Software-Defined Network Overlays

BY

OLUWAMAYOWA ADE ADELEKE
B.TECH., Ladoke Akintola University of Technology, 2010

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2015

Chicago, Illinois

To my Father, Jesus, and the Holy Spirit.

To all individuals and organizations that are working towards making communication networks more efficient.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Seferoglu, whose continuous encouragement and counsel has made this thesis possible. Also, I would like to thank my thesis committee members; Professor Yang and Professor Devroye for their time and insightful comments.

I would like to thank the Electrical and Computer Engineering administrators, particularly Tina Alvarado, Ala Wrobleski, and Erica Plys, whose timely advice has kept me on track while completing this thesis.

I would like to thank the PhantomNet-Emulab team at the University of Utah, particularly Professor van der Merwe as well as Kirk Webb, Binh Nguyen, Junchuk Cho. They have given me the opportunity to work on the test-bed, and kindly and patiently replied to all my questions.

My thanks also go to Pastor Lanre Otukoya and Pastor Shola Olaiya, whose words and prayers have made my Chicago days easier. I also appreciate all members of the Winners Chapel, Chicago for helping me not to miss my family too badly.

My big thanks go to my uncle and his family; Uncle Matt, Aunt Biola, Yemisi and Dolapo. I really appreciate that they have hosted me during my final year. Finally, I really thank my family; my father; Dn Surv Adetunji Adeleke, my mother; Dns FM Adeleke, and my sisters; Deola, Bukky, and Damola. I would like to thank them for being there for me all the time; I love you. I also thank my dear Kehinde; I sincerely appreciate your prayers and constant support.

ACKNOWLEDGEMENTS (continued)

Most importantly I thank God, the Almighty, for helping me to complete my MS thesis at UIC within the expected time frame.

OA

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The focus of this study is on software-defined overlay networks, that is, networks which accommodate both legacy network components such as conventional switches and routers as well as SDN-enabled network components. Design and development of software-defined overlay networks have potential to lead to incremental SDN deployment, which reduces the initial deployment costs and encourages the network operators to switch to SDN. Our goal is to understand and develop software-defined overlays, particularly focusing on the question of which network components should be SDN-enabled so that the benefits of SDNs are fully exploited. Towards this goal, we first conduct a comprehensive review of SDNs. Then, we focus on software-defined overlay networks. Finally, we implement a prototype networks to compare software-defined overlays with pure SDN (where all network components are SDN-enabled), and legacy networks (where none of the network components are SDN-enabled) via experiments conducted using the Emulab networking test-bed.

We observe from our experiments that software-defined overlay networks are able to exploit the full potential of SDNs, if the SDN-enabled switches are deployed by taking into account network topologies. The performance of the SDN overlay network is comparable to the performance of the full SDN and the full legacy network. We believe that our results would provide useful insights on the incremental deployment of SDNs on top of the existing legacy networks.

CHAPTER 1

INTRODUCTION

1.1 Motivation and Problem Statement

The usage of computer communication networks, especially the Internet, has increased significantly over the last few decades both in terms of number of users and average data traffic. In the last decade, the number of Internet users increased from about nine hundred million users to almost three billion, which is quite significant. Furthermore, the number of devices connected to the Internet is estimated to be around seven billion based on the current world population [1].

The growth in total traffic is also significant; in the last decade, the amount of annual IP traffic over the Internet has increased by more than one thousand five hundred percent from 780 petabytes to more than 34,000 petabytes [2]. In the Unites States, in the same time period, the amount of monthly Internet traffic increased from 140,00 Terabytes to 4,100,000 terabytes. These statistics keep increasing and it is estimated that by 2018, the global Internet IP traffic will increase to 84,000 petabytes [3].

This exponential increase in data demand has made it necessary for network operators and service providers to find new mechanisms of handling traffic and network management. Software-defined networking (SDN) is a new paradigm in computer networking [4], and it has a potential of handling larger and increasing traffic in today's networks by providing more and better control functionalities.

In conventional networks, the control plane and the data plane are implemented, and closely bound together, so they require individual configuration of each network element to provide any additional functionalities. On the other hand, SDNs propose to separate the data-plane from the control plane, and move the control plane functionalities to a central controller [5], which is equipped with a global view of the network [6]. This makes it possible to develop simple applications that run on the controller to perform any required functionalities.

The SDN paradigm has introduced new aspects in research and industry by (i) providing more granularity in network control, (ii) faster implementation of new services, (iii) elimination of the need for specialized middle boxes, (iv) better security control, and (v) easier mechanisms to test, develop, and deploy new services.

However, majority of research on SDN assumes full SDN deployment; *i.e.*, all the network components should be SDN-enabled. Yet, this may not be possible in today's networks, where legacy devices, in addition to SDN-enabled devices, are commonly used. This thesis focuses on software-defined networking overlays, which is a hybrid network where SDN-enabled switches co-exist with legacy network components. This approach has a potential of increasing the deployment of SDN networks as it provides incremental development and deployment of SDNs. This approach also reduces high initial implementation costs of pure SDNs. Legacy routing algorithms are robust to link failures and have fast convergence time. SDN overlays can help network operators to benefit from these advantages of legacy networks while enjoying, to some degree, the flexibility provided by SDNs. This thesis explores how conventional legacy networks can be updated with SDN-enabled network components to create SDN overlays.

1.2 Objectives

The goal of this thesis is to understand and develop software-defined overlays, particularly focusing on the question of which network components should be SDN-enabled so that the benefits of SDNs are fully exploited. Towards this goal, we first conduct a comprehensive review of SDNs, then we focus on software-defined overlay networks. Finally, we implement a prototype to compare the performance of software-defined overlays with pure SDN (where all network components are SDN-enabled), and legacy networks (where none of the network components are SDN-enabled) via experiments conducted using the Emulab networking test-bed.

1.3 Thesis Outline

This thesis is organized as follows: Chapter one presents the problem statement, the objectives, and the outline of the thesis. Chapter two presents a comprehensive review of software-defined networking; it describes the SDN architecture, giving details of the SDN-enabled switch data plane, OpenFlow, SDN controllers and a summary of network programing languages. The third chapter gives a review of network overlay technologies and presents some existing research in hybrid SDN. The fourth chapter gives the project methodology and it presents the software and test-bed used in carrying out the experiments. The fourth chapter also gives a detailed description of the experiments performed, presenting prototype topologies and the results of the experiments. The final chapter provides our conclusions and future research directions.

CHAPTER 2

OVERVIEW OF SOFTWARE-DEFINED NETWORKING

2.1 The Need for Software-Defined Networks

Legacy networks are inherently difficult to manage and update, because most of the time each single network element needs to be configured to run often-proprietary, complex, distributed algorithms. These algorithms are usually integrated in the hardware and may not be compatible with devices from other vendors. This makes the change or modification of such algorithms difficult, if not impossible.

Unlike conventional legacy networks in which the intelligence and control are distributed across all network elements, SDN moves all the intelligence and control to a centralized controller, which is equipped with a global network view. This makes it possible to develop customized software on the controller side (control plane) to manage the network components (data plane), *i.e.*, it makes network components, hence networks, programmable. Thus, in the core of SDN there is a highly intelligent controller, which manages the forwarding behavior of the "dumb" network elements or switches in the data plane [7].

SDN provides a programmatic, logically centralized interface to control the data-plane network. Through this interface, a software program acts as a network controller by implementing several rules, including forwarding, by possibly reacting to topology and traffic variations [8]

In SDN, the idea of software–hardware separation and dynamic network programmability on a central controller have several advantages. First, data traffic can be grouped into finegrained classes (flows) based on a combination of all parameters that cuts across the protocol stack and each of these flows can be treated differently at each switch. Hence the flow-based nature of SDN offers more granularity (increased level of detail) in network control, i.e. in an SDN-enabled switch, packets can be dropped, forwarded, or re-routed based on any combination of parameters that cut across protocol stack. Second, due to the fact that the data plane elements do not have to be intelligent, SDN network components cost less than the conventional network components. SDN also removes the need for specialized, expensive middle-boxes, because all middle-box functionalities can be programmed on the SDN controller. Furthermore, network management is made much easier, because SDN networks do not have to run complex distributed software. Advanced traffic engineering, network slicing, network virtualization, and several other network functionalities can be realized in a simple manner via SDN. In addition, the separation of the network software from the hardware implies that the two aspects can evolve separately and at a faster rate, hence leading to a faster rate of innovation in network research.

2.2 Software-Defined Networking Architecture

A simplified SDN architecture, as described by the Open Networking Foundation [9] - the regulatory organization for OpenFlow, is shown in Figure 1. The SDN architecture separates the network into the data plane, the control plane, and the application layer, which is implemented above the control layer [10].

The interfaces between the control layer and the infrastructure layer are referred to as the southbound interfaces. OpenFlow [7] is just one example of such interfaces, although it is the most popular, because it is an open source protocol. Another southbound interface is the Cisco proprietary OpFlex protocol [11].

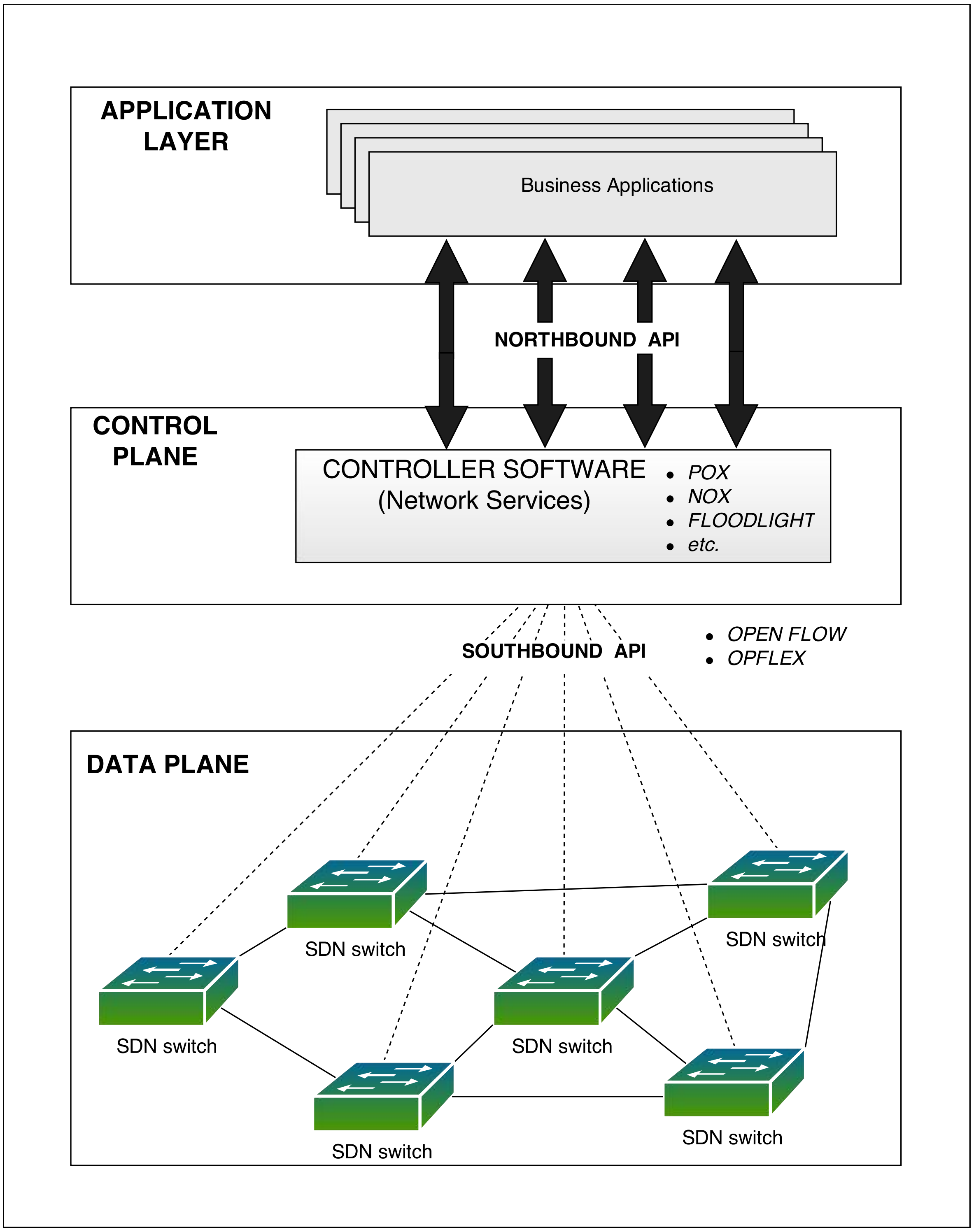

Figure 1. Software-defined networking architecture

The northbound interfaces provide communication between the control plane and the application programs [6]. High-level programs developed by using various languages can communicate with the controller through these northbound APIs. Each of the layers and interfaces in the SDN architecture are discussed in the subsequent sections.

2.2.1 The Data Plane

In general, the data plane (also referred to as the forwarding plane) consists of all physical devices that are involved in processing and forwarding of packets. These devices can be switches, routers, firewalls, gateways, and all middle-boxes. In conventional networks, these devices are pre-programmed or configured to carry out required specialized functions. However, in SDN data plane, all these devices are replaced with simple SDN-enabled switches. An SDN switch can be viewed as a switching device that can make forwarding decisions based on any combination of parameters from layers 2, 3 and 4 of the network stack, utilizing flow-based switching.

SDN switches are often classified as software or hardware switches. The most popular OpenFlow enabled software switch is the Open vSwitch [12], which can be installed on Linux-based devices to make them function as a virtual switch with interfaces of the Linux-based device as the switch's interfaces. The Open vSwitch can also be installed on ASIC switches. Other OpenFlow enabled software switches include the Pantou/OpenWrt switch, the Ofsoftswitch, and the Indigo. Pantou OpenWRT can be installed on some commercial wireless routers to convert them into OpenFlow enabled wireless switches, the Ofsoftswitch [13] can also be installed on Linux machines while Indigo [14] can be installed on commercial switch ASICs, to make them function as SDN-enabled devices.

In this thesis we use Open vSwitch software running on the Linux platform, to implement our SDN-enabled switches. The Open vSwitch can also be ported to the NetFPGA platform to get cheaper switching gear for research purposes and small-scale applications. There are also implementations for wireless routers via OpenWrT on some supported simple access points.

On the other hand, there are also a number of commercially available SDN-enabled hardware switches by Cisco, Big switch, Arista, Brocade, Dell, HP, and many others [15].

2.2.2 South Bound API - OpenFlow Protocol

In the SDN architecture, the southbound API defines the means of communication between the control and data planes. A number of southbound protocols have been created, some of which are open source, while the others are vendor-proprietary. Due to the extensive use of the OpenFlow protocol in research and industry, we provide a detailed explanation of OpenFlow next.

The OpenFlow protocol was introduced by [7]. The initial purpose for developing the OpenFlow protocol was to boost innovation in campus networks. It was also created to enable computer networking research experiments to be tested on real networks, alongside regular network traffic without causing disruption. In addition, it was also created to enable real-time programmatic control of networks.

OpenFlow is a protocol that enables communication between a centralized controller(s) and the OpenFlow enabled switches in the SDN architecture. The controller defines the functions including the forwarding behavior of all switches. The first packet of each new incoming flow into a controller's domain is transferred from the switch of entry into the controller on a secure interface via OpenFlow protocol. The controller determines the appropriate behavior for all connected SDN switches for that specific flow based on instructions specified in the application layer. Then, the controller forwards control messages with appropriate flow entries to all the switches in the network, and sends the packet back to the switch of entry, again via OpenFlow. After this setting, all matching packets in the flow are forwarded directly by the switches based on the flow entries that have been specified, without connecting to the controller, until specified idle time-out or hard time-out expires [16].

It is proposed in [7] that OpenFlow enabled switches must have an interface through which they can receive control messages from the controller and a secure communication channel, to transfer data and commands between the control plane and the forwarding plane.

Furthermore, OpenFlow switches must have a flow table. Rules in the flow tables, called flow entries, consist of match fields and corresponding actions that specify how the switch handles each incoming flow. The flow table operates in a match-action processing. OpenFlow match fields include destination and source MAC addresses, IP addresses, port numbers, VLAN tag and several other fields. The OpenFlow switch specification 1.4 [17] gives a list of forty-two flow marching fields.

Each flow is specified by any possible combination of matching fields and will have actions associated with it. The set of flow actions include output (to select the physical output port through which packets should be sent out of a switch), drop (to drop all packets corresponding to that flow entry), set (to change the source & destination IP, MAC, Port, VLAN etc. for all packets in that flow) and many others. The full list can be found in [17].

2.2.3 Control Plane

One of the basic functionalities of OpenFlow is to connect the data plane to the control plane. The control plane in an SDN is a logically centralized entity that performs all calculations based on a global view of the network and sets flow entries on all switches. The controller is responsible for maintaining all network functionalities and for distribution of appropriate instructions to the network devices. Also, it is responsible for determining how to handle packets without valid flow entries [18].

In SDN, all the intelligence in the network is abstracted into the controller, which is in charge of installing and deleting flow entries from tables. The controller decides the routes that each packet takes by specifying the appropriate output ports for all incoming flows. Usually, data plane elements send the first packet in a flow to the controller, the controller then decides the appropriate paths for all packets that belong to the flow by processing this packet, then the controller sends appropriate flow entries which show what to do with the packets to the data plane elements.

In this setup, an administrator can effectively control the flow of packets within the network, great flexibility by developing software programs on top of the controller. There are several controller platforms that have been developed in the literature; some of these are open-source and available for free while others are not. A comprehensive listing of popular SDN controllers are provided in [15].

Due to the fact that the controller platform of choice for our implementations (provided in chapter 4) was the POX controller, which is an extension of NOX, a detailed description of the

NOX and POX controllers is given below.

2.2.3.1 NOX

One of the first OpenFlow standard controller platforms is the NOX controller [19]. It was proposed as an open source network operating system, and was developed mainly in C++, by Nicira - a company started by the initial inventors of the OpenFlow protocol.

In a NOX controlled network, a set of switches is connected to one or more controllers where the network management software applications run. NOX provides a network view that includes switch level topology, locations of user hosts, middle-boxes, and all network elements and services. It also provides a reasonable level of flow-based granularity. NOX works perfectly with OpenFlow; if a packet doesn’t match flow entries on a switch, it is forwarded to one of the control processes on the NOX [16].

Applications that are created on the NOX controller use information from the packets to determine whether or not to forward traffic and generate the appropriate set of actions. In NOX, events can be generated directly by OpenFlow messages such as switch join, switch leave, packet received, and switch statistics received. Other events can also be generated directly from NOX processes [16].

2.2.3.2 POX

POX was developed as an extension of the NOX controller [20]. It can be basically considered as a Python version of NOX. It is used as a conventional teaching platform for software-defined networking. The POX controller is targeted for research purposes and is extensively used in combination with Mininet: a prototyping and virtualization software that can be used to emulate SDN networks containing many hosts and switches in a personal computer.

Therefore, in this project, POX has been selected as our controller platform.

Having presented a brief overview of SDN, we go on to discuss overlay networks and hybrid SDN technologies in the next chapter.

CHAPTER 3

OVERLAY TECHNOLOGIES AND HYBRID SDN

3.1 The Concept of Overlays

Software-defined network overlays combine SDN with overlay networking concept. An overlay network is a logical or virtual network built on top of an underlying physical network [21]. Overlay networks are composed of nodes, which are a subset of the nodes in a physical network, and are connected by virtual/logical links that correspond to paths on that physical network.

Overlay networks can be used to provide specialized routing, isolation, security, multicast, mobility and several other services on legacy networks by modifying legacy devices or adding new network devices. By deploying overlays, new services can be enabled in existing networks without expensive hardware costs that may be associated with installation of the new services. In this setup, only the specific nodes that require specialized services need to be configured. Overlay networks are often built over existing networks using tunneling or encapsulation to provide new services in networks with little or almost no changes to the network infrastructure. Since our proposed SDN overlay networks will combine overlay technologies with SDN, we present a brief description of overlay networking in the following sections.

3.2 Overlay Networking Technologies

In large datacenters, customers may require dedicated nodes in an isolated network. In such networks, several customers may share the same physical infrastructure. However, each customer will have a logically separate network that will be incapable of communicating with other customers' networks, even when they use the same physical nodes and links. By deploying overlay networks, the problem can be easily solved and the physical network can provide connectivity among nodes, while the overlay network handles high-level network policies using tunnels and encapsulation to isolate traffic for each customer.

One of the simplest methods of implementing overlay networks is by using virtual local area networks (VLANs). VLANs can be used to separate a physical network into two or more isolated logical networks, each with a unique VLAN identifier. The network devices or ports with the same VLAN identifier can communicate, while the network elements with different VLANs cannot communicate.

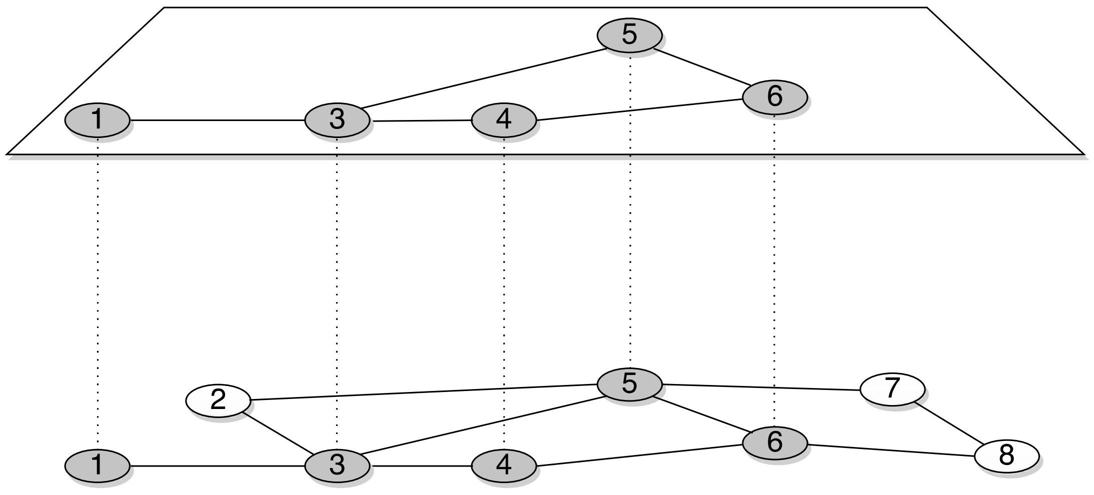

Figure 2: A VLAN overlay network. The VLAN in the overlay plane consists only of shaded nodes in the underlying physical network. Un-shaded nodes cannot communicate with the shaded nodes even though they may be physically connected.

VLANs are adequate for creating overlays in most small-scale enterprise networks. However, in today's datacenter networks, the most popular overlay network technologies are the Virtual Extensible Local Area Network (VXLAN) and the Network Virtualization using Generic Routing Encapsulation (NVGRE) overlay technologies. The two techniques are able to handle a larger number of overlay networks, overcoming the limitations of VLAN.

In VXLAN, an IP/UDP packet encapsulates the Ethernet frame. Frame encapsulation is done at virtual tunnel endpoints (VTEP). VXLAN uses a 24-bit VXLAN header in the encapsulation process providing up to 16 million virtual L2 networks. The VXLAN frame is presented below.

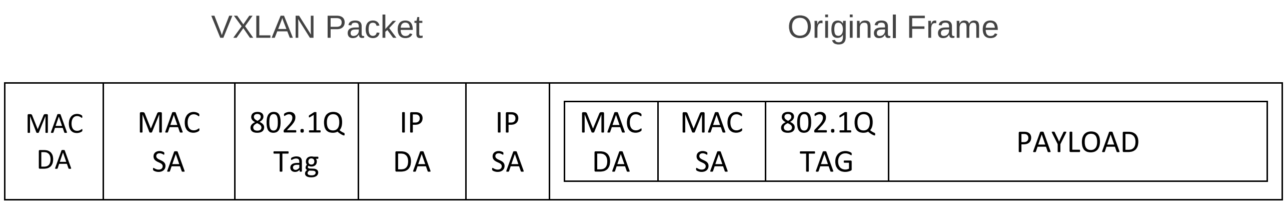

Figure 3. The VXLAN frame, showing how the original frame is encapsulated in another layer of layer 3 and layer 2 headers.

Network Virtualization using Generic Routing Encapsulation (NVGRE) [22] is another popular overlay technology. At the sending end, NVGRE encapsulates an Ethernet Frame in an additional IP header, appending a twenty-four bit TNI, and then sends it through an IP tunnel. The receiving endpoint removes the encapsulation and forwards the packet to the actual destination MAC, thus enabling the creation of virtual Layer 2 networks that can span multiple physical Layer 3 IP networks.

3.3 Benefits of SDN Overlays

Having discussed the two main enabling technologies for our proposed SDN overlays (i.e. SDN technology and network overlay technologies), we briefly discuss the potential benefits of having software-defined overlay networks in this section.

The advantages of SDN, which are discussed in the previous chapter, make it crucial to deploy SDN in legacy networks. Although, full SDN deployment offers the best of benefits in terms of flexibility and programmability, a lot can be gained by deploying SDN infrastructure alongside legacy network hardware, either in the form of overlays.

An overlay of SDN on a legacy network will enable the implementation of specialized policies and functions on traffic that flows through the SDN-enabled devices in such networks. SDN overlays can be designed to ensure that the SDN-enabled devices handle the higher level policy decisions, leaving the underlying legacy network to do the work of maintaining simple connectivity across the network data path [8]. With this design, the hybrid or overlay network can make use of almost all the benefits associated with full SDN.

An advantage is that without much change to existing network, overlay SDN can make it possible to restructure the physical networks, and upgrade them without much change to the SDN controller policy. Since the legacy network is focused on maintaining connectivity, in the event that there are minor changes or upgrades, the network will not be affected or require changing of policies on the SDN-enabled devices. An SDN overlay can be designed to have very fast response to link failures. Since routing algorithms on legacy devices with individual control plane can quickly react to failures by making local decisions [23]. In the event of a link failure in the underlay network, the distributed routing algorithms will quickly respond with alternative routes in the SDN overlay network.

Another major benefit for the deployment of SDN overlay networks is that it enables incremental deployment of SDN on legacy networks. This is important for organizations with existing legacy networks in the process of migration to SDN, especially because replacing all equipment in an existing network might be prohibitively expensive.

3.4 Approaches to Hybrid SDN

Before we discuss our methodology for implementing software-defined network overlays, we examine some related work in hybrid SDN in this section. Hybrid SDN networks are networks that contain both legacy and SDN-enabled devices. We define SDN overlay network as a type of hybrid SDN network where overlay technologies are used to maintain communication between SDN and legacy devices. Four types of hybrid SDN deployment models: topology-based, service-based, class-based and integrated hybrid SDN are presented in [23].

The topology-based SDN requires a physical separation of nodes within the network into two zones: SDN zones and legacy network zones. Each node in the network belongs to only one zone. An example is Google’s B4 network [24], which we discuss later in this section.

In the service-based model, both SDN and legacy networks are used to provide specific services across the network. In this setup, it is possible for a single node to be part of both the SDN and the legacy network. For instance, the legacy network may handle basic network connectivity functions, while SDN is used to provide special services like traffic engineering, access control, load balancing etc. [23]. An example of service based SDN is the OSHI [25].

In the class-based hybrid SDN model, all devices in the network have both legacy and SDN functionality. That is, each device can act as a legacy router and as a SDN switch simultaneously. However, the SDN controller and the legacy routing algorithm are designed to control distinct slices or classes of traffic flows. For instance, an SDN may be used to move all HTTP traffic across a network while all other traffic is controlled by legacy routing techniques.

Integrated hybrid SDN refers to networks in which SDN-enabled and legacy devices operate together. In integrated hybrids, SDN communicates directly with the legacy routing protocol and uses the legacy protocol to either inject routes into the node information base, or to modify legacy network settings [23]. Examples of integrated hybrids in research are route flow and software-defined exchange points (SDX) [26].

3.4.1 Google B4

The Google B4 is a good example of topology-based SDN. B4 is the Google private WAN network that connects all Google's internal datacenters across the globe. Although Google maintains legacy networking for its customer-facing network, it uses the B4 SDN network for internal communication among its datacenters. The two isolated portions of the network communicate at selected points creating a form of topology-based hybrid. At each node on the Google B4 network, specialized servers enable distributed routing and central traffic engineering as a routing overlay [24]. In the B4 network, the Quagga open-source routing daemon running BGP/ISIS was used in conjunction with a SDN Routing Application Proxy (RAP) [24].

3.4.2 Open Source Hybrid IP/SDN (OSHI)

OSHI is an implementation of hybrid SDN that utilizes an open source software switch, which supports both SDN and legacy networking on the same device. The OSHI node combines Quagga routing daemon for OSPF routing and the open vSwitch software for flow-based switching on a Linux box. The SDN Capable Switch (SCS) is connected to the set of physical network interfaces belonging to the integrated IP/SDN network, while the IP forwarding engine is connected to a set of virtual ports of the SCS [25].

OSHI uses VLAN tagging to distinguish between traffic to be processed by either SDN or IP routing. OSHI have been deployed and tested on Mininet [27] and on the Ofelia test-bed [26].

3.4.3 SDX

Software-defined Internet exchange SDX [26] applies SDN to legacy BGP WAN networks to give better control and flexibility to BGP interconnections between ISPs at Internet exchange points where multiple BGP networks meet. SDX uses an integrated hybrid SDN approach, in which legacy routers running BGP communicate with the open flow enabled SDX switch.

The SDX switch has two main components; a policy compiler and a route server, operating simultaneously on the same switch. The SDN component of the switch is the policy compiler. It is a SDN controller application based on pyretic programming language (a programming language for the POX controller platform). The route server serves as the legacy network component. The route server used was ExaBGP: a python based router software for Linux machines [26]. ExaBGP was configured to run the BGP routing protocol. When the route server receives BGP adverts from participating ASes, it processes them and computes best paths to each destination. The route server then sends the routes to the SDX policy compiler. The best allowed paths, as determined by the SDX policy compiler application running on the SDN controller, are then advertised back to the connected ASes via the route server [26].

In summary, a common property of all the technologies described in this section is that all of them utilize a route server or daemon (running a distributed routing protocol), in conjunction with one or more SDN controller programs. However the link between the SDN and legacy networks are implemented using various techniques. OSHI uses VLANs; Google B4 employs specialized software in conjunction with IP in IP tunneling. Our implementation of SDN overlays relies on virtual extensible local area network (VXLAN) overlays to bridge between SDN and legacy routing algorithms.

CHAPTER 4

METHODOLOGY AND IMPLEMETATION

4.1 Methodology

Our goal in this project is to understand the benefits of SDN in a legacy network without replacing all devices in the network with SDN-enabled switches. To achieve this goal, we consider SDN overlay networks. In this section, we show how SDN overlays can be implemented by combining techniques drawn from our review of hybrid SDN technologies with VXLAN overlay technology. We also show that the SDN overlay network can utilize all benefits of SDN, without hurting throughput performance. Deploying SDN overlay networks provides incremental deployment from full legacy network to a full SDN network. A simple sketch of the proposed topology of an SDN overlay network is given in Fig. 4.

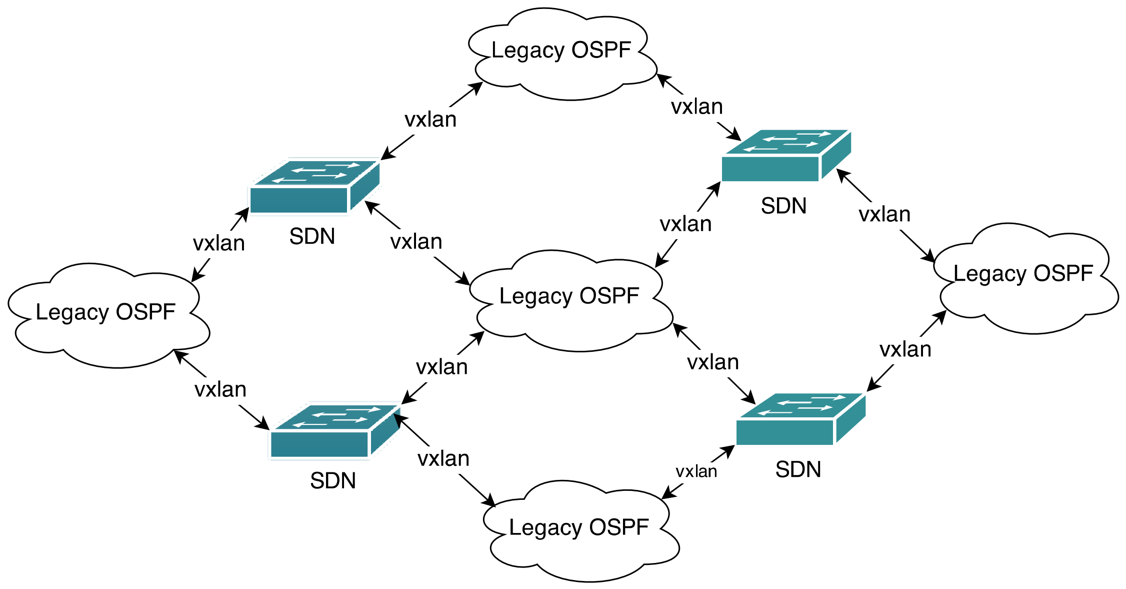

Figure 4: An SDN overlay network in which SDN switches interact with legacy networks through VXLAN tunnels.

In Fig. 4, all SDN switches are connected to a central controller (which not shown to make the figure clear) and they communicate with legacy devices using VXLAN tunnels.

We carried out a series of experiments on the Emulab test-bed [28] to demonstrate the feasibility of our proposed SDN overlay network using a prototype. The topology of our prototype SDN overlay network is presented in Fig.5. We carried out further experiments to compare the operation of the system with legacy networks and SDN networks, which we discuss later in this chapter. In particular, we implemented prototypes for a SDN overlay network, a full SDN network and a full legacy network, using physical devices on the Emulab test-bed and PhantomNet Testbed. Our experiments use open source software including Mininet [27], Quagga [29], POX controller [20], the Open vSwitch [30] and Iperf [31].

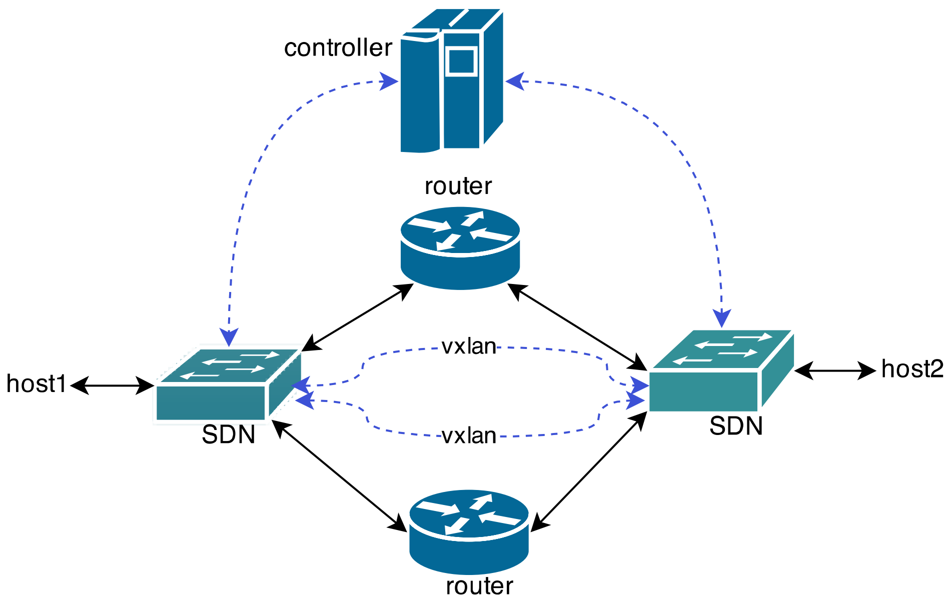

Figure 5. The prototype SDN overlay network with two SDN switches and two legacy routers.

In our test-bed (Emulab), SDN switches are implemented as open vSwitches running on Linux machines. The controller (POX) runs on a Linux device that has a direct connection with all the SDN switches. To enable legacy routing, we use the Quagga routing daemon to enable the open shortest path first (OSPF) routing protocol on our routers. All our devices are implemented as Linux operating devices in the Emulab test-bed. In the next sub-section, we provide a brief description of the Emulab test-bed environment, and we provide detailed descriptions of our experiments afterwards.

4.1.1 The Emulab Test-bed

The Emulab test-bed is operated by the Flux Research Group of the University of Utah. The test-bed allows researchers to develop, debug, and evaluate their research projects on physical hardware from remote locations. It is incorporated in GENI and Planet lab test-beds [28]. To use Emulab, an arbitrary network topology specified with a ns2 tcl [32] file, defines specification of the nodes and links in the network. The operating systems for each node, the properties for each link (bandwidth and delay), IP addresses and routing algorithm can also be specified for all devices using the topology file.

When an experiment is initiated on the Emulab test-bed, each node in the topology file is assigned to a physical device on the test-bed, and the required functionalities are activated. The links, specified delay and bandwidth in the topology file are realized by using Ethernet interfaces. After the initiation, necessary codes can be installed and run on any of the nodes to provide additional functionalities. Software to implement routing, SDN switching and any other functionalities may be installed on any node.

4.2 Experiment Setup

Three representative topologies were implemented out to evaluate the feasibility and performance benefits of the proposed SDN overlay networks. The detailed descriptions for each topology, and the results obtained are presented in this section. The topologies include the full-legacy, full SDN and SDN overlay networks. The experiments with these topologies help us to understand the performance of our proposed SDN overlay and compare it with full-legacy and full SDN deployments. Next we provide a description of the full-legacy network topology.

4.2.1 Legacy Network

The topology implemented for the case of a legacy network is shown in Fig.6. Four legacy routers are implemented using Linux machines. The ns2 file that specifies the topology, is provided in Appendix 1. Note that by specifying the command "*$ns rtproto Manual*" on the topology file, we instruct Emulab to apply no routing algorithm on the network. After the experiment was initiated, we installed the Quagga routing daemon on each of the routers and the open shortest path first (OSPF) routing protocol is configured on all of them.

Routers r1 and r4 are physically connected via two paths. A fast path through r2 have 1Mbps links with 0ms delay, drop tail queuing and loss rate of 0.01%. The second path through r3 have 500kbps links with 0ms delay, drop tail queuing and loss rate 0.01%, implying that the path through r2 is much better than the path through r3.

We installed the Iperf throughput performance measurement software on the two end hosts to generate and measure throughput across the network. The results of TCP and UDP throughput tests are presented in Fig. 7. In this setup OSPF finds the best route, which is via r2. Fig. 7 confirms that, and OSPF selects the 1Mbps routes between h1 and h2. Fig. 8 provides the jitter and the percentage of dropped packets.

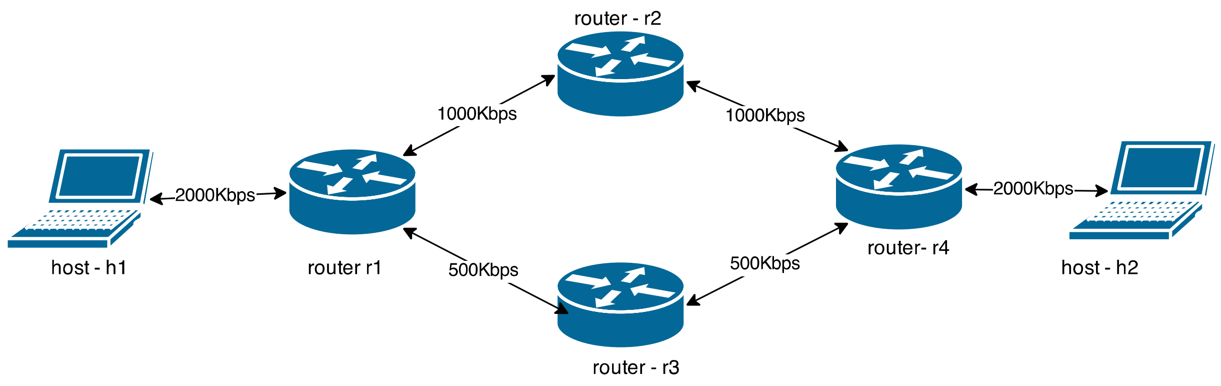

Figure 6. Full legacy network prototype, showing routers r1 and r4 connected through two paths: a 1Mbps path through r2 and a slower 500Kbps link through r3.

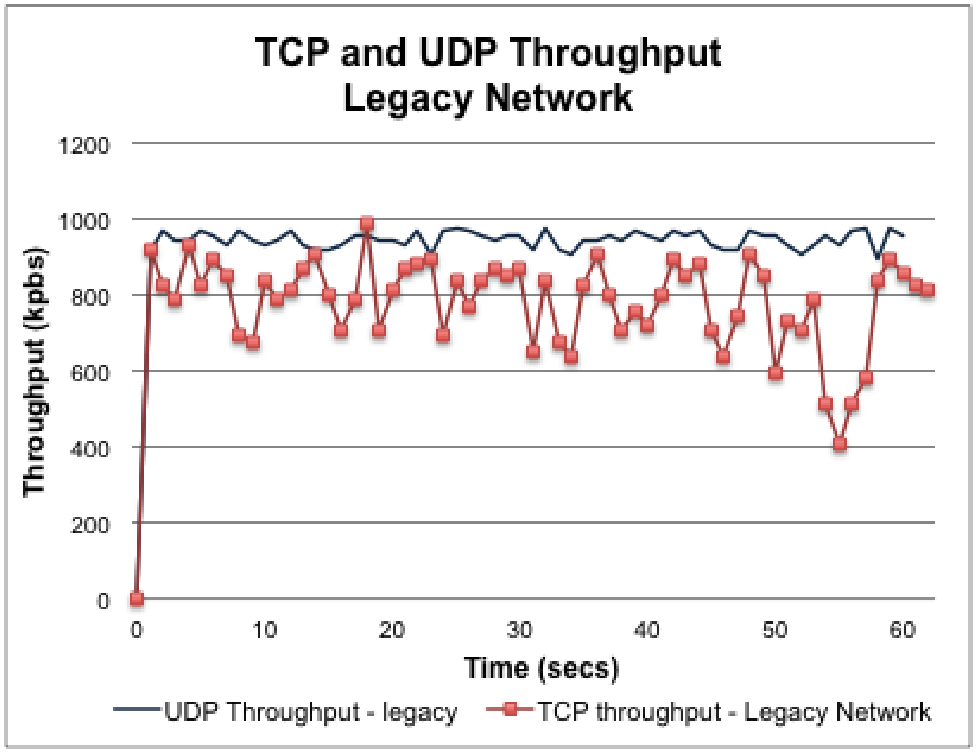

Figure 7. Throughput vs. time graph for the full legacy network, showing UDP and TCP throughputs. As expected, the legacy network selects the 1000Mbps path between r1 and r4.

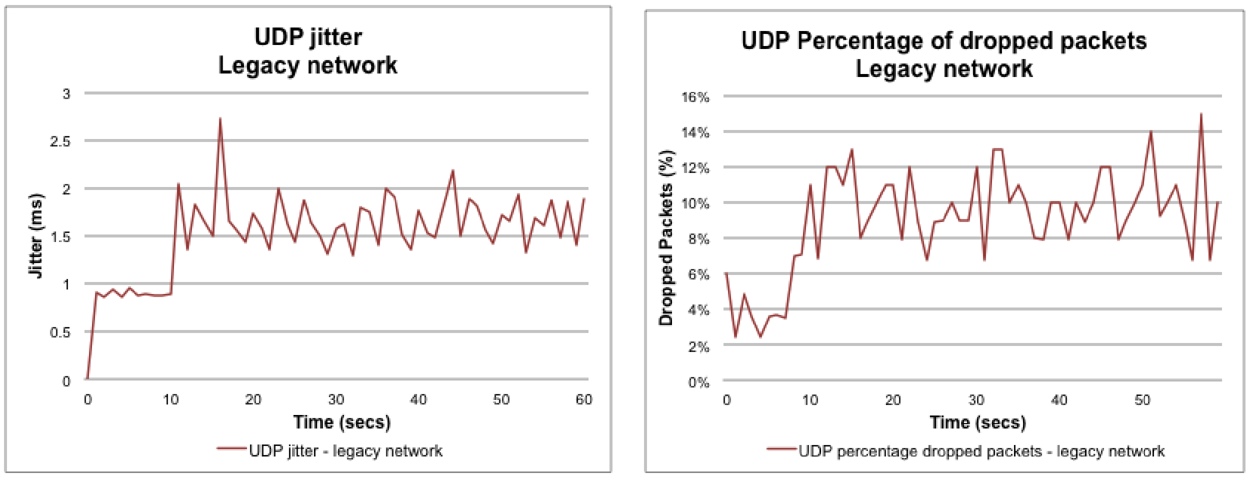

Figure 8. UDP jitter and percentage of dropped packets vs. time for the full legacy network.

4.2.2 Full SDN Network

The topology we consider for the case of full SDN network is shown in Figure 9. The specification of the topology using ns2 is presented in Appendix 2a. On Emulab, we implemented the SDN switches with open vSwitch software running on a computer with a SMORE-SDN disk image. In the topology h1 and h2 are end hosts, c1 is the SDN controller, and ofs1, ofs2, ofs3 and ofs4 are SDN switches. As in the full legacy prototype network, there are two paths between switches ofs1 and ofs4. That is, there is a high-speed path through ofs2 with transmission rates 1Mbps, and a slower path through ofs3 with transmission rates of 500kbps. The controller c1 is connected to each of the SDN switches through dedicated interfaces on a different network.

After initiating the experiment, we ran Open vSwitch on the SDN nodes. Afterwards, a POX controller application is created in the controller c1, to manage the routing of end-to-end traffic between the two hosts. The controller configuration codes are presented in Appendix 2b. We designed our controller application to forward all http traffic (TCP port 80) through the high-speed path (1Mbps) and all other traffic through the slower path (500kbps).

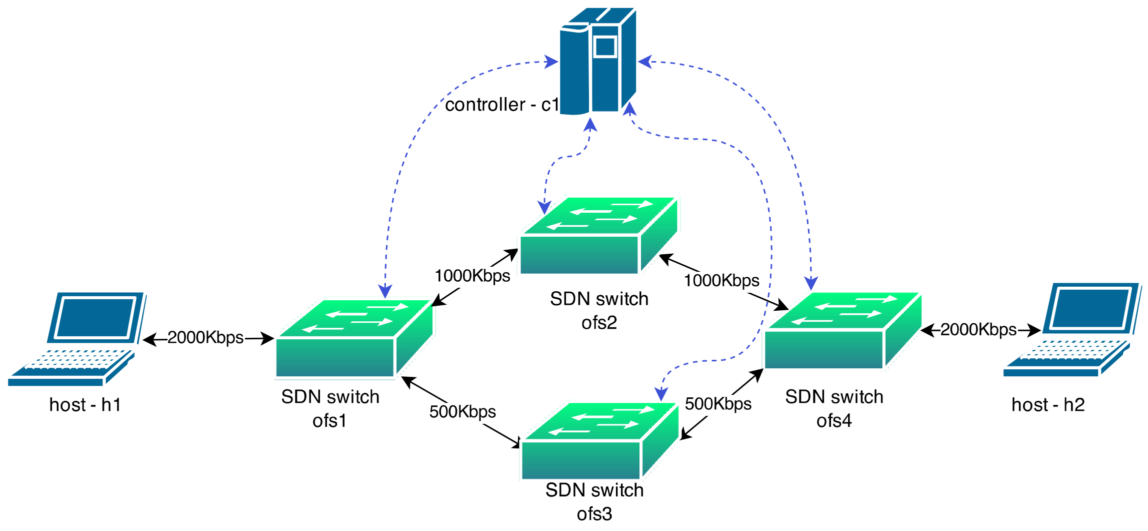

Figure 9. The full SDN network topology. Dotted lines indicate a direct connection from each SDN switch to a central controller through dedicated interfaces.

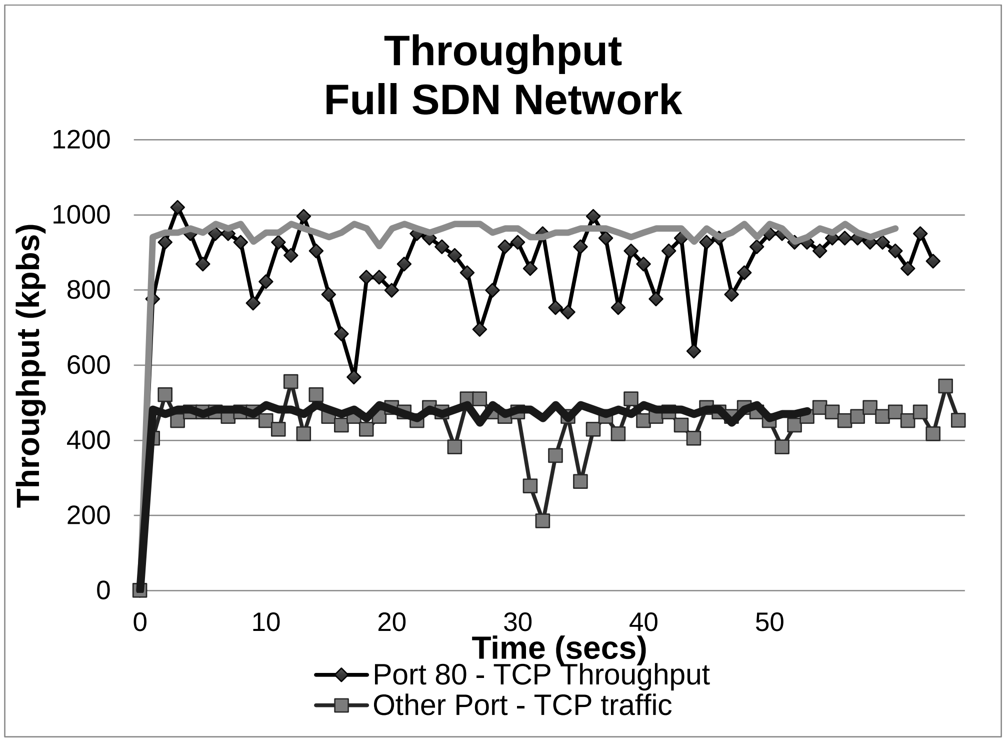

Figure 10. Throughput vs. time for the full SDN topology. HTTP traffic (port number 80) are routed through the 1000Mbps path while other traffic is forwarded through the 500Mbps path.

After installing the Iperf software for network performance measurement on hosts h1 and h2, we measured TCP and UDP throughput as a function of time (Fig. 10). As expected all http traffic is routed through the high-speed path and other traffic was routed through the slower path. The graphs for the throughput, jitter and percentage dropped packets are presented in Fig. 11.

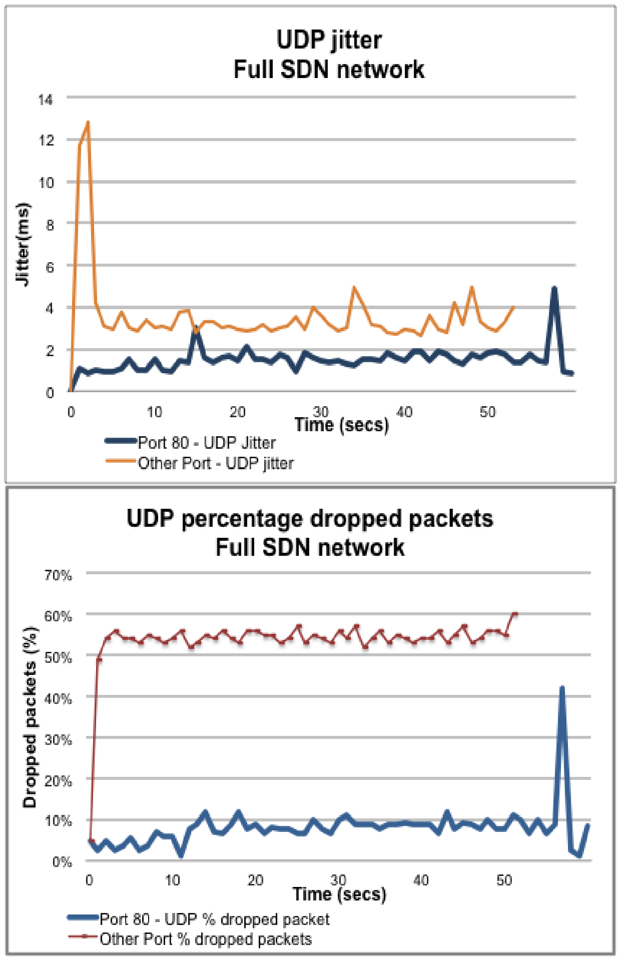

Figure 11. UDP jitter and percentage dropped packets vs. time for the full SDN network. There is more jitter and more packet are dropped for the non-HTTP traffic because they are routed through a lower speed path.

In the next sets of experiments, to show additional benefits of SDN over legacy networks, we simultaneously transmit a file on TCP port 80 and on another file through an arbitrary port

(port 87) from h1 to h2. We start by transmitting on port 80. After 30 seconds we start transmitting on port 87. We observe from the results in Fig. 12 that the SDN Controller is able to transmit both of the two flows simultaneously via independent paths. The file transmitted on TCP port 80 is sent via ofs2 at 1Mbps and the second file transmitted on TCP port 87 is sent via ofs3 at 500Kbps. The additional data transmitted TCP port 87 does not adversely affect the transmission via TCP port 80. Thus the throughput between hosts h1 and h2 is increased to a maximum of 1.5Mbps as compared to 1Mbps observed in the legacy network implementation.

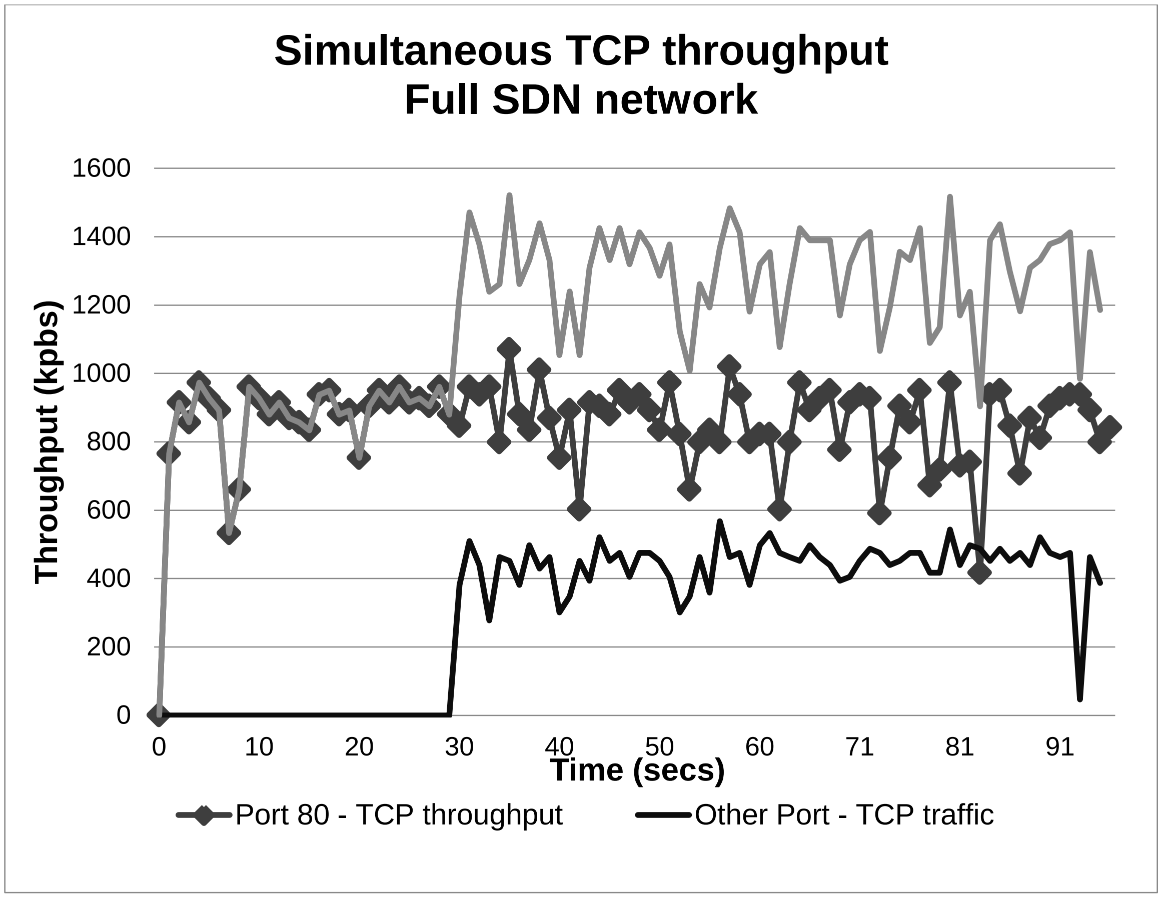

Figure 12. Simultaneous transmission full SDN network. We start by sending HTTP data on port 80 (1000Mbps), after 30 seconds, we begin to transmitting data with other port number simultaneously. SDN enables simultaneous transmission of the two traffic classes making total throughput 1500Mbps possible.

4.2.3 SDN Overlay Network

In the final experiment, we consider a topology of an SDN overlay network on the Emulab Testbed. The topology is shown in Fig. 13 and the configuration file for the network is presented in Appendix 3a.

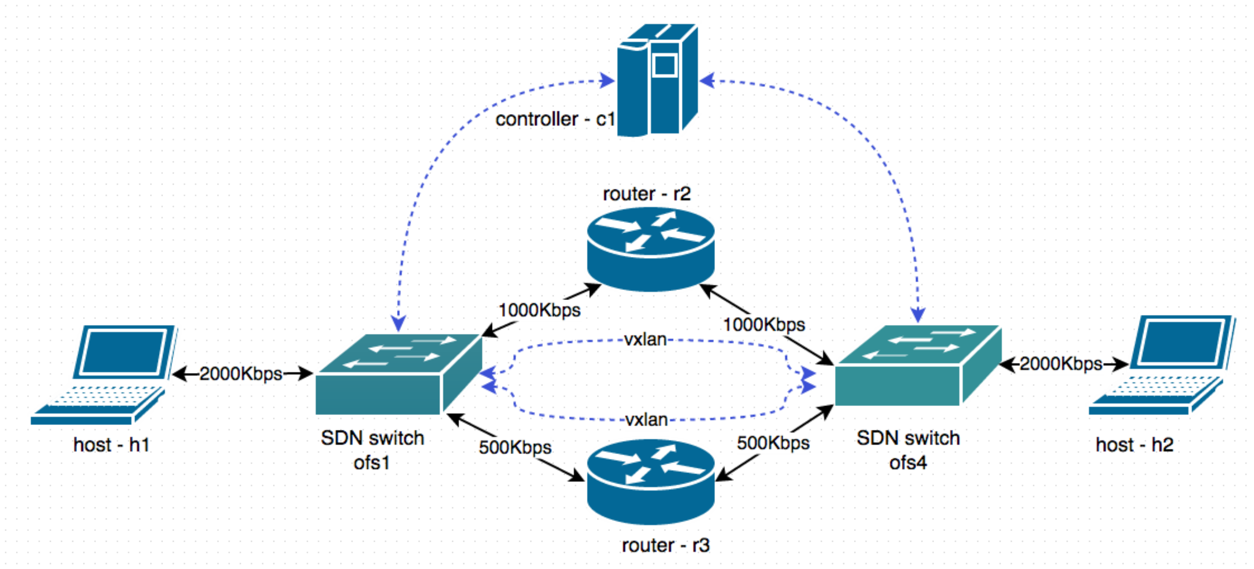

Figure 13. The hybrid SDN topology, showing VXLAN overlay tunnels across the legacy network. When a controller is connected to switches ofs1 and ofs4, the configuration makes the controller see the tunnel endpoints on each switch as Ethernet interfaces, with ofs1 and ofs2 appearing as if they are directly connected.

In this set of experiments, we implement the SDN switches by installing open vSwitch software on ofs1 and ofs4. We then configured an underlay legacy network by creating appropriate routes on ofs1, ofs4, r2 and r3 to enable independent layer 3 communications between ofs1 and ofs4, on the fast 1Mbps path and on the 500Kbps path. Afterwards, we created

VXLAN tunnels between ofs1 and ofs4 through the previously configured legacy network. The commands used to create the tunnels are presented next.

```
ofs1# ovs-vsctl add-port br0 vtep -- set interface vtep type=vxlan
options:remote_ip=10.1.5.3 ofport_request=10
ofs1# ovs-vsctl add-port br0 vtep2 -- set interface vtep type=vxlan
options:remote_ip=10.1.6.3 ofport_request=20

ofs2# ovs-vsctl add-port br0 vtep -- set interface vtep type=vxlan
options:remote_ip=10.1.3.3 ofport_request=10
ofs1# ovs-vsctl add-port br0 vtep2 -- set interface vtep type=vxlan
options:remote_ip=10.1.4.3 ofport_request=20
```

The first set of commands, show the configuration of two VXLAN tunnel end points labeled vtep, and vtep2 on the SDN switch ofs1. The tunnel endpoints are configured with the IP address of the destination switch interfaces to which packets will be delivered. The first tunnel endpoint, vtep, sends packets through the 1Mbps path, while vtep2 sends packets through the 500Kbps path. The next set of commands is also applied on ofs2 to create similar VXLAN endpoints on it.

After creating VXLAN tunnel endpoints on each switch, we configure the open vSwitches on ofs1 and ofs2 with virtual interfaces (including the VXLAN endpoint). We then connected the SDN switches to the POX controller. The controller program is similar to the one used for implementing our full SDN topology; it contains modifications to support the VXLAN interfaces used in the SDN overlay topology. The controller code we developed is also presented in appendix 3b. After creating the topology, Iperf tests are performed to measure the throughput between host 1 and host 2. Note that, our goal is to see if the SDN overlay topology can perform exactly as the full-SDN topology. The results obtained are explained next.

Based on observations of throughput in Fig. 14., the performance of the SDN overlay topology is very similar to the performance of the full-SDN prototype. HTTP packets (port 80) are routed through the high-speed links (1Mbps) as specified in the controller instructions, while other packets are routed through the slower links (500 Kbps) as expected. The throughputs for simultaneous transmission of traffic on port 80 and traffic on other ports are also measured. The results (shown in Fig 16) are similar to that of the full SDN network in Fig. 12.

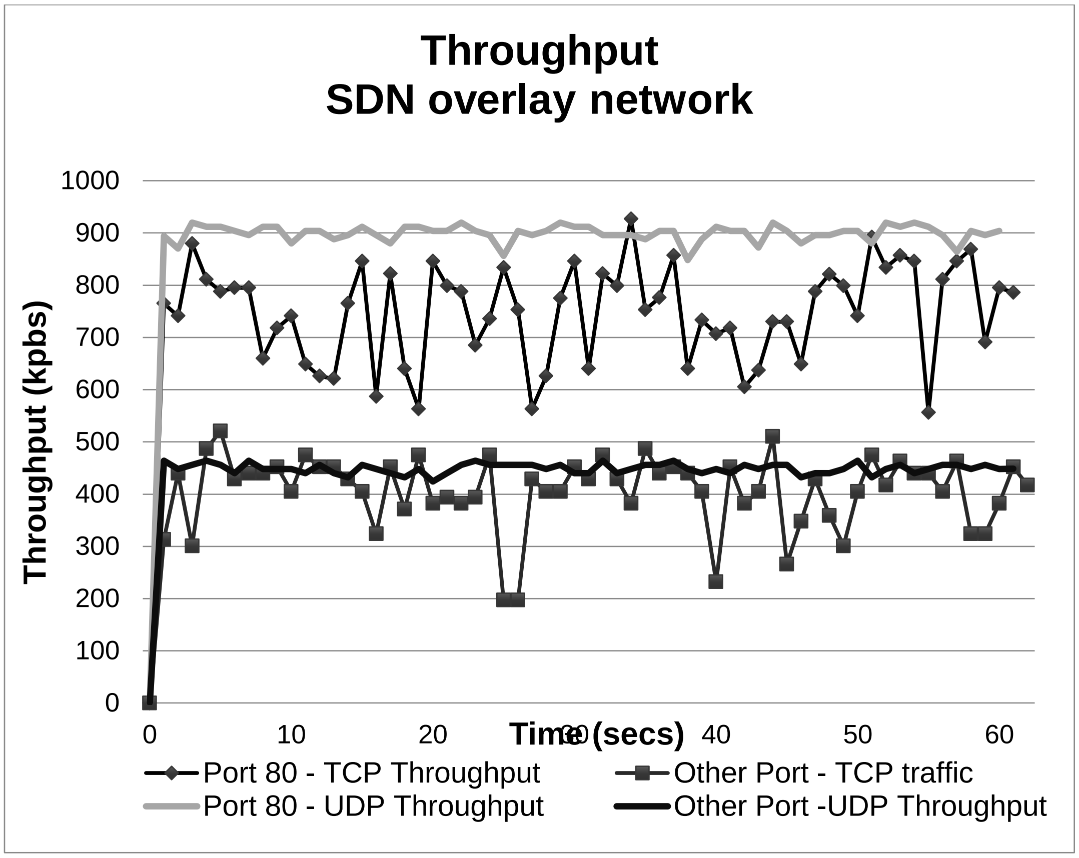

Figure 14. Throughput vs. time in SDN overlay network. HTTP traffic (TCP port 80) is routed through the 1Mbps path while other traffic is forwarded through the 500Mbps path. The results are comparable with that of the full SDN topology in Fig. 10.

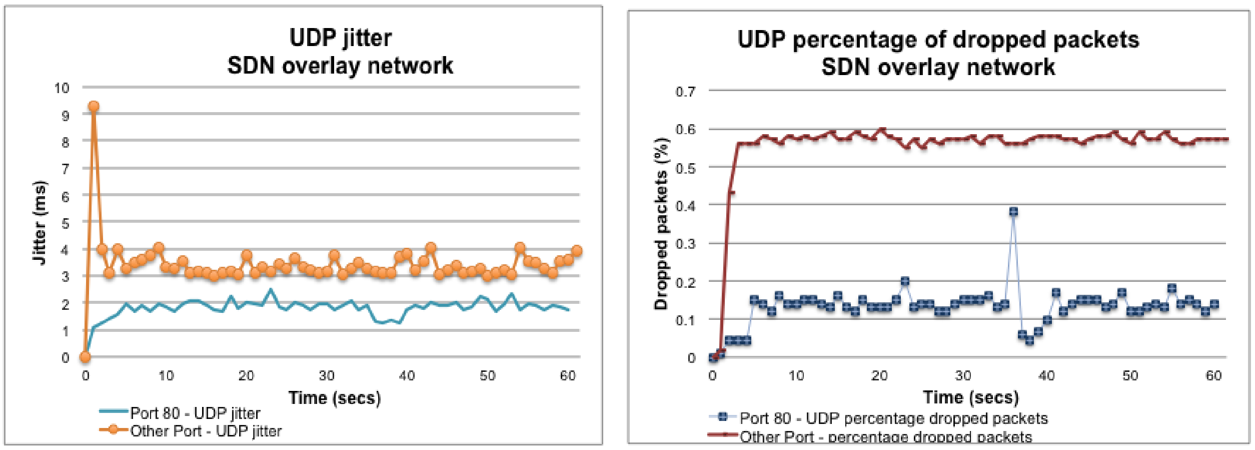

Figure 15. UDP jitter and percentage of dropped packets vs. time for the SDN overlay network. The results are similar to those obtained for full legacy and full SDN provided in Fig. 8 and 11.

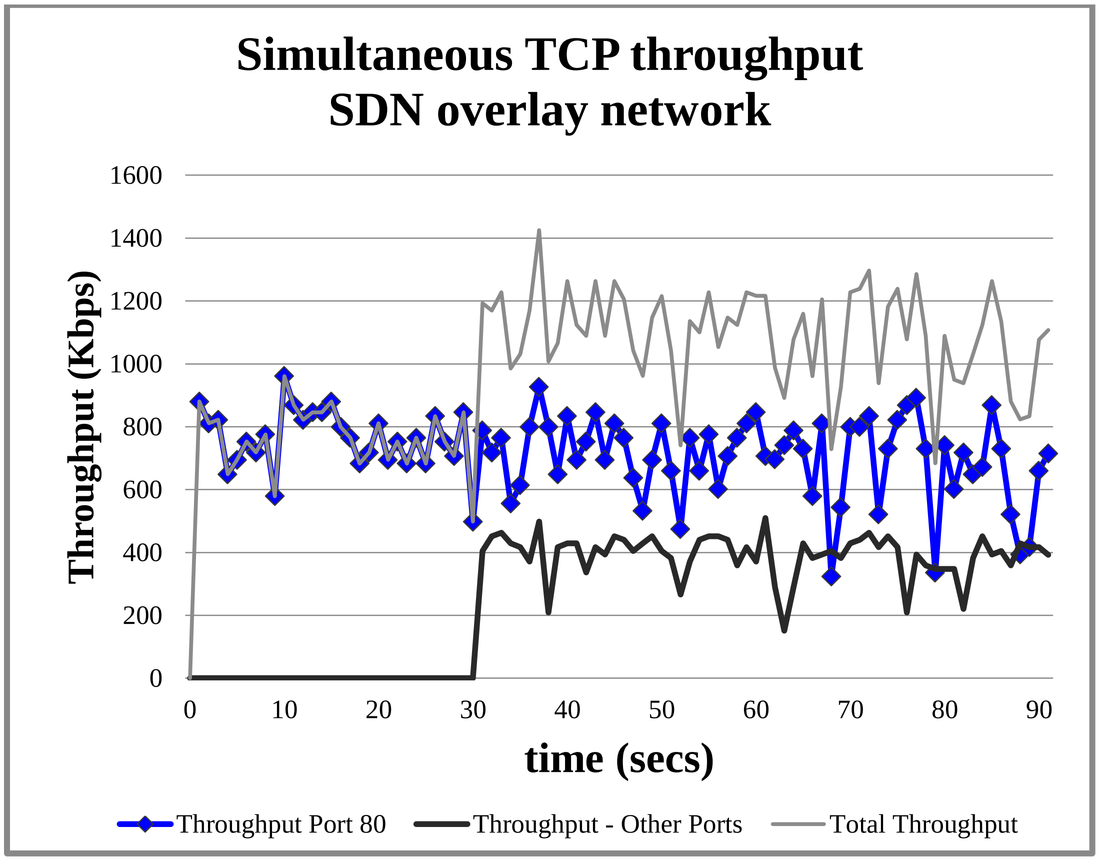

Figure 16. Simultaneous transmission in hybrid SDN network. We start by sending HTTP data on port 80 (1Mbps), after 30 seconds, we begin to transmitting data with other port number simultaneously. The overlay SDN enables simultaneous transmission of the two traffic classes making total throughput 1.5Mbps possible. This result is very similar to what was obtained for the full SDN topology in Fig. 12.

4.3 Comparing Legacy, Full-SDN and SDN-Overlay Network Throughput

In this section, we provide a comparison of the performance for HTTP traffic on legacy network, full SDN, and overlay SDN. The graph presented in Fig. 17. compares the UDP throughput for the three networks, while Table 1. gives a comparison the average values of TCP throughput, UDP throughput, jitter, percentage of dropped packets and throughput for simultaneous transmission of two flows.

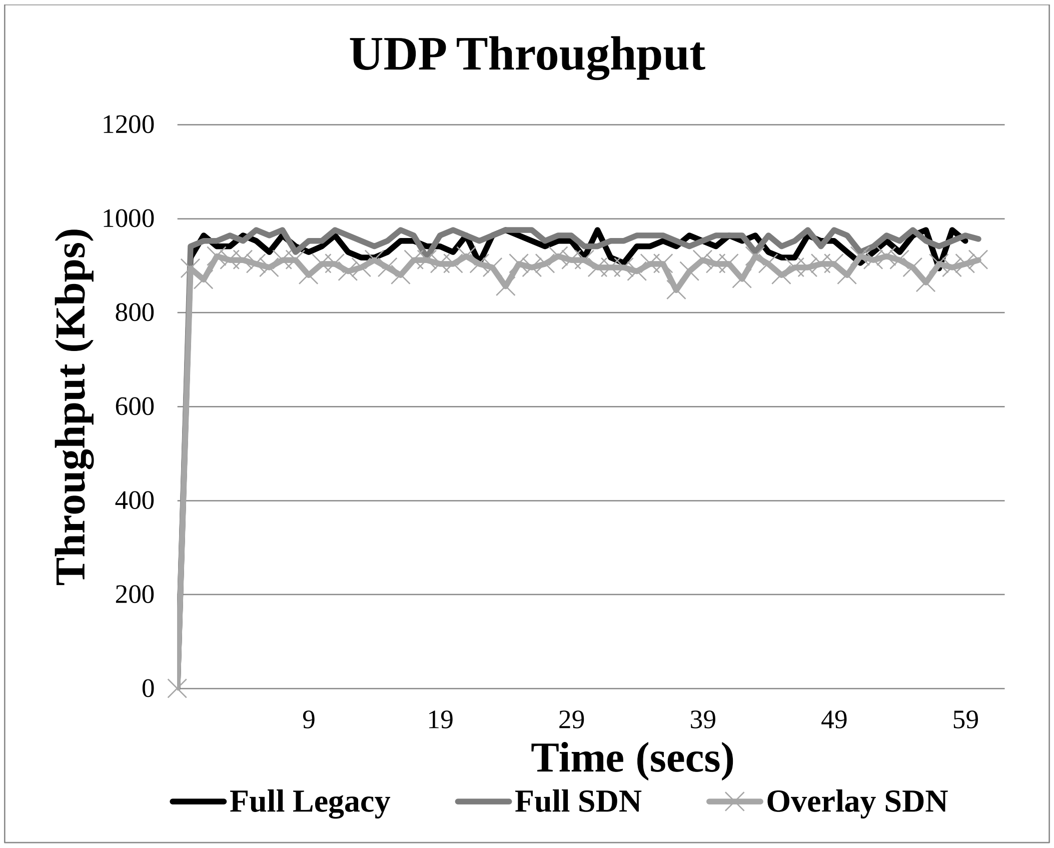

Figure 17. Throughput comparison of SDN overlay with full SDN and full legacy network.

Table 1: Comparison of SDN overlay network performance with full legacy and full SDN networks.

	Full legacy network	Full SDN network	SDN overlay network
Average TCP throughput (Kbps)	738.37	881.93	758.70
Average UDP throughput (Kbps)	943.23	955.33	898.93
Average jitter (seconds)	1.64	1.57	1.83
Average dropped packets (%)	9.94	8.87	14.04
Average throughput for simultaneous flows transmission (Mbps)	734.00	1288.53	1056.60

Fig. 17. and Table 1. show that the performance of the proposed SDN overlay network is found to be comparable with that of full SDN and full legacy networks. The TCP throughput and UDP throughput for the overlay network are compatible with the full SDN and full legacy network. The jitter, the percentage of dropped packets and the throughput for simultaneous transmission also show comparable behavior. The overlay network seems to have a slightly smaller UDP throughput. The very little difference could be attributed to additional processing required to perform the overlay functions, and can be ignored since the overlay system is able to take full advantage of the benefits of SDN to easily implement specialized services like traffic engineering and load-balancing without replacing all devices with SDN switches.

CHAPTER 5

CONCLUSION

5.1 Conclusion

In the previous chapters, we carried out a study of SDN and overlay technologies. We examined existing research in hybrid SDN and gave descriptions of some Hybrid SDN architectures that have been proposed in literature. We presented an SDN overlay architecture that combines SDN and OSPF legacy routing with VXLAN overlay technology. We implemented a prototype of the SDN overlay, and we carried out experiments to determine the performance of the network. We also compared its performance with similar full legacy and full SDN networks.

In our experiments, we observe that the performance of the SDN overlay network is comparable to the performance of the full SDN and the full legacy network. We also observe that we were able to implement all the functionalities of a full SDN network on the SDN overlay.

Therefore, we conclude that by deploying SDN overlays, it is possible to have full SDN functionality on a network without replacing all legacy devices with SDN-enabled switches. This result helps to save cost for organizations that seek to migrate to SDN

5.2 Future Directions

We intend to carry out further research to improve on SDN overlays by considering other overlay technologies besides VXLAN. We will create experiments on more complex topologies and we will also consider specific applications of SDN overlays to wireless networks. Methods investigated, and finally, we will develop a means of automating the configuration of SDN overlays on networks.

REFERENCES

[1] Statista inc., "Number of worldwide internet users from 2000 to 2014 (in millions)," 2014. [Online]. Available: http://www.statista.com/statistics/273018/number-of-internet-users-worldwide/. [Accessed 2014].

[2] Wikipedia, 2014. [Online]. Available: http://en.wikipedia.org/wiki/Internet_traffic..

[3] Cisco, 2014. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html.

[4] A. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira and A. Valadarsky, "VeriCon: towards verifying controller programs in software-defined networks", Proceedings of ACM SIGPLAN Conference, 2014.

[5] F. Bari, A. R. Roy, S. R. Chowdhury, Q Zhang, M. F. Zhani, R. Ahmed, and R. Boutaba, "Dynamic Controller Provisioning in Software-Defined Networks", 9th CNSM and Workshops, 2013.

[6] W. Braun, M.Menth, "Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices". Future Internet, 2014.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker and J. Turner, "Openflow: enabling innovation in campus networks," in *ACM SIGCOMM Comp. Commun. Rev. 38 (2)*, 2008.

[8] D. Levin, M. Canini, S. Schmid and A. Feldman, "Panopticon: Reaping the Benefits of Partial SDN Deployment in Enterprise Networks," in *USENIX ATC*, 2014.

[9] Open Networking foundation, "Software-Defined Networking: The New Norm for Networks," 2012. [Online].

[10] P. Wang, J. Luo, W. Li, Y. Qu, "NCPP: A Network Control Programmale Platform of Trustworthy Controllable Network," IEEE International Conference on Distributed Computing Systems Workshops, 2009.

[11] Cisco Systems inc., "OpFlex: An Open Policy Protocol," 2014. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-731302.pdf. [Accessed 2014].

REFERENCES (continued)

[12] Open vSwitch, "Production Quality, Multilayer Open Virtual Switch," 2014. [Online]. Available: http://openvswitch.org/. [Accessed 2014].

[13] CPqD, "CPqD OpenFlow 1.3 Software Switch," 2014. [Online]. Available: http://cpqd.github.io/ofsoftswitch13/. [Accessed 2014].

[14] Big Switch, "Indigo Project," 2014. [Online]. Available: http://www.projectfloodlight.org/indigo/. [Accessed 2014].

[15] B. A. A. Nunes, M. Mendonca, X. Nguyen, K. Obraczka and T. Turletti, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks," *HAL-INRA 00825087, version 5,* 2014.

[16] Nicira Networks, Inc., "Evolution Of The Ethane Architecture", Technical Report AFRL-RI-RS-TR-2009-41 For The Air Force Research Laboratory, Information directorate new york, 2009.

[17] Open Networking Foundation, "Openflow switch specification v1.4," 2013. [Online].

[18] M. Jammala, T. Singh, A. Shami, R. Asalb and Y. Lic, "Software-Defined Networking: State of the Art and Research Challenges," *Elsevier's Journal of Computer Networks,* 2013.

[19] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown and S. Shenker, "NOX: towards an operating system for networks," in *ACM SIGCOMM*, 2008.

[20] noxrepo.org, "About POX," 2012. [Online]. Available: http://www.noxrepo.org/pox/about-pox/. [Accessed 2014].

[21] J. Qadir1, N. Ahmed and N Ahad, "Building programmable wireless networks: an architectural survey", URASIP Journal on Wireless Communications and Networking, 2014.

[22] A. Wang, M. Iyer, R. Dutta, G.N. Rouskas and I.Baldine, "Network Virtualization: Technologies, Perspectives, and Frontiers" IEEE Journal of Lightwave Technology, vol.31, no.4, pp.523,537, Feb.15, 2013

[23] S. Vissicchio, L. Vanbever and O. Bonaventure, "Networks, Opportunities and Research Challenges of Hybrid Software-Defined Networking," *Computer communication Review,* 2014.

REFERENCES (continued)

[24] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart and A. Vahdat, “B4: Experience with a Globally-Deployed Software Defined WAN,” in *SIGCOMM*, 2013.

[25] S. Salsano, P. L. Ventre, L. Prete, G. Siracusano, M. Gerola and E. Salvadori, "OSHI - Open Source Hybrid IP/SDN networking (and its emulation on Mininet and on distributed SDN testbeds)," in *EWSDN*, Budapest, 2014.

[26] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark and E. Katz-Bassett, “SDX: a software-defined internet exchange,” in *SIGCOMM*, 2014.

[27] B. Lantz, B. Brandon Heller and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *HotNets 2010,* 2010.

[28] Flux Research Group, University of Utah, "Emulab," 2014. [Online]. Available: http://www.flux.utah.edu/project/emulab.

[29] K. Ishiguro, "Quagga - a Routing Software Package for TCP/IP Networks," 2013. [online] http://www.nongnu.org/quagga/docs/quagga.pdf. [Accessed 2014]

[30] "Open vSwitch.," 2013. [Online] Available: http://www.openvswitch.org/. [Accessed 2014].

[31] A.Tirumala, F. Qin, J. Dugan, J. Ferguson, K. Gibbs, "Iperf: The TCP/UDP bandwidth measurement tool" [online] - http://dast. nlanr. net/Projects, 2005

[32] The Network Simulator - ns2, http://www.isi.edu/nsnam/ns/..[Accessed 2014]

[33] C. N. Chuah, "Overlay Networks - Indirection & Virtualization DIMACS Tutorial on Algorithms for Next Generation Networks," [Online]. Available: http://www.ece.ucdavis.edu/rubinet .

[34] N. Feamster, J. Rexford and E. Zegura, "Road to SDN," in *ACM Computer Communications Review,* 2014.

[35] M. Casado, N. McKeown, M. J. Freedman, J. Pettit, J. Luo and S. Shenker, "ETHANE: Taking Control of the Enterprise," in *ACM SIGCOMM,* 2007.

REFERENCES (continued)

[36] A. Köpsel, H.Woesner, "OFELIA – Pan-European Test Facility for OpenFlow Experimentation " pg 311 - 312, Towards a service based Internet, *Springer*, 2010.

[37] N. Foster, M. J. Freedman, A. Guha, R. Harrison, N. P. Katta, C. Monsanto, J. Reich, M. Reitblatt, J. Rexford, C. Schlesinger, A. Story and D. Walker, "Languages for Software-Defined Networks," *IEEE Communication Magazine*, February 2013.

[38] A. Voellmy, H. Kim and N. Feamster, "Procera: A Language for High-Level Reactive Network Control," in *HotSDN 2012*, 2012.

[39] J. Luo, J. Pettit, M. Casado, J. Lockwood and N. McKeown, "Prototyping Fast, Simple, Secure Switches for ETHANE," in *IEEE Symposium - High-Performance Interconnects, HOTI 2007*, 2007.

[40] M. Casado, N. McKeown, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh and S. Shenker, "SANE: A Protection Architecture for Enterprise Networks," in *USENIX Security Symposium*, 2006.

[41] Flux Research Group, University of Utah, "Phantomnet Testbed," 2014. [Online]. Available: Phantomnet.org.

APPENDICES

APPENDIX A

Codes: Full Legacy Network

Topology File – Legacy Network

```
set ns [new Simulator]
source tb_compat.tcl
set h1 [$ns node]
set h2 [$ns node]
set r1 [$ns node]
set r2 [$ns node]
set r3 [$ns node]
set r4 [$ns node]
set linkh1r1 [$ns duplex-link $h1 $r1 100Mb 50ms DropTail]
tb-set-link-loss $linkh1r1 0.01
set linkr1r2 [$ns duplex-link $r1 $r2 10Mb 50ms DropTail]
tb-set-link-loss $linkr1r2 0.01
set linkr1r3 [$ns duplex-link $r1 $r3 100Mb 50ms DropTail]
tb-set-link-loss $linkr1r3 0.01
set linkr2r4 [$ns duplex-link $r2 $r4 10Mb 50ms DropTail]
tb-set-link-loss $linkr2r4 0.01
set linkr3r4 [$ns duplex-link $r3 $r4 100Mb 50ms DropTail]
tb-set-link-loss $linkr3r4 0.01
set linkr4h2 [$ns duplex-link $r4 $h2 100Mb 50ms DropTail]
tb-set-link-loss $linkr4h2 0.01
tb-set-node-os $h1 UBUNTU12-64-STD
tb-set-node-os $h2 UBUNTU12-64-STD
tb-set-node-os $r1 UBUNTU12-64-STD
tb-set-node-os $r2 UBUNTU12-64-STD
tb-set-node-os $r3 UBUNTU12-64-STD
tb-set-node-os $r4 UBUNTU12-64-STD
$ns rtproto Manual
# Go!
$ns run
```

APPENDIX B

Codes: Full SDN Network

Topology File – Full SDN Network

```
set ns [new Simulator]
source tb_compat.tcl
set h1 [$ns node]
set h2 [$ns node]
set ofs1 [$ns node]
set ofs2 [$ns node]
set ofs3 [$ns node]
set ofs4 [$ns node]
set c1 [$ns node]
set linkh1ofs1 [$ns duplex-link $h1 $ofs1 2000Kb 0ms DropTail]
tb-set-link-loss $linkh1ofs1 0.01
set linkofs1ofs3 [$ns duplex-link $ofs1 $ofs3 500Kb 0ms DropTail]
tb-set-link-loss $linkofs1ofs3 0.01
set linkofs1ofs2 [$ns duplex-link $ofs1 $ofs2 1000Kb 0ms DropTail]
tb-set-link-loss $linkofs1ofs2 0.01
set linkofs2ofs4 [$ns duplex-link $ofs2 $ofs4 1000Kb 0ms DropTail]
tb-set-link-loss $linkofs2ofs4 0.01
set linkofs3ofs4 [$ns duplex-link $ofs3 $ofs4 500Kb 0ms DropTail]
tb-set-link-loss $linkofs3ofs4 0.01
set linkofs4h2 [$ns duplex-link $ofs4 $h2 2000kb 0ms DropTail]
tb-set-link-loss $linkofs4h2 0.01
tb-set-node-os $h1 UBUNTU12-64-STD
tb-set-node-os $h2 UBUNTU12-64-STD
tb-set-node-os $ofs1 "PhantomNet/SMORE-SDN"
tb-set-node-os $ofs2 "PhantomNet/SMORE-SDN"
tb-set-node-os $ofs3 "PhantomNet/SMORE-SDN"
tb-set-node-os $ofs4 "PhantomNet/SMORE-SDN"
tb-set-node-os $c1 UBUNTU12-64-STD
# Go!
$ns rtproto Manual
$ns run
```

APPENDIX B (continued)

Controller code – Full SDN Network

```
#excerpts of ControllerFullSDN.py
#controller codes for the Full SDN network

import pox.openflow.libopenflow_01 as of
import pox.openflow.discovery
from pox.core import core
from pox.lib.addresses import EthAddr, EthAddr
from pox.lib.util import dpidToStr
...
...
class FullSDN(EventMixin):
    def __init__(self):
        ...
        ...
        #define switch ids and host addresses
        ofs1_dpid = '00-04-23-b7-19-70'
        ofs2_dpid = '00-04-23-b7-1c-e2'
        ofs3_dpid = '00-04-23-b7-3e-a2'
        ofs4_dpid = '00-04-23-b7-19-02'
        h1_mac = '00:04:23:b1:f0:ac'
        h2_mac = '00:04:23:b7:1a:f8'
        ...
        ...
        #A dictionary that shows gives the output interface
        #for each flow, at each of-switch based on the
        #tcp port number
        self.portmap80 = {
          (ofs1_dpid, EthAddr (h1_mac)): 5,
          (ofs1_dpid, EthAddr (h2_mac)): 4,
          (ofs2_dpid, EthAddr (h1_mac)): 4,
          (ofs2_dpid, EthAddr (h2_mac)): 5,
          (ofs4_dpid, EthAddr (h1_mac)): 4,
          (ofs4_dpid, EthAddr (h2_mac)): 3}
        self.portmapOther = {
          (ofs1_dpid, EthAddr (h1_mac)): 5,
          (ofs1_dpid, EthAddr (h2_mac)): 2,
          (ofs3_dpid, EthAddr (h1_mac)): 3,
          (ofs3_dpid, EthAddr (h2_mac)): 2,
          (ofs4_dpid, EthAddr (h1_mac)): 2,
          (ofs4_dpid, EthAddr (h2_mac)): 3}
    ...
    ...
    def _handle_ConnectionUp(self, event):
        dpid = dpidToStr(event.dpid)
        log.debug("Switch %s has come up.", dpid)
    ...
    ...
```

APPENDIX B (continued)

```
def _handle_PacketIn (self, event):
        #Function that defines how incoming packets are handled
        mypkt = event.parsed

        def create_fwdrule(event,mypkt,outport):
            #define function to install forwarding rules
            rule = of.ofp_flow_mod()
            rule.match = of.ofp_match.from_packet(packet, event.port)
            rule.actions.append(of.ofp_action_output(port = outport))
            rule.data = event.ofp
            rule.in_port = event.port
            event.connection.send(rule)

        def forwardpkt (fwd_msg = None):
            this_dpid = dpid_to_str(event.dpid)
            if packet.dst.is_multicast:
                #flood
                return
            else:
                try:
                    tcppp = 0
                    if event.parsed.find('tcp'):
                        tcppp = event.parsed.find('tcp')
                    elif event.parsed.find('udp'):
                        tcppp = event.parsed.find('udp')
                    if tcppp.dstport == 80:
                        key_port = 80
                        log.debug("key tcport is   "+str(key_port)+"   bloc")
                        outport = self.portmap80 [(dpid_to_str (
                        event.dpid), packet.dst)]
                    else:
                        key_port = 27 #dummy port value for non TCP
                        log.debug("key tcport is "+str(key_port))
                        outport = self.portmapOther[(dpid_to_str
                        (event.dpid), packet.dst)]
                        log.debug("new tcp pkt @ swh:"+dpid_to_str
                        (event.dpid)+" tcpt:"+str(key_port)+ "
                        dst:"+str(packet.dst)+" ==> _port="+str(outport))
                    create_fwdrule(event,packet,outport)
                    log.debug("flow matched & installed")

                except AttributeError:
                    log.debug("no TCP port, flood")
                    create_fwdrule(event,packet,of.OFPP_FLOOD)
        fowardpkt()
...
...
def launch():
    pox.openflow.discovery.launch()
    core.registerNew(VideoSlice)
```

APPENDIX C

Codes: Hybrid SDN Network

<u>Topology File – Hybrid SDN</u>

```
set ns [new Simulator]
source tb_compat.tcl
set h1 [$ns node]
set h2 [$ns node]
set ofs1 [$ns node]
set r2 [$ns node]
set r3 [$ns node]
set ofs4 [$ns node]
set c1 [$ns node]
set linkh1ofs1 [$ns duplex-link $h1 $ofs1 2000Kb 0ms DropTail]
tb-set-link-loss $linkh1ofs1 0.01
set linkofs1r2 [$ns duplex-link $ofs1 $r2 1000Kb 0ms DropTail]
tb-set-link-loss $linkofs1r2 0.01
set linkofs1r3 [$ns duplex-link $ofs1 $r3 500Kb 0ms DropTail]
tb-set-link-loss $linkofs1r3 0.01
set linkr2ofs4 [$ns duplex-link $r2 $ofs4 1000Kb 0ms DropTail]
tb-set-link-loss $linkr2ofs4 0.01
set linkr3ofs4 [$ns duplex-link $r3 $ofs4 500Kb 0ms DropTail]
tb-set-link-loss $linkr3ofs4 0.01
set linkofs4h2 [$ns duplex-link $ofs4 $h2 2000Kb 0ms DropTail]
tb-set-link-loss $linkofs4h2 0.01
tb-set-node-os $h1 UBUNTU12-64-STD
tb-set-node-os $h2 UBUNTU12-64-STD
tb-set-node-os $ofs1 "PhantomNet/SMORE-SDN"
tb-set-node-os $r2  UBUNTU12-64-STD
tb-set-node-os $r3  UBUNTU12-64-STD
tb-set-node-os $ofs4  "PhantomNet/SMORE-SDN"
tb-set-node-os $c1  UBUNTU12-64-STD
# Go!
$ns rtproto Manual
$ns run
```

APPENDIX C (continued)

Controller code – Hybrid SDN

```
#excerpts of controllerHybridSDN.py
#controller code for the Hybrid SDN network

import pox.openflow.libopenflow_01 as of
import pox.openflow.discovery
from pox.core import core
from pox.lib.addresses import EthAddr, EthAddr
from pox.lib.util import dpidToStr
from collections import defaultdict
...
...
class HybridSDN(EventMixin):
    def __init__(self):
        ...
        ...
        #define switch ids and host addresses
        ofs1_dpid = '00-04-23-b7-1e-1e'
        ofs4_dpid = '00-04-23-b7-26-ae'
        h1_mac = '00:04:23:b7:19:62'
        h2_mac = '00:04:23:b7:1b:49'
        ...
        ...
        #A dictionary that shows gives the output interface
        #for each flow, at each of-switch based on the
        #tcp port number
        self.portmap80 = {
          (ofs1_dpid, EthAddr (h1_mac)): 2,
          (ofs1_dpid, EthAddr (h2_mac)): 11,
          (ofs4_dpid, EthAddr (h1_mac)): 11,
          (ofs4_dpid, EthAddr (h2_mac)): 4}
        self.portmapOther = {
          (ofs1_dpid, EthAddr (h1_mac)): 2,
          (ofs1_dpid, EthAddr (h2_mac)): 12,
          (ofs4_dpid, EthAddr (h1_mac)): 12,
          (ofs4_dpid, EthAddr (h2_mac)): 4}
    ...
    ...
    def _handle_ConnectionUp(self, event):
        dpid = dpidToStr(event.dpid)
        log.debug("Switch %s has come up.", dpid)
    ...
    ...
    def _handle_PacketIn (self, event):
        Function that defines how incoming packets are handled
        mypkt = event.parsed
```

APPENDIX C (continued)

```
        def create_fwdrule(event,mypkt,outport):
            #define function to install forwarding rules
            rule = of.ofp_flow_mod()
            rule.match = of.ofp_match.from_packet(packet, event.port)
            rule.actions.append(of.ofp_action_output(port = outport))
            rule.data = event.ofp
            rule.in_port = event.port
            event.connection.send(rule)


        def forwardpkt (fwd_msg = None):
            this_dpid = dpid_to_str(event.dpid)
            if packet.dst.is_multicast:
                flood()
                return
            else:
                try:
                    tcppp = 0
                    if event.parsed.find('tcp'):
                        tcppp = event.parsed.find('tcp')
                    elif event.parsed.find('udp'):
                        tcppp = event.parsed.find('udp')
                    if tcppp.dstport == 80:
                        key_port = 80
                        log.debug("key tcport is "+str(key_port)+" bloc")
                        outport = self.portmap80[ (dpid_to_str (event.dpid),
                        packet.dst)]
                    else:
                        key_port = 27 #dummy port non TCP traffic
                        log.debug("key tcport is "+str(key_port))
                        outport = self.portmapOther[(dpid_to_str (event.dpid),
                        packet.dst)]
                        log.debug("new tcp pkt @ swh:"+dpid_to_str (event.dpid)+"
                        tcpt:"+str(key_port)+" dst:"+str(packet.dst)+" ==>
                        out_port="+str(outport))
                    create_fwdrule(event,packet,outport)
                    log.debug("flow matched & installed")

                except AttributeError:
                    log.debug("no TCP port, flood")
                    create_fwdrule(event,packet,of.OFPP_FLOOD)
        fowardpkt()
...
...
def launch():
    pox.openflow.discovery.launch()
    pox.openflow.spanning_tree.launch()
    core.registerNew(Tcpportslicing)
```

VITA

NAME	Oluwamayowa Ade Adeleke
EDUCATION	B.Tech., Electronic and Electrical Engineering, Ladoke Akintola University of Technology, Ogbomoso, Nigeria, 2010
	M.S., Electrical and Computer Engineering, University of Illinois at Chicago, Chicago, Illinois, 2015
TEACHING AND PROFESSIONAL EXPERIENCE:	Graduate Teaching Assistant, Electrical and Computer Engineering Department, University of Illinois 2014 – 2014
	Transmission Planning Engineer (Trainee), Alcatel Lucent Nigeria 2011 – 2012
	Intern – Mobile Architecture and Design Department, Alcatel Lucent Nigeria 2009-2009
HONORS:	Postgraduate Scholarship Awards, National Information Technology development Agency, Nigeria, 2013/2014
	Undergraduate University Scholarship, Nigerian National Petroleum Corporation & ExxonMobil Nigeria, 2006-2010
PROFESSIONAL MEMBERSHIP:	Institute of Electrical and Electronics Engineers
	Association for Computing Machinery