dpset.EventDP` event in line 3 uses the decorator that informs the controller that the application captures events which occur whenever the switch is connected/disconnected from the controller. These events are then handled by the function `datapath_handler` shown in line 4. Each time the OpenFlow switch gets connected to the controller, the `ev.enter` was triggered (line 5) which resulted in the registering of the switch to the `WebControlApi`. The switch registration to `WebControlApi` exposes the switch's configuration in the web UI. The switch would appear in the network topology and is represented with a switch icon.

Once registered with the web server, events were managed and updated in the web UI.

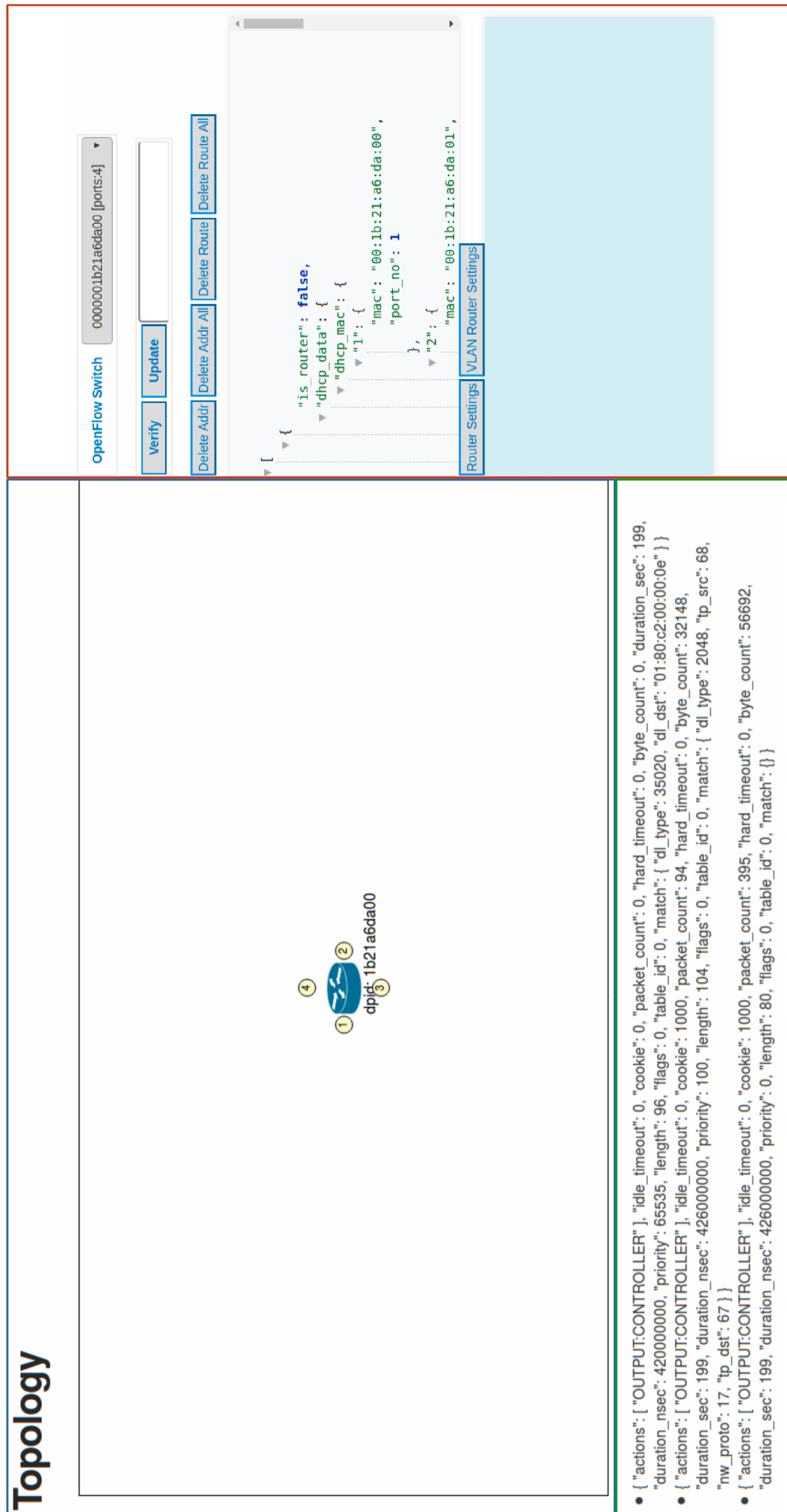


Figure 6.8: The Full View of The Web UI.

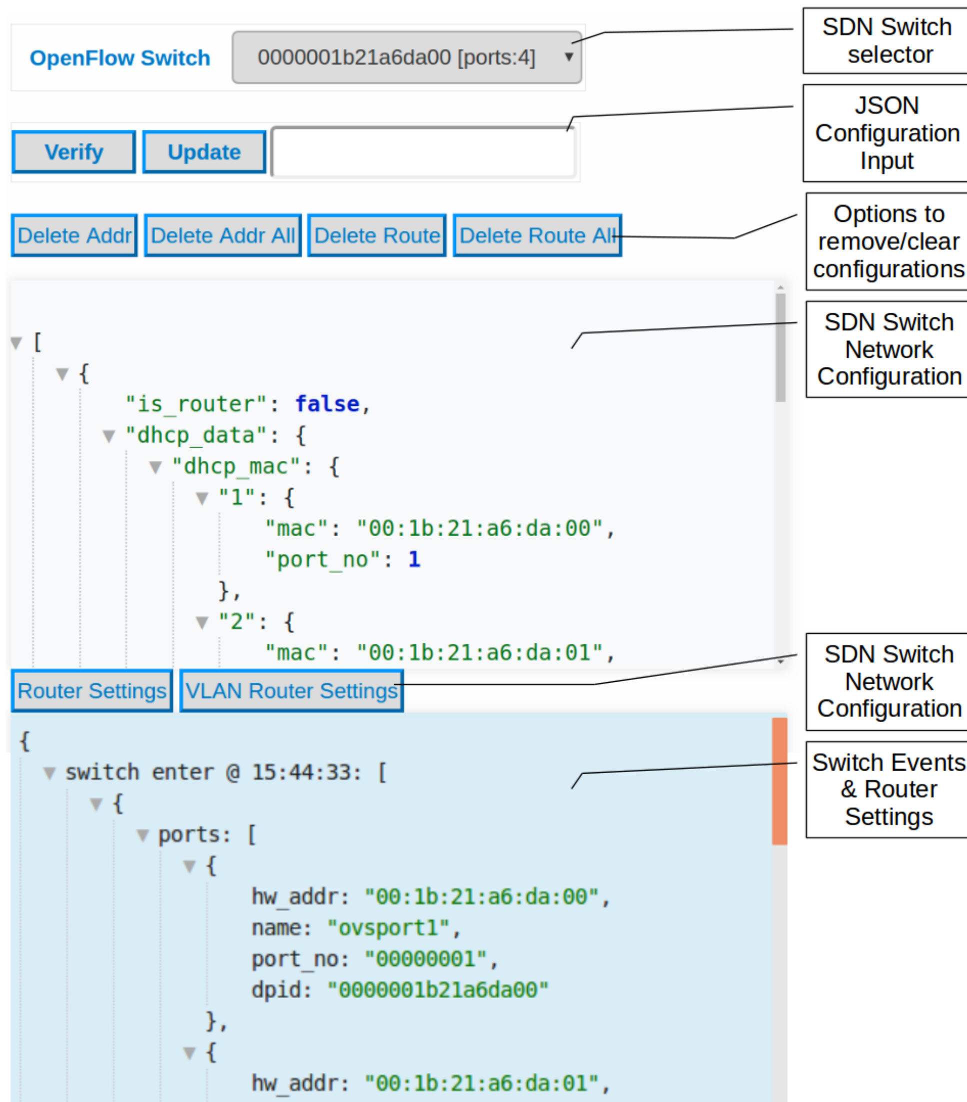


Figure 6.9: Configuration Toolbar Showing Settings for an OpenFlow Switch.

Listing 6.3: Code snippet from `base.py`

```
1 class Base(app_manager.RyuApp):
2   #...
3   @set_ev_cls(dpset.EventDP, dpset.DPSET_EV_DISPATCHER)
4   def datapath_handler(self, ev):
5     if ev.enter:
6       WebControlApi.register_ofswitch(ev.dp)
7     else:
8       WebControlApi.unregister_ofswitch(ev.dp)
9   #...
```

Table 6.5 shows some events with their description. These events are used to update the topology viewer in the web UI.

Table 6.5: Events for the Web UI.

Event	**Description**
event.EventSwitchEnter	Triggered when the switch connects to the controller. Causes a graphical image of the switch to appear in the UI.
event.EventSwitchLeave	Triggered when the switch gets disconnected from the controller. Results in the removal of the graphical switch.
event.EventLinkAdd	Triggered when a managed switch is directly connected to another managed switch. Creates a line to show the link connection of the switches.
event.EventLinkDelete	Triggered when the link is disconnected from two managed switches. Results in the removal of the link.
event.EventHostAdd	Triggered when a host is discovered on a switch's port. Displays a small circle representing a host.

6.6.2 Web File Server Application

This application serves up the HTML files, script files and image resource files. The **WebAPI** application, defined as the Python class `WebAPI(app_manager.RyuApp)`, is saved in the `webapi.py` script file. Within the class, a web controller class called `GUIServerController` handles the process.

Listing 6.4: Code snippet from `webapi.py`

```
class GUIServerController(ControllerBase):
  def __init__(self, req, link, data, **config):
    super(GUIServerController, self).__init__(req, link, data, **config)
    path = "%s/html/" % PATH
    self.static_app = DirectoryApp(path)

  @route('topology', '/{filename:[^/]*}')
  def static_handler(self, req, **kwargs):
    if kwargs['filename']:
      req.path_info = kwargs['filename']
    return self.static_app(req)
```

From Listing 6.4, the path, defined by `path = "%s/html/"` (in line 4) defines the location where the web files are located. The `DirectoryApp(path)` (line 5) returns an application that dispatches requests based on the `path_info` (line 10). Any requested file from the web UI (line 9) would then be used as the `path_info` which results in files served that are in the subdirectory `html`. Below shows the list of web files within the `html` directory and their function.

- **d3.js** – JavaScript library used for the visualisation of the network topology.
- **form.js** – Script file that handles form data formatting within the Configuration Toolbar seen in Figure 6.9. It is also used to make REST calls to the controller by making use of the JQuery library.
- **index.html** – The main HTML file that displays the main web UI screen.
- **jquery.js** – JavaScript library used for REST calls and JSON data formatting.
- **jquery.json-viewer.css** – Cascade Style Sheet styling that describes how JSON data is presented.
- **jquery.json-viewer.js** – Script file that transforms the JSON data received from the controller into an HTML tree structure seen in Figure 6.9. It uses the JQuery library.
- **router.svg** – Image file of a router as seen in Figure 6.8.
- **ryu.topology.css** – Cascade Style Sheet styling that describes how elements within the web UI are presented.

- **ryu.topology.js** – Script file that collects information from the controller using URLs with the function *topology* given in Table 6.4. It makes use of the D3.js library to translate data from the controller into a visual structure, i.e. network topology.

The web UI was adapted from Ryu controller source code. A Configuration Toolbar was developed to allow a user to interact with OpenFlow devices from within this UI. Originally, querying and controlling an OpenFlow device meant using an external REST client by sending JSON configurations through that client. This toolbar will, however, makes the process of communicating with an OpenFlow device easier. As mentioned before, multiple files were used to enable the functionality of a REST client.

The `index.html`, `d3.js`, `router.svg` and `ryu.topology.css` were taken from the source code from the Ryu controller, where the `index.html` file was adapted to include the view of the Configuration Toolbar. The remaining files provide the functionality of a REST client and displays feedback that comes from an OpenFlow device.

6.6.3 L2Switch Application

During packet processing, the functionality of a layer 2 switch (covered in Section 4.4), performs the following actions:

1. Modify the address of packet or send the packet from a specified port.
2. Send the unknown packet to the controller (Packet-In).
3. Send the packet from the controller through the specified port (Packet-Out).

The first action is executed based on flow entries within the OpenFlow table. However, the other two actions involve the **L2Switch** application. The application was adapted from [110]. This application is defined in Python as `L2Switch(app_manager.RyuApp)` saved in the script file `l2switch.py`. Within the class, the Packet-In and Packet-Out functions are implemented as follows (Listing 6.5):

Listing 6.5: Code snippet from `l2switch.py`

```
1 class L2Switch(app_manager.RyuApp):
2   #...
3   @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
```

```
4   def _packet_in_handler(self, ev):
5     #...
6     dst = eth.dst
7     src = eth.src
8     dpid = datapath.id
9     self.global_mac_to_port.setdefault(dpid, {})
10
11    self.global_mac_to_port[dpid][src] = in_port
12
13    if dst in self.global_mac_to_port[dpid]:
14      out_port = self.global_mac_to_port[dpid][dst]
15    else:
16      out_port = ofproto.OFPP_FLOOD
17
18    actions = [parser.OFPActionOutput(port) for port in out_port]
19
20    # install a flow to avoid packet_in next time
21    if out_port != ofproto.OFPP_FLOOD:
22      match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
23      ofp_helper.add_flow(COOKIE, datapath, 0, match, actions, idle_timeout=IDLE_TIMEOUT)
24
25    data = msg.data
26    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,in_port=in_port,
          ↪ actions=actions, data=data)
27    datapath.send_msg(out)
28  #...
```

When a packet is sent by the OpenFlow switch, the `EventOFPPacketIn` event (line 3) is triggered and is handled by the function `_packet_in_handler` decorated by `@set_ev_cls` decorator (Section 4.3.2). As seen above, the MAC source and destination are noted along with the data path ID (dpid) of the switch (lines 6 – 8). The MAC table, saved as the variable `global_mac_to_port` (line 9) contains MAC addresses of nodes connected at each port. The destination address is used to search the table (line 13). If present, the received packet is forwarded to the corresponding port (line 14) otherwise flood the packet (line 16). Line 18 creates an OpenFlow rule such that similar packets are forwarded to the corresponding port in future. Lines 21 to 23 add the rule to the OpenFlow switch while lines 25 to 27 send the packet back to the switch.

6.6.4 Router Application

The router application, adapted from the REST router (`rest_router.py`) included in the Ryu framework, supports normal and VLAN based routing. The configuration of the router is implemented by defining the address data as structured as "A.B.C.D/M". When two or more routers are involved, extra information is required. This includes: defining

the gateway as structured as "A.B.C.D" and also defining static routes. The process of configuring the VLAN router is similar, the only difference is that the VLAN ID is required.

The application is defined as `class RouterAPI(app_manager.RyuApp)` with the script file saved as `router.py`. Within the class, the behaviour of the router is managed by the corresponding object ‘`Router`’ or ‘`VlanRouter`’. Packet-In messages are handled by the function `packet_in_handler` shown in Listing 6.6.

Listing 6.6: Code snippet from `router.py`

```
1  class RouterAPI(app_manager.RyuApp):
2    #...
3    def packet_in_handler(self, msg, header_list):
4      # Check invalid TTL (for OpenFlow V1.2/1.3)
5      ofproto = self.dp.ofproto
6      if ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION or \
7          ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
8        if msg.reason == ofproto.OFPR_INVALID_TTL:
9          self._packetin_invalid_ttl(msg, header_list)
10         return
11
12     # Analyze event type.
13     if ARP in header_list:
14       self._packetin_arp(msg, header_list)
15       return
16
17     if IPV4 in header_list:
18       rt_ports = self.address_data.get_default_gw()
19       if header_list[IPV4].dst in rt_ports:
20         # Packet to router’s port.
21         if ICMP in header_list:
22           if header_list[ICMP].type == icmp.ICMP_ECHO_REQUEST:
23             self._packetin_icmp_req(msg, header_list)
24             self.logger.debug("ICMP Packet to router’s port")
25             return
26         elif TCP in header_list or UDP in header_list:
27           self._packetin_tcp_udp(msg, header_list)
28           self.logger.debug("TCP/UDP Packet to router’s port")
29           return
30       else:
31         # Packet to internal host or gateway router.
32         self._packetin_to_node(msg, header_list)
33         self.logger.debug("Packet to internal host or gateway router")
34         return
```

Listing 6.6 shows the code snippet of the router application. Upon receiving a packet for processing, the TTL value is verified (lines 3 to 8). The function `_packetin_invalid_ttl` handles invalid TTL values by sending an ICMP TTL error that informs the sender that the packet took too many hops to route. If the received packet is an ARP message, it is

handled by the function `_packetin_arp`. Routers may use the information to learn host MAC addresses, to update routing tables, or to reply to ARP messages. An IPv4 packet is handled according to the type of packet. If the packet is either an ICMP echo, ping message (line 22), a TCP/UDP packet on the router's port (line 26), or a packet headed to an internal host or gateway (line 30), the action would be to send an ICMP echo reply, send an ICMP port unreachable error or forward packet to an internal host/gateway. This covers the core functionality of the router implemented as the router application.

6.6.5 DHCP Application

Part of the responsibilities assigned to the **Base** application is to define flow rules that capture DHCP messages. These packets are then handled by the **SimpleDHCPServer** application (process given in Section 4.4.4.3). This application has the following functions: (1) maintain a shared pool of IP addresses, (2) offer an IP address from this shared pool, and (3) acknowledge the offered IP if it is requested. The application is defined as a class using the *RyuApp* component. The defined class, `SimpleDHCPServer(app_manager.RyuApp)`, was saved as the script file `dhcp.py` adapted from [110]. Within the class, arriving DHCP frames are processed as shown in Listing 6.7:

Listing 6.7: Code snippet from `dhcp.py`

```
1  class SimpleDHCPServer(app_manager.RyuApp):
2    #...
3    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
4    def _packet_in_handler(self, ev):
5      openflow_sw = self._OF_SWITCH_LIST_.get(datapath.id)
6      #...
7      if msgType == dhcp.DHCP_DISCOVER or not openflow_sw.mac_to_client_ip.has_key(dhcpPacket.
           ↪ chaddr):
8        self.handle_dhcp_discover(openflow_sw, dhcpPacket, datapath, in_port)
9      elif msgType == dhcp.DHCP_REQUEST:
10       self.handle_dhcp_request(openflow_sw, dhcpPacket, datapath, in_port)
11     #...
```

DHCP messages received are either a DHCP_DISCOVER or DHCP_REQUEST. The `handle_dhcp_discover` function will offer an IP address to received DHCP_DISCOVER messages by sending a DHCP_OFFER with the offered IP, while the `handle_dhcp_request` will send an ACK message if the offered IP address is requested. This, thereby, perfoms the process of IP assignment.

6.6.6 Running the Controller Software and Applications

The execution of these applications is done by running the controller and the script files. To run the Ryu controller and the applications described in the previous sections, in the terminal, the following command is entered.

```
>>> ryu-manager --observe-links router.py base.py l2switch.py dhcp.py webapi.py
```

Listing 6.8 shows the logs from the Ryu controller during start up. The important thing to note is the local web address and port number of the web UI given in line 26 since it is used to access the web UI.

Listing 6.8: Logs from starting Ryu controller.

```
tinashe@Controller:~$ ryu-manager --observe-links router.py base.py l2switch.py dhcp.py webapi
    ↪ .py
loading app router.py
loading app base.py
loading app l2switch.py
loading app dhcp.py
loading app webapi.py
loading app ryu.controller.ofp_handler
loading app ryu.app.rest_topology
loading app ryu.app.ws_topology
loading app ryu.app.ofctl_rest
loading app ryu.controller.ofp_handler
instantiating app None of Switches
creating context switches
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu.app.ws_topology of WebSocketTopology
instantiating app router.py of RestRouterAPI
instantiating app base.py of Base
instantiating app dhcp.py of SimpleDHCPServer
instantiating app ryu.app.rest_topology of TopologyAPI
instantiating app l2switch.py of L2Switch
instantiating app ryu.app.ofctl_rest of RestStatsApi
instantiating app webapi.py of GUIFileServer
(3787) wsgi starting up on http://0.0.0.0:8080
#...
```

6.7 Summary

This chapter detailed the installation and configurations for the multi-layer switch that was built in the previous chapter. The installation covered the integration of OVS-DPDK

and the hardware. Efforts to improve the performance include the optimising of the running code by making use of features available in the hardware such as extensions supported by the processor. The result was an OpenFlow-enabled switch with five physical Ethernet ports highlighted in Section 6.5.3 (four data plane ports and one management port).

This chapter also introduced the testing methodology as well as the implementation of SDN applications. The tests outlined here are used in Chapter 7 to measure the IO, L2 and L3 performances. The tools used in these tests were: `TestPMD`, TRex traffic generator and the trace tool from OVS. The `TestPMD` tool measures the IO forwarding throughput. TRex measures the throughput for L2 and L3 packet processing as well as the latency for IO, L2 and L3. Finally, the trace tool verified the L2 and L3 functionality of the flow entries in the switch. The SDN applications implemented a set of features as seen in some commercial applications which include flow management, topology update, and traffic monitoring/classification accessed via the web UI (Section 6.6.1).

Chapter 7

Switch Benchmark Tests and Results

This chapter presents results that document the performance and functionality of the multi-layer SDN switch. The initial tests described in this chapter are the subject of a conference paper that the author wrote and published in 2018 [18]. Section 7.1 shows the format in which the tests, discussed in this chapter, are organised. Section 7.2 relates to throughput testing and assessing the performance of IO packet transfer to determine if the functioning of the switch meets the link rate of 1 Gbps when all ports are under load. Section 7.3 evaluates the performance of the switch when L2 packet processing is involved, to examine if the CPU is the bottleneck. This section also covers the throughput and delays introduced by packet processing and compares results between a single core and multi-core processing. Section 7.4 evaluates the performance of L3 routing. This includes packet processing, core utilisation and latency. Finally, Section 7.5 verifies the controller-switch interaction and validates the OpenFlow functionality of the switch.

7.1 Tests General Structure

The format followed in the presentation of tests is as follows:

- Description of the test.
- Description of the testing environment.
- Results obtained.
- Analysis of results.

The benchmarks in this chapter tested the multi-layer switch using a packet generator following the RFC 2544 test methodology. The packet generator was deployed with the following components which were an Intel® 760G (780L) Motherboard, an Intel® i5-6400 CPU @ 2.7GHz, 8GB Single Channel @ 2133MHz RAM, and the Intel® 82580 Quad 1GbE Card as its physical ports. The four Ethernet interfaces were connected directly to the Ethernet ports of the switch and tests were taken to evaluate it. The traffic generator implemented used the TRex network benchmark tool. The set up of this test is shown in Figure 7.1.

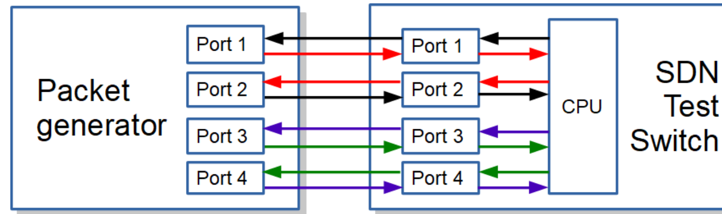


Figure 7.1: Multi-layer Switch Benchmark Setup.

TRex [109] was used for the benchmarking of four 1 Gbps Ethernet interfaces to measure the bi-directional transfer simultaneously on all four ports. Both the switch and TRex negotiated a link speed of 1 Gbps between their connections. Traffic from TRex was sent over port pairs. The traffic used in this process used the UDP protocol and different frame sizes as defined in Section 5.4. TRex evaluated the traffic from its paired port and tallied the rate of bytes received.

The tests, for single core and multi-core packet processing, were carried out to provide an indication of the behaviour of the switch when processing frames at multiple frame rates. These tests measured the overall packet per second (`pps`) and bits per second (`bps`). The relationship between `pps` and `bps` is shown in Equation 7.1 [92, 94].

$$Packet\ Per\ Second(pps) = \frac{Bits\ Per\ Second(bps)}{(FrameSize + 20) \times 8} \tag{7.1}$$

The 20 bytes defined in the equation is the header size of an IP packet as seen in Figure 7.2. The frame size from the equation is the payload of the IP packet.

7.2 IO Forwarding Test

During this test, the multi-layer switch was configured for only IO transfers. This was to ensure that the switch strictly forward frames without any packet processing. The Ether-

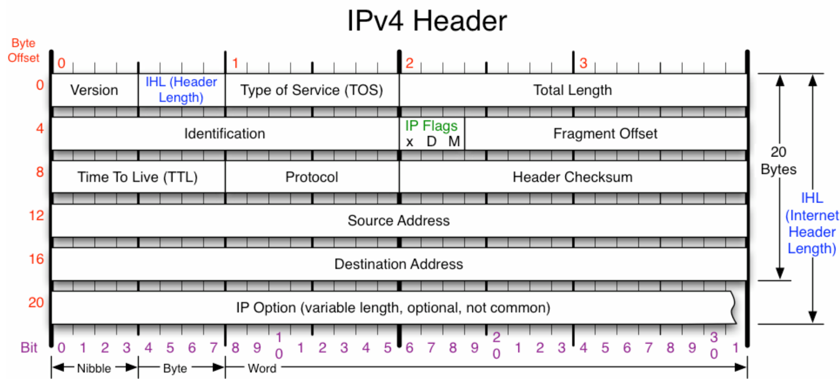


Figure 7.2: Representation of an IPv4 packet. Source [17]

net ports on the switch were connected in a loop to make use of the *loopback* configuration from the DPDK library. Included in the DPDK library are sample tools and applications that provide for testing and further development. An application named `TestPMD` [82] was used to evaluate the IO of the hardware switch for packet forwarding. `TestPMD` is also able to directly access the ports present on the Intel® 82580 NIC hardware (a sample is shown in Appendix A.1). The evaluation of both the throughput and latency were done over a minimum period of 60 seconds using frames with sizes between 64 bytes and 1518 bytes. Figure 7.3 shows the setup for the IO test.

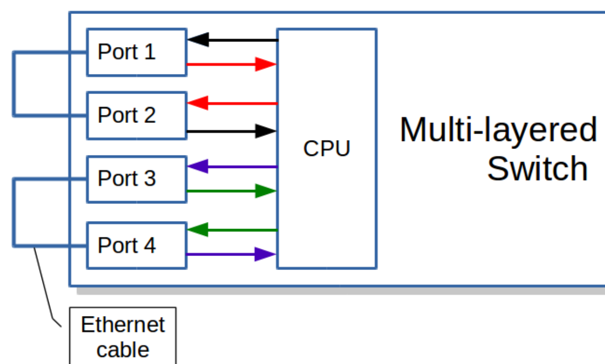


Figure 7.3: DPDK IO Benchmark Setup.

7.2.1 Results

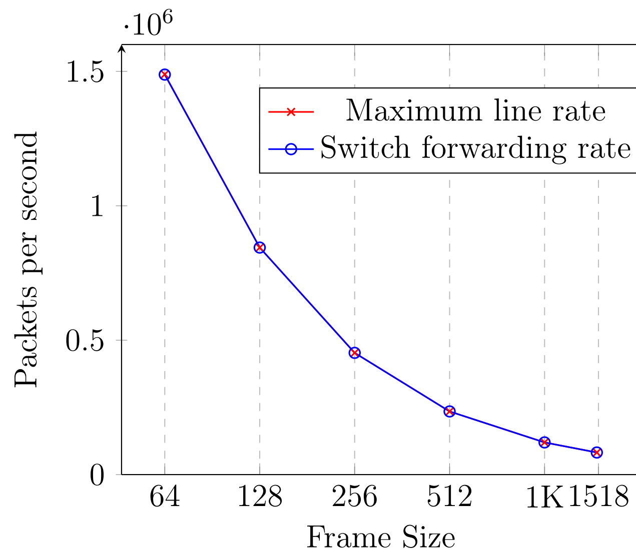


Figure 7.4: The IO Forwarding Rate of the Multi-layer Switch. Source [18].

Table 7.1: IO Forwarding Rate for the Multi-layer Switch.

Frame size	**IO forwarding rate (pps)**	**Theoretical line rate (pps)** [1]	**Throughput rate (bps)**	**Throughput %**
64	1487911	1488905	999876192	99.988
128	844575	844594	999976800	99.997
256	452888	452898	999976704	99.997
512	234957	234962	999976992	99.998
1024	119728	119731	999968256	99.997
1518	81273	81274	999982992	99.998

Figure 7.4 and Table 7.1 shows the IO forwarding rate under multiple frame sizes during the execution of the experiments [18]. The gathered information show that the frame size of 64 bytes had the least performance, averaging 1487911 packets per second. This amounts to 99.988% of the theoretical line speed of 1 Gbps. Following this was the performance of near line rates which peaked at 99.998% for frames sizes between 128 and 1518 bytes.

Table 7.2 shows the average, maximum, and jitter latencies for IO forwarding. The average latency gradually increased as the frame size increased. This is expected since larger packets require more time to travel from the ingress port, to main memory, then out the egress port. The maximum latency however, experienced a rather unique trend. It

[1] These values were calculated using Equation 7.1, where the `bps` is 1 Gigabit/s

Table 7.2: IO Latency for the Multi-layer Switch. Source [18]

Packet size (bytes)	**Average latency (μs)**	**Maximum latency (μs)**	**Jitter (μs)**
64	14	103	1
128	15	210	1
256	18	69	3
512	22	67	5
1024	30.25	73	5
1518	36.75	86	1

was observed that 128 byte sized frames experienced the highest latency while the lowest latency was observed for 512 byte frames. This outcome is a result of the behaviour of the switch; the maximum latency is mostly affected at higher packets per second (pps). The jitter, however gradually increased from 1μs to 5μs as the frame sizes increased from 64 bytes to 1024 bytes. The jitter, however, drops back to 1μs for the 1518 byte frame.

7.2.2 IO Summary

The throughput results show that the IO of the switch is able to forward traffic at near line rate speed for all measured packet sizes as seen in Table 7.1. This means that the IO transfer rate achieved by the switch for frames moving between the ingress and egress port was not hindered by the path between the NIC, CPU, and main memory. The average latency measured for IO operations from the ingress port to an egress port was less 40 μs.

7.3 Layer 2 Benchmarking

Switches and routers function together so that information is correctly forwarded through the network. This feature involves processing of packets based on the information found in the frame header. The results gathered from the IO benchmarks had shown that the performance of the multi-layer switch regarding the internal IO were at line rate speeds hinting that the NIC, the interconnection and memory were able to maintain the link speed as frames moved in and out the switch. In this section, the sets of tests aim to find out what overhead was incurred while the CPU processes the packets, in other words, to find the effect of packet processing on throughput, latency and frame loss. The throughput desired for this implementation would be 1 Gbps on all the ports, naturally.

By referring to the results found in the IO test, comparing the IO transfer rate and the L2 packet processing would reveal the extra overhead that the CPU introduces when performing L2 packet processing.

7.3.1 Results – Single Core

Table 7.3: Layer 2 Forwarding Rate.

Packet size (bytes)	**TRex line rate (PPS)**	**Packet loss %**	**Core Utilisation %**
64	1480760	0.00267	91
128	844352	0	80
256	452695	0	54
512	234790	0	28
1024	119280	0	12
1500	82025	0	8

The results in Table 7.3 shows the packet loss and the CPU utilisation for each packet size taken in earlier experiments [18]. Figure 7.5 shows the CPU utilisation of four PMD threads whose total CPU utilisation is 91%. The result shows packet loss during these tests. The packet loss seen for 64 byte packet was 0.00267%. It also shows the single core CPU utilisation which peaked at 91% during the highest PPS. The CPU utilisation however, decreased as the PPS decreased. This is expected since larger packets require a lower number of operations.

```
openvswitch@openvswitch:~$ sudo ovs-appctl dpif-netdev/pmd-rxq-show
pmd thread numa_id 0 core_id 1:
  isolated : false
  port: ovsport1          queue-id:  0  pmd usage: 25 %
  port: ovsport2          queue-id:  0  pmd usage: 19 %
  port: ovsport3          queue-id:  0  pmd usage: 25 %
  port: ovsport4          queue-id:  0  pmd usage: 25 %
openvswitch@openvswitch:~$
```

Figure 7.5: PMD Thread Utilisation on a Single Core for 64 Byte Frames.

7.3.2 Analysis – Single Core

These sets of tests showed the behaviour of the CPU during L2 processing. The packet loss experienced was seen to occur only during the beginning of the tests. It was found that the loss was caused by the time taken by the switch to cache actions from the flow table in the `openvswichd` program to the `user-space data-plane` (Section 5.3). The `user-space`

`data-plane` is seen to improve packet processing by caching actions. Once these actions were cached, no losses were experienced even after running the tests multiple times. This behaviour may seem to have a potentially negative impact on the performance, but in the real world, TCP/IP network connection information is not spontaneously sent in bulk before the communication between the sender and receiver is initialised [111, 112]. As a result, this initialisation gives the switch enough time to cache actions of flow entries in the `user-space data-plane` before data transfer commences, and as such, the packet loss experienced would unlikely occur. High CPU usage was observed, especially for smaller packet sizes, during higher PPS. This may result in a potential lagging that delays packet processing. The CPU may end up being the bottleneck of the system, especially as more advanced computations such as routing and VLAN tagging would require greater CPU resources.

7.3.3 Results – Three Cores

In these sets of tests, three cores where enabled instead of one. This was to evaluate the significance of adding more cores to the switch's capabilities. Table 7.4 shows the L2 switching latencies (average, maximum and jitter) and PMD thread utilisation on three cores.

```
openvswitch@openvswitch:~$ sudo ovs-appctl dpif-netdev/pmd-rxq-show
pmd thread numa_id 0 core_id 1:
  isolated : false
  port: ovsport3           queue-id:  0  pmd usage: 49 %
pmd thread numa_id 0 core_id 2:
  isolated : false
  port: ovsport2           queue-id:  0  pmd usage: 42 %
  port: ovsport4           queue-id:  0  pmd usage: 41 %
pmd thread numa_id 0 core_id 3:
  isolated : false
  port: ovsport1           queue-id:  0  pmd usage: 48 %
openvswitch@openvswitch:~$
```

Figure 7.6: PMD Thread Utilisation on Three Cores for 64 byte Frames.

Figure 7.6 shows the PMD utilisation for each core during 64 byte frame tests. The addition of CPU cores effectively reduced the maximum core usage to 83% (core 2) while the other cores utilised 49% (core 1) and 48% (core 3). Two PMD threads ran on *core 2* that is the reason why it shows higher core utilisation. The comparison between Table 7.2 and Table 7.4 shows the overhead experienced in average latency between IO forwarding and L2 packet processing. The results show a slight increase in latency between IO and L2 processing. However, the overall *maximum latency* was observed to be between 1400μs and 1800μs with the highest *maximum latency* being for 64 byte frames.

Table 7.4: L2 Switching Latency, Jitter and CPU Utilisation for 3 Cores

Packet size (bytes)	**Core Utilisation 1 : 2 : 3**	**Average latency (μs)**	**Maximum latency (μs)**	**Jitter (μs)**
64	49 : 83 : 48	14.75	1766	1
128	27 : 54 : 26	15	1526	2
256	14 : 28 : 14	18	1568	3
512	7 : 14 : 7	22.5	1660	5
1024	3 : 6 : 3	31.75	1734	5
1280	3 : 6 : 2	37	1446	6
1518	2 : 4 : 2	38	1580	2

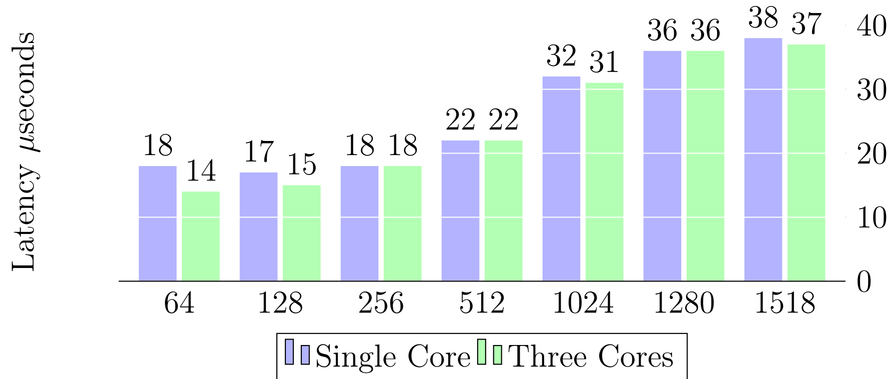


Figure 7.7: Average Latencies for Single Core and Three Cores (Lower is better).

Figure 7.7 and Figure 7.8 show the comparison between average latency and maximum core utilisation of single core and multi-core (for core 2). The addition of cores reduced the average latency especially for smaller frame sizes (shown in Figure 7.7). Not only did the addition of cores reduced the maximum CPU utilisation, it also reduced average latency, improving the performance of the multi-layer switch.

7.3.4 Analysis – Three Cores

Each port was configured with a single queue creating a total of four PMD threads which were defined as `ovsport1`, `ovsport2`, `ovsport3`, and `ovsport4`. Cores 1 and 3 are each running a single PMD thread, while core 2 ran two threads. The assignment of PMD threads to cores was done automatically [79]. The OVS-DPDK framework include commands for load balancing between each PMD threads across all running cores. This means that the switch's PMD threads can be distributed across cores for efficient use of the CPU.

The performance of the switch was seen to have improved when other cores were enabled

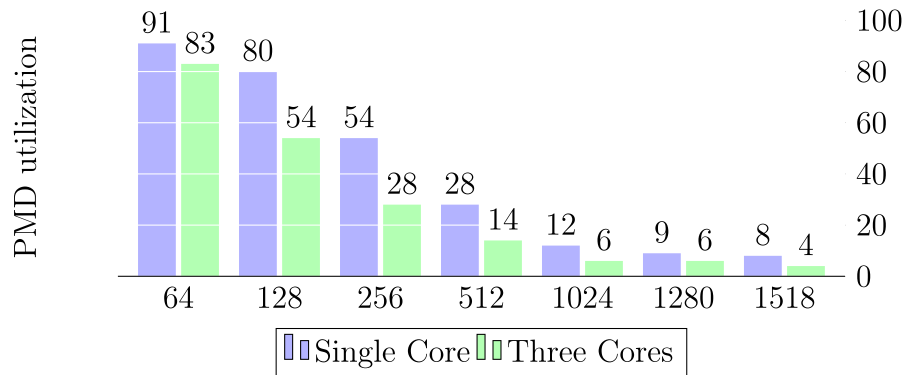


Figure 7.8: Maximum CPU Utilisation Per Core (lower is better).

(when comparing the results in Figure 7.5 (single core) and Figure 7.6 (three cores)). This resulted in a lower CPU utilisation. The DPDK library takes advantage of the multi-threaded architecture which allows the processing of packets in bulk. Hence, the adding of more cores reduced the average latency due to packets being processed simultaneously in a multi-threaded environment.

7.3.5 Layer 2 Benchmarking Summary

These tests illustrate the performance and behaviour of the switch when L2 processing is considered. The average latency between the IO forwarding and L2 packet processing generally experienced a slight increase of less than 1 μsecond. This means that, on average, packet processing under L2 load has little effect on the delay. The majority of the delay is due to IO operations between the NIC, the CPU and the main memory. The maximum latency, unlike the average latency, showed a significant increase. This may be caused by the queuing of packets, a limitation within the NIC, or memory access during L2 processing.

The CPU utilisation for both single core and three cores tests increased as the packet size decreased. This leads to conclude, reasonably, that higher pps require more CPU resources. On average, the changes in delay were minor, demonstrating that L2 packet processing had little effect on average latency.

7.4 Layer 3 Benchmarking

L3 processing incurs a greater penalty, due to the number of actions needed to fulfil the behaviour of a router. This section shows the performance characteristics of the switch during L3 benchmarks.

7.4.1 Results

Table 7.5: Layer 3 Switching Latency, Jitter and CPU Utilisation for 3 Cores.

Packet size (bytes)	**Core Utilisation 1 : 2 : 3**	**Average latency (μs)**	**Maximum latency (μs)**	**Jitter (μs)**
64	56 : 86 : 54	91.77	1781	1
128	31 : 62 : 30	499	2082	1
256	16 : 32 : 16	42	1766	3
512	8 : 16 : 8	47	1476	5
1024	4 : 8 : 4	55	1329	8
1280	3 : 6 : 3	62	1527	6
1518	2 : 4 : 2	67	1070	11

Table 7.5 shows the CPU utilisation and latencies during L3 routing with different frame sizes. Comparing the results from Table 7.4 (L2 latencies) and Table 7.5 (L3 latencies), there is a significant change that is evident in the *average latency* between L2 and L3 packet processing. This is due to the L3 packet processing that requires altering the header information for each packet before outputting to a port. CPU utilisation also followed a similar trend, where the maximum per core utilisation peaked at 86% (42% + 44%) for 64 byte frames (shown in Figure 7.9). In spite of the added complexity, the switch maintained link speed without dropping of packets. Figure 7.10 shows the comparison between L2 and L3 packet processing for multiple frame sizes.

```
openvswitch@openvswitch:~$ sudo ovs-appctl dpif-netdev/pmd-rxq-show
pmd thread numa_id 0 core_id 1:
  isolated : false
  port: ovsport3          queue-id:  0  pmd usage: 56 %
pmd thread numa_id 0 core_id 2:
  isolated : false
  port: ovsport2          queue-id:  0  pmd usage: 44 %
  port: ovsport4          queue-id:  0  pmd usage: 42 %
pmd thread numa_id 0 core_id 3:
  isolated : false
  port: ovsport1          queue-id:  0  pmd usage: 54 %
```

Figure 7.9: Thread Utilisation for 64 Byte Frame.

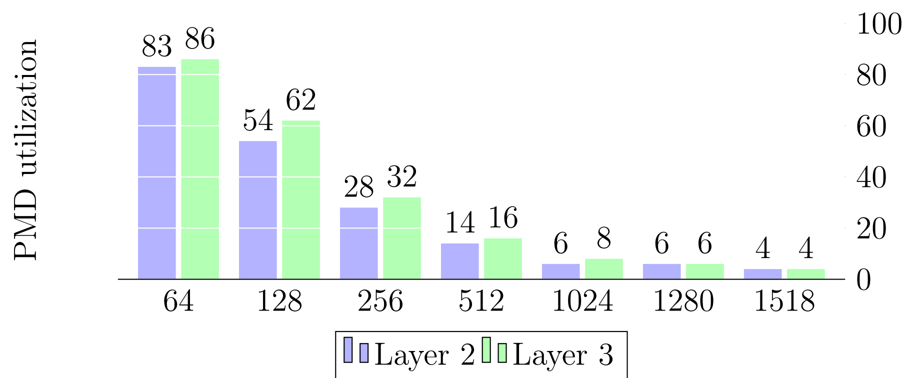


Figure 7.10: Comparison Between Layer 2 and Layer 3 CPU utilisation.

7.4.2 Layer 3 Benchmarking Summary

The trace tool used earlier showed the actions that the switch executes during L2/L3 benchmarking (Section 6.5.4.3 and Section 6.5.4.4 respectively). It is evident that routing requires greater CPU resources because of the number of actions required to process a single packet. As such, a higher CPU utilisation was seen in the benchmarking tests for L3 compared to L2 forwarding (shown in Figure 7.10), resulting in increased delay.

7.5 OpenFlow Compliance Testing

This section covers tests ran to validate the OpenFlow functionality of the multi-layer switch. This is meant to check if the switch meets the function requirements defined in the OpenFlow specification. Faucet's unit-test framework reduces the time required to manually test the hardware through automated testing [54]. The framework simplified the process of testing and verified that the switch fulfilled both feature-level and system-level interoperability.

7.5.1 Compliance Results

Feature-level and system-level testing was performed by executing multiple unit tests. A total of 141 tests were ran which included testing the switch's features such as multiple tables flow injection, VLAN switching, IPv4 and IPv6 routing (static and BGP routing protocol), access control lists (ACLs), port mirroring and policy-based forwarding among

Table 7.6: Summary Results for Compliance Testing.

Test	Result	Remark
Group Table Untagged IPv4 Route Test	909 Mbps to 10.0.2.1, 903 Mbps to 10.0.1.1	Routing between two hosts, 10.0.1.1 and 10.0.2.1
Group Table Untagged IPv6 Route Test	893 Mbps to fc00::20:1, 896 Mbps to fc00::10:1	Routing between two hosts, fc00::10:1 and fc00::20:1
IPv4 Tuple Test	pushed 1024 IPv4 tuples	Testing table capabilities
IPv6 Tuple Test	pushed 1024 IPv6 tuples	Testing table capabilities
Single L2 Learn MACs On Port Test	verified 4096 hosts learned in 120 sec	Learn 4086 hosts and verifying connectivity
Single L3 Learn MACs On Port Test	verified 2048 hosts learned in 60 sec	Learn 2048 hosts and verifying connectivity
Untagged ApplyMeter Test	10000 packets transmitted, 7925 received, 20% loss	Test ACL with drop lossy meter
Evaluation of 141 tests	OK	elapse time 9234.158s

others. Table 7.6 shows a summary of the unit tests conducted from the 141 tests ran. Appendix B.1 shows the full results.

The results show that the switch supports multiple OpenFlow switch features. The group table routing for IPv4 and IPv6 utilised the OpenFlow Group table. The benchmarked speed for group table routing was 909 Mbps and 903 Mbps between hosts in an IPv4 network while routing for IPv6 was 893 and 896 Mbps. In the tuple test, the framework tested the up to 1024 table rules for both IPv4 and IPv6. However, the switch supports a maximum of 1'000'000 flows per table with a total of 254 tables (shown in Appendix A.2). This shows that the switch is able to hold a large number of flow table entries. An important function of the switch is the ability to learn host MAC addresses and allow necessary connectivity between hosts in reasonable time. The results also show that the multi-layer switch is capable of learning L2 MAC addresses for 4096 hosts in 120 seconds, while the learning and verification of L3 MAC addresses for 2048 hosts was done in 60 seconds.

7.5.2 Compliance Summary

Faucet's comprehensive tests ran 141 unit tests, where the network was virtualised in Mininet. The switch supported the group table forwarding, an experimental features not

fully supported by all OpenFlow devices [54]. The tuple tests show how much time the switch would require to learn and verify end-hosts when connected within a network of thousands of machines. This defines how much time the switch requires to learn rules necessary for switching and routing between all nodes of the network. The results had shown that this switch is able to maintain flow entries for thousands of hosts.

The Faucet controller uses a physically centralised controller due its small footprint size and simple architecture. A physically distributed SDN controller architecture is a better solution for large-scale SDN networks [91].

7.6 Summary

The chapter began by detailing the hardware and setup for the tests that were performed. The first group of tests was to check whether the IO operations within the multi-layer switch could sustain forwarding at 1 Gbps for different frame sizes f rom the minimum to maximum (64 bytes to 1518 bytes). The initial results showed that the hardware supported IO forwarding at a rate of millions of packets per second (Table 7.1). This was followed by the measurement of the CPU performance, in terms of L2 packet processing and latency (Section 7.3). It was observed that L2 processing had little effect on the average latency, where the average latency of the switch was less than 40 μs.

The L3 benchmark results show that L3 computation is a more expensive operation in terms of latency where the delay doubles and in some cases is five times the latency seen in IO forwarding. Nonetheless, the average latency for all frame sizes was less than 100 μs. Finally in Section 7.5, the compliance for OpenFlow features for this multi-layer switch were tested.

Chapter 8

Real Life Testing

This chapter demonstrates three small-scale networking use cases where the multi-layer switch, the Ryu controller, and a selection of applications are used. These examples reflect real-world setups and show how one would build, configure, and manage an SDN network in support of specific use cases. The chapter includes the administration of the multi-layer switch, applying network configurations for specified network topologies, verify network connectivity between each host, and finally benchmarking the network.

Section 8.1 gives an overview of the use cases presented. Section 8.2 illustrates the process of configuring a L2 network, while Section 8.3 and Section 8.4 demonstrate the application of L3 routing and VLAN partitioning. Finally, Section 8.5 concludes the chapter by observing bandwidth usage and analysing the switching/routing tables for the traffic within each SDN network.

8.1 Overview

The use cases described in the sections below cover the core functionality of the multi-layer switch which includes forwarding (L2) and routing (L3). This chapter aims to also demonstrate functions that are seen in certain commercial designs, such as traffic isolation through VLANs and network device monitoring. In an effort to simplify the process of implementing these functions, the applications discussed in Chapter 6 were used in the configuration of policies to define the behaviour of the network.

Each use case is presented as follows:

1. The topology of the network use case.
2. Provide network configuration entered through the web interface.
3. The display of flow entries added to the switch's OpenFlow table.
4. Testing of relevant environment. This includes tests such as host to host connectivity and bandwidth benchmarks.
5. Analysis of the results of the realised network.

8.1.1 Policy Configuration

The configuration of policies in each use case were entered only through the web user interface (UI). JSON objects were used as inputs to the web UI to configure each switch. A JSON object defines the policy/behaviour that maintains the network. Within the UI, the configuration process followed three steps: (1) creating the policy, (2) verifying the policy, and (3) applying the policy. From a network administrator's point of view, the UI shows three key areas (the web UI is shown in Figure C.1 in the Appendix) which are: the graphical section which shows view the topology of the network, the table entries section which allows one to view the state of the OpenFlow table for the selected switch, and the configuration toolbar which is where the policies (JSON objects) are added for a switch (the configuration toolbar is shown in Figure C.2 in the Appendix). Within the configuration toolbar, the user can add, verify, and apply policies.

8.1.2 Specification of Hosts Used

Table 8.1: Hosts Hardware & Software Specifications.

Item	Description
Platform	H170 PRO GAMING
Chipset	Intel® 760G (780L)
CPU	Intel(R) Core™i5-6400 CPU (6MB Cache, 2.70GHz), 4 core, 4 threads
Memory	2x4GB, Single Channel @ 2133MHz
Operating System	Ubuntu Server 16.04 64-bit (GNU/Linux 4.15.0-39-generic x86_64)
NIC	Intel® Ethernet Connection (2) I219-V

Table 8.1 shows the hardware and software specifications for each connected host, and Figure 8.1 shows the overview structure used in each use case. The switch was directly connected to four hosts using Ethernet cables. The link speed over each Ethernet port was 1 Gbps. The same physical network topology was used in each use case where the network configuration was entered into the web UI. The network benchmark tests were ran using iPerf network-benchmarking tool such that all data ports on the switch were tested simultaneously. Each test was conducted for minimum period of 60s to determine the data rate under continuous load. This was to observe the throughput supported by the switch.

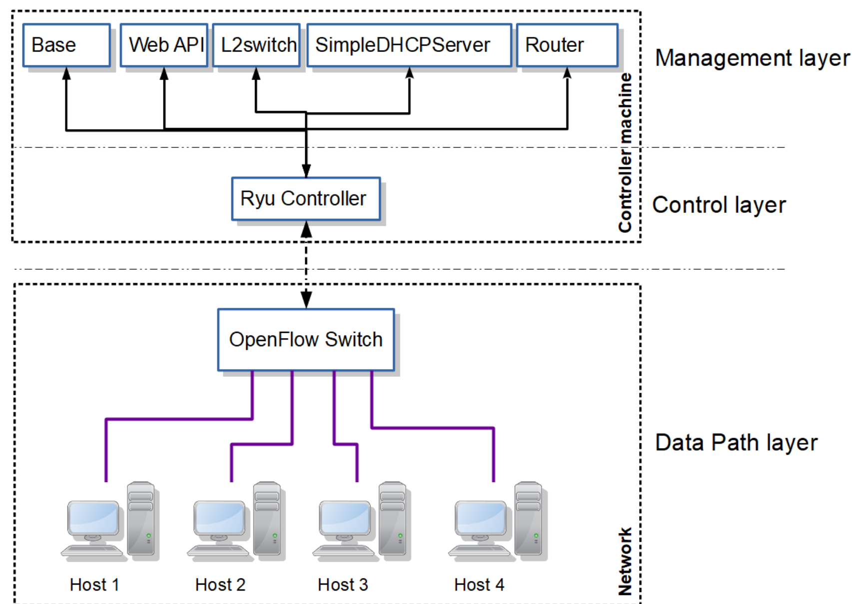


Figure 8.1: SDN Network Architecture.

8.2 Hub and Learning Switch Use Case

L2 functionality enables a switch to support communication among devices within the same network by making use of the source and destination MAC addresses. The decision taken on how to forward traffic is performed within the controller. To demonstrate this concept, the network was setup as shown in Figure 8.1. Each host was assigned an IP by the DHCP service offered by the `SimpleDHCPServer` application where all IPs were selected from the same subnet. Once the switch was configured and IPs were assigned, the `L2Switch` application managed the logic/behaviour of an L2 switch.

8.2.1 Configuration

The policy to define L2 functionality is as follows:

1. Enable DHCP service, where the IP address of the DHCP server is 10.0.0.1.
2. IP addresses must be assigned from the same subnet. Hence, the CIDR (Classless Inter-Domain Routing) was set to 10.0.0.0/24.
3. Initialise the multi-layer switch as a L2 switch, i.e. to only use L2 functionality.

The following JSON object was created to achieve the policy defined above.

```
1 {
2   "subnet":{"all":"10.0.0.0/24"},
3   "dhcp_ips":{"all":"10.0.0.1"},
4   "is_router":false,
5   "enable_dhcp":true
6 }
```

The JSON object above initialises all given ports on the switch to be under the same subnet of 10.0.0.0/24, shown in line 2. The DHCP service was enabled to allow automatic allocation of IP addresses where the DHCP server's IP is 10.0.0.1 (lines 5 and 3). Line 4 configures the multi-layer switch to only use L2 functionality.

8.2.1.1 Verifying Controller Settings

During this stage of the configuration, the controller was queried to verify the created policy. The message replied is shown in Listing 8.1.

The received message shows the acknowledgement for the defined policy and validates if the policy is applicable to the switch with id `0000001b21a6da00` (i.e. the ID of the switch, Section 6.5.2.4). Listing 8.2 shows the confirmation message from the controller after applying the policy.

8.2.1.2 Verifying Hosts Settings

By executing the Linux `ifconfig` command, the IP addresses and the interface names are shown. The result shows IP addresses for Host 1, Host 2, Host 3, and Host 4. Listing 8.3 shows IP address for Host 1 and in Appendix D.1 shows IP addresses for Host 2, Host 3, and Host 4. This shows that the entered policy correctly configured the network.

Listing 8.1: Validation of Configuration.

```
{
  "switch_id": "0000001b21a6da00",
  "command_result": [
    {
    "result": "success",
    "details": [
      "Add subnet 10.0.0.0/24 [id=1]",
      "Add subnet 10.0.0.0/24 [id=2]",
      "Add subnet 10.0.0.0/24 [id=3]",
      "Add subnet 10.0.0.0/24 [id=4]",
      "Add dhcp_ips: 10.0.0.1 [id=1]",
      "Add dhcp_ips: 10.0.0.1 [id=2]",
      "Add dhcp_ips: 10.0.0.1 [id=3]",
      "Add dhcp_ips: 10.0.0.1 [id=4]",
      "DHCP set [enable=True]"
    ]
    }
  ]
}
```

Listing 8.2: Confirmation For the Configured Input.

```
[
  "Success",
  "Updating switch values",
  "Configure as:",
  "DHCP: True",
  "DHCP addresses: {1: '10.0.0.1', 2: '10.0.0.1', 3: '10.0.0.1', 4: '10.0.0.1'}",
  "Router: False",
  "Subnets: {1: '10.0.0.0/24', 2: '10.0.0.0/24', 3: '10.0.0.0/24', 4: '10.0.0.0/24'}"
]
```

Listing 8.3: Host 1 Network Configuration.

```
host1@ubuntu:~$ ifconfig
enp0s31f6 Link encap:Ethernet  HWaddr 30:5a:3a:7c:cd:9d
          inet addr:10.0.0.254  Bcast:10.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::325a:3aff:fe7c:cd9d/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3580 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1243 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1210116 (1.2 MB)  TX bytes:426174 (426.1 KB)
          Interrupt:16 Memory:f7100000-f7120000
...
```

8.2.2 Results

Information and logs were gathered during the IP addressing, ping testing, and L2 switching (OpenFlow entries). Appendix D.2 shows logs from the DHCP application during the allocation of IP addresses. The logs show the interaction and the order that the IPs were addressed. The `L2Switch` application maintained the L2 logic within the network by building the forwarding table. The connectivity between Host 1 and other hosts was verified using ping tests which were successful as shown in Listing 8.4. It was observed

that the initial ping message going to each host took longer compared to consecutive pings. This is due to the time taken in learning of MAC addresses, and the adding of flows during the first ping.

Listing 8.4: Host 1 pinging Host 2 (IP: 10.0.0.253), Host 3 (IP: 10.0.0.252), and Host 4 (IP: 10.0.0.251).

```
host1@ubuntu:~$ ping -c 4 10.0.0.253
PING 10.0.0.253 (10.0.0.253) 56(84) bytes of data.
64 bytes from 10.0.0.253: icmp_seq=1 ttl=64 time=20.0 ms
64 bytes from 10.0.0.253: icmp_seq=2 ttl=64 time=0.207 ms
64 bytes from 10.0.0.253: icmp_seq=3 ttl=64 time=0.189 ms
64 bytes from 10.0.0.253: icmp_seq=4 ttl=64 time=0.188 ms

--- 10.0.0.253 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 0.188/5.146/20.002/8.577 ms

host1@ubuntu:~$ ping -c 4 10.0.0.252
PING 10.0.0.252 (10.0.0.252) 56(84) bytes of data.
64 bytes from 10.0.0.252: icmp_seq=1 ttl=64 time=19.3 ms
64 bytes from 10.0.0.252: icmp_seq=2 ttl=64 time=0.196 ms
64 bytes from 10.0.0.252: icmp_seq=3 ttl=64 time=0.197 ms
64 bytes from 10.0.0.252: icmp_seq=4 ttl=64 time=0.204 ms

--- 10.0.0.252 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 0.196/4.992/19.371/8.301 ms

host1@ubuntu:~$ ping -c 4 10.0.0.251
PING 10.0.0.251 (10.0.0.251) 56(84) bytes of data.
64 bytes from 10.0.0.251: icmp_seq=1 ttl=64 time=14.4 ms
64 bytes from 10.0.0.251: icmp_seq=2 ttl=64 time=0.196 ms
64 bytes from 10.0.0.251: icmp_seq=3 ttl=64 time=0.200 ms
64 bytes from 10.0.0.251: icmp_seq=4 ttl=64 time=0.197 ms

--- 10.0.0.251 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 0.196/3.768/14.482/6.185 ms
```

Once the policy was applied and the MAC addresses of the hosts were noted, the state of the flow table was taken. The flow table had the following flow entries:

Listing 8.5: OpenFlow Table for L2 Forwarding.


```
1 { "actions" : [ "OUTPUT:CONTROLLER" ], "idle_timeout": 0, "cookie": 0, "packet_count": 76, "
    ↪ hard_timeout": 0, "byte_count": 4560, "duration_sec": 2715, "duration_nsec": 944000000,
    ↪  "priority": 65535, "length": 96, "flags": 0, "table_id": 0, "match": { "eth_type":
    ↪ 35020, "eth_dst": "01:80:c2:00:00:0e" }},
2 { "actions" : [ "OUTPUT:CONTROLLER" ], "idle_timeout": 0, "cookie": 1000, "packet_count": 8, "
    ↪ hard_timeout": 0, "byte_count": 2736, "duration_sec": 2715, "duration_nsec": 959000000,
    ↪  "priority": 100, "length": 104, "flags": 0, "table_id": 0, "match": { "eth_type":
    ↪ 2048, "tp_src": 68, "nw_proto": 17, "tp_dst": 67 }},
3 { "actions" : [ "OUTPUT:1" ], "idle_timeout": 0, "cookie": 2000, "packet_count": 10, "
    ↪ hard_timeout": 0, "byte_count": 904, "duration_sec": 1579, "duration_nsec": 417000000,
    ↪ "priority": 1, "length": 96, "flags": 0, "table_id": 0, "match": { "eth_dst": "30:5a:3a
    ↪ :7c:cd:9d", "in_port": 2 }},
4 { "actions" : [ "OUTPUT:2" ], "idle_timeout": 0, "cookie": 2000, "packet_count": 9, "
    ↪ hard_timeout": 0, "byte_count": 806, "duration_sec": 1579, "duration_nsec": 410000000,
```

```
    "priority": 1, "length": 96, "flags": 0, "table_id": 0, "match": { "eth_dst": "30:5a:3a:7e:36:a0", "in_port": 1 }},
5 { "actions" : [ "OUTPUT:1" ], "idle_timeout": 0, "cookie": 2000, "packet_count": 10981819, "hard_timeout": 0, "byte_count": 15863485306, "duration_sec": 1517, "duration_nsec": 499000000, "priority": 1, "length": 96, "flags": 0, "table_id": 0, "match": { "eth_dst": "30:5a:3a:7c:cd:9d", "in_port": 3 }},
6 { "actions" : [ "OUTPUT:3" ], "idle_timeout": 0, "cookie": 2000, "packet_count": 10982450, "hard_timeout": 0, "byte_count": 15861036504, "duration_sec": 1517, "duration_nsec": 493000000, "priority": 1, "length": 96, "flags": 0, "table_id": 0, "match": { "eth_dst": "30:5a:3a:7c:d3:58", "in_port": 1 }},
7 { "actions" : [ "OUTPUT:1" ], "idle_timeout": 0, "cookie": 2000, "packet_count": 10, "hard_timeout": 0, "byte_count": 904, "duration_sec": 1510, "duration_nsec": 013000000, "priority": 1, "length": 96, "flags": 0, "table_id": 0, "match": { "eth_dst": "30:5a:3a:7c:cd:9d", "in_port": 4 }},
8 { "actions" : [ "OUTPUT:4" ], "idle_timeout": 0, "cookie": 2000, "packet_count": 9, "hard_timeout": 0, "byte_count": 806, "duration_sec": 1510, "duration_nsec": 009000000, "priority": 1, "length": 96, "flags": 0, "table_id": 0, "match": { "eth_dst": "30:5a:3a:7a:0d:b9", "in_port": 1 }},
9 { "actions" : [ "OUTPUT:2" ], "idle_timeout": 0, "cookie": 2000, "packet_count": 10, "hard_timeout": 0, "byte_count": 904, "duration_sec": 1347, "duration_nsec": 727000000, "priority": 1, "length": 96, "flags": 0, "table_id": 0, "match": { "eth_dst": "30:5a:3a:7e:36:a0", "in_port": 3 }},
10 { "actions" : [ "OUTPUT:3" ], "idle_timeout":0, "cookie":2000, "packet_count":9, "hard_timeout":0, "byte_count":806, "duration_sec":1347, "duration_nsec":725000000, "priority":1, "length":96, "flags":0, "table_id":0, "match":{ "eth_dst":"30:5a:3a:7c:d3:58", "in_port": 2 }},
11 { "actions" : [ "OUTPUT:2" ], "idle_timeout": 0, "cookie": 2000, "packet_count": 10988721, "hard_timeout": 0, "byte_count": 15863940960, "duration_sec": 1337, "duration_nsec": 318000000, "priority": 1, "length": 96, "flags": 0, "table_id": 0, "match": { "eth_dst": "30:5a:3a:7e:36:a0", "in_port": 4 }},
12 { "actions" : [ "OUTPUT:4" ], "idle_timeout": 0, "cookie": 2000, "packet_count": 10985823, "hard_timeout": 0, "byte_count": 15859424236, "duration_sec": 1337, "duration_nsec": 316000000, "priority": 1, "length": 96, "flags": 0, "table_id": 0, "match": { "eth_dst": "30:5a:3a:7a:0d:b9", "in_port": 2 }},
13 { "actions" : [ "OUTPUT:4" ], "idle_timeout": 0, "cookie": 2000, "packet_count": 10, "hard_timeout": 0, "byte_count": 904, "duration_sec": 1242, "duration_nsec": 607000000, "priority": 1, "length": 96, "flags": 0, "table_id": 0, "match": { "eth_dst":"30:5a:3a:7a:0d:b9", "in_port": 1 }},
14 { "actions" : [ "OUTPUT:3" ], "idle_timeout": 0, "cookie": 2000, "packet_count": 9, "hard_timeout": 0, "byte_count": 806, "duration_sec": 1242, "duration_nsec": 601000000, "priority": 1, "length": 96, "flags": 0, "table_id": 0, "match": { "eth_dst": "30:5a:3a:7c:d3:58", "in_port": 4 }},
15 { "actions" : [ "OUTPUT:CONTROLLER" ], "idle_timeout": 0, "cookie": 1000, "packet_count": 2706, "hard_timeout": 0, "byte_count": 198148, "duration_sec": 2715, "duration_nsec": 959000000, "priority": 0, "length": 80, "flags": 0, "table_id": 0, "match": {} }
```


The flows added by the `L2Switch` application were to allow communication between hosts connected to the switch. Lines 3 and 4 show flow entries for the bidirectional traffic between Host 1 (on port 1) and Host 2 (on port 2) where line 3 is the flow entry for traffic coming from port 2 going to port 1 while line 4 is the flow entry for traffic coming from port2 going to port 1. The remaining lines, 5 to 14, demonstrate the flow entries that enable bidirectional communication within the network.

8.2.3 Throughput Tests

The throughput benchmark measurements were from host to host, which required client-server pairs. Host 1 was paired to Host 2, and Host 3 to Host 4. This test measures the throughput for TCP traffic between each pair, through bidirectional streams of data. Listing 8.6 and Listing 8.7 shows the results for the network benchmarks for forwarding between the host pairs of Host 1 and Host 2, Host 3 and Host 4. The throughput recorded for L2 switching was 932 Mbit/s.

Listing 8.6: iPerf Benchmark Between Host 1 and Host 2.

```
host1@ubuntu:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 10.0.0.254 port 5001 connected with 10.0.0.253 port 51700
------------------------------------------------------------
Client connecting to 10.0.0.253, TCP port 5001
TCP window size:  264 KByte (default)
------------------------------------------------------------
[  5] local 10.0.0.253 port 50816 connected with 10.0.0.254 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-300.0 sec  32.5 GBytes   931 Mbits/sec
[  4]  0.0-300.0 sec  32.5 GBytes   931 Mbits/sec
```

Listing 8.7: iPerf Benchmark Between Host 4 and Host 3.

```
host4@ubuntu:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 10.0.0.251 port 5001 connected with 10.0.0.252 port 39610
------------------------------------------------------------
Client connecting to 10.0.0.251, TCP port 5001
TCP window size:  280 KByte (default)
------------------------------------------------------------
[  5] local 10.0.0.252 port 52154 connected with 10.0.0.251 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-300.0 sec  32.5 GBytes   931 Mbits/sec
[  4]  0.0-300.0 sec  32.5 GBytes   931 Mbits/sec
```

8.2.4 L2Switch Summary

The configuration of the network was initiated by defining a policy for L2 functionality. The policy, represented by a JSON object, defines the behaviour which is translated into OpenFlow rules and are installed as flow entries in the switch's table. These entries would instruct the switch to capture unknown and forward known L2 traffic. As seen in Section 8.2.2, the `L2switch` application learnt the MAC addresses for each connected host

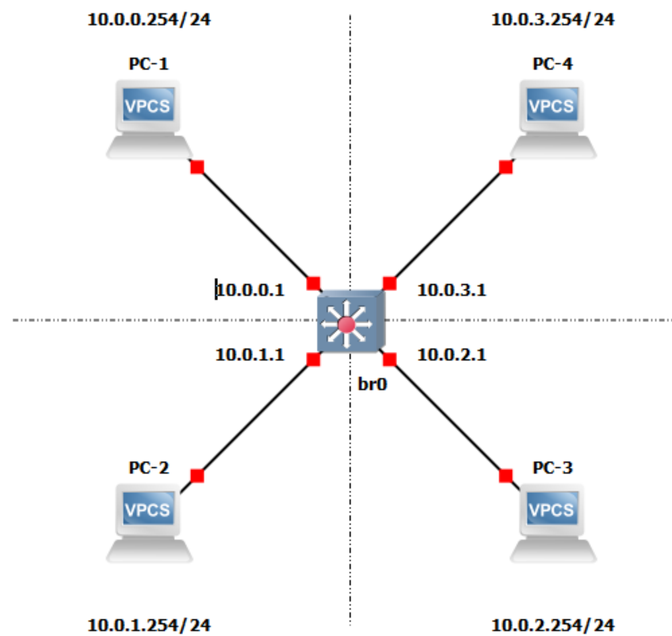


Figure 8.2: Routing Use Case Topology.

and added flow entries so that traffic is directed based on the MAC destination (shown in the logs). The result yielded a bandwidth of 931 Mbit/s, hence the total switching capacity was 7448 Mbit/s, as the aggregate of all four ports.

8.3 Routing Use Case

Routing enables communication between devices in different networks. This is done by identifying connected devices using IP addresses. Interfaces on a router each have an IP address, and as such, should reply to ping and ARP messages. The decision-making and behaviour of a router is established in the controller by the `Router` application (Section 6.6.4). To demonstrate this concept, four subnets were created. Figure 8.2 depicts the topology and IP addresses used in the experiment.

8.3.1 Configuration

Below are the configuration rules used to realise the network policy reflected in Figure 8.2 for L3 routing.

- IP addresses on each interface are to be on different subnets. Hence, the CIDRs are set to 10.0.0.0/24, 10.0.1.0/24, 10.0.2.0/24 and 10.0.3.0/24 on each interface.
- The interfaces for the switch were assigned IP addresses 10.0.0.1, 10.0.1.1, 10.0.2.1 and 10.0.3.1. This is to identify each of the router's interfaces. These IP addresses are also to be used as the addresses of the DHCP servers.
- Initialise the switch as a router; this use case uses L2 and L3 functionality. L2 functionality is handled by the *normal pipeline* (Section 3.1.3) of the router.

Using the above configuration and the web UI the following JSON object was created, entered, and applied.

Listing 8.8: Policy for L3 Routing and IP Allocation

```
1 {
2   "subnet":{"1":"10.0.0.0/24", "2":"10.0.1.0/24", "3":"10.0.2.0/24", "4":"10.0.3.0/24"},
3   "router_ip":{"1":"10.0.0.1", "2":"10.0.1.1", "3":"10.0.2.1", "4":"10.0.3.1"},
4   "dhcp_ips":{"1":"10.0.0.1", "2":"10.0.1.1", "3":"10.0.2.1", "4":"10.0.3.1"},
5   "is_router":true,
6   "enable_dhcp":true
7 }
```

The above policy configures the switch as a router with the DHCP service included. The interfaces on the router were defined to have the following DHCP server address and port number of 10.0.0.1 – port 1, 10.0.1.1 – port 2, 10.0.2.1 – port 3, and 10.0.3.1 – port 4.

8.3.1.1 Verifying Controller Settings

Verifying the input configurations was done by querying the controlled. This displayed the message from the web UI which is shown in Listing 8.9.

This confirms that the above policy was validated against the switch. The result below (Listing 8.10) shows the confirmation from the controller after applying the policy completing the configuration of the switch.

8.3.1.2 Verifying Hosts Settings

By running Linux tools, `ifconfig` and `ip`, information about the network such as IP address, gateway, and subnets were collected to verify the correctness of the applied

Listing 8.9: Validation of Configuration.

```
{
  "switch_id": "0000001b21a6da00",
  "command_result": [
    {
    "result": "success",
    "details": [
      "Add subnet 10.0.0.0/24 [id=1]",
      "Add subnet 10.0.1.0/24 [id=2]",
      "Add subnet 10.0.2.0/24 [id=3]",
      "Add subnet 10.0.3.0/24 [id=4]",
      "Add internal_router_ips: 10.0.0.1 [id=1]",
      "Add internal_router_ips: 10.0.1.1 [id=2]",
      "Add internal_router_ips: 10.0.2.1 [id=3]",
      "Add internal_router_ips: 10.0.3.1 [id=4]",
      "Add dhcp_ips: 10.0.0.1 [id=1]",
      "Add dhcp_ips: 10.0.1.1 [id=2]",
      "Add dhcp_ips: 10.0.2.1 [id=3]",
      "Add dhcp_ips: 10.0.3.1 [id=4]"
    ]
    }
  ]
}
```

Listing 8.10: Confirmation of Router settings.

```
[
  "Success",
  "Registering switch",
  "Configure as:",
  "DHCP: True",
  "DHCP addresses: {1: '10.0.0.1', 2: '10.0.1.1', 3: '10.0.2.1', 4: '10.0.3.1'}",
  "Router: True",
  "Router addresses: {1: '10.0.0.1', 2: '10.0.1.1', 3: '10.0.2.1', 4: '10.0.3.1'}",
  "Subnets: {1: '10.0.0.0/24', 2: '10.0.1.0/24', 3: '10.0.2.0/24', 4: '10.0.3.0/24'}"
]
```

policy. Below are the results for the configured network showing the interface and the routing information for each host. Listing 8.11 shows the network configurations for Host 1 which shows: the IP is defined as `10.0.0.254`, the gateway is defined as `10.0.0.1`, and the interface as `enp0s31f6`. The results for Host 2, Host 3, and Host 4's configurations are given in Appendix E.1

The results shown by the `ip` tool lists the routing table. The default route for Host 1 is given by `10.0.0.1` which is the IP address of one of the switch's interface.

8.3.2 Result

After applying the policy, logs were captured which shows the application adding flows. This is shown in Listing 8.12. The logs show the router application adding the first group of flows to the switch. The cookie (see Section 3.1.3.1) helps in keeping track

Listing 8.11: Host 1 Network Configurations.

```
host1@ubuntu:~$ ifconfig
enp0s31f6 Link encap:Ethernet  HWaddr 30:5a:3a:7a:0d:b9
          inet addr:10.0.0.254  Bcast:10.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::325a:3aff:fe7a:db9/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:30 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:866 (866.0 B)  TX bytes:2904 (2.9 KB)
          Interrupt:16 Memory:f7100000-f7120000
...
host1@ubuntu:~$ ip route list
default via 10.0.0.1 dev enp0s31f6
10.0.0.0/24 dev enp0s31f6  proto kernel  scope link  src 10.0.0.254
```

of installed flow entries. The cookies selected ranged between 1 and 4, to identify the grouping for flows added for each host. The installed flows during this process are shown in Listing 8.13. Flow entries shown in lines 1 – 4 were created during the event '`Set host MAC learning (packet in) flow`' seen in the logs (see Listing 8.12, lines 3, 7, 12, and 16) which is used to learn MAC addresses of connected hosts. Flows in lines 5 – 8 instructs the switch to perform L2 forwarding using the OpenFlow 'normal' pipeline. Lines 9 – 12 are IP handling flows. Packets matching these flows were sent to the controller so that the *Router* application learns the IP addresses. The flow entry defined in line 13 is used by the *Router* application to capture IPv4 traffic. The multi-layer switch (configured as a router), requires constant updating of its routing tables, hence, the capturing of IPv4 traffic.

Listing 8.12: Router Logs During Reactive Adding of Flows

```
1  [RT][INFO] switch_id=0000001b21a6da00: Join as router.
2  # Adding flows for IP address '10.0.0.1/24'
3  [RT][INFO] switch_id=0000001b21a6da00: Set host MAC learning (packet in) flow [cookie=0x1]
4  [RT][INFO] switch_id=0000001b21a6da00: Set IP handling (packet in) flow [cookie=0x1]
5  [RT][INFO] switch_id=0000001b21a6da00: Set L2 switching (normal) flow [cookie=0x1]
6  # Adding flows for IP address '10.0.1.1/24'
7  [RT][INFO] switch_id=0000001b21a6da00: Set host MAC learning (packet in) flow [cookie=0x2]
8  [RT][INFO] switch_id=0000001b21a6da00: Set IP handling (packet in) flow [cookie=0x2]
9  [RT][INFO] switch_id=0000001b21a6da00: Set L2 switching (normal) flow [cookie=0x2]
10 127.0.0.1 - - [22/Oct/2018 04:28:17] "GET /init/0000001b21a6da00 HTTP/1.1" 200 723 0.013672
11 # Adding flows for IP address '10.0.2.1/24'
12 [RT][INFO] switch_id=0000001b21a6da00: Set host MAC learning (packet in) flow [cookie=0x3]
13 [RT][INFO] switch_id=0000001b21a6da00: Set IP handling (packet in) flow [cookie=0x3]
14 [RT][INFO] switch_id=0000001b21a6da00: Set L2 switching (normal) flow [cookie=0x3]
15 # Adding flows for IP address '10.0.3.1/24'
16 [RT][INFO] switch_id=0000001b21a6da00: Set host MAC learning (packet in) flow [cookie=0x4]
17 [RT][INFO] switch_id=0000001b21a6da00: Set IP handling (packet in) flow [cookie=0x4]
18 [RT][INFO] switch_id=0000001b21a6da00: Set L2 switching (normal) flow [cookie=0x4]
```

Listing 8.13: Added Flows by Router Application.


```
1  { "actions": [ "OUTPUT:CONTROLLER" ], "idle_timeout": 0, "cookie": 1, "packet_count": 0, "
```

```
    ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 197, "duration_nsec": 438000000, "
    ↪ priority": 1038, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "
    ↪ nw_dst": "10.0.0.1" } }
2  { "actions": [ "OUTPUT:CONTROLLER" ], "idle_timeout": 0, "cookie": 2, "packet_count": 0, "
    ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 197, "duration_nsec": 438000000, "
    ↪ priority": 1038, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "
    ↪ nw_dst": "10.0.1.1" } }
3  { "actions": [ "OUTPUT:CONTROLLER" ], "idle_timeout": 0, "cookie": 3, "packet_count": 0, "
    ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 197, "duration_nsec": 433000000, "
    ↪ priority": 1038, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "
    ↪ nw_dst": "10.0.2.1" } }
4  { "actions": [ "OUTPUT:CONTROLLER" ], "idle_timeout": 0, "cookie": 4, "packet_count": 0, "
    ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 197, "duration_nsec": 433000000, "
    ↪ priority": 1038, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "
    ↪ nw_dst": "10.0.3.1" } }
5  { "actions": [ "OUTPUT:NORMAL" ], "idle_timeout": 0, "cookie": 1, "packet_count": 0, "
    ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 197, "duration_nsec": 438000000, "
    ↪ priority": 37, "length": 112, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "
    ↪ nw_src": "10.0.0.0/255.255.255.0", "nw_dst": "10.0.0.0/255.255.255.0" } }
6  { "actions": [ "OUTPUT:NORMAL" ], "idle_timeout": 0, "cookie": 2, "packet_count": 0, "
    ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 197, "duration_nsec": 438000000, "
    ↪ priority": 37, "length": 112, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "
    ↪ nw_src": "10.0.1.0/255.255.255.0", "nw_dst": "10.0.1.0/255.255.255.0" } }
7  { "actions": [ "OUTPUT:NORMAL" ], "idle_timeout": 0, "cookie": 3, "packet_count": 0, "
    ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 197, "duration_nsec": 433000000, "
    ↪ priority": 37, "length": 112, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "
    ↪ nw_src": "10.0.2.0/255.255.255.0", "nw_dst": "10.0.2.0/255.255.255.0" } }
8  { "actions": [ "OUTPUT:NORMAL" ], "idle_timeout": 0, "cookie": 4, "packet_count": 0, "
    ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 197, "duration_nsec": 433000000, "
    ↪ priority": 37, "length": 112, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "
    ↪ nw_src": "10.0.3.0/255.255.255.0", "nw_dst": "10.0.3.0/255.255.255.0" } }
9  { "actions": [ "OUTPUT:CONTROLLER" ], "idle_timeout": 0, "cookie": 1, "packet_count": 0, "
    ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 197, "duration_nsec": 438000000, "
    ↪ priority": 3, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "
    ↪ nw_dst": "10.0.0.0/255.255.255.0" } }
10 { "actions": [ "OUTPUT:CONTROLLER" ], "idle_timeout": 0, "cookie": 2, "packet_count": 0, "
    ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 197, "duration_nsec": 438000000, "
    ↪ priority": 3, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "
    ↪ nw_dst": "10.0.1.0/255.255.255.0" } }
11 { "actions": [ "OUTPUT:CONTROLLER" ], "idle_timeout": 0, "cookie": 3, "packet_count": 0, "
    ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 197, "duration_nsec": 434000000, "
    ↪ priority": 3, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "
    ↪ nw_dst": "10.0.2.0/255.255.255.0" } }
12 { "actions": [ "OUTPUT:CONTROLLER" ], "idle_timeout": 0, "cookie": 4, "packet_count": 0, "
    ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 197, "duration_nsec": 433000000, "
    ↪ priority": 3, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "
    ↪ nw_dst": "10.0.3.0/255.255.255.0" } }
13 { "actions": [ "OUTPUT:CONTROLLER" ], "idle_timeout": 0, "cookie": 3000, "packet_count": 0, "
    ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 197, "duration_nsec": 438000000, "
    ↪ priority": 2, "length": 88, "flags": 0, "table_id": 0, "match": { "dl_type": 2054 } }
```


To initiate traffic flow, ping messages were sent from Host 1 to Host 4. This allowed the multi-layer switch to begin the process of learning MAC addresses. As a result, more flows were added to the OpenFlow table. Listing 8.14 shows successful ping messages between

these hosts. This validates the correctness of the entered routing policy. Listing 8.15 shows the installed flow entries as a result of ping testing. The first behaviour of a router, described in Section 6.6.4, was displayed. During routing, the multi-layer switch contained flows that would decrease the TTL and set the destination MAC address (lines 1 – 4). Interfaces on the multi-layer switch demonstrated the second router capability by replying to ping messages as shown in Listing 8.16. This shows Host 3 receiving ping replies from an interface on the switch.

Listing 8.14: Pinging form Host 1 to Host 4 (10.0.3.254).

```
host1@ubuntu:~$ ping -c4 10.0.3.254
PING 10.0.3.254 (10.0.3.254) 56(84) bytes of data.
64 bytes from 10.0.3.254: icmp_seq=1 ttl=63 time=0.210 ms
64 bytes from 10.0.3.254: icmp_seq=2 ttl=63 time=0.200 ms
64 bytes from 10.0.3.254: icmp_seq=3 ttl=63 time=0.195 ms
64 bytes from 10.0.3.254: icmp_seq=4 ttl=63 time=0.190 ms

--- 10.0.3.254 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2997ms
rtt min/avg/max/mdev = 0.190/0.198/0.210/0.018 ms
```

Listing 8.15: Flows Added for Routing.


```
1 { "actions": [ "DEC_NW_TTL", "SET_FIELD: {eth_src:00:1b:21:a6:da:00}", "SET_FIELD: {eth_dst
    ↪ :30:5a:3a:7a:0d:b9}", "OUTPUT:1" ], "idle_timeout": 1800, "cookie": 1, "packet_count":
    ↪ 60842180, "hard_timeout": 0, "byte_count": 87829836856, "duration_sec": 50, "
    ↪ duration_nsec": 868000000, "priority": 36, "length": 136, "flags": 0, "table_id": 0, "
    ↪ match": { "dl_type": 2048, "nw_dst": "10.0.0.254" } }
2 { "actions": [ "DEC_NW_TTL", "SET_FIELD: {eth_src:00:1b:21:a6:da:01}", "SET_FIELD: {eth_dst
    ↪ :30:5a:3a:7c:d3:58}", "OUTPUT:2" ], "idle_timeout": 1800, "cookie": 2, "packet_count":
    ↪ 60832754, "hard_timeout": 0, "byte_count": 87829607700, "duration_sec": 37, "
    ↪ duration_nsec": 471000000, "priority": 36, "length": 136, "flags": 0, "table_id": 0, "
    ↪ match": { "dl_type": 2048, "nw_dst": "10.0.1.254" } }
3 { "actions": [ "DEC_NW_TTL", "SET_FIELD: {eth_src:00:1b:21:a6:da:03}", "SET_FIELD: {eth_dst
    ↪ :30:5a:3a:7e:36:a0}", "OUTPUT:4" ], "idle_timeout": 1800, "cookie": 4, "packet_count":
    ↪ 60796851, "hard_timeout": 0, "byte_count": 87839034654, "duration_sec": 24, "
    ↪ duration_nsec": 659000000, "priority": 36, "length": 136, "flags": 0, "table_id": 0, "
    ↪ match": { "dl_type": 2048, "nw_dst": "10.0.3.254" } }
4 { "actions": [ "DEC_NW_TTL", "SET_FIELD: {eth_src:00:1b:21:a6:da:02}", "SET_FIELD: {eth_dst
    ↪ :30:5a:3a:7c:cd:9d}", "OUTPUT:3" ], "idle_timeout": 1800, "cookie": 3, "packet_count":
    ↪ 60841207, "hard_timeout": 0, "byte_count": 87839733926, "duration_sec": 4, "
    ↪ duration_nsec": 411000000, "priority": 36, "length": 136, "flags": 0, "table_id": 0, "
    ↪ match": { "dl_type": 2048, "nw_dst": "10.0.2.254" } }
```

Listing 8.16: Pinging Router interfaces from Host 3.

```
host3@ubuntu:~$ ping -c4 10.0.3.1
PING 10.0.3.1 (10.0.3.1) 56(84) bytes of data.
64 bytes from 10.0.3.1: icmp_seq=1 ttl=64 time=7.28 ms
64 bytes from 10.0.3.1: icmp_seq=2 ttl=64 time=7.24 ms
64 bytes from 10.0.3.1: icmp_seq=3 ttl=64 time=7.53 ms
64 bytes from 10.0.3.1: icmp_seq=4 ttl=64 time=7.51 ms

--- 10.0.3.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 7.248/7.392/7.531/0.142 ms
host3@ubuntu:~$ ping -c1 10.0.2.1
PING 10.0.2.1 (10.0.2.1) 56(84) bytes of data.
64 bytes from 10.0.2.1: icmp_seq=1 ttl=64 time=14.5 ms

--- 10.0.2.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 14.558/14.558/14.558/0.000 ms
```

8.3.3 Throughput Tests

The throughput measurements were taken for host-to-host, that is from Host 1 to Host 3, and Host 2 to Host 4. Listing 8.17 and Listing 8.18 shows the results for the network benchmarks for routing between the host pairs Host 1 and Host 3, and host pair Host 2 and Host 4. The recorded throughput during routing was 931 Mbit/s. This meant that the total bandwidth of the device during routing was 7448 Mbit/s. The monitoring of the multi-layer switch, using the sFlow tool, generated the report shown in Figure 8.3 which shows the sFlow's network dashboard during L3 benchmarks. The results gathered show the frame/s during the tests, giving the average frame rate during each minute interval. The benchmark test performed in Listing 8.17 and Listing 8.18 correspond to the frame rate recorded between 6.06 AM and 6.12 AM.

Listing 8.17: iPerf benchmark between Host 1 and Host 2.

```
host1@ubuntu:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 10.0.2.254, TCP port 5001
TCP window size:  425 KByte (default)
------------------------------------------------------------
[  5] local 10.0.0.254 port 37694 connected with 10.0.2.254 port 5001
[  4] local 10.0.0.254 port 5001 connected with 10.0.2.254 port 47084
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-300.0 sec  32.5 GBytes   931 Mbits/sec
[  4]  0.0-300.0 sec  32.5 GBytes   931 Mbits/sec
```

Listing 8.18: iPerf benchmark between Host 4 and Host 3.

```
host4@ubuntu:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 10.0.1.254, TCP port 5001
TCP window size:  280 KByte (default)
------------------------------------------------------------
[  6] local 10.0.3.254 port 49996 connected with 10.0.1.254 port 5001
[  6]  0.0-300.0 sec  32.5 GBytes   931 Mbits/sec
[  5]  0.0-300.0 sec  32.5 GBytes   931 Mbits/sec
[  4] local 10.0.3.254 port 5001 connected with 10.0.1.254 port 40154
------------------------------------------------------------
```

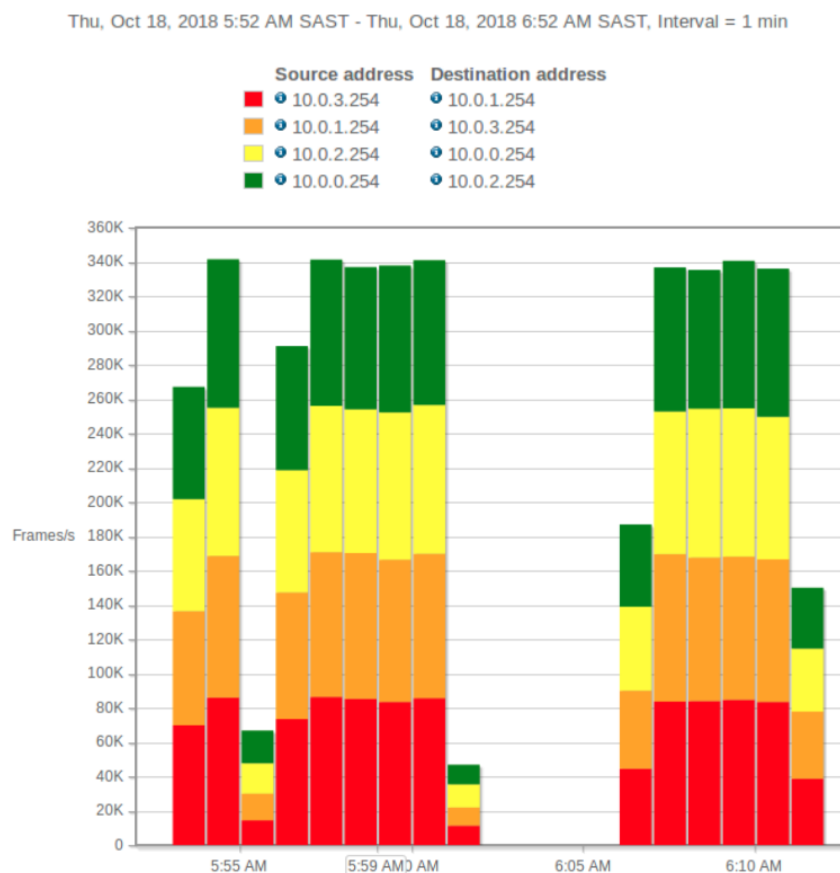


Figure 8.3: sFlow Monitoring Switch During iPerf Benchmarks for VLAN.

8.3.4 Routing Summary

The switch exhibited a performance of 931 Mbit/s on all four ports during bandwidth testing for L3 routing. In this network setup, network functions made use of L2, L3 and DHCP. The results shown included logs during the creation and configuration of the network. The results also show that the switch exhibited the expected behaviour of a

router which includes altering packet headers and decrementing the TTL as required. The throughput and bandwidth was 931 Mbits/s. This shows that the switch's total capaity to forward L3 traffic was 7448 Mbits/s.

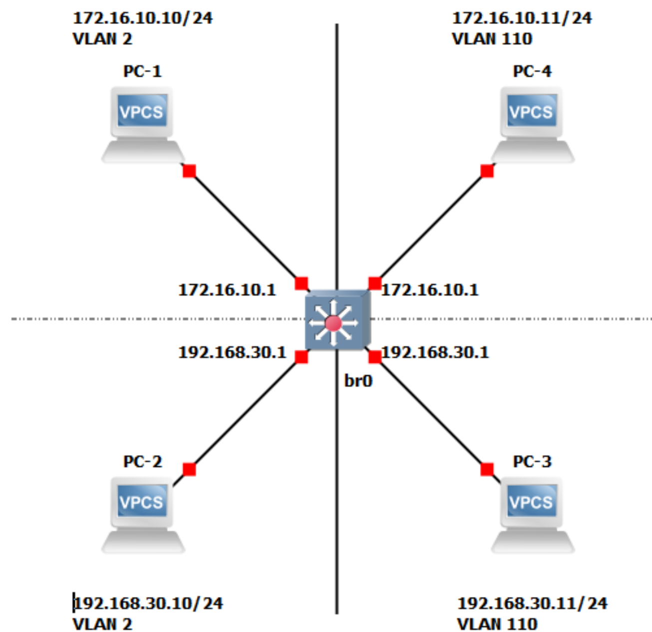


Figure 8.4: VLAN Use Case Topology.

8.4 VLAN Use Case

The 802.1Q protocol allows the partitioning of networks by using a single switch to host multiple networks, i.e. host different subnets. This allows the use of a single switch to act as multiple logically separated switches. As discussed before, an L2 switch is not able to route communication between subnets, hence routing between VLAN networks, requires a router with VLAN support. To demonstrate network partitioning, the topology illustrated in Figure 8.4 shows an example of partitioning the network using VLANs.

8.4.1 Configuration

Implemented for this use case, the switch was configured to have the following properties:

- IP addresses for the switch's interfaces are from two different subnets. The CIDR is set to 172.16.10.0/24 for port 1 and port 3, while the CIDR for ports 2 and 4 is defined as 192.168.30.0/24.
- The four interfaces of the multi-layer switch were assigned the following IP address: 172.16.10.1 for ports 1 and 3, and IP address 192.168.30.1 for ports 2 and 4. These were used for routing.

- Figure 8.4 shows that the network was partitioned into two VLANs with ID 2 and 110 respectively. Host 1 and Host 3 are in the VLAN 2, while Host 2 and Host 4 are in the VLAN 110. By partitioning the network (in absence of routing capability between the VLANs), hosts in VLAN 2 are unable to communicate to hosts in VLAN 110 and vice-versa.

- The interfaces on each host was configured manually for VLAN 2 and 110. This required initialising the ports to use VLAN tagging, hence DHCP was disabled.

Using the above policy and the configuration toolbar from he web UI, the following was set and applied to the multi-layer switch. This policy was created for the switch to have two VLAN networks. The controller used this policy to create and add OpenFlow flow entries onto the switch (Listing 8.19).

Listing 8.19: Input Configuration for VLAN Network.

```
1 {
2   "vlan_ip":{
3     "2":["172.16.10.1/24","192.168.30.1/24"],
4     "110":["172.16.10.1/24","192.168.30.1/24"]
5   },
6   "is_router":true
7 }
```

8.4.1.1 Verifying Controller Settings

Verifying the policy displayed the following message in the web UI shown in Listing 8.20.

Listing 8.20: Validation of VLAN Configuration.

```
1 {
2   "switch_id": "0000001b21a6da00",
3   "command_result": [
4     {
5       "result": "success",
6       "details": [
7         "Add VLAN addresses {'vlan_ip': {'vlan_id': '2', 'address': ['172.16.10.1/24', '192.16
            ↪ 8.30.1/24']}} [vlan_id=1]",
8         "Add VLAN addresses {'vlan_ip': {'vlan_id': '110', 'address': ['172.16.10.1/24', '192.
            ↪ 168.30.1/24']}} [vlan_id=2]",
9       ]
10    }
11  ]
12 }
```

This confirmed that the created policy was validated against port structure of the switch. The details of the message show actions that the controller takes when the policy is

applied. This also hints that the switch's OpenFlow table will be updated with flows with VLAN IP addresses of 172.16.10.1/24 and 192.168.30.1/24 for VLAN 2 and VLAN 110.

The result in Listing 8.21 shows the controller reply message after applying the policy to the switch (0000001b21a6da00).

Listing 8.21: Confirmation For the Configured Input.

```
1 [
2   "Success",
3   "Updating switch values",
4   "Configure as:",
5   "DHCP: False",
6   "DHCP addresses: ",
7   "Router: True",
8   "VLAN router addresses: {'2': ['172.16.10.1/24', '192.168.30.1/24'], '110': ['172.16.10.1/24
      ↪ ', '192.168.30.1/24']}"
9 ]
```

The controller was queried to display the switch's settings. This is displayed in Listing 8.22.

Listing 8.22: Requested Switch's Configuration Returned by Controller.

```
1  [
2    {
3      "vlan_id": 2,
4      "address": [
5        {
6          "address_id": 2,
7          "address": "192.168.30.1/24"
8        },
9        {
10         "address_id": 1,
11         "address": "172.16.10.1/24"
12       }
13     ]
14   },
15   {
16     "vlan_id": 110,
17     "address": [
18       {
19         "address_id": 2,
20         "address": "192.168.30.1/24"
21       },
22       {
23         "address_id": 1,
24         "address": "172.16.10.1/24"
25       }
26     ]
27   }
28 ]
```

The above message tallies with the entered policy confirming the correctness of the created policy. This completed the configuration of the VLAN setup as shown in Figure 8.4. This procedure shows that creating an SDN network with subnet partitioning can be achieved using the JSON object shown in Listing 8.19.

Configuring The Host Machines

The DHCP service was not configured to operate with VLANs. Hence, at each host, the VLAN interfaces were added manually with the respective VLAN ID followed by registering the default gateway on each port. Listing 8.23 shows the configuration of Host 1 (Host 2 – 4 are shown configuration in Appendix F.1).

Listing 8.23: Configuring Interface `enp0s31f6` for VLAN 2 in Host 1.

```
1 >>> ip link add link enp0s31f6 name enp0s31f6.2 type vlan id 2
2 >>> ip addr add 172.16.10.10/24 dev enp0s31f6.2
3 >>> ip link set dev enp0s31f6.2 up
4 >>> ip route add default via 172.16.10.1
```

8.4.1.2 Verifying Hosts Settings

By executing the Linux tools `ifconfig` and `ip`, network configuration could be displayed including information about the IP address, the interface and the routing for the interfaces. Listing 8.24 shows the netowrk configuration for Host 1 (configurations for Host 2 – 4 in Appendix F.2).

Listing 8.24: Network Configuration for Host 1.

```
host1@ubuntu:~$ ip route list
default via 172.16.10.1 dev enp0s31f6.2
172.16.10.0/24 dev enp0s31f6.2  proto kernel  scope link  src 172.16.10.10
host1@ubuntu:~$ ifconfig
enp0s31f6.2 Link encap:Ethernet  HWaddr 30:5a:3a:7a:0d:b9
          inet addr:172.16.10.10  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::325a:3aff:fe7a:db9/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:59 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:2790 (2.7 KB)
```

8.4.2 Results

Once the network was up and running, traffic was observed to be circulating between the switch and the four hosts. As a result, OpenFlow entries were added to the switch. As discussed in the previous chapter, the source application of the OpenFlow entries can be determined by the value of the cookie field of the flow. The cookie grouping value for flow entries added is given by $VLAN\ ID \times 2^{32}$. Table 8.2 shows the cookies for flows added for each VLAN ID.

Table 8.2: Cookie Grouping for VLAN 2 and 110

VLAN ID	Base Cookie	Value	Value (HEX)
2	2×2^{32}	8589934592	0x200000000
110	110×2^{32}	472446402560	0x6E00000000

The `ping` tool in Linux was used to verify connectivity between hosts. The configured switch was set up as a VLAN router that had partitioned the network into two. By partitioning the network, the expected behaviour was to limit the communication between the two VLAN networks (in the absence of a routing capabilities between the two VLANs). The results displayed in Listing 8.25 – Listing 8.27 show that hosts were only able to ping interfaces of hosts and router within the same VLAN ID. Listing 8.25 and Listing 8.26 shows successful ping messages between Host 4 and router interface. Listing 8.27 shows unsuccessful ping messages. This result is expected since the logically partitioned router was unable to forward these messages. Logs from the router shown in Listing 8.28 shows the switch is unable to reach the network. The pings from Host 4 (192.168.30.11) to Host 1 (172.16.10.10) in different VLANs were unsuccessful and the controller explicitly replied with ICMP messages as unreachable.

Listing 8.25: Host 4 Pinging Router Interface in Same Subnet.

```
host4@ubuntu:~$ ping -c10 172.16.10.1
PING 172.16.10.1 (172.16.10.1) 56(84) bytes of data.
64 bytes from 172.16.10.1: icmp_seq=1 ttl=64 time=7.22 ms
64 bytes from 172.16.10.1: icmp_seq=2 ttl=64 time=3.02 ms
64 bytes from 172.16.10.1: icmp_seq=3 ttl=64 time=6.54 ms
64 bytes from 172.16.10.1: icmp_seq=4 ttl=64 time=9.26 ms
64 bytes from 172.16.10.1: icmp_seq=5 ttl=64 time=7.29 ms
64 bytes from 172.16.10.1: icmp_seq=6 ttl=64 time=8.89 ms
64 bytes from 172.16.10.1: icmp_seq=7 ttl=64 time=7.82 ms
64 bytes from 172.16.10.1: icmp_seq=8 ttl=64 time=6.22 ms
64 bytes from 172.16.10.1: icmp_seq=9 ttl=64 time=8.48 ms
64 bytes from 172.16.10.1: icmp_seq=10 ttl=64 time=8.38 ms

--- 172.16.10.1 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9013ms
rtt min/avg/max/mdev = 3.029/7.317/9.260/1.711 ms
```

Appendix F.7 shows the settled OpenFlow table. The table had a total of 25 flow entries. Flows added by the router application had the cookie parameter defined as hex values of `0x200000001, 0x200000002, 0x6e00000001, 0x6e00000002`. This is observed in the logs shown in Appendix F.8. These values translate to cookie values of `8589934593, 8589934594` for VLAN2 and `472446402561, 472446402562` for VLAN 110 in the OpenFlow table. Other supporting logs show the router application adding flows during pinging (Appendix F.9). sFlow recorded the source-destination traffic in bits/s shown in Figure 8.5, and the corresponding source VLANs, shown in Figure 8.6.

Listing 8.26: Host 4 Pinging Router Interface in Same VLAN but Different Subnet.

```
host4@ubuntu:~$ ping -c10 192.168.30.1
PING 192.168.30.1 (192.168.30.1) 56(84) bytes of data.
64 bytes from 192.168.30.1: icmp_seq=1 ttl=64 time=6.23 ms
64 bytes from 192.168.30.1: icmp_seq=2 ttl=64 time=5.69 ms
64 bytes from 192.168.30.1: icmp_seq=3 ttl=64 time=7.77 ms
64 bytes from 192.168.30.1: icmp_seq=4 ttl=64 time=8.86 ms
64 bytes from 192.168.30.1: icmp_seq=5 ttl=64 time=5.89 ms
64 bytes from 192.168.30.1: icmp_seq=6 ttl=64 time=7.69 ms
64 bytes from 192.168.30.1: icmp_seq=7 ttl=64 time=8.21 ms
64 bytes from 192.168.30.1: icmp_seq=8 ttl=64 time=5.79 ms
64 bytes from 192.168.30.1: icmp_seq=9 ttl=64 time=5.17 ms
64 bytes from 192.168.30.1: icmp_seq=10 ttl=64 time=6.64 ms

--- 192.168.30.1 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9015ms
rtt min/avg/max/mdev = 5.174/6.797/8.860/1.185 ms
```

Listing 8.27: Host 4 Pinging Host 1 in Different VLANs.

```
host4@ubuntu:~$ ping -c10 172.16.10.10
PING 172.16.10.10 (172.16.10.10) 56(84) bytes of data.
From 192.168.30.1 icmp_seq=1 Destination Host Unreachable
From 192.168.30.1 icmp_seq=2 Destination Host Unreachable
From 192.168.30.1 icmp_seq=3 Destination Host Unreachable
From 192.168.30.1 icmp_seq=4 Destination Host Unreachable
From 192.168.30.1 icmp_seq=5 Destination Host Unreachable
From 192.168.30.1 icmp_seq=6 Destination Host Unreachable
From 192.168.30.1 icmp_seq=7 Destination Host Unreachable
From 192.168.30.1 icmp_seq=8 Destination Host Unreachable
From 192.168.30.1 icmp_seq=9 Destination Host Unreachable
From 192.168.30.1 icmp_seq=10 Destination Host Unreachable

--- 172.16.10.10 ping statistics ---
10 packets transmitted, 0 received, +10 errors, 100% packet loss, time 9008ms
```

8.4.3 Throughput iPerf

The throughput benchmark measurements were for host-to-host, that is from Host 1 to Host 2 and Host 3 to Host 4. This test aimed to see what the throughput was for TCP communication between each pair of server and client, bidirectional data transmission. These tests were run simultaneously on all hosts. Listing 8.29 and Listing 8.30 shows the results for the network benchmarks for VLAN routing between the host pairs 1 and 2, 3 and 4. The throughput for VLAN Routing was 929Mbit/s. By making use of the sFlow tool, the generated report shows the hosts and their associated TCP ports.

8.4.4 VLAN Summary

The process of partitioning the network only required setting up the VLANs by applying the configuration of Listing 8.19 in the web UI. This essentially configured the switch as a

Listing 8.28: Log Messages from the Router Application.

```
1 #...
2 [RT][INFO] switch_id=0000001b21a6da00: Receive IP packet from [192.168.30.11] to an internal
    ↪ host [172.16.10.10].
3 [RT][INFO] switch_id=0000001b21a6da00: Send ARP request (flood)
4 [RT][INFO] switch_id=0000001b21a6da00: ARP reply wait timer was timed out.
5 [RT][INFO] switch_id=0000001b21a6da00: Send ICMP destination unreachable to [172.16.10.10].
6 #...
```

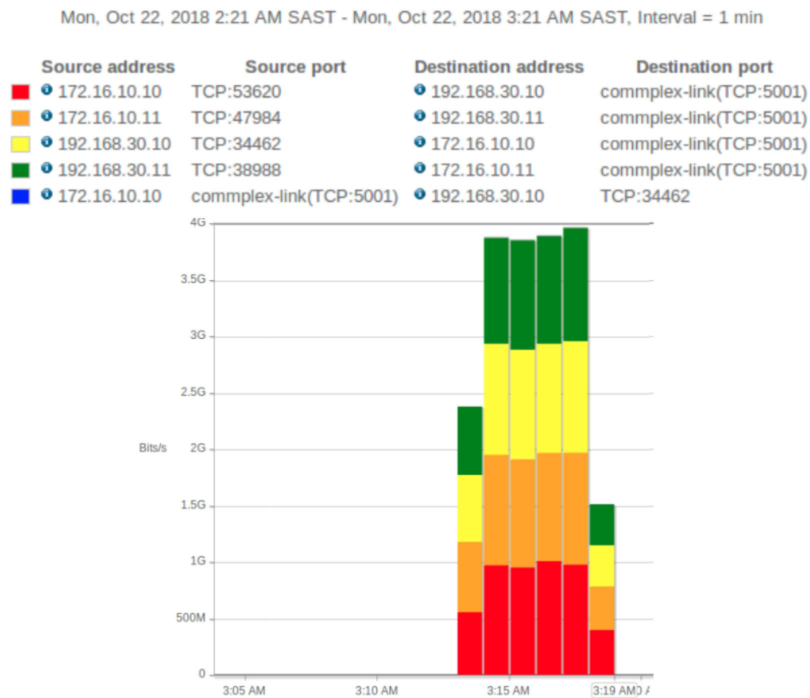


Figure 8.5: sFlow Monitoring Source-Destination traffic for VLAN.

Listing 8.29: iPerf benchmark between Host 1 and Host 2.

```
host1@ubuntu:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 172.16.10.10 port 5001 connected with 192.168.30.10 port 34462
------------------------------------------------------------
Client connecting to 192.168.30.10, TCP port 5001
TCP window size:  706 KByte (default)
------------------------------------------------------------
[  6] local 172.16.10.10 port 53620 connected with 192.168.30.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-300.0 sec  32.4 GBytes   929 Mbits/sec
[  4]  0.0-300.0 sec  32.4 GBytes   929 Mbits/sec
```

router with two VLANs, VLAN 2 and VLAN 110. It was seen that successful partitioning between the VLANs was enforced in the controller and traffic between VLANs was not routed. The benchmarks of the VLAN network peaked at 929 Mbit/s. The switch is able

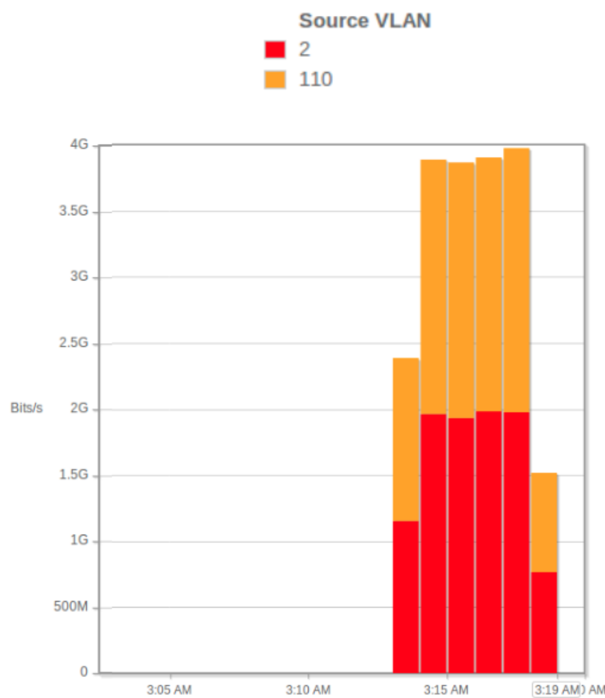


Figure 8.6: sFlow Monitoring Source VLANs.

Listing 8.30: iPerf benchmark between Host 3 and Host 4.

```
host4@ubuntu:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 172.16.10.11 port 5001 connected with 192.168.30.11 port 38988
------------------------------------------------------------
Client connecting to 192.168.30.11, TCP port 5001
TCP window size:  264 KByte (default)
------------------------------------------------------------
[  6] local 172.16.10.11 port 47984 connected with 192.168.30.11 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-300.0 sec  32.4 GBytes   929 Mbits/sec
[  4]  0.0-300.0 sec  32.4 GBytes   929 Mbits/sec
```

to handle a total VLAN routing capacity of 7432 Mbit/s on all of its ports. These results were also monitored externally using sFlow.

8.5 Summary

This chapter demonstrated that the multi-layer switch is able to serve as an L2 switch, router or a VLAN router. The use cases covered also demonstrated the application of the

developed SDN applications. The experiments have shown also the easy extensibility of the system.

The processes illustrated in this chapter for network monitoring and management of the underlying network demonstrated a simple model of network management through centralised decisions and policy making. The policies defined for each use case resulted in automated flow management, topology update and traffic monitoring.

Chapter 9

Conclusion

The main goal of this thesis was to develop an inexpensive, multi-layer SDN switch that could be used in conjunction with a small-medium scale network. This work should provide useful information for network administrators undertaking the creation of SDN networks faced with limited resources. It also provides various tools to manage, benchmark, and evaluate network hardware and network bandwidth. The software tools selected were open source, i.e. freely available. This chapter concludes the thesis by providing an analysis of the work done, summarising contributions made and presenting suggestions for future work.

9.1 Achieved Objectives

This section revisits the objectives outlined in Section 1.2, to evaluate the degree to which these objectives were met.

The following are the objectives set in Section 1.2:

1. Investigate open source tools and commodity hardware to implement an inexpensive multi-layer SDN switch.
2. Evaluate the performance of the developed switch.
3. Provide a simplified structure for network applications development and integrate common functions present in certain commercial implementations.
4. Demonstrate a simplified process of network management.

9.1.1 Tools to Use for Developing an SDN Switch

The investigation of commodity hardware revealed three major forms of networking hardware: general-purpose CPU, FPGAs and ASICs. By considering the requirement of an inexpensive, general-purpose CPU offers the cheapest solution but possibly with limited performance. On the other hand, they offer the greatest flexibility (This is because CPUs implement programs that run as software in main memory). Hence changes in functionality is quick and seamless. So, the switch was realised as a software switch that runs on commodity hardware. The software tools implemented were free and open. This helped achieve the goal of implementing an inexpensive hardware switch.

The OVS and DPDK framework enabled high speed packet processing in software for physical interfaces. The scale at which the switch operates was deemed be feasible for small-medium networks with port speeds of 1 GBit/s, suitable for research, office, campus networks or even private cloud environments.

An important question on the suitability of commodity hardware was to research further about system architecture. This would ensure that there was optimal configuration for the system. This investigated the interconnection between the NIC (Intel® 82580), CPU and memory. The Intel® 82580 network card interfaced directly with the CPU over the recommended 4 lanes. This minimised the distance between NIC and main memory. And so, did not limit the rate of data transfer between a port on the NIC and the user program `ovs-vswitchd`.

9.1.2 Performance of the Switch

The second objective was to evaluate the performance of the multi-layer SDN switch. This was achieved by performing sets of tests on the switch. These included IO, L2, L3 performance and bandwidth as well as VLAN tagging networks. The results gathered were throughput and latency measurements for IO, L2 and L3 forwarding. Using the TRex traffic generator, the switch maintained line speed, where the average forwarding time was less than 100 μs. Using iPerf, the results of L2 and L3 achieved speeds of 931 MBits/s; the results of VLAN achieved speeds of 929 MBits/s. The performance of the switch is therefore suitable for small-medium scale applications.

9.1.3 Simplified Structure for Network Applications

The third objective was to provide simplified method of implementing SDN networking including network applications that perform functions found in commercial implementations. This was demonstrated Through Ryu SDN controller which hosted network applications. The platform for programming included API calls to the Ryu controller simplifying the development of applications. The Ryu framework also comes with example implementations reducing the time to develop applications.

The functions tested were the functioning of an L2 switch, a router, DHCP service and VLAN.

9.1.4 Simplified the Process of Network Management

The fourth objective was to express a simplified way for network management. This was demonstrated by the development of a web interface to configure the system. This reduces the time to configure the network, improving the efficiency of the system.

The web interface also displays the network topology for visualising the structure of the network. The user can take advantage of the interface by creating JSON objects to set policies, while the controller and SDN applications handle low-level commands to the underlying network devices.

9.2 Limitations

Naturally, there are some limitations to the OpenFlow multi-layer switch implementation. The measurements done during the evaluation of the switch cover the core functionality and does not completely cover the behaviour for all the features of the switch nor cover the entire functions of the controller. eg, the switch's capability such as MAC and VLAN learning for ingress port, VLAN trunking and access ports, Generic Routing Encapsulation (GRE), Virtual Extensible LAN (VXLAN) , Stateless Transport Tunneling (STT), and Locator/Identifier Separation Protocol (LISP) tunnelling, e.t.c, while the controller is able to extend on these capabilities. Another limitation was the measurements of the latency, which was based on the CPU, software timers, rather than using hardware timers [113].

9.3 Future work

Below are the considerations for future development, beginning with the data plane switch.

- Extending the functions within the web UI by introducing additional applications such as load balancers and policy engines.
- To further reduce costs, ARM processors (which are found in mobile devices) may substitute the Intel processor, offering multi-threaded architecture which the OVS-DPDK can take advantage of.
- Use the switch to provide a way to test the migration from traditional to SDN networks, as done by Google in [114] or proposed migration structure seen in [54].
- Deploy the switch in a production environment to judge its robustness.

9.4 Summary

The work done in this thesis detailed the development of an inexpensive but flexible SDN switch using open source software and commodity hardware. The testing of the switch also provided a guide for building a small-scale SDN network.

Bibliography

[1] S. Subramanian and S. Voruganti, *Software-Defined Networking (SDN) with OpenStack*. Packt Publishing, 2016.

[2] M. Campista, M. Rubinstein, I. Moraes, L. Costa, and O. C. M. B. Duarte, "Challenges and research directions for the future internetworking," *Communications Surveys & Tutorials, IEEE*, vol. 16, pp. 1050–1079, 01 2014.

[3] D. Sharma, "Training report on software defined networking taken at florida international university, florida usa," Sep 2015. Report 12E1DAECM4XP008, [Online]. Available: https://www.researchgate.net/publication/281979574. [Accessed: Sep. 22, 2018].

[4] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, and R. Gopal, "RFC5810 - Forwarding and Control Element Separation (ForCES) Protocol Specification." The Forwarding and Control Element Separation (ForCES) Protocol Specification webpage on IETF Tools, Mar 2010. [Online]. Available: https://tools.ietf.org/html/rfc5810. [Accessed: Sep. 12, 2018].

[5] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "RFC6241 - Network Configuration Protocol (NETCONF)." The Network Configuration Protocol (NETCONF) webpage on IETF Tools, Jun 2011. [Online]. Available: https://tools.ietf.org/html/rfc6241. [Accessed: Sep. 12, 2018].

[6] Open Networking Foundation, *OpenFlow Switch Specification*. The Linux Foundation, 03 2015. Version 1.3.5.

[7] V. Kumar and S. Priya, "OpenFlow Version RoadMap," May 2016. [Online]. Available: http://kspviswa.github.io/OpenFlow_Version_Roadmap.html. [Accessed: May. 14, 2018].

[8] K. Benzekki, A. El Fergougui, and A. El Belrhiti El Alaoui, "Software-defined networking (sdn): A survey," *Security and Communication Networks*, 02 2017.

[9] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch." PowerPoint presentation at the meeting of USENIX Association, Oakland, CA, 2015.

[10] R. H. Nir Yechiel, Principal Product Manager. (2016) Boosting the NFV datapath with RHEL OpenStack Platform webpage on Red Hat Stack, Jun 2018. [Online]. Available: `https://redhatstackblog.redhat.com/2016/02/10/boosting-the-nfv-datapath-with-rhel-openstack-platform/`. [Accessed: Jun. 02, 2018].

[11] "Home – DPDK," Apr 2018. [Online]. Available: `http://dpdk.org`. [Accessed: Sep. 28, 2018].

[12] F. Antonio and B. B. Prakash. OVS-DPDK Datapath Classifier | Intel®Software, Oct 2016. [Online]. Available: `https://software.intel.com/en-us/articles/ovs-dpdk-datapath-classifier`. [Accessed: Jun. 28, 2018].

[13] RYU Project Team, *RYU SDN Framework*. Dec 2016. [Online]. Available: `https://osrg.github.io/ryu-book/en/Ryubook.pdf`. [Accessed: Sep. 13, 2018].

[14] "Architecture — Python documentation," Jan 2019. [Online]. Available: `https://faucet.readthedocs.io/en/latest/architecture.html`. [Accessed: Jan. 17, 2019].

[15] R. Droms, "RFC2131 - Network Configuration Protocol (NETCONF)." The Dynamic Host Configuration Protocol (DHCP) webpage on IETF Tools, Mar 2011. [Online]. Available: `https://tools.ietf.org/html/rfc2131`. [Accessed: Nov. 30, 2018].

[16] T. Graf, "Ovs stateful services." PowerPoint presentation at the meeting of FOSDEM Brussles, 2015.

[17] "Segment pada Transmission Control Protocol (TCP)," Apr 2016. [Online]. Available: `https://nhirtthirsty.wordpress.com/2016/04/20/segment-pada-transmission-control-protocol-tcp`. [Accessed: Jan. 29, 2019].

[18] T. Magwenzi, A. Terzoli, and M. Tsietsi, "Towards a low-cost software-defined ethernet switch using open-source components," in *Southern Africa Telecommunication Networks and Applications Conference (SATNAC) 2018*, pp. 168–173, Sep 2018.

[19] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network and openflow: From concept to implementation," *IEEE Communications Surveys Tutorials*, vol. 16, pp. 2181–2206, Fourthquarter 2014.

[20] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, pp. 114–119, February 2013.

[21] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for SDN? Implementation challenges for software-defined networks," in *IEEE Communications Magazine*, vol. 51, pp. 36–43, Jul 2013.

[22] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *SIGCOMM Comput. Commun. Rev.*, vol. 43, pp. 99–110, ACM, Aug 2013.

[23] H. Kim, J. Kim, and Y.-B. Ko, "Developing a cost-effective openflow testbed for small-scale software defined networking," in *16th International Conference on Advanced Communication Technology*, pp. 758–761, Feb 2014.

[24] J. Kempf, S. Whyte, J. Ellithorpe, P. Kazemian, M. Haitjema, N. Beheshti, S. Stuart, and H. Green, "Openflow mpls and the open source label switched router," in *Proceedings of the 23rd International Teletraffic Congress*, pp. 8–14, Sep 2011.

[25] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang, "Serverswitch: A programmable and high performance platform for data center networks," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pp. 15–28, 2011.

[26] N. Networks, "Northbound Networks," Jan 2019. [Online; accessed 29. Jan. 2019].

[27] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A roadmap for traffic engineering in sdn-openflow networks," *Comput. Netw.*, vol. 71, pp. 1–30, Oct. 2014.

[28] G. Cuba, J. M. Becerra, G. Bartra, and C. Santivanez, "Pucplight: A sdn/openflow controller for an academic campus network," pp. 1–4, 10 2016.

[29] T. Benson, A. Akella, and D. Maltz, "Unraveling the complexity of network management," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, (Berkeley, CA, USA), pp. 335–348, 2009.

[30] A. Agarwal, "Inter-Datacenter WAN with centralized TE using SDN and OpenFlow," Feb 2012. [Online]. Available: `https://www.opennetworking.org/wp-content/uploads/2013/02/cs-googlesdn.pdf`. [Accessed: Sep. 13, 2018].

[31] Cisco, "Software-Defined Networking: Why We Like It and How We Are Building On It." Whitepaper, 2013.

[32] Open Networking Foundation, "Sdn architecture issue 1.1," Tech. Rep. 1.1, Open Networking Foundation, 2275 E. Bayshore Road, Suite 103, Palo Alto, CA 94303, 2016.

[33] ONF Solution Brief, "OpenFlow-enabled SDN and network functions virtualization," 2014. [Online]. Available: `https://www.opennetworking.org/wp-content/uploads/2013/05/sb-sdn-nvf-solution.pdf`. [Accessed: Sep. 13, 2018].

[34] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," in *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar 2008.

[35] C. M. Kozierok, *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*. William Pollock, 2005.

[36] T. D. Nadeau and K. Gray, *SDN: Software Defined Networks*. O'Reilly Media, Inc., 2013.

[37] M. Hadley, D. Nicol, and R. Smith, "Software-defined networking redefines performance for ethernet control systems," in *Power and Energy Automation Conference*, 2017.

[38] Cisco, "Enterprise campus 3.0 architecture: Overview and framework," Apr 2008. [Online]. Available: `https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Campus/campover.html`. [Accessed: Jun. 30, 2018].

[39] Open Networking Foundation, "SDN in the Campus Environment," Tech. Rep. 1.1, Open Networking Foundation, 2275 E. Bayshore Road, Suite 103, Palo Alto, CA 94303, 2013.

[40] B. A. A. Nunes, M. Mendonca, X. N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys Tutorials*, vol. 16, pp. 1617–1634, Third 2014.

[41] Linux Foundation, "Open Networkin Foundation," Aug 2018. [Online]. Available: `https://www.opennetworking.org`. [Accessed: Aug. 17, 2018].

[42] M. Karakus and A. Durresi, "Quality of service (qos) in software defined networking (sdn): A survey," *Journal of Network and Computer Applications*, vol. 80, pp. 200 – 218, 2017.

[43] J. C. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Mudigonda, P. Sharma, and Y. Turner, "Corybantic: towards the modular composition of sdn control programs," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, p. 1, ACM, 2013.

[44] P. Tijare and D. Vasudevan, "The Northbound APIs of Software Defined Networks," *INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY*, vol. 5, 2016.

[45] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, 2000. University of California, Irvine.

[46] R. Battle and E. Benson, "Bridging the semantic web and web 2.0 with representational state transfer (rest)," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 6, no. 1, pp. 61–69, 2008.

[47] OSGil™ Alliance, "Architecture – OSGi™ Alliance," Sep 2018. [Online]. Available: `https://www.osgi.org/developer/architecture`. [Accessed: Sep. 13, 2018].

[48] L. Yang, R. Dantu, T. Anderson, and R. Gopal, "RFC3746 - The Forwarding and Control Element Separation (ForCES) Framework webpage on IETF Tools." The Forwarding and Control Element Separation (ForCES) Framework webpage on IETF Tools, Apr 2004. [Online]. Available: `https://tools.ietf.org/html/rfc3746`. [Accessed: Sep. 12, 2018].

[49] J. Appel, C. Dogan, A. Fuetsch, S. Guanglu, R. Howald, D. Kashiwa, P. Lopez, N. McKeown, G. Parulkar, and A. Vahdat, "ONF Strategic Plan," Feb 2018. [Online]. Available: `https://www.opennetworking.org/wp-content/uploads/2018/03/ONF-Strategic-Plan.pdf`. [Accessed: Sep. 13, 2018].

[50] D. Pitt, "Key Benefits of OpenFlow-Based SDN - Open Networking Foundation," Jul 2012. [Online]. Available: `https://www.opennetworking.org/news-and-events/blog/key-benefits-of-openflow-based-sdn/`. [Accessed: Jan. 13, 2018].

[51] Open Networking Fundation, "Software-Defined Networking: The New Norm for Networks," *ONF White Paper*, vol. 2, pp. 2–6, 2012.

[52] T. Hollingsworth, "OpenFlow 1.3 Support: Why It Matters," Jul 2014. [Online]. Available: `https://www.networkcomputing.com/networking/openflow-13-support-why-it-matters/602695705`. [Accessed: Sep. 14, 2018].

[53] SDxCentral. (2016) The Future of Network Virtualization and SDN Controllers Report on SDxCentral.

[54] J. Bailey and S. Stuart, "Faucet: Deploying sdn in the enterprise," *Commun. ACM*, vol. 60, pp. 45–49, Dec. 2016.

[55] Open Networking Foundation, *OpenFlow Switch Specification.* The Linux Foundation, 12 2009. Version 1.0.

[56] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "Past: Scalable ethernet for data centers," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, (New York, NY, USA), pp. 49–60, ACM, 2012.

[57] Open Networking Foundation, "ONF SDN Evolution," Tech. Rep. 1.0, Open Networking Foundation, 2275 E. Bayshore Road, Suite 103, Palo Alto, CA 94303, S 2016.

[58] I. Englander, *The Architecture of Computer Hardware, System Software, and Networking.* John Wiley & Sons, Inc., 2009.

[59] Z. Ullah, M. K. Jaiswal, and R. C. C. Cheung, "Z-tcam: An sram-based architecture for tcam," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, pp. 402–406, Feb 2015.

[60] H. Wong, V. Betz, and J. Rose, "Comparing fpga vs. custom cmos and the impact on processor microarchitecture," in *Proceedings of the 23rd International Teletraffic Congress ??*, pp. 8–14, Mar 2011.

[61] Intel, "Open vSwitch Enables SDN and NFV Transformation." Whitepaper, 2015.

[62] Intel, "DPDK Boosts Packet Processing, Performance, and Throughput," Aug 2018. [Online]. Available: https://www.intel.co.za/content/www/za/en/communications/data-plane-development-kit.html. [Accessed: Sep. 13, 2018].

[63] Y. Qi, B. Xu, F. He, B. Yang, J. Yu, and J. Li, "Towards high-performance flow-level packet processing on multi-core network processors," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '07, (New York, NY, USA), pp. 17–26, ACM, 2007.

[64] A. Cooke, "FPGAs vs. CPU: What's best for data acquisition?," Dec 2016. [Online]. Available: https://www.curtisswrightds.com/news/blog/fpgas-vs-cpu-whats-best-for-data-acquisition.html. [Accessed: Jan. 17, 2019].

[65] T. Santhanam, "CAM (Content Addressable Memory) VS TCAM (Ternary Content Addressable Memory)," Mar 2011. [Online]. Available: https://community.cisco.com/t5/networking-documents/cam-content-addressable-memory-vs-tcam-ternary-content/ta-p/3107938. [Accessed: Jan. 29, 2019].

[66] N. Jain and M. K. Jain, "Current status of network processors," *International Journal of Computer Applications*, vol. 98, no. 12, 2014.

[67] Cisco, "Cisco Nexus 9300-EX Platform Switches Architecture." Whitepaper, 2017.

[68] P. Rygielski, M. Seliuchenko, S. Kounev, and M. Klymash, "Performance analysis of sdn switches with hardware and software flow tables," in *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools on 10th EAI International Conference on Performance Evaluation Methodologies and Tools*, (ICST, Brussels, Belgium, Belgium), pp. 80–87, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2017.

[69] W. Miao, G. Min, Y. Wu, H. Wang, and J. Hu, "Performance modelling and analysis of software-defined networking under bursty multimedia traffic," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 12, pp. 77:1–77:19, Sept. 2016.

[70] Big Switch Networks, "Indigo - Open Source OpenFlow Switches - Indigo - Project Floodlight - OpenFlow news and projects," Sep 2018. [Online]. Available: http://www.projectfloodlight.org/indigo. [Accessed: Sep. 09, 2018].

[71] Linux Foundation, "Open vSwitch," Aug 2018. [Online]. Available: http://www.openvswitch.org. [Accessed: Sep. 13, 2018].

[72] NTT Network Innovation Laboratories, "Lagopus vswitch," Oct 2018. [Online]. Available: https://lagopus.github.io. [Accessed: Oct. 21, 2018].

[73] FlowForwarding, "Flowforwarding/linc-switch: Openflow software switch written in erlang," Sep 2018. [Online]. Available: https://github.com/FlowForwarding/LINC-Switch. [Accessed: Sep. 28, 2018].

[74] "Trafficlab/of11softswitch: Openflow 1.1 softswitch," Sep 2018. [Online]. Accessed 28. Sep. 2018. https://github.com/TrafficLab/of11softswitch. [Accessed: Sep. 28, 2018].

[75] E. L. Fernandes, "Cpqd/ofsoftswitch13: Openflow 1.3 switch," Sep 2018. [Online]. Accessed 28. Sep. 2018. https://github.com/CPqD/ofsoftswitch13.

[76] Pica8, "Resources - Pica8," Sep 2018. [Online]. Available: https://www.pica8.com/resources/?resourcelib_category=datasheets. [Accessed: Sep. 14, 2018].

[77] Netronome, "Agilio CX SmartNICs - Netronome," Sep 2018. [Online]. Available: https://www.netronome.com/products/agilio-cx. [Accessed: Sep. 13, 2018].

[78] The Open vSwitch website. [Online]. Available: https://www.openvswitch.org. [Accessed: Jun. 01, 2018].

[79] The Linux Foundation, *Open vSwitch, Release 2.9.90*, 2018. [Online]. Available: https://media.readthedocs.org/pdf/openvswitch/latest/openvswitch.pdf. [Accessed: Sep. 13, 2018].

[80] Mellanox, "Mellanox Technologies: ASAP2 - Accelerated Switch and Packet Processing," Sep 2018. [Online]. Available: http://www.mellanox.com/page/asap2?mtag=asap2. [Accessed: Sep. 13, 2018].

[81] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), pp. 117–130, USENIX Association, 2015.

[82] Linux Foundation, *Data Plane Development Kit (DPDK) 17.11.3.* The Linux Foundation, Apr 2018. [Online]. Available: https://readthedocs.org/projects/dpdk/downloads/epub/v17.11/. [Accessed: Apr. 17, 2018].

[83] F. Antonio and B. B. Prakash, "OVS-DPDK Datapath Classifier | Intel® Software," Sep 2018. Available: https://software.intel.com/en-us/articles/ovs-dpdk-datapath-classifier. [Accessed: Sep. 13, 2018].

[84] OpenDaylight, "OpenDaylight," Sep 2018. [Online]. Available: https://www.opendaylight.org. [Accessed: Sep. 13, 2018].

[85] Open Network Foundation, "Open Network Operating System," Sep 2018. [Online]. Available: https://onosproject.org. [Accessed: Sep. 13, 2018].

[86] The Open Networking Lab, "Introducing ONOS - a SDN network operating system for Service Providers," tech. rep., ON.Lab., 2014.

[87] Ryu SDN Framework Community, "Ryu SDN Framework," May 2018. [Online]. Available: https://osrg.github.io/ryu. [Accessed: Sep. 13, 2018].

[88] Nicira, Inc, "Nicira vendor extentions [source code]," Dec 2012. [Online]. Available: https://github.com/osrg/openvswitch/blob/master/include/openflow/nicira-ext.h. [Accessed: Jan. 31, 2019].

[89] Ryu Development Team, *Ryu Documentation, Release 4.24*, 2018. [Online]. Available: https://media.readthedocs.org/pdf/ryu/latest/ryu.pdf. [Accessed: Apr. 17, 2018].

[90] Faucet SDN Controller, "Faucet SDN Controller," Jul 2018. [Online]. Available: https://faucet.nz. [Accessed: Sep. 13, 2018].

[91] F. Bannour, S. Souihi, and A. Mellouk, "Distributed sdn control: Survey, taxonomy and challenges," *IEEE Communications Surveys & Tutorials*, vol. PP, pp. 1–1, 12 2017.

[92] S. Bradner and J. McQuaid, "RFC2544 - Benchmarking Methodology for Network Interconnect Devices webpage on IETF Tools," Mar 1999. [Online]. Available: https://tools.ietf.org/html/rfc2544. [Accessed: Jun. 02, 2018].

[93] V. Gueant, "iPerf - The TCP, UDP and SCTP network bandwidth measurement tool," Nov 2018. [Online] Available: https://iperf.fr. [Accessed: Nov. 30, 2018].

[94] IEEE, "IEEE Standard for Ethernet," *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)*, pp. 1–4017, March 2016.

[95] J. Abley, "RFC7042 - IANA Considerations and IETF Protocol and Documentation Usage for IEEE 802 Parameterss webpage on IETF Tools," Oct 2013. [Online]. Available: https://tools.ietf.org/html/rfc7042. [Accessed: Jun. 02, 2018].

[96] J. Kurose and K. Ross, *Computer Nnetworking A Top-Down Approach*. Addison-Wesley, 6th ed., 2013.

[97] M. Rowe, "What does GT/s mean, anyway?," Mar 2007. [Online]. Available: https://www.edn.com/electronics-news/4380071/What-does-GT-s-mean-anyway-. [Accessed: Mar. 17, 2018].

[98] L. Mike Micheletti, "Eecatalog - the engineers guide to the electronics industry. a refresher on 8b/10b encoding | pci express," Jun 2012. [Online]. Available: http://eecatalog.com/pcie/2012/06/13/a-refresher-on-8b10b-encoding/. [Accessed: Mar. 17, 2018].

[99] J. Gasparakis and P. P. Waskiewicz Jr., "Open vswitch manual. design considerations for efficient network applications with intel® multi-core processor-based systems on linux.," tech. rep., Intel, Jul 2010. Available [Online] https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/multi-core-processor-based-linux-paper.pdf. [Accessed: Mar. 17, 2018].

[100] Linux Foundation, "The DPDK Supported Hardware webpage on DPDK website," Jan 2019. [Online]. Available: https://core.dpdk.org/supported/. [Accessed: Jan. 29, 2019].

[101] Intel Corporation, *Intel®82580EB/82580DB Gigabit Ethernet Controller*, Sep 2010. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82580-eb-db-gbe-controller-datasheet.pdf. [Accessed: Apr. 17, 2018].

[102] Debian, "Hugepages - Debian Wiki," Jun 2017. [Online]. Available: https://wiki.debian.org/Hugepages. [Accessed: Nov. 30, 2018].

[103] I. Corporation, "Single- and Multichannel Memory Modes," Nov 2017. [Online]. Available: https://www.intel.co.za/content/www/za/en/support/articles/000005657/boards-and-kits.html. [Accessed: Jan. 29, 2018].

[104] S. Burke, "RAM Performance Benchmark: Single-Channel vs. Dual-Channel - Does It Matter?," Mar 2014. Accessed on: 29. Jan.

2019. [Online]. Available: https://www.gamersnexus.net/guides/1349-ram-how-dual-channel-works-vs-single-channel?showall=1.

[105] R. Coelho, "Does dual-channel memory make difference in gaming performance? - Hardware Secrets," Nov 2015. [Online]. Available: https://www.hardwaresecrets.com/does-dual-channel-memory-make-difference-in-gaming-performance. [Accessed: Jan. 29, 2018].

[106] "ovs/NEWS at branch-2.9 Âů openvswitch/ovs [Source code]," Oct 2018. [Online]. Available: https://github.com/openvswitch/ovs/blob/branch-2.9/NEWS. [Accessed: Dec. 21, 2018].

[107] Intel, "Intel® Core™ i5-3570 Processor (6M Cache, up to 3.80 GHz) Product Specifications," Dec 2018. [Online]. Available: https://ark.intel.com/products/65702/Intel-Core-i5-3570-Processor-6M-Cache-up-to-3-80-GHz- [Accessed: Dec. 11, 2018].

[108] "sFlow.org - Making the Network Visible," Jan 2019. Accessed 17 January 2019, [Online]. Available: https://sflow.org.

[109] The TRex website. Accessed 17 May 2018, [Online]. Available: https://trex-tgn.cisco.com.

[110] "Source NAT with DHCP [source code]," June 2016. [Online]. Available: https://github.com/John-Lin/nat. [Accessed Sep. 13, 2018].

[111] M. Allman, V. Paxson, and E. Blanton, "RFC5681 - TCP Congestion Control webpage on IETF Tools." The TCP Congestion Control webpage on IETF Tools, 2009. Accessed 31 January 2018, [Online]. https://tools.ietf.org/html/rfc5681.

[112] W.-c. Feng, D. D. Kandlur, D. Saha, and K. G. Shin, "Techniques for eliminating packet loss in congested tcp/ip networks," *Ann Arbor*, vol. 1001, p. 63130, 1997.

[113] Cisco, *TRex Stateless support*. Cisco, 2019. Accessed February 2019. [Online]. Available: https://trex-tgn.cisco.com/trex/doc/trex_stateless.pdf.

[114] Google Inc., "Inter-Datacenter, WAN with centralized TE using SDN and OpenFlow," 2012.

Appendices

Appendix A

TestPMD Info

Listing A.1: `TestPMD` info for port 0 of Intel® 82580 NIC.

```
testpmd> show port info 0
********************* Infos for port 0  *********************
MAC address: 00:1B:21:A6:D3:BC
Driver name: net_e1000_igb
Connect to socket: 0
memory allocation on the socket: 0
Link status: down
Link speed: 0 Mbps
Link duplex: half-duplex
MTU: 1500
Promiscuous mode: enabled
Allmulticast mode: disabled
Maximum number of MAC addresses: 24
Maximum number of MAC addresses of hash filtering: 0
VLAN offload:
  strip on
  filter on
  qinq(extend) off
Hash key size in bytes: 40
Redirection table size: 128
Supported flow types:
  ipv4
  ipv4-tcp
  ipv4-udp
  ipv6
  ipv6-tcp
  ipv6-udp
  user defined 15
  user defined 16
  user defined 17
Max possible RX queues: 8
Max possible number of RXDs per queue: 4096
Min possible number of RXDs per queue: 32
RXDs number alignment: 8
Max possible TX queues: 8
Max possible number of TXDs per queue: 4096
Min possible number of TXDs per queue: 32
TXDs number alignment: 8
```

A.1 OpenFlow Table Description

Listing A.2: Table Description for Table 0.

```
openvswitch@openvswitch:~$ sudo ovs-ofctl dump-table-features br0 -O OpenFlow13
  table 0:
    metadata: match=0xffffffffffffffff write=0xffffffffffffffff
    max_entries=1000000
    instructions (table miss and others):
      next tables: 1-253
      instructions: meter,apply_actions,clear_actions,write_actions,write_metadata,goto_table
      Write-Actions and Apply-Actions features:
        actions: output group set_field strip_vlan push_vlan mod_nw_ttl dec_ttl set_mpls_ttl
            ↪ dec_mpls_ttl push_mpls pop_mpls set_queue
        supported on Set-Field: tun_id tun_src tun_dst tun_ipv6_src tun_ipv6_dst tun_flags
            ↪ tun_gbp_id tun_gbp_flags tun_erspan_idx tun_erspan_ver tun_erspan_dir
            ↪ tun_erspan_hwid tun_metadata0 tun_metadata1 tun_metadata2 tun_metadata3
            ↪ tun_metadata4 tun_metadata5 tun_metadata6 tun_metadata7 tun_metadata8
            ↪ tun_metadata9 tun_metadata10 tun_metadata11 tun_metadata12 tun_metadata13
            ↪ tun_metadata14 tun_metadata15 tun_metadata16 tun_metadata17 tun_metadata18
            ↪ tun_metadata19 tun_metadata20 tun_metadata21 tun_metadata22 tun_metadata23
            ↪ tun_metadata24 tun_metadata25 tun_metadata26 tun_metadata27 tun_metadata28
            ↪ tun_metadata29 tun_metadata30 tun_metadata31 tun_metadata32 tun_metadata33
            ↪ tun_metadata34 tun_metadata35 tun_metadata36 tun_metadata37 tun_metadata38
            ↪ tun_metadata39 tun_metadata40 tun_metadata41 tun_metadata42 tun_metadata43
            ↪ tun_metadata44 tun_metadata45 tun_metadata46 tun_metadata47 tun_metadata48
            ↪ tun_metadata49 tun_metadata50 tun_metadata51 tun_metadata52 tun_metadata53
            ↪ tun_metadata54 tun_metadata55 tun_metadata56 tun_metadata57 tun_metadata58
            ↪ tun_metadata59 tun_metadata60 tun_metadata61 tun_metadata62 tun_metadata63
            ↪ metadata in_port in_port_oxm pkt_mark ct_mark ct_label reg0 reg1 reg2 reg3 reg4
            ↪  reg5 reg6 reg7 reg8 reg9 reg10 reg11 reg12 reg13 reg14 reg15 xreg0 xreg1 xreg2
            ↪  xreg3 xreg4 xreg5 xreg6 xreg7 xxreg0 xxreg1 xxreg2 xxreg3 eth_src eth_dst
            ↪ vlan_tci vlan_vid vlan_pcp mpls_label mpls_tc mpls_ttl ip_src ip_dst ipv6_src
            ↪ ipv6_dst ipv6_label nw_tos ip_dscp nw_ecn nw_ttl arp_op arp_spa arp_tpa arp_sha
            ↪  arp_tha tcp_src tcp_dst udp_src udp_dst sctp_src sctp_dst icmp_type icmp_code
            ↪ icmpv6_type icmpv6_code nd_target nd_sll nd_tll nsh_flags nsh_spi nsh_si nsh_c1
            ↪  nsh_c2 nsh_c3 nsh_c4 nsh_ttl
....
```

Appendix B

Faucet Compliance Tests

B.1 Test Results

Listing B.1: Faucet full test results.

```
1  ----------------------------------------------------------------------
2  Ran 141 tests in 9234.158s
3
4  test results
5  ============
6
7
8  faucet_mininet_test_unit.FaucetSanityTest.test_listening        OK
9  faucet_mininet_test_unit.FaucetSanityTest.test_portmap  OK
10 faucet_mininet_test_unit.FaucetSanityTest.test_untagged OK
11 faucet_mininet_test_unit.FaucetConfigReloadAclTest.test_port_acls       OK
12 faucet_mininet_test_unit.FaucetConfigReloadTest.test_add_unknown_dp     OK
13 faucet_mininet_test_unit.FaucetConfigReloadTest.test_port_change_acl    OK
14 faucet_mininet_test_unit.FaucetConfigReloadTest.test_port_change_perm_learn     OK
15 faucet_mininet_test_unit.FaucetConfigReloadTest.test_port_change_vlan   OK
16 faucet_mininet_test_unit.FaucetConfigReloadTest.test_tabs_are_bad       OK
17 faucet_mininet_test_unit.FaucetConfigStatReloadAclTest.test_port_acls   OK
18 faucet_mininet_test_unit.FaucetDeleteConfigReloadTest.test_delete_interface     OK
19 faucet_mininet_test_unit.FaucetDestRewriteTest.test_switching   OK
20 faucet_mininet_test_unit.FaucetDestRewriteTest.test_untagged    OK
21 faucet_mininet_test_unit.FaucetEthSrcMaskTest.test_untagged     OK
22 faucet_mininet_test_unit.FaucetExperimentalAPITest.test_untagged        OK
23 faucet_mininet_test_unit.FaucetGroupTableTest.test_group_exist  OK
24 faucet_mininet_test_unit.FaucetGroupTableTest.test_untagged     OK
25 faucet_mininet_test_unit.FaucetGroupTableUntaggedIPv4RouteTest.test_untagged    OK
26 faucet_mininet_test_unit.FaucetGroupTableUntaggedIPv6RouteTest.test_untagged    OK
27 faucet_mininet_test_unit.FaucetIPv4TupleTest.test_tuples        OK
28 faucet_mininet_test_unit.FaucetIPv6TupleTest.test_tuples        OK
29 faucet_mininet_test_unit.FaucetMaxHostsPortTest.test_untagged   OK
30 faucet_mininet_test_unit.FaucetMultiOutputTest.test_untagged    OK
```

```
31 faucet_mininet_test_unit.FaucetNailedFailoverForwardingTest.test_untagged       OK
32 faucet_mininet_test_unit.FaucetNailedForwardingTest.test_untagged       OK
33 faucet_mininet_test_unit.FaucetRouterConfigReloadTest.test_router_config_reload OK
34 faucet_mininet_test_unit.FaucetSingleHostsTimeoutPrometheusTest.test_untagged   OK
35 faucet_mininet_test_unit.FaucetSingleL2LearnMACsOnPortTest.test_untagged        OK
36 faucet_mininet_test_unit.FaucetSingleL3LearnMACsOnPortTest.test_untagged        OK
37 faucet_mininet_test_unit.FaucetSingleUntaggedIPv4ControlPlaneTest.test_fping_controller OK
38 faucet_mininet_test_unit.FaucetSingleUntaggedIPv4ControlPlaneTest.test_untagged OK
39 faucet_mininet_test_unit.FaucetSingleUntaggedIPv6ControlPlaneTest.test_fping_controller OK
40 faucet_mininet_test_unit.FaucetSingleUntaggedIPv6ControlPlaneTest.test_untagged OK
41 faucet_mininet_test_unit.FaucetSingleUntaggedInfluxTooSlowTest.test_untagged    OK
42 faucet_mininet_test_unit.FaucetStackStringOfDPUntaggedTest.test_untagged        OK
43 faucet_mininet_test_unit.FaucetTaggedAndUntaggedTest.test_separate_untagged_tagged       OK
44 faucet_mininet_test_unit.FaucetTaggedAndUntaggedVlanGroupTest.test_untagged     OK
45 faucet_mininet_test_unit.FaucetTaggedAndUntaggedVlanTest.test_untagged  OK
46 faucet_mininet_test_unit.FaucetTaggedBroadcastTest.test_tagged  OK
47 faucet_mininet_test_unit.FaucetTaggedGroupTableTest.test_group_exist    OK
48 faucet_mininet_test_unit.FaucetTaggedGroupTableTest.test_tagged OK
49 faucet_mininet_test_unit.FaucetTaggedICMPv6ACLTest.test_icmpv6_acl_match        OK
50 faucet_mininet_test_unit.FaucetTaggedICMPv6ACLTest.test_tagged  OK
51 faucet_mininet_test_unit.FaucetTaggedIPv4ControlPlaneTest.test_ping_controller  OK
52 faucet_mininet_test_unit.FaucetTaggedIPv4ControlPlaneTest.test_tagged   OK
53 faucet_mininet_test_unit.FaucetTaggedIPv4RouteTest.test_tagged  OK
54 faucet_mininet_test_unit.FaucetTaggedIPv6ControlPlaneTest.test_ping_controller  OK
55 faucet_mininet_test_unit.FaucetTaggedIPv6ControlPlaneTest.test_tagged   OK
56 faucet_mininet_test_unit.FaucetTaggedIPv6RouteTest.test_tagged  OK
57 faucet_mininet_test_unit.FaucetTaggedPopVlansOutputTest.test_tagged     OK
58 faucet_mininet_test_unit.FaucetTaggedProactiveNeighborIPv4RouteTest.test_tagged
       ↪ OKfaucet_mininet_test_unit.
59 faucet_mininet_test_unit.FaucetTaggedProactiveNeighborIPv6RouteTest.test_tagged OK
60 faucet_mininet_test_unit.FaucetTaggedScaleTest.test_tagged      OK
61 faucet_mininet_test_unit.FaucetTaggedSwapVidMirrorTest.test_tagged      OK
62 faucet_mininet_test_unit.FaucetTaggedSwapVidOutputTest.test_tagged      OK
63 faucet_mininet_test_unit.FaucetTaggedTargetedResolutionIPv4RouteTest.test_tagged        OK
64 faucet_mininet_test_unit.FaucetTaggedTest.test_tagged   OK
65 faucet_mininet_test_unit.FaucetTaggedWithUntaggedTest.test_tagged       OK
66 faucet_mininet_test_unit.FaucetUntagged8021XTest.test_untagged  OK
67 faucet_mininet_test_unit.FaucetUntaggedACLMirrorDefaultAllowTest.test_eapol_mirrored    OK
68 faucet_mininet_test_unit.FaucetUntaggedACLMirrorDefaultAllowTest.test_untagged  OK
69 faucet_mininet_test_unit.FaucetUntaggedACLMirrorTest.test_eapol_mirrored        OK
70 faucet_mininet_test_unit.FaucetUntaggedACLMirrorTest.test_untagged      OK
71 faucet_mininet_test_unit.FaucetUntaggedACLTcpMaskTest.test_port5001_blocked     OK
72 faucet_mininet_test_unit.FaucetUntaggedACLTcpMaskTest.test_port5002_notblocked  OK
73 faucet_mininet_test_unit.FaucetUntaggedACLTcpMaskTest.test_port_gt1023_blocked  OK
74 faucet_mininet_test_unit.FaucetUntaggedACLTcpMaskTest.test_untagged     OK
75 faucet_mininet_test_unit.FaucetUntaggedACLTest.test_port5001_blocked    OK
76 faucet_mininet_test_unit.FaucetUntaggedACLTest.test_port5002_notblocked OK
77 faucet_mininet_test_unit.FaucetUntaggedACLTest.test_untagged    OK
78 faucet_mininet_test_unit.FaucetUntaggedApplyMeterTest.test_untagged     OK
79 faucet_mininet_test_unit.FaucetUntaggedBGPDualstackDefaultRouteTest.test_untagged       OK
80 faucet_mininet_test_unit.FaucetUntaggedBGPIPv4DefaultRouteTest.test_untagged    OK
81 faucet_mininet_test_unit.FaucetUntaggedBGPIPv4RouteTest.test_untagged   OK
82 faucet_mininet_test_unit.FaucetUntaggedBGPIPv6DefaultRouteTest.test_untagged    OK
83 faucet_mininet_test_unit.FaucetUntaggedBGPIPv6RouteTest.test_untagged   OK
84 faucet_mininet_test_unit.FaucetUntaggedBroadcastTest.test_untagged      OK
85 faucet_mininet_test_unit.FaucetUntaggedCDPTest.test_untagged    OK
```

```
86  faucet_mininet_test_unit.FaucetUntaggedControllerNfvTest.test_untagged  OK
87  faucet_mininet_test_unit.FaucetUntaggedDPACLTest.test_port5001_blocked  OK
88  faucet_mininet_test_unit.FaucetUntaggedDPACLTest.test_port5002_notblocked       OK
89  faucet_mininet_test_unit.FaucetUntaggedDPACLTest.test_untagged  OK
90  faucet_mininet_test_unit.FaucetUntaggedExpireIPv4InterVLANRouteTest.test_untagged       OK
91  faucet_mininet_test_unit.FaucetUntaggedGroupHairpinTest.test_untagged   OK
92  faucet_mininet_test_unit.FaucetUntaggedHUPTest.test_untagged    OK
93  faucet_mininet_test_unit.FaucetUntaggedHairpinTest.test_untagged        OK
94  faucet_mininet_test_unit.FaucetUntaggedHostMoveTest.test_untagged       OK
95  faucet_mininet_test_unit.FaucetUntaggedHostPermanentLearnTest.test_untagged     OK
96  faucet_mininet_test_unit.FaucetUntaggedIPv4ControlPlaneFuzzTest.test_flap_ping_controller
    ↪       OK
97  faucet_mininet_test_unit.FaucetUntaggedIPv4ControlPlaneFuzzTest.test_fuzz_controller    OK
98  faucet_mininet_test_unit.FaucetUntaggedIPv4ControlPlaneFuzzTest.test_ping_fragment_controller
    ↪   OK
99  faucet_mininet_test_unit.FaucetUntaggedIPv4ControlPlaneFuzzTest.test_untagged   OK
100 faucet_mininet_test_unit.FaucetUntaggedIPv4InterVLANRouteTest.test_untagged     OK
101 faucet_mininet_test_unit.FaucetUntaggedIPv4LACPTest.test_untagged       OK
102 faucet_mininet_test_unit.FaucetUntaggedIPv4PolicyRouteTest.test_untagged        OK
103 faucet_mininet_test_unit.FaucetUntaggedIPv4RouteTest.test_untagged      OK
104 faucet_mininet_test_unit.FaucetUntaggedIPv6ControlPlaneFuzzTest.test_flap_ping_controller
    ↪       OK
105 faucet_mininet_test_unit.FaucetUntaggedIPv6ControlPlaneFuzzTest.test_fuzz_controller    OK
106 faucet_mininet_test_unit.FaucetUntaggedIPv6ControlPlaneFuzzTest.test_untagged   OK
107 faucet_mininet_test_unit.FaucetUntaggedIPv6InterVLANRouteTest.test_untagged     OK
108 faucet_mininet_test_unit.FaucetUntaggedIPv6RATest.test_ndisc6   OK
109 faucet_mininet_test_unit.FaucetUntaggedIPv6RATest.test_ra_advertise     OK
110 faucet_mininet_test_unit.FaucetUntaggedIPv6RATest.test_rdisc6   OK
111 faucet_mininet_test_unit.FaucetUntaggedIPv6RATest.test_rs_reply OK
112 faucet_mininet_test_unit.FaucetUntaggedIPv6RATest.test_untagged OK
113 faucet_mininet_test_unit.FaucetUntaggedIPv6RouteTest.test_untagged      OK
114 faucet_mininet_test_unit.FaucetUntaggedInfluxDownTest.test_untagged     OK
115 faucet_mininet_test_unit.FaucetUntaggedInfluxTest.test_untagged OK
116 faucet_mininet_test_unit.FaucetUntaggedInfluxUnreachableTest.test_untagged      OK
117 faucet_mininet_test_unit.FaucetUntaggedLLDPBlockedTest.test_untagged    OK
118 faucet_mininet_test_unit.FaucetUntaggedLLDPDefaultFallbackTest.test_untagged    OK
119 faucet_mininet_test_unit.FaucetUntaggedLLDPTest.test_untagged   OK
120 faucet_mininet_test_unit.FaucetUntaggedLogRotateTest.test_untagged      OK
121 faucet_mininet_test_unit.FaucetUntaggedLoopTest.test_untagged   OK
122 faucet_mininet_test_unit.FaucetUntaggedMaxHostsTest.test_untagged       OK
123 faucet_mininet_test_unit.FaucetUntaggedMeterParseTest.test_untagged     OK
124 faucet_mininet_test_unit.FaucetUntaggedMirrorTest.test_untagged OK
125 faucet_mininet_test_unit.FaucetUntaggedMixedIPv4RouteTest.test_untagged OK
126 faucet_mininet_test_unit.FaucetUntaggedMixedIPv6RouteTest.test_untagged OK
127 faucet_mininet_test_unit.FaucetUntaggedMultiDBWatcherTest.test_untagged OK
128 faucet_mininet_test_unit.FaucetUntaggedMultiMirrorSepTest.test_untagged OK
129 faucet_mininet_test_unit.FaucetUntaggedMultiMirrorTest.test_untagged    OK
130 faucet_mininet_test_unit.FaucetUntaggedMultiVlansOutputTest.test_untagged       OK
131 faucet_mininet_test_unit.FaucetUntaggedNoCombinatorialBroadcastTest.test_untagged       OK
132 faucet_mininet_test_unit.FaucetUntaggedNoPortUnicastFloodTest.test_untagged     OK
133 faucet_mininet_test_unit.FaucetUntaggedNoReconfACLTest.test_untagged    OK
134 faucet_mininet_test_unit.FaucetUntaggedNoVLanUnicastFloodTest.test_untagged     OK
135 faucet_mininet_test_unit.FaucetUntaggedOutputOnlyTest.test_untagged     OK
136 faucet_mininet_test_unit.FaucetUntaggedOutputOverrideTest.test_untagged OK
137 faucet_mininet_test_unit.FaucetUntaggedOutputTest.test_untagged OK
138 faucet_mininet_test_unit.FaucetUntaggedPortUnicastFloodTest.test_untagged       OK
```

```
139 faucet_mininet_test_unit.FaucetUntaggedPrometheusGaugeTest.test_untagged        OK
140 faucet_mininet_test_unit.FaucetUntaggedRandomVidTest.test_untagged      OK
141 faucet_mininet_test_unit.FaucetUntaggedSameVlanIPv6RouteTest.test_untagged      OK
142 faucet_mininet_test_unit.FaucetUntaggedTcpIPv4IperfTest.test_untagged   OK
143 faucet_mininet_test_unit.FaucetUntaggedTcpIPv6IperfTest.test_untagged   OK
144 faucet_mininet_test_unit.FaucetUntaggedTest.test_untagged       OK
145 faucet_mininet_test_unit.FaucetUntaggedVLANACLTest.test_port5001_blocked        OK
146 faucet_mininet_test_unit.FaucetUntaggedVLANACLTest.test_port5002_notblocked     OK
147 faucet_mininet_test_unit.FaucetUntaggedVLANACLTest.test_untagged        OK
148 faucet_mininet_test_unit.FaucetUntaggedVLanUnicastFloodTest.test_untagged       OK
149 faucet_mininet_test_unit.FaucetSanityTest.test_listening        OK
150 faucet_mininet_test_unit.FaucetSanityTest.test_portmap  OK
151 faucet_mininet_test_unit.FaucetSanityTest.test_untagged OK
152 faucet_mininet_test_unit.FaucetConfigReloadAclTest.test_port_acls       OK
153 faucet_mininet_test_unit.FaucetConfigReloadTest.test_add_unknown_dp     OK
154 faucet_mininet_test_unit.FaucetConfigReloadTest.test_port_change_acl    OK
155 faucet_mininet_test_unit.FaucetConfigReloadTest.test_port_change_perm_learn     OK
156 faucet_mininet_test_unit.FaucetConfigReloadTest.test_port_change_vlan   OK
157 faucet_mininet_test_unit.FaucetConfigReloadTest.test_tabs_are_bad       OK
158 faucet_mininet_test_unit.FaucetConfigStatReloadAclTest.test_port_acls   OK
159 faucet_mininet_test_unit.FaucetDeleteConfigReloadTest.test_delete_interface     OK
160 faucet_mininet_test_unit.FaucetDestRewriteTest.test_switching   OK
161 faucet_mininet_test_unit.FaucetDestRewriteTest.test_untagged    OK
162 faucet_mininet_test_unit.FaucetEthSrcMaskTest.test_untagged     OK
163 faucet_mininet_test_unit.FaucetExperimentalAPITest.test_untagged        OK
164 faucet_mininet_test_unit.FaucetGroupTableTest.test_group_exist  OK
165 faucet_mininet_test_unit.FaucetGroupTableTest.test_untagged     OK
166 faucet_mininet_test_unit.FaucetGroupTableUntaggedIPv4RouteTest.test_untagged    OK
167 faucet_mininet_test_unit.FaucetGroupTableUntaggedIPv6RouteTest.test_untagged    OK
168 faucet_mininet_test_unit.FaucetIPv4TupleTest.test_tuples        OK
169 faucet_mininet_test_unit.FaucetIPv6TupleTest.test_tuples        OK
170 faucet_mininet_test_unit.FaucetMaxHostsPortTest.test_untagged   OK
171 faucet_mininet_test_unit.FaucetMultiOutputTest.test_untagged    OK
172 faucet_mininet_test_unit.FaucetNailedFailoverForwardingTest.test_untagged       OK
173 faucet_mininet_test_unit.FaucetNailedForwardingTest.test_untagged       OK
174 faucet_mininet_test_unit.FaucetRouterConfigReloadTest.test_router_config_reload OK
175 faucet_mininet_test_unit.FaucetSingleHostsTimeoutPrometheusTest.test_untagged   OK
176 faucet_mininet_test_unit.FaucetSingleL2LearnMACsOnPortTest.test_untagged        OK
177 faucet_mininet_test_unit.FaucetSingleL3LearnMACsOnPortTest.test_untagged        OK
178 faucet_mininet_test_unit.FaucetSingleUntaggedIPv4ControlPlaneTest.test_fping_controller OK
179 faucet_mininet_test_unit.FaucetSingleUntaggedIPv4ControlPlaneTest.test_untagged OK
180 faucet_mininet_test_unit.FaucetSingleUntaggedIPv6ControlPlaneTest.test_fping_controller OK
181 faucet_mininet_test_unit.FaucetSingleUntaggedIPv6ControlPlaneTest.test_untagged OK
182 faucet_mininet_test_unit.FaucetSingleUntaggedInfluxTooSlowTest.test_untagged    OK
183 faucet_mininet_test_unit.FaucetStackStringOfDPUntaggedTest.test_untagged        OK
184 faucet_mininet_test_unit.FaucetTaggedAndUntaggedTest.test_separate_untagged_tagged      OK
185 faucet_mininet_test_unit.FaucetTaggedAndUntaggedVlanGroupTest.test_untagged     OK
186 faucet_mininet_test_unit.FaucetTaggedAndUntaggedVlanTest.test_untagged  OK
187 faucet_mininet_test_unit.FaucetTaggedBroadcastTest.test_tagged  OK
188 faucet_mininet_test_unit.FaucetTaggedGroupTableTest.test_group_exist    OK
189 faucet_mininet_test_unit.FaucetTaggedGroupTableTest.test_tagged OK
190 faucet_mininet_test_unit.FaucetTaggedICMPv6ACLTest.test_icmpv6_acl_match        OK
191 faucet_mininet_test_unit.FaucetTaggedICMPv6ACLTest.test_tagged  OK
192 faucet_mininet_test_unit.FaucetTaggedIPv4ControlPlaneTest.test_ping_controller  OK
193 faucet_mininet_test_unit.FaucetTaggedIPv4ControlPlaneTest.test_tagged   OK
194 faucet_mininet_test_unit.FaucetTaggedIPv4RouteTest.test_tagged  OK
```

```
195 faucet_mininet_test_unit.FaucetTaggedIPv6ControlPlaneTest.test_ping_controller  OK
196 faucet_mininet_test_unit.FaucetTaggedIPv6ControlPlaneTest.test_tagged  OK
197 faucet_mininet_test_unit.FaucetTaggedIPv6RouteTest.test_tagged  OK
198 faucet_mininet_test_unit.FaucetTaggedPopVlansOutputTest.test_tagged  OK
199 faucet_mininet_test_unit.FaucetTaggedProactiveNeighborIPv4RouteTest.test_tagged OK
200 faucet_mininet_test_unit.FaucetTaggedProactiveNeighborIPv6RouteTest.test_tagged OK
201 faucet_mininet_test_unit.FaucetTaggedScaleTest.test_tagged  OK
202 faucet_mininet_test_unit.FaucetTaggedSwapVidMirrorTest.test_tagged  OK
203 faucet_mininet_test_unit.FaucetTaggedSwapVidOutputTest.test_tagged  OK
204 faucet_mininet_test_unit.FaucetTaggedTargetedResolutionIPv4RouteTest.test_tagged  OK
205 faucet_mininet_test_unit.FaucetTaggedTest.test_tagged  OK
206 faucet_mininet_test_unit.FaucetTaggedWithUntaggedTest.test_tagged  OK
207 faucet_mininet_test_unit.FaucetUntagged8021XTest.test_untagged  OK
208 faucet_mininet_test_unit.FaucetUntaggedACLMirrorDefaultAllowTest.test_eapol_mirrored  OK
209 faucet_mininet_test_unit.FaucetUntaggedACLMirrorDefaultAllowTest.test_untagged  OK
210 faucet_mininet_test_unit.FaucetUntaggedACLMirrorTest.test_eapol_mirrored  OK
211 faucet_mininet_test_unit.FaucetUntaggedACLMirrorTest.test_untagged  OK
212 faucet_mininet_test_unit.FaucetUntaggedACLTcpMaskTest.test_port5001_blocked  OK
213 faucet_mininet_test_unit.FaucetUntaggedACLTcpMaskTest.test_port5002_notblocked  OK
214 faucet_mininet_test_unit.FaucetUntaggedACLTcpMaskTest.test_port_gt1023_blocked  OK
215 faucet_mininet_test_unit.FaucetUntaggedACLTcpMaskTest.test_untagged  OK
216 faucet_mininet_test_unit.FaucetUntaggedACLTest.test_port5001_blocked  OK
217 faucet_mininet_test_unit.FaucetUntaggedACLTest.test_port5002_notblocked OK
218 faucet_mininet_test_unit.FaucetUntaggedACLTest.test_untagged  OK
219 faucet_mininet_test_unit.FaucetUntaggedApplyMeterTest.test_untagged  OK
220 faucet_mininet_test_unit.FaucetUntaggedBGPDualstackDefaultRouteTest.test_untagged  OK
221 faucet_mininet_test_unit.FaucetUntaggedBGPIPv4DefaultRouteTest.test_untagged  OK
222 faucet_mininet_test_unit.FaucetUntaggedBGPIPv4RouteTest.test_untagged  OK
223 faucet_mininet_test_unit.FaucetUntaggedBGPIPv6DefaultRouteTest.test_untagged  OK
224 faucet_mininet_test_unit.FaucetUntaggedBGPIPv6RouteTest.test_untagged  OK
225 faucet_mininet_test_unit.FaucetUntaggedBroadcastTest.test_untagged  OK
226 faucet_mininet_test_unit.FaucetUntaggedCDPTest.test_untagged  OK
227 faucet_mininet_test_unit.FaucetUntaggedControllerNfvTest.test_untagged  OK
228 faucet_mininet_test_unit.FaucetUntaggedDPACLTest.test_port5001_blocked  OK
229 faucet_mininet_test_unit.FaucetUntaggedDPACLTest.test_port5002_notblocked  OK
230 faucet_mininet_test_unit.FaucetUntaggedDPACLTest.test_untagged  OK
231 faucet_mininet_test_unit.FaucetUntaggedExpireIPv4InterVLANRouteTest.test_untagged  OK
232 faucet_mininet_test_unit.FaucetUntaggedGroupHairpinTest.test_untagged  OK
233 faucet_mininet_test_unit.FaucetUntaggedHUPTest.test_untagged  OK
234 faucet_mininet_test_unit.FaucetUntaggedHairpinTest.test_untagged  OK
235 faucet_mininet_test_unit.FaucetUntaggedHostMoveTest.test_untagged  OK
236 faucet_mininet_test_unit.FaucetUntaggedHostPermanentLearnTest.test_untagged  OK
237 faucet_mininet_test_unit.FaucetUntaggedIPv4ControlPlaneFuzzTest.test_flap_ping_controller
      ↪  OK
238 faucet_mininet_test_unit.FaucetUntaggedIPv4ControlPlaneFuzzTest.test_fuzz_controller  OK
239 faucet_mininet_test_unit.FaucetUntaggedIPv4ControlPlaneFuzzTest.test_ping_fragment_controller
      ↪  OK
240 faucet_mininet_test_unit.FaucetUntaggedIPv4ControlPlaneFuzzTest.test_untagged  OK
241 faucet_mininet_test_unit.FaucetUntaggedIPv4InterVLANRouteTest.test_untagged  OK
242 faucet_mininet_test_unit.FaucetUntaggedIPv4LACPTest.test_untagged  OK
243 faucet_mininet_test_unit.FaucetUntaggedIPv4PolicyRouteTest.test_untagged  OK
244 faucet_mininet_test_unit.FaucetUntaggedIPv4RouteTest.test_untagged  OK
245 faucet_mininet_test_unit.FaucetUntaggedIPv6ControlPlaneFuzzTest.test_flap_ping_controller
      ↪  OK
246 faucet_mininet_test_unit.FaucetUntaggedIPv6ControlPlaneFuzzTest.test_fuzz_controller  OK
247 faucet_mininet_test_unit.FaucetUntaggedIPv6ControlPlaneFuzzTest.test_untagged  OK
```

```
248 faucet_mininet_test_unit.FaucetUntaggedIPv6InterVLANRouteTest.test_untagged     OK
249 faucet_mininet_test_unit.FaucetUntaggedIPv6RATest.test_ndisc6   OK
250 faucet_mininet_test_unit.FaucetUntaggedIPv6RATest.test_ra_advertise     OK
251 faucet_mininet_test_unit.FaucetUntaggedIPv6RATest.test_rdisc6   OK
252 faucet_mininet_test_unit.FaucetUntaggedIPv6RATest.test_rs_reply OK
253 faucet_mininet_test_unit.FaucetUntaggedIPv6RATest.test_untagged OK
254 faucet_mininet_test_unit.FaucetUntaggedIPv6RouteTest.test_untagged      OK
255 faucet_mininet_test_unit.FaucetUntaggedInfluxDownTest.test_untagged     OK
256 faucet_mininet_test_unit.FaucetUntaggedInfluxTest.test_untagged OK
257 faucet_mininet_test_unit.FaucetUntaggedInfluxUnreachableTest.test_untagged      OK
258 faucet_mininet_test_unit.FaucetUntaggedLLDPBlockedTest.test_untagged    OK
259 faucet_mininet_test_unit.FaucetUntaggedLLDPDefaultFallbackTest.test_untagged    OK
260 faucet_mininet_test_unit.FaucetUntaggedLLDPTest.test_untagged   OK
261 faucet_mininet_test_unit.FaucetUntaggedLogRotateTest.test_untagged      OK
262 faucet_mininet_test_unit.FaucetUntaggedLoopTest.test_untagged   OK
263 faucet_mininet_test_unit.FaucetUntaggedMaxHostsTest.test_untagged       OK
264 faucet_mininet_test_unit.FaucetUntaggedMeterParseTest.test_untagged     OK
265 faucet_mininet_test_unit.FaucetUntaggedMirrorTest.test_untagged OK
266 faucet_mininet_test_unit.FaucetUntaggedMixedIPv4RouteTest.test_untagged OK
267 faucet_mininet_test_unit.FaucetUntaggedMixedIPv6RouteTest.test_untagged OK
268 faucet_mininet_test_unit.FaucetUntaggedMultiDBWatcherTest.test_untagged OK
269 faucet_mininet_test_unit.FaucetUntaggedMultiMirrorSepTest.test_untagged OK
270 faucet_mininet_test_unit.FaucetUntaggedMultiMirrorTest.test_untagged    OK
271 faucet_mininet_test_unit.FaucetUntaggedMultiVlansOutputTest.test_untagged       OK
272 faucet_mininet_test_unit.FaucetUntaggedNoCombinatorialBroadcastTest.test_untagged       OK
273 faucet_mininet_test_unit.FaucetUntaggedNoPortUnicastFloodTest.test_untagged     OK
274 faucet_mininet_test_unit.FaucetUntaggedNoReconfACLTest.test_untagged    OK
275 faucet_mininet_test_unit.FaucetUntaggedNoVLanUnicastFloodTest.test_untagged     OK
276 faucet_mininet_test_unit.FaucetUntaggedOutputOnlyTest.test_untagged     OK
277 faucet_mininet_test_unit.FaucetUntaggedOutputOverrideTest.test_untagged OK
278 faucet_mininet_test_unit.FaucetUntaggedOutputTest.test_untagged OK
279 faucet_mininet_test_unit.FaucetUntaggedPortUnicastFloodTest.test_untagged       OK
280 faucet_mininet_test_unit.FaucetUntaggedPrometheusGaugeTest.test_untagged        OK
281 faucet_mininet_test_unit.FaucetUntaggedRandomVidTest.test_untagged      OK
282 faucet_mininet_test_unit.FaucetUntaggedSameVlanIPv6RouteTest.test_untagged      OK
283 faucet_mininet_test_unit.FaucetUntaggedTcpIPv4IperfTest.test_untagged   OK
284 faucet_mininet_test_unit.FaucetUntaggedTcpIPv6IperfTest.test_untagged   OK
285 faucet_mininet_test_unit.FaucetUntaggedTest.test_untagged       OK
286 faucet_mininet_test_unit.FaucetUntaggedVLANACLTest.test_port5001_blocked        OK
287 faucet_mininet_test_unit.FaucetUntaggedVLANACLTest.test_port5002_notblocked     OK
288 faucet_mininet_test_unit.FaucetUntaggedVLANACLTest.test_untagged        OK
289 faucet_mininet_test_unit.FaucetUntaggedVLanUnicastFloodTest.test_untagged       OK
```

Appendix C

Web UI

C.1 Full View of the web UI

{NEXT PAGE}

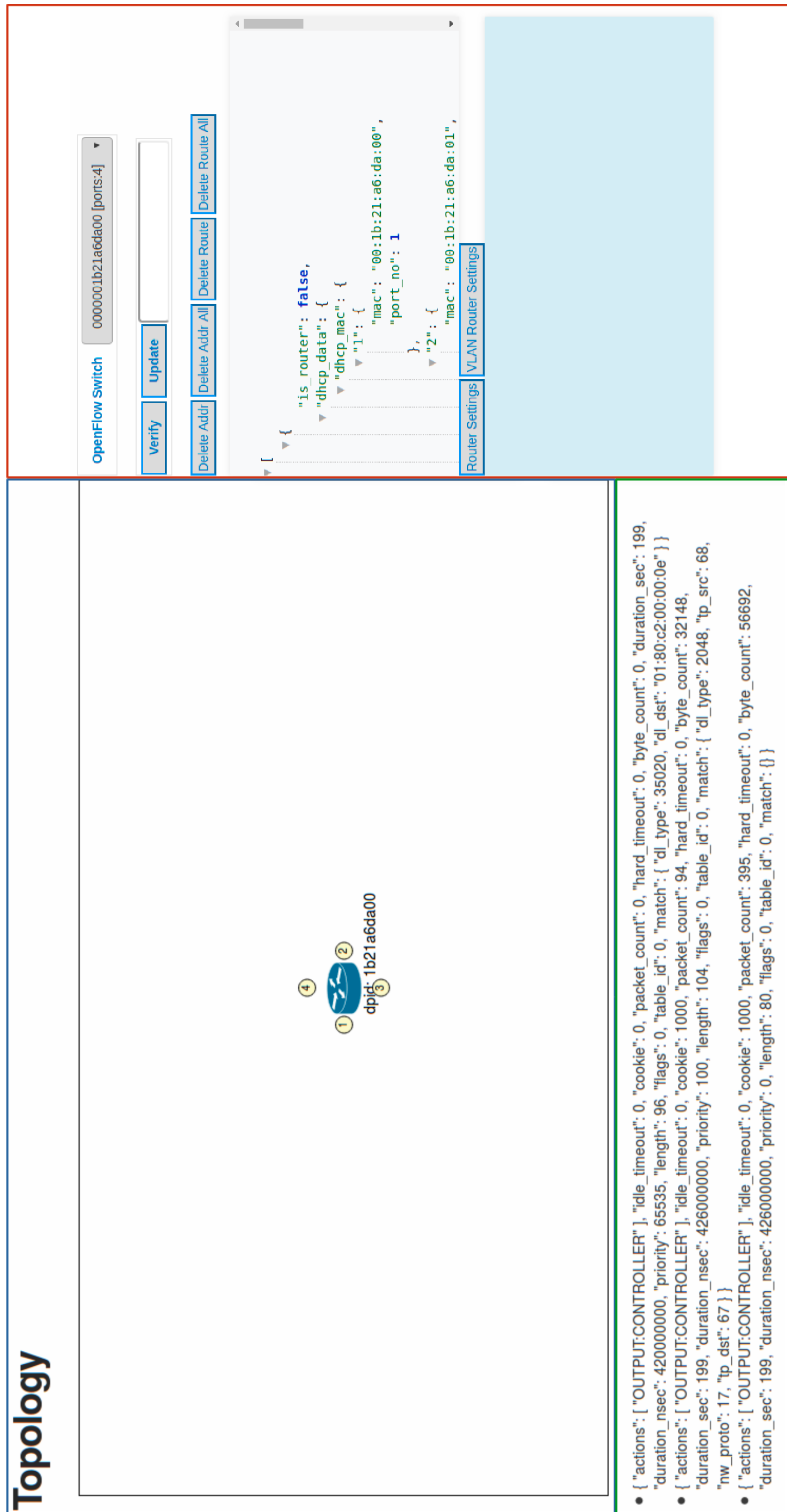


Figure C.1: The Full View of The Web UI.

C.2 Configuration Toolbar

OpenFlow Switch 0000001b21a6da00 [ports:4]

Verify Update

Delete Addr | Delete Addr All | Delete Route | Delete Route All


```
[
  {
    "is_router": false,
    "dhcp_data": {
      "dhcp_mac": {
        "1": {
          "mac": "00:1b:21:a6:da:00",
          "port_no": 1
        },
        "2": {
          "mac": "00:1b:21:a6:da:01",
```


Router Settings | VLAN Router Settings


```
{
  switch enter @ 15:44:33: [
    {
      ports: [
        {
          hw_addr: "00:1b:21:a6:da:00",
          name: "ovsport1",
          port_no: "00000001",
          dpid: "0000001b21a6da00"
        },
        {
          hw_addr: "00:1b:21:a6:da:01",
```


SDN Switch selector

JSON Configuration Input

Options to remove/clear configurations

SDN Switch Network Configuration

SDN Switch Network Configuration

Switch Events & Router Settings

Figure C.2: Configuration Toolbar Showing Settings for an OpenFlow Switch.

Appendix D

Use Case: Layer 2

D.1 IP Addresses

Listing D.1: Host 2 Network Configuration.

```
host2@ubuntu:~$ ifconfig
enp0s31f6 Link encap:Ethernet  HWaddr 30:5a:3a:7e:36:a0
          inet addr:10.0.0.253  Bcast:10.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::325a:3aff:fe7e:36a0/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4251 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1333 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1270258 (1.2 MB)  TX bytes:425570 (425.5 KB)
          Interrupt:16 Memory:f7100000—f7120000
....
```

Listing D.2: Host 3 Network Configuration.

```
host3@ubuntu:~$ ifconfig
enp0s31f6 Link encap:Ethernet  HWaddr 30:5a:3a:7c:d3:58
          inet addr:10.0.0.252  Bcast:10.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::325a:3aff:fe7c:d358/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3365 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1410 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1136334 (1.1 MB)  TX bytes:482866 (482.8 KB)
          Interrupt:16 Memory:f7100000—f7120000
....
```

Listing D.3: Host 4 Network Configuration.

```
host4@ubuntu:~$ ifconfig
enp0s31f6 Link encap:Ethernet  HWaddr 30:5a:3a:7a:0d:b9
          inet addr:10.0.0.251  Bcast:10.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::325a:3aff:fe7a:db9/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:349 errors:0 dropped:0 overruns:0 frame:0
          TX packets:170 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:74726 (74.7 KB)  TX bytes:46580 (46.5 KB)
          Interrupt:16 Memory:f7100000-f7120000
....
```

D.2 DHCP Logs During IP Allocation

Listing D.4: DHCP Application Logs During IP allocation.

```
1  127.0.0.1 - - [18/Aug/2018 08:54:17] "GET /init/0000116528699904 HTTP/1.1" 200 379 0.009102
2  [DHCP][INFO] @ 2018-08-18 08:54:25.048271>    switch:116528699904 Managed by DHCP
3  [DHCP][INFO] @ 2018-08-18 08:54:25.048513>      port:1  Receive DHCP message type
       ↪ DHCP_DISCOVER
4  client_ip_addr 10.0.0.254
5  [DHCP][INFO] @ 2018-08-18 08:54:25.048704>      port:1  Send DHCP message type DHCP_OFFER
6  [DHCP][INFO] @ 2018-08-18 08:54:25.049582>      {'30:5a:3a:7c:cd:9d': '10.0.0.254'}
7  [DHCP][INFO] @ 2018-08-18 08:54:25.049749>      {'10.0.0.254': 1}
8  [DHCP][INFO] @ 2018-08-18 08:54:25.049935>      IP pool len 253
9  [DHCP][INFO] @ 2018-08-18 08:54:25.055254>    switch:116528699904 Managed by DHCP
10 [DHCP][INFO] @ 2018-08-18 08:54:25.055426>      port:1  Receive DHCP message type DHCP_REQUEST
11 [DHCP][INFO] @ 2018-08-18 08:54:25.055546>      port:1  Send DHCP message type DHCP_ACK
12 [DHCP][INFO] @ 2018-08-18 08:55:06.396331>    switch:116528699904 Managed by DHCP
13 [DHCP][INFO] @ 2018-08-18 08:55:06.396580>      port:2  Receive DHCP message type DHCP_REQUEST
14 client_ip_addr 10.0.0.253
15 [DHCP][INFO] @ 2018-08-18 08:55:06.396843>      port:2  Send DHCP message type DHCP_OFFER
16 [DHCP][INFO] @ 2018-08-18 08:55:06.397726>      {'30:5a:3a:7c:cd:9d': '10.0.0.254', '30:5a:3a
       ↪ :7e:36:a0': '10.0.0.253'}
17 [DHCP][INFO] @ 2018-08-18 08:55:06.397912>      {'10.0.0.253': 2, '10.0.0.254': 1}
18 [DHCP][INFO] @ 2018-08-18 08:55:06.398098>      IP pool len 252
19 [DHCP][INFO] @ 2018-08-18 08:55:09.558645>    switch:116528699904 Managed by DHCP
20 [DHCP][INFO] @ 2018-08-18 08:55:09.558899>      port:2  Receive DHCP message type DHCP_REQUEST
21 [DHCP][INFO] @ 2018-08-18 08:55:09.559068>      port:2  Send DHCP message type DHCP_ACK
22 [DHCP][INFO] @ 2018-08-18 08:55:09.640380>    switch:116528699904 Managed by DHCP
23 [DHCP][INFO] @ 2018-08-18 08:55:09.640626>      port:3  Receive DHCP message type DHCP_REQUEST
24 client_ip_addr 10.0.0.252
25 [DHCP][INFO] @ 2018-08-18 08:55:09.640890>      port:3  Send DHCP message type DHCP_OFFER
26 [DHCP][INFO] @ 2018-08-18 08:55:09.641896>      {'30:5a:3a:7c:cd:9d': '10.0.0.254', '30:5a:3a
       ↪ :7c:d3:58': '10.0.0.252', '30:5a:3a:7e:36:a0': '10.0.0.253'}
27 [DHCP][INFO] @ 2018-08-18 08:55:09.642032>      {'10.0.0.252': 3, '10.0.0.253': 2,
       ↪ '10.0.0.254': 1}
28 [DHCP][INFO] @ 2018-08-18 08:55:09.642178>      IP pool len 251
29 [DHCP][INFO] @ 2018-08-18 08:55:12.867275>    switch:116528699904 Managed by DHCP
30 [DHCP][INFO] @ 2018-08-18 08:55:12.867519>      port:3  Receive DHCP message type DHCP_REQUEST
31 [DHCP][INFO] @ 2018-08-18 08:55:12.867671>      port:3  Send DHCP message type DHCP_ACK
32 [DHCP][INFO] @ 2018-08-18 08:55:12.944377>    switch:116528699904 Managed by DHCP
33 [DHCP][INFO] @ 2018-08-18 08:55:12.944619>      port:4  Receive DHCP message type DHCP_REQUEST
34 client_ip_addr 10.0.0.251
```

```
35 [DHCP][INFO] @ 2018–08–18 08:55:12.944811>      port:4  Send DHCP message type DHCP_OFFER
36 [DHCP][INFO] @ 2018–08–18 08:55:12.945658>      {'30:5a:3a:7c:cd:9d': '10.0.0.254', '30:5a:3a
       ↪ :7a:0d:b9': '10.0.0.251', '30:5a:3a:7c:d3:58': '10.0.0.252', '30:5a:3a:7e:36:a0':
       ↪ '10.0.0.253'}
37 [DHCP][INFO] @ 2018–08–18 08:55:12.945868>      {'10.0.0.252': 3, '10.0.0.253': 2,
       ↪ '10.0.0.251': 4, '10.0.0.254': 1}
38 [DHCP][INFO] @ 2019–02–13 08:55:12.946021>      IP pool len 250
39 [DHCP][INFO] @ 2019–02–13 08:55:15.367329>    switch:116528699904 Managed by DHCP
40 [DHCP][INFO] @ 2019–02–13 08:55:15.367572>      port:4  Receive DHCP message type DHCP_REQUEST
41 [DHCP][INFO] @ 2019–02–13 08:55:15.367727>      port:4  Send DHCP message type DHCP_ACK
```

Appendix E

Use Case: Layer 3

E.1 Host Network Configurations

Listing E.1: Host 2 Network Configurations.

```
host2@ubuntu:~$ ifconfig
enp0s31f6 Link encap:Ethernet  HWaddr 30:5a:3a:7c:d3:58
          inet addr:10.0.1.254  Bcast:10.0.1.255  Mask:255.255.255.0
          inet6 addr: fe80::325a:3aff:fe7c:d358/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:334 errors:0 dropped:0 overruns:0 frame:0
          TX packets:191 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:75852 (75.8 KB)  TX bytes:50744 (50.7 KB)
          Interrupt:16 Memory:f7100000-f7120000
...
host2@ubuntu:~$ ip route list
default via 10.0.1.1 dev enp0s31f6
10.0.1.0/24 dev enp0s31f6  proto kernel  scope link  src 10.0.1.254
```

Listing E.2: Host 3 Network Configurations.

```
host4@ubuntu:~$ ifconfig
enp0s31f6 Link encap:Ethernet  HWaddr 30:5a:3a:7c:cd:9d
          inet addr:10.0.2.254  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::325a:3aff:fe7c:cd9d/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:11 errors:0 dropped:0 overruns:0 frame:0
          TX packets:143 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1186 (1.1 KB)  TX bytes:11620 (11.6 KB)
          Interrupt:16 Memory:f7100000-f7120000
....
host4@ubuntu:~$ ip route list
default via 10.0.2.1 dev enp0s31f6
10.0.2.0/24 dev enp0s31f6  proto kernel  scope link  src 10.0.2.254
```

Listing E.3: Host 4 Network Configurations.

```
host3@ubuntu:~$ ifconfig
enp0s31f6 Link encap:Ethernet  HWaddr 30:5a:3a:7e:36:a0
          inet addr:10.0.3.254  Bcast:10.0.3.255  Mask:255.255.255.0
          inet6 addr: fe80::325a:3aff:fe7e:36a0/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:302 errors:0 dropped:0 overruns:0 frame:0
          TX packets:189 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:68902 (68.9 KB)  TX bytes:45258 (45.2 KB)
          Interrupt:16 Memory:f7100000-f7120000
....
host3@ubuntu:~$ ip route list
default via 10.0.3.1 dev enp0s31f6
10.0.3.0/24 dev enp0s31f6  proto kernel  scope link  src 10.0.3.254
```

E.2 DHCP Logs

Listing E.4: DHCP application logs during IP allocation

```
1  [DHCP][INFO] @ 2018-10-22 04:32:03.421104>    switch:116528699904 Managed by DHCP
2  [DHCP][INFO] @ 2018-10-22 04:32:03.421352>      port:1  Receive DHCP message type
       ↪ DHCP_DISCOVER
3  client_ip_addr 10.0.0.254
4  [DHCP][INFO] @ 2018-10-22 04:32:03.421612>      port:1  Send DHCP message type DHCP_OFFER
5  [DHCP][INFO] @ 2018-10-22 04:32:03.422568>      {'30:5a:3a:7a:0d:b9': '10.0.0.254'}
6  [DHCP][INFO] @ 2018-10-22 04:32:03.422764>      {'10.0.0.254': 1}
7  [DHCP][INFO] @ 2018-10-22 04:32:03.422906>      IP pool len 253
8  [DHCP][INFO] @ 2018-10-22 04:32:03.432072>    switch:116528699904 Managed by DHCP
9  [DHCP][INFO] @ 2018-10-22 04:32:03.432626>      port:1  Receive DHCP message type DHCP_REQUEST
10 [DHCP][INFO] @ 2018-10-22 04:32:03.432970>      port:1  Send DHCP message type DHCP_ACK
11 [DHCP][INFO] @ 2018-10-22 04:32:36.768076>    switch:116528699904 Managed by DHCP
12 [DHCP][INFO] @ 2018-10-22 04:32:36.768350>      port:4  Receive DHCP message type
       ↪ DHCP_DISCOVER
13 client_ip_addr 10.0.3.254
14 [DHCP][INFO] @ 2018-10-22 04:32:36.768580>      port:4  Send DHCP message type DHCP_OFFER
15 [DHCP][INFO] @ 2018-10-22 04:32:36.769307>      {'30:5a:3a:7a:0d:b9': '10.0.0.254', '30:5a:3a
       ↪ :7e:36:a0': '10.0.3.254'}
16 [DHCP][INFO] @ 2018-10-22 04:32:36.769446>      {'10.0.3.254': 4, '10.0.0.254': 1}
17 [DHCP][INFO] @ 2018-10-22 04:32:36.769546>      IP pool len 253
18 [DHCP][INFO] @ 2018-10-22 04:32:40.776150>    switch:116528699904 Managed by DHCP
19 [DHCP][INFO] @ 2018-10-22 04:32:40.776318>      port:4  Receive DHCP message type
       ↪ DHCP_DISCOVER
20 client_ip_addr 10.0.3.254
21 [DHCP][INFO] @ 2018-10-22 04:32:40.776434>      port:4  Send DHCP message type DHCP_OFFER
22 [DHCP][INFO] @ 2018-10-22 04:32:40.777012>      {'30:5a:3a:7a:0d:b9': '10.0.0.254', '30:5a:3a
       ↪ :7e:36:a0': '10.0.3.254'}
23 [DHCP][INFO] @ 2018-10-22 04:32:40.777131>      {'10.0.3.254': 4, '10.0.0.254': 1}
24 [DHCP][INFO] @ 2018-10-22 04:32:40.777218>      IP pool len 253
25 [DHCP][INFO] @ 2018-10-22 04:32:44.784966>    switch:116528699904 Managed by DHCP
26 [DHCP][INFO] @ 2018-10-22 04:32:44.785157>      port:4  Receive DHCP message type
       ↪ DHCP_DISCOVER
27 client_ip_addr 10.0.3.254
28 [DHCP][INFO] @ 2018-10-22 04:32:44.785257>      port:4  Send DHCP message type DHCP_OFFER
29 [DHCP][INFO] @ 2018-10-22 04:32:44.785730>      {'30:5a:3a:7a:0d:b9': '10.0.0.254', '30:5a:3a
```

```
       ↪ :7e:36:a0': '10.0.3.254'}
30 [DHCP][INFO] @ 2018-10-22 04:32:44.785845>      {'10.0.3.254': 4, '10.0.0.254': 1}
31 [DHCP][INFO] @ 2018-10-22 04:32:44.785923>      IP pool len 253
32 [DHCP][INFO] @ 2018-10-22 04:32:48.794158>    switch:116528699904 Managed by DHCP
33 [DHCP][INFO] @ 2018-10-22 04:32:48.794325>      port:4  Receive DHCP message type
       ↪ DHCP_DISCOVER
34 client_ip_addr 10.0.3.254
35 [DHCP][INFO] @ 2018-10-22 04:32:48.794442>      port:4  Send DHCP message type DHCP_OFFER
36 [DHCP][INFO] @ 2018-10-22 04:32:48.795285>      {'30:5a:3a:7a:0d:b9': '10.0.0.254', '30:5a:3a
       ↪ :7e:36:a0': '10.0.3.254'}
37 [DHCP][INFO] @ 2018-10-22 04:32:48.795431>      {'10.0.3.254': 4, '10.0.0.254': 1}
38 [DHCP][INFO] @ 2018-10-22 04:32:48.795531>      IP pool len 253
39 [DHCP][INFO] @ 2018-10-22 04:32:52.803470>    switch:116528699904 Managed by DHCP
40 [DHCP][INFO] @ 2018-10-22 04:32:52.803675>      port:4  Receive DHCP message type
       ↪ DHCP_DISCOVER
41 client_ip_addr 10.0.3.254
42 [DHCP][INFO] @ 2018-10-22 04:32:52.803875>      port:4  Send DHCP message type DHCP_OFFER
43 [DHCP][INFO] @ 2018-10-22 04:32:52.804553>      {'30:5a:3a:7a:0d:b9': '10.0.0.254', '30:5a:3a
       ↪ :7e:36:a0': '10.0.3.254'}
44 [DHCP][INFO] @ 2018-10-22 04:32:52.804702>      {'10.0.3.254': 4, '10.0.0.254': 1}
45 [DHCP][INFO] @ 2018-10-22 04:32:52.804853>      IP pool len 253
46 [DHCP][INFO] @ 2018-10-22 04:32:56.812774>    switch:116528699904 Managed by DHCP
47 [DHCP][INFO] @ 2018-10-22 04:32:56.813008>      port:4  Receive DHCP message type
       ↪ DHCP_DISCOVER
48 client_ip_addr 10.0.3.254
49 [DHCP][INFO] @ 2018-10-22 04:32:56.813317>      port:4  Send DHCP message type DHCP_OFFER
50 [DHCP][INFO] @ 2018-10-22 04:32:56.814646>      {'30:5a:3a:7a:0d:b9': '10.0.0.254', '30:5a:3a
       ↪ :7e:36:a0': '10.0.3.254'}
51 [DHCP][INFO] @ 2018-10-22 04:32:56.814856>      {'10.0.3.254': 4, '10.0.0.254': 1}
52 [DHCP][INFO] @ 2018-10-22 04:32:56.814999>      IP pool len 253
53 [DHCP][INFO] @ 2018-10-22 04:33:00.821412>    switch:116528699904 Managed by DHCP
54 [DHCP][INFO] @ 2018-10-22 04:33:00.821608>      port:4  Receive DHCP message type
       ↪ DHCP_DISCOVER
55 client_ip_addr 10.0.3.254
56 [DHCP][INFO] @ 2018-10-22 04:33:00.821809>      port:4  Send DHCP message type DHCP_OFFER
57 [DHCP][INFO] @ 2018-10-22 04:33:00.822484>      {'30:5a:3a:7a:0d:b9': '10.0.0.254', '30:5a:3a
       ↪ :7e:36:a0': '10.0.3.254'}
58 [DHCP][INFO] @ 2018-10-22 04:33:00.822628>      {'10.0.3.254': 4, '10.0.0.254': 1}
59 [DHCP][INFO] @ 2018-10-22 04:33:00.822734>      IP pool len 253
60 [DHCP][INFO] @ 2018-10-22 04:33:36.977542>    switch:116528699904 Managed by DHCP
61 [DHCP][INFO] @ 2018-10-22 04:33:36.977778>      port:3  Receive DHCP message type
       ↪ DHCP_DISCOVER
62 client_ip_addr 10.0.2.254
63 [DHCP][INFO] @ 2018-10-22 04:33:36.977992>      port:3  Send DHCP message type DHCP_OFFER
64 [DHCP][INFO] @ 2018-10-22 04:33:36.979018>      {'30:5a:3a:7c:cd:9d': '10.0.2.254', '30:5a:3a
       ↪ :7a:0d:b9': '10.0.0.254', '30:5a:3a:7e:36:a0': '10.0.3.254'}
65 [DHCP][INFO] @ 2018-10-22 04:33:36.979213>      {'10.0.2.254': 3, '10.0.3.254': 4,
       ↪ '10.0.0.254': 1}
66 [DHCP][INFO] @ 2018-10-22 04:33:36.979346>      IP pool len 253
67 [DHCP][INFO] @ 2018-10-22 04:33:36.986201>    switch:116528699904 Managed by DHCP
68 [DHCP][INFO] @ 2018-10-22 04:33:36.986439>      port:3  Receive DHCP message type DHCP_REQUEST
69 [DHCP][INFO] @ 2018-10-22 04:33:36.986592>      port:3  Send DHCP message type DHCP_ACK
70 [DHCP][INFO] @ 2018-10-22 04:33:49.765645>    switch:116528699904 Managed by DHCP
71 [DHCP][INFO] @ 2018-10-22 04:33:49.765852>      port:4  Receive DHCP message type
       ↪ DHCP_DISCOVER
72 client_ip_addr 10.0.3.254
```

```
73 [DHCP][INFO] @ 2018-10-22 04:33:49.766012>      port:4  Send DHCP message type DHCP_OFFER
74 [DHCP][INFO] @ 2018-10-22 04:33:49.766709>      {'30:5a:3a:7c:cd:9d': '10.0.2.254', '30:5a:3a
       ↪ :7a:0d:b9': '10.0.0.254', '30:5a:3a:7e:36:a0': '10.0.3.254'}
75 [DHCP][INFO] @ 2018-10-22 04:33:49.766869>      {'10.0.2.254': 3, '10.0.3.254': 4,
       ↪ '10.0.0.254': 1}
76 [DHCP][INFO] @ 2018-10-22 04:33:49.766986>      IP pool len 253
77 [DHCP][INFO] @ 2018-10-22 04:33:49.774697>    switch:116528699904 Managed by DHCP
78 [DHCP][INFO] @ 2018-10-22 04:33:49.774902>      port:4  Receive DHCP message type DHCP_REQUEST
79 [DHCP][INFO] @ 2018-10-22 04:33:49.775035>      port:4  Send DHCP message type DHCP_ACK
80 [DHCP][INFO] @ 2018-10-22 04:34:57.404079>    switch:116528699904 Managed by DHCP
81 [DHCP][INFO] @ 2018-10-22 04:34:57.404362>      port:2  Receive DHCP message type
       ↪ DHCP_DISCOVER
82 client_ip_addr 10.0.1.254
83 [DHCP][INFO] @ 2018-10-22 04:34:57.404603>      port:2  Send DHCP message type DHCP_OFFER
84 [DHCP][INFO] @ 2018-10-22 04:34:57.405729>      {'30:5a:3a:7c:cd:9d': '10.0.2.254', '30:5a:3a
       ↪ :7a:0d:b9': '10.0.0.254', '30:5a:3a:7c:d3:58': '10.0.1.254', '30:5a:3a:7e:36:a0':
       ↪ '10.0.3.254'}
85 [DHCP][INFO] @ 2018-10-22 04:34:57.405941>      {'10.0.2.254': 3, '10.0.3.254': 4,
       ↪ '10.0.0.254': 1, '10.0.1.254': 2}
86 [DHCP][INFO] @ 2018-10-22 04:34:57.406104>      IP pool len 253
87 [DHCP][INFO] @ 2018-10-22 04:34:57.416905>    switch:116528699904 Managed by DHCP
88 [DHCP][INFO] @ 2018-10-22 04:34:57.417170>      port:2  Receive DHCP message type DHCP_REQUEST
89 [DHCP][INFO] @ 2018-10-22 04:34:57.417332>      port:2  Send DHCP message type DHCP_ACK
```

Appendix F

Use Case: VLAN

F.1 Host Port Configuration

Listing F.1: Configuring Interface `enp0s31f6` for VLAN 2 for Host 1.

```
1 >>> ip link add link enp0s31f6 name enp0s31f6.2 type vlan id 2
2 >>> ip addr add 192.168.30.10/24 dev enp0s31f6.2
3 >>> ip link set dev enp0s31f6.2 up
4 >>> ip route add default via 192.168.30.1
```

Listing F.2: Configuring Interface `enp0s31f6` for VLAN 110 for Host 3.

```
1 >>> ip link add link enp0s31f6 name enp0s31f6.110 type vlan id 110
2 >>> ip addr add 172.16.10.11/24 dev enp0s31f6.110
3 >>> ip link set dev enp0s31f6.110 up
4 >>> ip route add default via 172.16.10.1
```

Listing F.3: Configuring Interface `enp0s31f6` for VLAN 110 for Host 4.

```
1 >>> ip link add link enp0s31f6 name enp0s31f6.110 type vlan id 110
2 >>> ip addr add 192.168.30.11/24 dev enp0s31f6.110
3 >>> ip link set dev enp0s31f6.110 up
4 >>> ip route add default via 192.168.30.1
```

F.2 VLAN Network Configurations for Host 2, Host 3 and Host 4.

Listing F.4: Network Configuration for Host 2.

```
host2@ubuntu:~$ ip route list
default via 192.168.30.1 dev enp0s31f6.2
192.168.30.0/24 dev enp0s31f6.2  proto kernel  scope link  src 192.168.30.10
host2@ubuntu:~$ ifconfig
enp0s31f6.2 Link encap:Ethernet  HWaddr 30:5a:3a:7c:d3:58
          inet addr:192.168.30.10  Bcast:0.0.0.0  Mask:255.255.255.255
          inet6 addr: fe80::325a:3aff:fe7c:d358/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:308 errors:0 dropped:0 overruns:0 frame:0
          TX packets:24 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:12936 (12.9 KB)  TX bytes:1320 (1.3 KB)
....
```

Listing F.5: Network Configuration for Host 3.

```
host3@ubuntu:~$ ip route list
192.168.30.0/24 dev enp0s31f6.110  proto kernel  scope link  src 192.168.30.11
host3@ubuntu:~$ ifconfig
enp0s31f6.110 Link encap:Ethernet  HWaddr 30:5a:3a:7e:36:a0
          inet addr:192.168.30.11  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::325a:3aff:fe7e:36a0/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:648 (648.0 B)
....
```

Listing F.6: Network Configuration for Host 4.

```
host4@ubuntu:~$ ip route list
default via 172.16.10.1 dev enp0s31f6.110
172.16.10.0/24 dev enp0s31f6.110  proto kernel  scope link  src 172.16.10.11
host4@ubuntu:~$ ifconfig
enp0s31f6.110 Link encap:Ethernet  HWaddr 30:5a:3a:7c:cd:9d
          inet addr:172.16.10.11  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::325a:3aff:fe7c:cd9d/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:16 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:984 (984.0 B)
....
```

F.3 Settled OpenFlow Table

Listing F.7: OpenFlow Table After Learning Hosts


```
1 { "actions": ["OUTPUT:CONTROLLER"], "idle_timeout": 0, "cookie": 0, "packet_count": 0, "hard_timeout": 0, "byte_count": 0, "duration_sec": 2551, "duration_nsec": 173000000, "priority": 65535, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 35020, "dl_dst": "01:80:c2:00:00:0e" } }
2 { "actions": ["OUTPUT:CONTROLLER"], "idle_timeout": 0, "cookie": 8589934593, "packet_count": 10, "hard_timeout": 0, "byte_count": 1020, "duration_sec": 1845, "duration_nsec": 406000000, "priority": 1038, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "dl_vlan": "2", "nw_dst": "172.16.10.1" } }
3 { "actions": ["OUTPUT:CONTROLLER"], "idle_timeout": 0, "cookie": 8589934594, "packet_count": 10, "hard_timeout": 0, "byte_count": 1020, "duration_sec": 1845, "duration_nsec": 406000000, "priority": 1038, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "dl_vlan": "2", "nw_dst": "192.168.30.1" } }
4 { "actions": ["OUTPUT:CONTROLLER"], "idle_timeout": 0, "cookie": 472446402561, "packet_count": 12, "hard_timeout": 0, "byte_count": 1224, "duration_sec": 1845, "duration_nsec": 330000000, "priority": 1038, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "dl_vlan": "110", "nw_dst": "172.16.10.1" } }
5 { "actions": ["OUTPUT:CONTROLLER"], "idle_timeout": 0, "cookie": 472446402562, "packet_count": 19, "hard_timeout": 0, "byte_count": 1938, "duration_sec": 1845, "duration_nsec": 330000000, "priority": 1038, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "dl_vlan": "110", "nw_dst": "192.168.30.1" } }
6 { "actions": ["DEC_NW_TTL", "SET_FIELD: {eth_src:00:1b:21:a6:da:03}", "SET_FIELD: {eth_dst:30:5a:3a:7c:cd:9d}", "OUTPUT:4"], "idle_timeout": 1800, "cookie": 472446402561, "packet_count": 2, "hard_timeout": 0, "byte_count": 204, "duration_sec": 25, "duration_nsec": 552000000, "priority": 1036, "length": 136, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "dl_vlan": "110", "nw_dst": "172.16.10.11" } }
7 { "actions": ["DEC_NW_TTL", "SET_FIELD: {eth_src:00:1b:21:a6:da:02}", "SET_FIELD: {eth_dst:30:5a:3a:7e:36:a0}", "OUTPUT:3"], "idle_timeout": 1800, "cookie": 472446402562, "packet_count": 26, "hard_timeout": 0, "byte_count": 2652, "duration_sec": 11, "duration_nsec": 948000000, "priority": 1036, "length": 136, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "dl_vlan": "110", "nw_dst": "192.168.30.11" } }
8 { "actions": ["DEC_NW_TTL", "SET_FIELD: {eth_src:00:1b:21:a6:da:01}", "SET_FIELD: {eth_dst:30:5a:3a:7c:d3:58}", "OUTPUT:2"], "idle_timeout": 1800, "cookie": 8589934594, "packet_count": 17, "hard_timeout": 0, "byte_count": 1734, "duration_sec": 8, "duration_nsec": 603000000, "priority": 1036, "length": 136, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "dl_vlan": "2", "nw_dst": "192.168.30.10" } }
9 { "actions": ["DEC_NW_TTL", "SET_FIELD: {eth_src:00:1b:21:a6:da:00}", "SET_FIELD: {eth_dst:30:5a:3a:7a:0d:b9}", "OUTPUT:1"], "idle_timeout": 1800, "cookie": 8589934593, "packet_count": 17, "hard_timeout": 0, "byte_count": 1734, "duration_sec": 4, "duration_nsec": 90000000, "priority": 1036, "length": 136, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "dl_vlan": "2", "nw_dst": "172.16.10.10" } }
10 { "actions": ["OUTPUT:NORMAL"], "idle_timeout": 0, "cookie": 8589934593, "packet_count": 0, "hard_timeout": 0, "byte_count": 0, "duration_sec": 1845, "duration_nsec": 406000000, "priority": 1037, "length": 112, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "dl_vlan": "2", "nw_src": "172.16.10.0/255.255.255.0", "nw_dst": "172.16.10.0/255.255.255.0" } }
11 { "actions": ["OUTPUT:NORMAL"], "idle_timeout": 0, "cookie": 8589934594, "packet_count": 0, "hard_timeout": 0, "byte_count": 0, "duration_sec": 1845, "duration_nsec": 406000000, "priority": 1037, "length": 112, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "dl_vlan": "2", "nw_src": "192.168.30.0/255.255.255.0", "nw_dst": "192.168.30.0/255.255.255.0" } }
12 { "actions": ["OUTPUT:NORMAL"], "idle_timeout": 0, "cookie": 472446402561, "packet_count": 0, "hard_timeout": 0, "byte_count": 0, "duration_sec": 1845, "duration_nsec": 330000000, "priority": 1037, "length": 112, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "dl_vlan": "110", "nw_src": "172.16.10.0/255.255.255.0", "nw_dst":
```

```
   ↪ "172.16.10.0/255.255.255.0" } }
13 { "actions": ["OUTPUT:NORMAL"], "idle_timeout": 0, "cookie": 472446402562, "packet_count": 0,
   ↪ "hard_timeout": 0, "byte_count": 0, "duration_sec": 1845, "duration_nsec": 212000000, "
   ↪ priority": 1037, "length": 112, "flags": 0, "table_id": 0, "match": { "dl_type": 2048,
   ↪ "dl_vlan": "110", "nw_src": "192.168.30.0/255.255.255.0", "nw_dst":
   ↪ "192.168.30.0/255.255.255.0" } }
14 { "actions": ["OUTPUT:CONTROLLER"], "idle_timeout": 0, "cookie": 8589934593, "packet_count":
   ↪ 0, "hard_timeout": 0, "byte_count": 0, "duration_sec": 1845, "duration_nsec":
   ↪ 406000000, "priority": 1003, "length": 104, "flags": 0, "table_id": 0, "match": { "
   ↪ dl_type": 2048, "dl_vlan": "2", "nw_dst": "172.16.10.0/255.255.255.0" } }
15 { "actions": ["OUTPUT:CONTROLLER"], "idle_timeout": 0, "cookie": 8589934594, "packet_count":
   ↪ 0, "hard_timeout": 0, "byte_count": 0, "duration_sec": 1845, "duration_nsec":
   ↪ 406000000, "priority": 1003, "length": 104, "flags": 0, "table_id": 0, "match": { "
   ↪ dl_type": 2048, "dl_vlan": "2", "nw_dst": "192.168.30.0/255.255.255.0" } }
16 { "actions": ["OUTPUT:CONTROLLER"], "idle_timeout": 0, "cookie": 472446402561, "packet_count":
   ↪  0, "hard_timeout": 0, "byte_count": 0, "duration_sec": 1845, "duration_nsec":
   ↪ 330000000, "priority": 1003, "length": 104, "flags": 0, "table_id": 0, "match": { "
   ↪ dl_type": 2048, "dl_vlan": "110", "nw_dst": "172.16.10.0/255.255.255.0" } }
17 { "actions": ["OUTPUT:CONTROLLER"], "idle_timeout": 0, "cookie": 472446402562, "packet_count":
   ↪  1, "hard_timeout": 0, "byte_count": 102, "duration_sec": 1845, "duration_nsec":
   ↪ 330000000, "priority": 1003, "length": 104, "flags": 0, "table_id": 0, "match": { "
   ↪ dl_type": 2048, "dl_vlan": "110", "nw_dst": "192.168.30.0/255.255.255.0" } }
18 { "actions": [], "idle_timeout": 0, "cookie": 8589934592, "packet_count": 1480, "hard_timeout
   ↪ ": 0, "byte_count": 115472, "duration_sec": 1845, "duration_nsec": 406000000, "priority
   ↪ ": 1002, "length": 64, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "dl_vlan
   ↪ ": "2" } }
19 { "actions": [], "idle_timeout": 0, "cookie": 472446402560, "packet_count": 843, "hard_timeout
   ↪ ": 0, "byte_count": 66522, "duration_sec": 1845, "duration_nsec": 330000000, "priority
   ↪ ": 1002, "length": 64, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "dl_vlan
   ↪ ": "110" } }
20 { "actions": ["OUTPUT:CONTROLLER"], "idle_timeout": 0, "cookie": 1000, "packet_count": 0, "
   ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 31771, "duration_nsec": 444000000, "
   ↪ priority": 100, "length": 104, "flags": 0, "table_id": 0, "match": { "dl_type": 2048, "
   ↪ tp_src": 68, "nw_proto": 17, "tp_dst": 67 } }
21 { "actions": ["OUTPUT:CONTROLLER"], "idle_timeout": 0, "cookie": 3000, "packet_count": 159, "
   ↪ hard_timeout": 0, "byte_count": 9540, "duration_sec": 1845, "duration_nsec": 406000000,
   ↪  "priority": 2, "length": 88, "flags": 0, "table_id": 0, "match": { "dl_type": 2054 } }
22 { "actions": [], "idle_timeout": 0, "cookie": 0, "packet_count": 0, "hard_timeout": 0, "
   ↪ byte_count": 0, "duration_sec": 1845, "duration_nsec": 406000000, "priority": 2, "
   ↪ length": 64, "flags": 0, "table_id": 0, "match": { "dl_type": 2048 } }
23 { "actions": ["OUTPUT:CONTROLLER"], "idle_timeout": 0, "cookie": 1000, "packet_count": 64794,
   ↪ "hard_timeout": 0, "byte_count": 4032864, "duration_sec": 31771, "duration_nsec":
   ↪ 444000000, "priority": 0, "length": 80, "flags": 0, "table_id": 0, "match": {} }
24 { "actions": ["OUTPUT:NORMAL"], "idle_timeout": 0, "cookie": 3000, "packet_count": 0, "
   ↪ hard_timeout": 0, "byte_count": 0, "duration_sec": 1845, "duration_nsec": 406000000, "
   ↪ priority": 1, "length": 80, "flags": 0, "table_id": 0, "match": {} }
```

Listing F.8: Logs After Applying of VLAN Policy.

```
1 #....
2 [RT][INFO] switch_id=0000001b21a6da00: Set default route (drop) flow [cookie=0x200000000]
3 [RT][INFO] switch_id=0000001b21a6da00: Set host MAC learning (packet in) flow [cookie=0
  ↪ x200000001]
4 [RT][INFO] switch_id=0000001b21a6da00: Set IP handling (packet in) flow [cookie=0x200000001]
5 [RT][INFO] switch_id=0000001b21a6da00: Set L2 switching (normal) flow [cookie=0x200000001]
6 # Adding VLAN 'address': '192.168.30.1/24'
```

```
7  [RT][INFO] switch_id=0000001b21a6da00: Set host MAC learning (packet in) flow [cookie=0
       ↪ x200000002]
8  [RT][INFO] switch_id=0000001b21a6da00: Set IP handling (packet in) flow [cookie=0x200000002]
9  [RT][INFO] switch_id=0000001b21a6da00: Set L2 switching (normal) flow [cookie=0x200000002]
10 127.0.0.1 - - [22/Oct/2018 02:35:19] "GET /init/0000001b21a6da00 HTTP/1.1" 200 388 0.301416
11 # Adding VLAN 'address': '172.16.10.1/24'
12 [RT][INFO] switch_id=0000001b21a6da00: Set default route (drop) flow [cookie=0x6e00000000]
13 [RT][INFO] switch_id=0000001b21a6da00: Set host MAC learning (packet in) flow [cookie=0
       ↪ x6e00000001]
14 [RT][INFO] switch_id=0000001b21a6da00: Set IP handling (packet in) flow [cookie=0x6e00000001]
15 [RT][INFO] switch_id=0000001b21a6da00: Set L2 switching (normal) flow [cookie=0x6e00000001]
16 #....
```

Listing F.9: Logs During Ping Messaging to Router Interface.

```
1  #....
2  [RT][INFO] switch_id=0000001b21a6da00: Set implicit routing flow [cookie=0x200000002]
3  [RT][INFO] switch_id=0000001b21a6da00: Receive ARP request from [192.168.30.10] to router port
       ↪  [192.168.30.1].
4  [RT][INFO] switch_id=0000001b21a6da00: Send ARP reply to [192.168.30.10]
5  [RT][INFO] switch_id=0000001b21a6da00: Set implicit routing flow [cookie=0x6e00000001]
6  [RT][INFO] switch_id=0000001b21a6da00: Receive ARP request from [172.16.10.11] to router port
       ↪ [172.16.10.1].
7  [RT][INFO] switch_id=0000001b21a6da00: Send ARP reply to [172.16.10.11]
8  #....
9  [RT][INFO] switch_id=0000001b21a6da00: Receive ICMP echo request from [192.168.30.10] to
       ↪ router port [172.16.10.1].
10 [RT][INFO] switch_id=0000001b21a6da00: Send ICMP echo reply to [192.168.30.10].
11 #....
12 [RT][INFO] switch_id=0000001b21a6da00: Receive ICMP echo request from [192.168.30.10] to
       ↪ router port [192.168.30.1].
13 [RT][INFO] switch_id=0000001b21a6da00: Send ICMP echo reply to [192.168.30.10].
14 #....
```