

# Use of Mininet as Research TestBed

*Thesis submitted in partial fulfillment of requirements*

*For the degree of*

**Master of Computer Application**

of

Computer Science and Engineering Department

of

Jadavpur University

by

**Shubham Gupta**

**Regn. No. - 149893 of 2019-2020**

**Exam Roll No. - MCA226032**

*under the supervision of*

**Mridul Sankar Barik**

Assistant Professor

Department of Computer Science and Engineering

JADAVPUR UNIVERSITY

Kolkata, West Bengal, India

2022

## Certificate from the Supervisor

This is to certify that the work embodied in this thesis entitled "**Use of Mininet as Research TestBed**" has been satisfactorily completed by **Shubham Gupta** (Registration Number 149893 of 2019–20; Class Roll No. 001910503033; Examination Roll No. *MCA226032*). It is a bona-fide piece of work carried out under my supervision and guidance at Jadavpur University, Kolkata for partial fulfilment of the requirements for the awarding of the **Master of Computer Application** degree of the Department of Computer Science and Engineering, Faculty of Engineering and Technology, Jadavpur University, during the academic year 2021 – 22.

---

**Mridul Sankar Barik,**  
Assistant Professor,  
Department of Computer Science and Engineering,  
Jadavpur University.  
(Supervisor)

Forwarded By:

---

**Prof. Anupam Sinha,**  
Head,  
Department of Computer Science and Engineering,  
Jadavpur University.

---

**Prof. Chandan Mazumdar,**  
Dean,  
Faculty of Engineering & Technology,  
Jadavpur University.

Department of Computer Science and Engineering  
Faculty of Engineering And Technology  
Jadavpur University, Kolkata - 700 032

## Certificate of Approval

This is to certify that the thesis entitled ”**Use of Mininet as Research TestBed**” is a bona-fide record of work carried out by **Shubham Gupta** (Registration Number 149893 of 2019 – 20; Class Roll No. 001910503033; Examination Roll No. *MCA226032*) in partial fulfilment of the requirements for the award of the degree of **Master of Computer Application** in the **Department of Computer Science and Engineering, Jadavpur University**, during the period of January 2022 to June 2022. It is understood that by this approval, the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein but approve the thesis only for the purpose of which it has been submitted.

**Examiners:**

---

(Signature of The Examiner)

---

(Signature of The Supervisor)

Department of Computer Science and Engineering  
Faculty of Engineering And Technology  
Jadavpur University, Kolkata - 700 032

## **Declaration of Originality and Compliance of Academic Ethics**

I hereby declare that the thesis entitled "**Use of Mininet as Research TestBed**" contains literature survey and original research work by the undersigned candidate, as a part of his degree of **Master of Computer Application** in the **Department of Computer Science and Engineering, Jadavpur University**. All information have been obtained and presented in accordance with academic rules and ethical conduct.

I also declare that, as required by these rules and conduct, I have fully cited and referenced all materials and results that are not original to this work.

**Name:** Shubham Gupta

**Examination Roll No.:** MCA226032

**Registration No.:** 149893 of 2019 – 20

**Thesis Title:** Use of Mininet as Research TestBed

**Signature of the Candidate:**

## ACKNOWLEDGEMENT

I am pleased to express my gratitude and regards towards my Project Guide **Shri Mridul Sankar Barik**, Assistant Professor, Department of Computer Science and Engineering, Jadavpur University, without whose valuable guidance, inspiration and attention towards me, pursuing my project would have been impossible.

Last but not the least, I express my regards towards my friends and family for bearing with me and for being a source of constant motivation during the entire term of the work.

---

**Shubham Gupta**

MCA Final Year

Exam Roll No. - MCA226032

Regn. No. - 149893 of 2019 – 20

Department of Computer Science and Engineering,  
Jadavpur University.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.2	Research Statement . . . . .	1
1.3	Contribution of the Thesis . . . . .	1
1.4	Outline of the Thesis . . . . .	2
<b>2</b>	<b>Background on Software Defined Networking</b>	<b>3</b>
2.1	Introduction . . . . .	3
2.2	Software-Defined Networking . . . . .	3
2.3	Evolution of SDN . . . . .	5
2.3.1	Active Network . . . . .	5
2.3.2	Separation of Concerns . . . . .	5
2.3.3	OpenFlow . . . . .	6
2.3.4	Current State of SDN . . . . .	6
2.4	SDN Architecture . . . . .	8
<b>3</b>	<b>Background on Mininet</b>	<b>10</b>
3.1	Introduction . . . . .	10
3.2	Mininet Commands . . . . .	11
3.3	Mininet Topologies . . . . .	12
3.3.1	Minimal . . . . .	13
3.3.2	Single . . . . .	14
3.3.3	Reversed . . . . .	15
3.3.4	Linear . . . . .	16
3.3.5	Tree . . . . .	17
3.4	Custom Topologies . . . . .	18
<b>4</b>	<b>Background on SDN Controllers</b>	<b>20</b>
4.1	NOX . . . . .	21
4.2	POX . . . . .	21
4.3	Ryu . . . . .	23
4.4	Floodlight . . . . .	23
4.5	ONOS . . . . .	23

<b>5</b>	<b>Mininet as SDN TestBed</b>	<b>25</b>
5.1	Testbed Design . . . . .	25
5.2	Distributed SDN . . . . .	29
<b>6</b>	<b>Mininet as 5G TestBed</b>	<b>30</b>
6.1	Contribution of SDN and NFV in 5G . . . . .	30
6.2	Using Mininet for Testing . . . . .	31
6.2.1	vRouter: Virtual Router . . . . .	31
6.2.2	IPRAN: Internet Protocol Radio Access Network . . . . .	32
6.2.3	MCORD: Mobile-CORD . . . . .	32
<b>7</b>	<b>Conclusion and Future Work</b>	<b>34</b>
7.1	Conclusion . . . . .	34
7.2	Future Work . . . . .	35
<b>A</b>	<b>Mininet Experiments</b>	<b>36</b>
A.1	Experiment 1 . . . . .	36
A.2	Experiment 2 . . . . .	51
A.2.1	Custom Topology Script for mn . . . . .	51
A.2.2	Standalone Python Script . . . . .	54

# List of Figures

2.1	Three-tier OS Model . . . . .	8
2.2	Three-tier SDN Model . . . . .	8
3.1	Emulating a Network Consisting of Two Hosts Connected to a Switch and a Controller	10
3.2	Minimal Topology . . . . .	13
3.3	Single Topology . . . . .	14
3.4	Reversed Topology . . . . .	15
3.5	Linear Topology . . . . .	16
3.6	Tree Topology . . . . .	17
3.7	Custom Topology . . . . .	18
5.1	Physical Architecture . . . . .	26
6.1	vRouter . . . . .	31
6.2	IP RAN . . . . .	32
A.1	Running of_tutorial.py provided by POX . . . . .	37
A.2	tcpdump on terminal to Host h2 . . . . .	39
A.3	tcpdump on terminal to Host h3 . . . . .	39
A.4	Mininet connected to modified python module, running tests . . . . .	46
A.5	POX controller logging debug messages . . . . .	46
A.6	Flow entries found on switch S1 . . . . .	49
A.7	Much improved bandwidth . . . . .	49



## **Abstract**

As new network technologies evolve, the need for proper testing arises. Various platforms are available which serve as robust and secure testbeds for Software Defined-Networks (SDN). But these platforms often use additional hardware and/or heavyweight software which run resource intensive processes. Access to such platforms is quite expensive as well. This thesis discusses the potential of a lightweight open-source network emulation tool called Mininet, to be used as a testbed for research in SDNs and 5G networks. It discusses the benefits of using such a tool and also describes its major issues that cause inaccurate analysis.

# Chapter 1

## Introduction

### 1.1 Background

Networking technologies are advancing at a rapid pace. This requires novel ideas to evolve quickly in order to solve the problems at the earliest. There is no lack of talented individuals who come up with such solutions on a regular basis. But with the solutions, arise the need for testing. Without proper testing, you can not conclude whether or not a certain idea can be converted to a best fitting solution. There has been a lot of development around Software Defined-Networks [23] and Network Function Virtualization [27]. These technologies are widely used currently. But developing a testbed for such networks is expensive and complex, as it requires additional hardware and/or proprietary heavy-weight virtualization tools.

### 1.2 Research Statement

In the light of the current situation, we have tried to find whether an alternative testbed, comprising of lightweight virtualization tools, would lower the costs associated with testing. One such tool is Mininet. Mininet is not only a lightweight network emulator, but it is also simple and quite convenient to use. In this thesis, we try to find out the merits and demerits of using Mininet as a testbed for 5G and SDN in general. We will do a comparative study where we refer to the works of other researchers and draw conclusions from them.

### 1.3 Contribution of the Thesis

The thesis surveys Mininet - its features and looks into its ability to virtualize and manage networks. It also compares different SDN controllers with Mininet and also explores the use of Mininet for research in SDN and 5G.

## 1.4 Outline of the Thesis

This thesis is divided into seven chapters. Chapter 1 introduces the topic of the thesis. Chapter 2 gives a background on Software Defined Networking - it introduces the technology, describes its evolution, its architecture, the protocols it uses and discusses the need of SDN in the modern day. Chapter 3 offers an introduction to Mininet and its API - its uses, the way it works, and a few examples of implementation. Chapter 4 describes various Network Operating Systems that are currently used and compares them. Chapter 5 surveys the use of Mininet as an SDN TestBed and also briefly touches on distributed SDN. In Chapter 6, the thesis gives an idea of how 5G is implemented and how SDN and NFV combine to help 5G achieve its demands. It is followed by a survey of how Mininet would perform as a TestBed for 5G. Finally, Chapter 7 summarizes the main conclusions of the thesis and presents an outlook for future work.

## Chapter 2

# Background on Software Defined Networking

### 2.1 Introduction

A typical computer network is complex. It consists of a number of different kinds of networking devices. Apart from switches and routers, there are network middleboxes such as firewalls, load balancers and network address translators. A traditional layer-2 switch or router usually runs control software that is proprietary and closed. For configuring these devices, network administrators use the configuration interface provided to them. These interfaces vary from vendor to vendor and sometimes also vary in different devices from the same vendor. There are some network management tools which allow the administrators to manage different devices on the network from a central point. But they have their limitations, and often they focus only on particular protocols or mechanisms. As a result, operational complexity and operational costs are seen to rise.

Apart from vendor dependence, there are a number of limitations of the traditional network which make it impossible to meet current market requirements. Networking technology has been driven by sets of protocols which are architected to connect hosts over long distances, network link speeds and topologies. To meet market needs, many high performing, reliable and secure protocols are developed and evolved over time. But the problem is that these protocols are designed in isolation. That is, each of these protocols solve specific problems and as a result fail to provide a fundamental abstraction. This has resulted in the networks being more complex. This also hinders the ability of the network to scale. With addition of thousands of networking devices, the network complexity will only increase. Managing such networks would be a nightmare. Imagine implementing a network-wide policy. Hundreds of devices will have to be configured. Not only is this a tedious and time consuming task, but it potentially leads to inconsistencies as well.

### 2.2 Software-Defined Networking

Software-Defined Networking [23] (SDN) is a novel concept for the network engineering field. SDN is paving way for designing and managing networks in non-traditional ways. This newer architecture has two main defining characteristics. First is the separation of concerns, when it comes to the data

plane (which simply forwards network traffic) and the control plane (which can be thought of as the brain of the device - decides how to handle network traffic) of the networking devices. Second, the SDN architecture facilitates the consolidation of the control plane. It does so by allowing a single software to act as a central control software, which can control the data planes of multiple network devices. The software that controls these networking devices is generally called the SDN controller. There are many such popular controllers.

The primary reason for separating the planes, is abstraction. Typically an application developer does not need to know about the underlying hardware in order to build applications. The hardware layer is abstracted by the operating system. It is a similar case for networking applications. In traditional networking devices, the control and data planes are tightly coupled to each other. Hence a network application developer, would need to keep in mind the specificities of the various devices from different vendors. Having the ability to separate the control plane from the data forwarding plane, helps in achieving this abstraction. It also paves the way for building a common interface through which a central controller can control the forwarding behaviours of multiple switches, which was earlier not possible as all the switches had the control planes tightly coupled to the forwarding planes.

The communication between the SDN controller and these networking devices occur through means of a Secure-Shell (SSH) connection and well defined Application Programming Interfaces (APIs). The OpenFlow protocol is a common and well known example of such an API. The OpenFlow protocol defines the requirements and the standards for communication between an SDN controller and the agent network device. The agent devices in such cases are called OpenFlow switches, which basically refers to any OpenFlow compliant networking device. It may behave like a switch, a router, a network address translator (NAT) or any other networking middleboxes. The behaviour entirely depends on the instructions sent by the controller. The OpenFlow switches have tables which store these information as records known as flow rules or flow entries. These entries populate the table, which is called the flow-table. There is at least one flow-table on an OpenFlow switch and there can be many more. The flow entries store information which help the switch match network traffic and perform actions on the matching traffic. Actions such as forwarding, dropping or flooding are defined on each of such flow-rules.

Over time SDN has slowly grown into a significant technology in the networking world. Now, there are a number of commercial switches that support OpenFlow. Network application developers have used such platforms to build load balancers, dynamic access control and many more useful applications and network features. At present, there are SDN industry consortia like Open Networking Foundation and the Open Daylight initiative. Many top information-technology companies are a part of those.

The core ideas of SDN have been evolving over quite a long period. We can think of the old telephone networks where there was a separation of the control plane and the data plane. It was done to simplify the network, which in-turn helped manage the network better. Over the years the idea has transformed and it has evolved into the present day SDN. The SDN controllers give programmers the ability to innovate, which was not possible in the earlier closed networks of telephone services.

## 2.3 Evolution of SDN

### 2.3.1 Active Network

In the mid-1990s to the early 2000s we had seen the Internet bloom with applications that went beyond the mere file transfers and emails over the wire. The diverse applications and the increase in the use by the general users drove the researchers to test and deploy ideas for creating better network services. The researchers would first design and test the new network protocols in their own personal labs. They would approach it by opening up the network control, based on the idea of re-programming a standalone computer. A conventional network can not be programmed. A newer approach was taken called *active networking* [19] where the network control was done through a programming interface that exposed resources on individual network devices and supported the construction of custom functionality to apply to a subset of packets passing through the node. Traditional networks provide a transport mechanism to transfer bits from one end of the system to another, with minimal computation. In contrast, the active networks not only allow the network nodes to perform computations on the data but also allow their users to inject customized programs into the nodes of the network, that may modify, store or redirect the user data flowing through the network. Such programmable networks opened many new doors for possible applications that were unimaginable with traditional data networks. The researchers' community working on active networking, pursued two programming models:

- capsule model: here the programs to execute were carried in-band in data packets
- programmable router/switch model: here the programs to execute were established by out-of-band mechanisms

The cost reduction of computing was the main driving force which encouraged active networking. Active networking research projects made use of the advances in rapid code compilation. Advances were made in programming languages such as Java. Newer concepts such as virtual machine technology also came into play. Also, there was a strong pull from the market for active networking. The demands and requirements were similar to ones we see in modern SDN.

### 2.3.2 Separation of Concerns

It was the early 2000s, when network operators had to seek better approaches to some network-management functions such as traffic engineering. The increase in traffic volumes and the need for network reliability and performance led to these approaches. Using only traditional routing protocols, it would not have been possible to optimize these functions. As a result, research was done to find pragmatic and short term approaches that were standard-driven or deployable using the existing protocols.

The conventional routers and switches had the control plane and the data plane tightly coupled. This caused many network-management tasks, such as debugging configuration problems and controlling routing behaviour, quite challenging. To address these problems, efforts to separate the control and data planes began to emerge.

Equipment vendors started implementing packet-forwarding logic directly in hardware, separate from the control plane software. In addition to that, Internet Service Providers (ISPs) were having trouble to manage the ever increasing size of their networks and the demands for more reliability. These trends led to two inventions: (i) open interface between the data and control planes and (ii)

logically centralized control of the network. These two inventions are the main defining feature of the modern SDN.

### 2.3.3 OpenFlow

In mid 2000s, once the separation of the control and data planes was successful, there was a need to build a common and open API for the north-bound interface, i.e, the interface between the controller and the agent devices. At that point OpenFlow came to being. OpenFlow [10] introduced a balance between a fully programmable network and it's pragmatic approach. The OpenFlow API quickly evolved with the design of controller platforms such as NOX, which allowed the creation of control applications.

The OpenFlow compliant switch [11] has a table which contains packet-handling rules, which are commonly referred to as flow-rules or flow-entries. Each rule is composed of a pattern (which matches the bits in the packet headers), an action list (indicating actions such as forward, drop, flood, send packet to controller), few counters (to track the packets) and priority values. On receiving an incoming packet, the OpenFlow switch matches the packet with the flow-entries and the entry with the highest priority is chosen. The actions associated with that entry follow and the counters are incremented.

OpenFlow is only one of the first protocols to have gained so much popularity in the SDN world. But it is a key component as it started the networking software revolution. It defined a programmable and dynamic network protocol which could help manage traffic among switches and routers independent of the vendor that made the underlying hardware.

### 2.3.4 Current State of SDN

Since its inception, SDN has evolved into a reputable networking technology that is offered by top networking vendors such as Juniper, Cisco, VMware and many others. It has been embraced and widely used by cloud providers, with Google, Facebook, Microsoft being the most public about its adoption. While most of their platforms and solutions are proprietary, they have open sourced some individual components in an effort to catalyze a wider adoption. Also, network operators like AT&T, DT, NTT and Comcast publicly talk about their plans to use SDN-based solutions; especially in their access networks. The Open Networking Foundation develops many open-source SDN tools and technologies as well. Brad Casemore, IDC research vice president, Data Center Networks said in an interview [16] : “We’re now at a point where SDN is better understood, where its use cases and value propositions are familiar to most datacenter network buyers and where a growing number of enterprises are finding that SDN offerings offer practical benefits,” Casemore said. “With SDN growth and the shift toward software-based network automation, the network is regaining lost ground and moving into better alignment with a wave of new application workloads that are driving meaningful business outcomes.”

A number of trends have played into the development of the central idea behind SDN. Moving data center function to the edge, distributing computing power to remote sites, supporting Internet of Things environments and adopting cloud computing - all of these tasks can be done easily and in an cost effective manner through a properly configured SDN environment.

The first widely adopted case for SDN was network virtualization. Virtual networks, including both Virtual Private Networks (VPNs) and Virtual Local Area Networks (VLANs), have been a part of the Internet for a long time. VLANs have historically proven useful within enterprises, where

they are used to isolate different organizational groups, such as departments or labs, giving each of them the appearance of having their own private LAN. However, these early forms of virtualization were quite limited in scope and lacked many of the advantages of SDN. We can think of them as virtualizing the address space of a network but not all its other properties, such as firewall policies or higher-level network services like load balancing. The original idea behind using SDN to create virtual networks is widely credited to the team at Nicira, whose approach is described in an NSDI paper by Teemu Koponen and colleagues. The key insight was that modern clouds required networks that could be programmatically created, managed, and torn down, without a sysadmin having to manually configure, say, VLAN tags on some number of network switches. By separating the control plane from the data plane, and logically centralizing the control plane, it became possible to expose a single API entry point for the creation, modification, and deletion of virtual networks. This meant that the same automation systems that were being used to provision compute and storage capacity in a cloud (such as OpenStack [12] at the time) could now programmatically provision a virtual network with appropriate policies to interconnect those other resources.

The role of SDN in private and hybrid cloud adoption seems natural. Big SDN players such as Cisco, VMware and Juniper have all made moves to tie together enterprise data center and cloud worlds. ACI Anywhere package from Cisco, would for example let policies be configured through Cisco's SDN APIC (Application Policy Infrastructure Controller). It will use native APIs offered by a public-cloud provider to orchestrate changes within both the public and private cloud environments.

SDN also enables a variety of security benefits. Customers can split a network connection between the end user and the data center, while having different security settings for the different types of network traffic. One single network could have one public-facing, low-security network that does not let any sensitive data to traverse through it and another segment could have a finer grained remote access control with software-based firewall and encryption policies defined on it, which would allow sensitive data to traverse through it.

Another area where SDN is being used is the SD-WAN, i.e., Software-Defined Wide Area Network. It is a natural application of the SDN that brings the benefits of an SDN technology to a WAN. SD-WAN allows companies aggregate a variety of network connections - including 4g LTE, DSL and MPLS - into a network edge location and have a platform for software management which can fire up new sites, set security policies and prioritize network traffic. The SD-WAN infrastructure market is growing at a very high rate.



## 2.4 SDN Architecture

The SDN model has a 3-tier architecture. An analogy with the 3-tier architecture of a typical operating system will make it easier to understand the SDN model. The two models are quite similar.

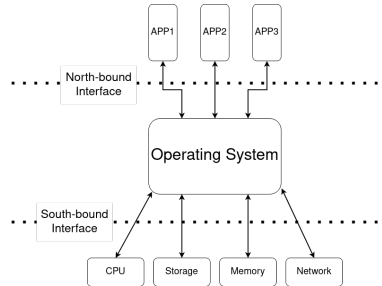


Figure 2.1: Three-tier OS Model

A typical desktop operating system can be broken down into three basic layers. First, there is the operating system (OS) itself. It acts as a middleman or an interface, managing access from applications to underlying hardware. It also comes with core services to aid in the process. On the lower layer, you have hardware. The OS is responsible for managing the system hardware on the lower layer, for example, the CPU, storage, memory, network interfaces, etc. Instead of calling it the lower layer, we can also call it the south of the operating system. Above the OS, or on the north-side, are the applications. The ability to develop or add/remove applications makes the system a flexible one, which can be tailored to the users' specific needs.

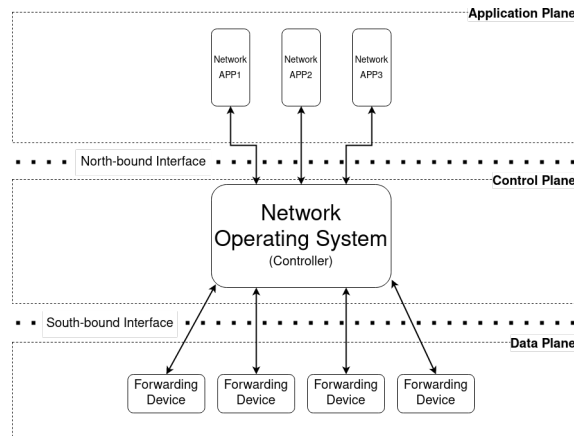


Figure 2.2: Three-tier SDN Model

For the SDN model, this will look quite similar to the OS model. Instead of the OS, the middle layer for SDN is the network operating system (NOS). This is also commonly called an SDN controller. The NOS will typically have core services to aid in its job of interfacing with network

nodes and for providing a programmable interface to the network applications. On the south side, instead of hardware like the processor, storage and memory, we have network forwarding device. The forwarding devices receive packets, take action on those packets and update counters. Types of actions include dropping the packet, modifying packet headers, and sending packets out of single or multiple ports. The instructions for how to handle packets originate with the SDN controller. Like the OS model, on the top or the north side are applications. Now we can call them network applications. Just like the apps on top of a standard OS, these apps can serve many purposes. But of course with SDN, they are all network focused.

The separation of the planes and open interfacing helps developers build creative network applications which was earlier not possible. There are a number of popular SDN controllers with extensive APIs which help developers interact with the forwarding devices. Some of them are NOX, POX and Floodlight. For the south-bound interface, the most commonly used protocol is OpenFlow. It provides standards for communication between the controller and the forwarding devices. OpenFlow has a strong community and is always evolving.

## Chapter 3

# Background on Mininet

### 3.1 Introduction

Mininet [4] is a network emulation software which lets users create virtual networks with hosts, switches and controllers. To be more precise, it is a network emulation orchestration system. It is lightweight and is based on Linux. Compared to simulators, Mininet (an emulator) uses lightweight process-based virtualization [5] to run the various components of the network (such as the hosts and switches) on a single OS kernel. This essentially means that each of the virtual components are hosted as independent processes. As a result of which, the users get to experience a realistic virtual network where they can run real application code and can also connect the virtual network to other real networks.

In short, these virtual devices act like their real counterparts. It is just that they are created using software rather than hardware. For the most part, their behavior is similar to discrete hardware elements.

Mininet is a useful tool for users who want to learn about Software-Defined Networking systems and OpenFlow. It allows the users to create, customize, test and share SDN networks.

Mininet includes a CLI, using which you can create virtual networks out of the box, without any programming. Mininet also provides an extensible Python API [6] for the same.

Creating a minimal topology (*consisting of an OpenFlow reference controller, an OpenFlow kernel switch and two hosts*) using Mininet is as simple as running Mininet.

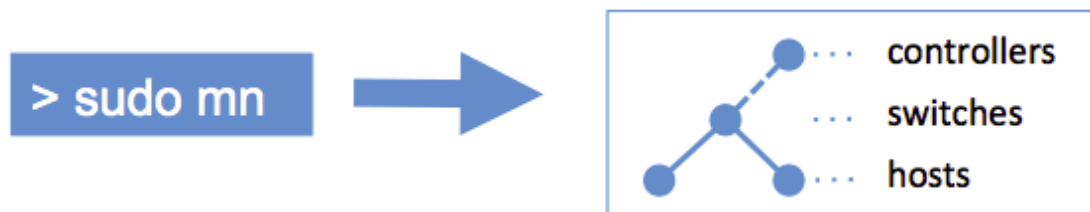


Figure 3.1: Emulating a Network Consisting of Two Hosts Connected to a Switch and a Controller

## 3.2 Mininet Commands

Mininet comes with a list of commands to navigate and re-configure the network. These also allow you to perform various tests and experiments on the network. Some of the commands are as follows:

**help** Once you are in Mininet, you can use this command to list all the commands available in Mininet

```
mininet> help
Documented commands (type help <topic>):
=====
EOF          gterm  iperfudp  nodes    pingpair
py           switch xterm    dpctl    help
link        noecho pingpairfull quit     time
dump        intfs  links     pingall  ports
sh          wait   exit      iperf    net
pingallfull px     source    x
```

Also, you can use **help** before any command to get a description of that command itself. For instance, if you want to know the description of **nodes**

```
mininet> help nodes
List all nodes.
mininet>
```

**nodes** displays a list of all the nodes in the network

```
mininet> nodes
available nodes are:
c0 h1 h2 s1
mininet>
```

**links** reports the status of the links. Here you can see on which interfaces the hosts are connected with the switch and also the status of those links.

```
mininet> links
h1-eth0<->s1-eth1 (OK OK)
h2-eth0<->s1-eth2 (OK OK)
mininet>
```

**dump** command dumps information about all the nodes

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=65198>
<Host h2: h2-eth0:10.0.0.2 pid=65200>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,
s1-eth2:None pid=65205>
<Controller c0: 127.0.0.1:6653 pid=65191>
mininet>
```

**pingall** is used to send ping requests between all hosts

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
mininet>
```

**iperf** and **iperfudp** are used to perform TCP and UDP bandwidth tests between two nodes respectively

```
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['34.2 Gbits/sec', '34.2 Gbits/sec']
mininet>
mininet> iperfudp bw h1 h2
*** Iperf: testing UDP bandwidth between h1 and h2
*** Results: ['bw', '11.8 Kbits/sec', '11.8 Kbits/sec']
mininet>
```

**sh** is a command used to execute external shell commands. For example, if you want to launch vim editor or wireshark from within Mininet.

```
mininet> sh vim
```

### 3.3 Mininet Topologies

Mininet provides a number of default topologies. As mentioned earlier, you can create a minimal topology by just running Mininet without any options and arguments. You can also create network topologies such as single, reversed, linear and tree, only by passing in the values of some command-line arguments. In this section, we will go through each one of them. Mininet follows a naming convention to name its elements and its components. This method of naming is applied when the

names are not explicitly defined by the user. Switches are named in the format **sX** from s1 to sN. Hosts are named in the format **hX** from h1 to hN. Host interfaces are named in the format **hX-ethY** - where, hX is the host name and ethY is an interface on that host. For example, the first interface of host h1 is named "h1-eth0" and the second interface of host h3 is named "h3-eth1". Just note that, in switches, interface numbering begins from 1. Hence, the first port of switch s1 is named "s1-eth1".

### 3.3.1 Minimal

This is the default topology that you get when you run Mininet. It consists of two hosts connected to an OpenFlow kernel switch and that switch is controlled by an OpenFlow reference controller that is running locally.

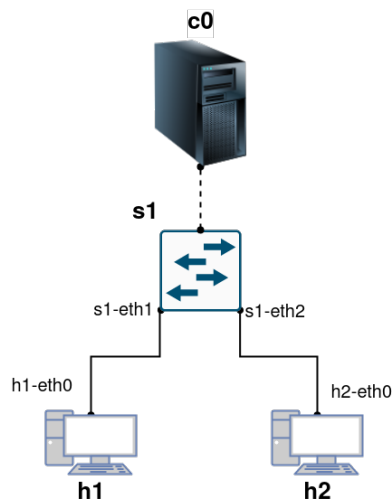


Figure 3.2: Minimal Topology

To obtain this topology, you just have to execute Mininet without any options or commandline arguments. Alternatively, you can explicitly mention the topology to be minimal.

**\$ sudo mn --topo minimal**

```
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo: s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
mininet>
```

### 3.3.2 Single

A single topology is one with a single OpenFlow switch and a number of hosts connected to that switch.

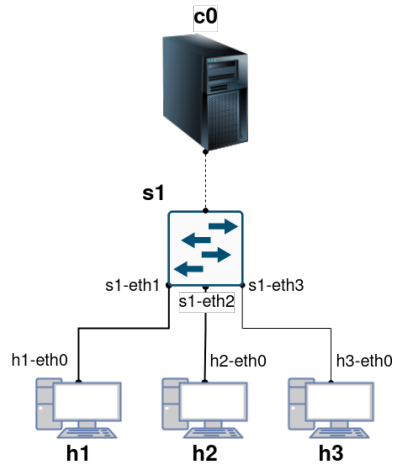


Figure 3.3: Single Topology

To create a topology where there are 3 hosts connected to a single switch. Use the following.

```
$ sudo mn --topo single,3
```

```
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
h3 h3-eth0:s1-eth3
s1 lo: s1-eth1:h1-eth0 s1-eth2:h2-eth0 s1-eth3:h3-eth0
c0
mininet>
```

### 3.3.3 Reversed

This is similar to the single topology, i.e., this also has a single switch connected to a number of hosts. But the hosts are connected in reverse order. Suppose, we have 3 hosts, then h1 will be connected to s1-eth3 and h3 will be connected to s1-eth1.

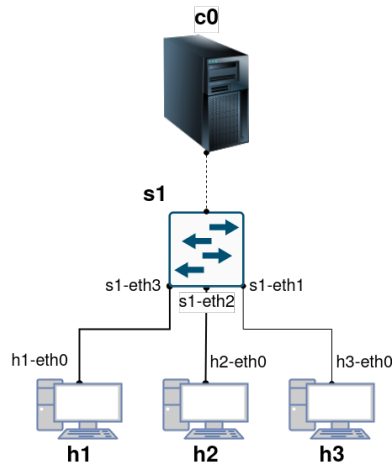


Figure 3.4: Reversed Topology

To create a topology where there are 3 hosts connected to a single switch, in reverse order. Use the following.

```
$ sudo mn --topo reversed,3
```

```
mininet> net
h1 h1-eth0:s1-eth3
h2 h2-eth0:s1-eth2
h3 h3-eth0:s1-eth1
s1 lo: s1-eth1:h3-eth0 s1-eth2:h2-eth0 s1-eth3:h1-eth0
c0
mininet>
```



### 3.3.4 Linear

A linear topology consists of  $k$  switches and  $k$  hosts. Each switch is connected to a single host. The switches are connected to one another as well.

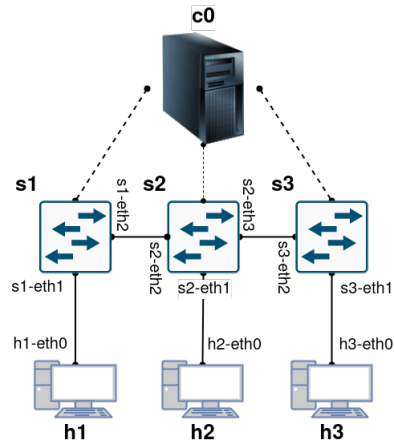


Figure 3.5: Linear Topology

To create a linear topology with 3 hosts and 3 switches. Use the following.

```
$ sudo mn --topo linear,3
```

```
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s3-eth1
s1 lo: s1-eth1:h1-eth0 s1-eth2:s2-eth2
s2 lo: s2-eth1:h2-eth0 s2-eth2:s1-eth2 s2-eth3:s3-eth2
s3 lo: s3-eth1:h3-eth0 s3-eth2:s2-eth3
c0
mininet>
```

### 3.3.5 Tree

As the name suggests, the tree topology consists of a tree like structure. The nodes form a perfect tree, i.e., all the non-terminal nodes contain the same number of children and all the terminal nodes are at the same level. The hosts are present as the leaves of the tree. All the other levels consist of switches. The depth and fanout of the tree are passed in as command-line arguments.

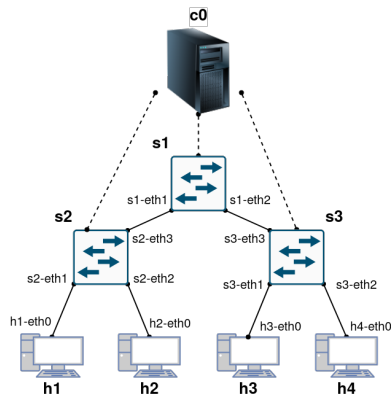


Figure 3.6: Tree Topology

To create a tree topology with a fanout of 2 and depth of 2. Use the following.

```
$ sudo mn --topo tree,fanout=2,depth=2
```

```
mininet> net
h1 h1-eth0:s2-eth1
h2 h2-eth0:s2-eth2
h3 h3-eth0:s3-eth1
h4 h4-eth0:s3-eth2
s1 lo: s1-eth1:s2-eth3 s1-eth2:s3-eth3
s2 lo: s2-eth1:h1-eth0 s2-eth2:h2-eth0 s2-eth3:s1-eth1
s3 lo: s3-eth1:h3-eth0 s3-eth2:h4-eth0 s3-eth3:s1-eth2
c0
mininet>
```

### 3.4 Custom Topologies

Mininet provides an extensive API with the help of which you can create your own custom topologies. All you need is a few lines of Python code. For instance, you can create a topology with two switches connected to each other, where the first switch is linked to 3 hosts and the second switch is linked to 1 host.

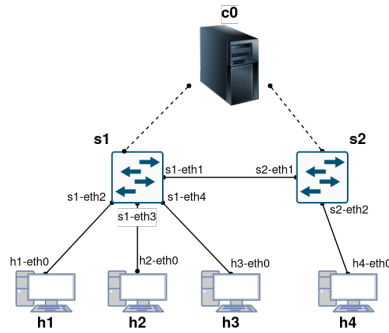


Figure 3.7: Custom Topology

You can create it with the following piece of code.

```
#!/usr/bin/env python3

from mininet.net import Mininet
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.node import OVSController, OVSSwitch

def create_custom_topo():
    "Creates an empty network and adds nodes to it"

    info('*** Creating Network\n')
    net = Mininet(controller=OVSController)

    info('*** Adding Controller\n')
    net.addController(name='c0')

    info('*** Adding Hosts\n')
    h1 = net.addHost(name='h1', ip='10.0.0.1')
    h2 = net.addHost(name='h2', ip='10.0.0.2')
    h3 = net.addHost(name='h3', ip='10.0.0.3')
    h4 = net.addHost(name='h4', ip='10.0.0.4')
    info(h1.name, h2.name, h3.name, h4.name, '\n')
```

```

info('*** Adding Switches\n')
s1 = net.addSwitch(name='s1', cls=OVSSwitch)
s2 = net.addSwitch(name='s2', cls=OVSSwitch)
info(s1.name, s2.name, '\n')

info('*** Adding Links\n')
net.addLink(node1=s1, node2=s2)
net.addLink(node1=h1, node2=s1)
net.addLink(node1=h2, node2=s1)
net.addLink(node1=h3, node2=s1)
net.addLink(node1=h4, node2=s2)
info('(s1, s2) ')
info('(h1, s1) ')
info('(h2, s1) ')
info('(h3, s1) ')
info('(h4, s2) \n')

net.start()
CLI(net)
net.stop()

if __name__ == '__main__':
    setLogLevel('info')
    create_custom_topo()

```

## Chapter 4

# Background on SDN Controllers

The SDN controller (or NOS), which forms the middle layer of the SDN model, provides services which can realize a distributed control plane and can also implement the concepts of ephemeral state management and centralization of control. It is the controller that communicates with the dumb forwarding devices. The controller sends instructions to these devices, following which necessary actions are taken by them. There are a number of popular SDN controllers out there. Some are from commercial vendors and the others from the open source community. In this chapter we will discuss about the various controllers available to us [26].

Currently, some of the commercial products include *vCloud* and *vSphere* by VMware, *NVP* by Nicira, *Trema* by NEC, *Floodlight* and *BNC* by Big Switch Networks, and *Contrail* by Juniper. Other popular controllers include NOX, POX, Floodlight, Ryu controllers.

There are some considerations that one must make before choosing an SDN controller. First, the programming language that the controller is written in. It will surely be easier to write code in a language you are familiar with. The language can also affect the performance as we know that not all programming languages work efficiently in all scenarios. Second most important consideration is the learning curve. If you need to develop and deploy module quickly, then it will be best to avoid the controllers which have a steep learning curve. The third consideration one must make is the strength of the controller's user base and the strength of its community. Like any other tool we use to develop applications, we need to rely on the community for help when we get stuck. A controller having a strong community, will not only help beginners but also help veterans when they get stuck. An active community also means that the product will be well maintained and constantly updated to match the needs of the current market. Another important factor when choosing a controller is the purpose for using it. Some controllers are re-implementation of the southbound API, i.e., the interface between the SDN controller and the forwarding devices. Using these APIs or protocols the controllers can directly speak with the forwarding switches. There are other controllers which implement the northbound API or the policy layer. These types of controllers allow the network operator or SDN programmers to write more complex high level programs at the expense of not being able to meddle directly with flow table entries. Pyretic and ProCera are examples of controllers which implement northbound APIs. Also, the goal of the user matters. For instance one might pick a controller which is better for research and experimentation while someone else might choose a controller which is robust and can be used in production.

## 4.1 NOX

NOX [8] is a first-generation OpenFlow controller. NOX was developed by Nicira and was donated to the research community. As a result, it became open source in 2008. In fact the early version of NOX was the first open source OpenFlow controller. It was then extended and supported by ON.LAB activity at Stanford University with major contributions from UC Berkeley. NOX is known for its stability and is widely used. There are two flavors of NOX, **NOX-Classic** - which is written in C++ and Python (but is not longer supported), and **NOX** (or new Nox) - which is written only in C++ and has a good performance in terms of speed. It has a clean codebase and is well maintained and supported by the community.

Users need to implement control applications in C++ when they are developing for NOX. It mainly supports OpenFlow v1.0 but there are some community forks of its public repository which support v1.1, v1.2 and v1.3.

The programming model of NOX is similar to that of many OpenFlow controllers. That is an asynchronous, event-based model. In it a controller registers for events and the programmer then writes event handlers that take specific actions or perform various tasks when those events are raised.

There are a number of scenarios where one could use NOX. One is if you are familiar with writing code in C++. Also, if you are willing to use very low-level facilities and semantics of OpenFlow. NOX is a pretty bare-bones implementation of the southbound API and it does not provide many high level abstractions. This will be helpful for those looking to meddle directly with the southbound API and extend it. Another good reason to use NOX is its high performance. This is due to the fact that it is developed in C++ which is a low level language.

NOX is primarily used in academic network research to develop SDN applications such as network protocol research. As a result of its widespread academic use, we can find examples and code snippets in abundance. That can be used as starter code for experimentation and projects.

Two popular NOX applications are SANE and Ethane. SANE is a way of representing the network as a filesystem. Ethane is used by Stanford University for centralized, network-wide security at the level of a traditional access control list.

## 4.2 POX

POX [14] is one of the most popular controllers today. It is the newer, Python-based version of NOX (or NOX in Python). It currently only supports OpenFlow v1.0 and extends special support for Open vSwitch. The idea behind introducing POX was to modify NOX-Classic to separate Python from it and return NOX to its C++ roots. As a result, NOX-Classic was discontinued and NOX was developed as a C++ API. For Python, the POX controller was then developed. It has a high level SDN API including a query-able topology graph and support for virtualization.

There are a several benefits to using POX. It is a very popular network software platform and has a strong community that maintains the code and supports the project. It is relatively easy for developers to read and write code. But it has a major drawback - Performance. Since it is written in Python, we can not expect it to match the performance of NOX.

A Python developer will naturally want to choose POX as the controller as the learning curve would be minimal in this case. Although one has to keep in mind the performance demerits. If you are not concerned about controller performance, then you can go ahead and use POX with Python.

POX is primarily used for rapid prototyping and experimentation. It is quite useful for research and demonstrations. It is also very helpful for users who are learning SDN concepts. You will not need much effort to build topologies using an emulation tool like Mininet and use POX as the controller for the forwarding switches.

Since POX is based on NOX, it is component based. A handful of components can be found out of the box which help the user get the controller up and running, emulating popular networking devices or middleboxes like a L2 learning switch or a loadbalancer. The component source code acts as a great starting point for developers building their own components. Most of the time a developer does not need to write applications from scratch. Often they just need to tweak the existing components to get their desired outcome.

Some of the components that you get with POX are:

- **py**: This causes POX to start an interactive Python interpreter that can be useful for debugging and interactive experimentation
- **forwarding.hub**: The hub example just installs wildcarded flood rules on every switch, essentially turning them all into ethernet hubs.
- **forwarding.l2.learning**: This makes OpenFlow switches act as a type of L2 learning switch. While this component learns L2 addresses, the flows it installs are exact-matches on as many fields as possible. For example, different TCP connections will result in different flows being installed.
- **forwarding.l2.pairs**: Like **l2.learning**, this component also makes OpenFlow switches act like a type of L2 learning switch. However, this one is probably just about the simplest possible way to do it correctly. Unlike **l2.learning**, **l2.pairs** installs rules based purely on MAC addresses.
- **forwarding.l3.learning**: Although the name suggests that this component is a router, it is not. It is kind of like a L3 learning switch. **l3.learning** does not care about conventional IP features like subnets; it just learns where IP addresses are. The users may require subnets and have gateways designated to those subnets. In order to facilitate it, this component provides an option for assigning fake gateways with an option called **--fakeways**.
- **openflow.spanning\_tree**: This component uses the discovery component to build a view of the network topology, constructs a spanning tree, and then disables flooding on switch ports that are not on the tree. The result is that topologies with loops no longer causes broadcast storms and infinite looping packets in the network. It has a couple of options:
  - no-flood**: disables flooding on all ports as soon as a switch connects; on some ports, it will be enabled later.
  - hold-down**: prevents altering of flood control until a complete discovery cycle has completed (and thus, all links have had an opportunity to be discovered).

There are many more components which you can find out in the official documentation.

## 4.3 Ryu

Ryu [15] is another component based, open source Python controller. The Ryu messaging service does not support components which are developed in other languages. It supports OpenFlow version 1.0, 1.2, 1.3, 1.4, 1.5 and various Nicira extensions. It also works with OpenStack (a cloud OS that controls storage, compute and networking resources in a data center and allows you to write cloud applications on top of that OS. It is open source and has a very large community). Ryu aims to be an operating system for SDN and it has various advantages such as OpenStack integration and the extended support of OpenFlow versions 1.2 and 1.3. Again, a major drawback is its performance because of its implementation in Python.

## 4.4 Floodlight

Floodlight [1] is an open source SDN controller implemented in Java. It is maintained by Big Switch Networks and is a fork from the Beacon Java OpenFlow controller which is also maintained by Big Switch Networks. Floodlight supports OpenFlow v1.0.

Floodlight has a well written documentation. Out of the mentioned controllers, this arguably has the best documentation. Floodlight also integrates well with the REpresentational State Transfer (REST) API. Floodlight also provides production level performance and integrates well with OpenStack. One of its major disadvantages is its steep learning curve.

You can pick Floodlight if you are familiar with Java. Also, when you need production level performance and support, Floodlight will deliver. Floodlight's integration with REST API is another factor to consider it.

We have discussed four popular controllers. It will be easier to pick a controller based on a table of their strengths and weaknesses.

## 4.5 ONOS

Open Network Operating System [9] or ONOS, is another open source SDN controller written in Java. It is the leading open source SDN controller for building SDN solutions. ONOS is also known as the only SDN controller platform that supports the transition from legacy brown field networks to SDN green field networks. It was developed in 2014. ONOS is a part of the Linux Foundation collaborative project.

ONOS can run as a distributed system on multiple servers, allowing the use of CPU and memory resources of many servers. It does so while providing fault tolerance in the face of server failure and potentially supporting live/rolling upgrades of hardware and software without interrupting network traffic.

Not only are the ONOS kernel and core services, written in Java, but the ONOS applications are written in the same language as well. Since ONOS runs on the Java Virtual Machine (JVM), it can run on several underlying OS platforms.

ONOS is known for its high performance, high availability and scalability. ONOS provides northbound abstractions for a global network view and network graphs, and also provides pluggable southbound for support for OpenFlow, P4Runtime and other newer or older protocols. These qualities allow ONOS to be used in production.



ONOS is widely used. It is used for interconnecting an SDN network with a traditional network using SDN-IP. It is also used for Data Center Network Virtualization. DC Network Virtualization is achieved with ONOS SONA. ONOS SONA is an ONOS based Virtual Network Management solution which is fully compatible with OpenStack. ONOS is also used to provide Virtual Private LAN Services (VPLS). These are only a few use cases, there are many more common use cases for ONOS.

	NOX	POX	Ryu	Floodlight	ONOS
Language	C++	Python	Python	Java	Java
Performance	Fast	Slow	Slow	Fast	Fast
OpenFlow	1.0	1.0	1.0 to 1.5	1.0 to 1.4	1.0 to 1.3
OpenStack	No	No	Yes	Yes	Yes (SONA)
Learning Curve	Moderate	Easy	Moderate	Steep	Steep

Table 4.1: Comparison of Controllers

## Chapter 5

# Mininet as SDN TestBed

Mininet is a pretty useful network orchestration tool. Its lightweight process virtualization technique combined with the use of linux namespaces and process grouping makes the virtual network appear as a real network with real networking devices. It can be used to teach and learn SDN concepts. But is it possible to create a proper, robust and accurate virtualized testbed for SDN? Let us see.

With technology progressing at a rapid pace, there is a need for new ideas to emerge quickly for finding solutions to our networking problems. And with this rapid evolution comes the need for proper testing. The problem with testing a system, such as an SDN, is that it often requires expensive equipment and sophisticated software. This not only requires skilled professionals to carry out tests but it also has a significant amount of cost associated with it. Even if money and time were not constraints, it requires a lot of effort and the systems to be set up are complex. As an alternative network virtualization tools like Mininet are widely used. Since Mininet is open-source, it is seen as a viable solution as it is ready to be used and can be implemented at no cost (or very low cost, taking into consideration the need for hardware upgradation). Mininet offers capabilities that can emulate the behavior of an actual network infrastructure. As a result, designing and deploying SDN technologies, which began in large data centers, can now be implemented on our personal computers. Mininet and similar network virtualization tools might seem like the perfect choice for an SDN testbed, but even the most sophisticated software can lack in areas such as security and robustness. Although emulation gives close to real results, testing the emulated system with reduced network workloads can impact the accuracy of the results. Such tests mask the problems that can actually be found on large scale systems.

Combining SDN and the concept of virtualization is quite beneficial as virtualization simplifies the complex SDN systems by maintaining its functionality, while converting the physical objects into ideal abstract objects.

### 5.1 Testbed Design

Various experiments were performed by researchers and they studied the results obtained using different kinds of test environments. For designing one such testbed [20], two different scenarios should be considered: First, the Mininet environment alone and second, the Mininet environment with an additional physical layer.

- A. **Mininet** As seen in chapter 3, Mininet is a popular network virtualization tool which can emulate real networks accurately. For the first part, let us consider a minimal Mininet SDN topology executed on a pre-built Linux-based Mininet VM installed on VirtualBox (which is a type 2 hypervisor) on a physical machine (running Ubuntu 22.04 as host OS). A minimal environment in Mininet consists of two virtual machines connected to a virtual OpenFlow Switch and is managed by Mininet's instantiated default OpenFlow reference controller.
- B. **Physical Layer** An additional physical layer is added on top of Mininet's environment for the second approach. The idea is to check whether this yields any significant benefits over Mininet alone, by strengthening the node isolation within Mininet. This is done by using a physical switch and an external controller which is running on a physical host.

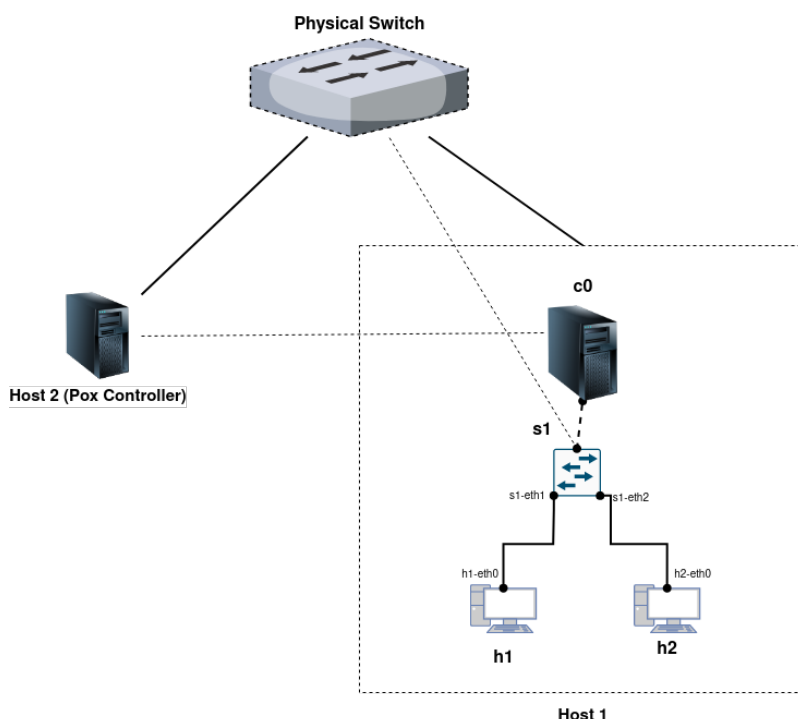


Figure 5.1: Physical Architecture

The network is set up in a way such that the hosts are isolated from one another. The Mininet VM PCs should not have access the rest of the network and are hence connected to a particular VLAN. Host 1 and Host 2 are connected to the physical switch on another VLAN which allows OpenFlow messages to be exchanged in an isolated way as well. IP addressing is done accordingly, giving each node appropriate access to the internal networks.

Upon testing the two environments, i.e., Mininet itself and Mininet with and additional physical layer, the following points were observed.

## 1. Isolation of nodes:

- *Mininet*

The way Mininet creates virtual hosts is by instantiating shells from the same process. These instances are the virtual hosts in Mininet. They do share the same process space but use unique network namespaces [2]. As a result, they attain the required isolation. But the problem with Mininet is that it assigns root privileges to the virtual hosts. This gives them access to not only view all the processes running in the main system, but also to kill them. So giving the VMs root privileges unnecessarily is a security threat. This can be overridden with a little workaround which involves creating a user with only sufficient rights and then connecting the the VM through SSH, followed by termination of the Mininet process. Although attainable, this does not come out of the box and is a bit complicated.

Also, having root privileges, the virtual hosts can create their own configurations or modify existing configurations. Since Mininet virtual hosts are just instances of the shell, they share the same process space. As a result, they can add, modify or remove configurations from the main system. This is a major breach of security, since the main system will be unaware of any changes made by the VMs.

Not only do the shell instances share process space, but they also share the file system. As a result there could be filename conflicts if two instances of the shell process is used to create the same file. One workaround for this would be to use Overlay Filesystem [13].

- *Physical Layer*

Having root users on a virtualized system can compromise VLAN use. This is because, the instantiated OvS is also another process which might get terminated accidentally. This would lead to the loss of VLAN labeling on them. Also, the VLANs defined on the network within Mininet can only isolate packet transmission and can not isolate file access and access to processes on a shared system. The dataflow from external devices (host 1, host 2 and the physical switch) can be measured accurately but the data flowing from the controller to the Mininet environment can not. All data flow relies on the Mininet environment. Since the VMs have root privileges, they can change the configurations. This might affect data flow and as a result, the analysis data would be inaccurate.

## 2. Resource Management:

- *Mininet*

Resource management in Mininet is poor. First, CPU has to be limited for each host. Although, you can assign more CPU to each of the VMs, it could overload the system. When CPU allocation is left to default, it is managed by Mininet itself. The performance drops considerably when another Mininet instance is run on the same environment. The resources are reallocated as a result of which, the capacity of each virtual host reduces. Also, due to sharing of resources, packet forwarding is slow.

Resource de-allocation is also not good enough on Mininet. When you terminate the Mininet process, you can not be sure of all the processes spawned by Mininet to be terminated. It is highly possible that some zombie processes remain in the system. These can accumulate and consume resources unnecessarily. Also, when terminating Mininet,

the configurations done earlier remain intact which also consumes memory resources. To avoid this, `--clean` option should always be used.

- *Physical Layer*

In Mininet environment alone, performance is seen to dip as soon as a second instance of Mininet is run on the same platform. Here, in a more complex system, it is even more problematic. The external controller running on a separate physical machine views the network as a whole. Having multiple instances of virtual switches and virtual controllers makes it difficult for the external controller to manage the SDN network. One can scale out the Mininet servers, but it brings in additional complexity. The reason being Mininet's absence of native clustering. Clustering is possible with extra configuration.

Although we can attain a more secure network with good isolation with an external switch and an external controller, inside Mininet all the components are instances of the main system. So, even though packet forwarding is completely isolated, the file system and the process space are not.

### 3. Data flow Analysis:

- *Mininet*

To analyse the data flow, `iperf` is popular tool. It is used to measure the maximum possible bandwidth in IP networks. But the problem with `iperf` is that it is fit for slow machines and smaller networks. A complex and bigger network would require additional Python scripts to measure the network bandwidth. Also, as mentioned earlier, packet forwarding in Mininet is slower than that of a real system (due to resource sharing). This affects the analysis results. One other potential problem noted is that the instanced controller communicates with the forwarding devices on the Loopback (Lo) interface. So, a large number of virtual hosts and virtual switches would make quite a clutter on a packet sniffing tool like Wireshark. A workaround for that is the use of MAC assignment feature provided by Mininet. This way it will be easier to track the messages exchanged between the controller and the other devices. However, the ports of the switches will still be randomly named.

- *Physical Layer*

Analysing data flow in the environment with additional physical layer, is not accurate either. We have discussed the reasons mentioned before: flow of data inside Mininet relies fully on the environment, there is a possibility of configurations being altered by the VMs and that Loopback (Lo) interface is used for all data exchange between the instanced controller and the other devices.

Mininet, although quite intuitive and easy to use, its environment is not suitable for testing complex networks. It is better to use heavyweight virtualization software/hardware for developing SDN testbeds as these tools provide proper isolation, close to real behavior and network performance. Mininet is good for testing smaller networks, and it is a perfect tool for learning about basic SDN concepts. But once you grasp the basics, Mininet would not be quite as effective as you will not be able to explore the advanced SDN concepts fully since it is not possible to adhere to the best practices.

## 5.2 Distributed SDN

With technological advancement and easy availability of the Internet, the need for fault tolerant and scalable systems has become stronger. It is the same in case of SDN operating systems (NOS) and its applications. Apart from fault tolerance and scalability, high availability is one other significant feature. For making the system highly available and scalable, one method is to scale up the current hardware. But this will only help in the short run, as we know that with scaling up comes a point of diminishing returns. That is, performance benefits can only be achieved till a certain point, beyond which upgrading your hardware will give your worst performance. To avoid such a situation it is recommended to scale out instead. For building a distributed SDN with its applications, you will need access to an expensive testbed which would offer reliability. But it is not always easy to get access to such expensive tools. Some options include using full system virtualization and heavyweight container, but these increase the complexity and reduce usability. As an alternative Mininet can be used to deploy a testbed of lightweight containers on a single machine.

Instead of using full machine virtualization and heavyweight containers, Mininet can be used. Mininet provides the capability of creating lightweight containers. This method does not require installing any additional software. For configurations that can not be satisfied by a single Mininet server, there is an experimental cluster support provided by Mininet (since version 2.2) which could be used. Even before this feature was included in Mininet, there had been some implementations of distributed Mininet systems. Two of the popular ones include P. Wette's MaxiNet [3] and Distrinet [18].

The `cluster.py` example script you will find on the public repositories of Mininet, is written by Bobby Lantz [24]. He demonstrates how easy it is to set up a distributed SDN on Mininet with the cluster. Although tools such as Virtual Emulab and CORE are better to use as a testbed for distributed SDN, Mininet provides a simpler, faster and a more convenient solution for network scenarios which are not complex in nature.

## Chapter 6

# Mininet as 5G TestBed

### 6.1 Contribution of SDN and NFV in 5G

Wireless Telecommunication networks or mobile networks have made a lot of progress over the years. We have seen the success achieved and the wide use of 4G cellular network. But even with all the progress that has been made in these years, there lies several problems with transmission. Some of the problems pertain to areas involving the interfaces, performance, handovers, costs, qualities, and many more. The newer technologies such as smart devices and Internet of Things (IoT) depends heavily on the reliability of the data flowing in wireless networks. The newest generation of cellular network - 5G, demands high data rates, lower power consumption, optimized management of resources and lower delays in network. Since 5G requires all of these, network management and orchestration has to be centralized and the various network functions need to be virtualized. This can not be achieved through traditional networks. As a result, Software Defined Networking (SDN) comes into play.

We have already discussed about the merits of SDN. Separation of planes and a logically central control not only help in reducing costs and breaking free from proprietorship but also help in converting the network into a virtualized one. This is an essential feature which allows complex conventional hardware (such as firewalls, routers, load-balancers, etc) to be converted to software, and a number of such software can run in parallel on a single piece of hardware.

Network Function Virtualization (NFV) [27] is the process of converting the mentioned network functionalities, which are conventionally provided by hardware, to software. These pieces of software are called Virtual Network Functions (VNFs). In essence, it is the virtualization of networking devices like firewalls, routers, etc. These VNFs are then run on servers and can be scaled out if and when needed. When scaled out, they are managed by a management and orchestration system (MANO). When combined with SDN, NFV offers some benefits over traditional network architecture. Some of them are: (a) reduced power consumption as it is possible to serve multiple VNFs on a single physical server, (2) rapid development and deployment is possible because of the separation of planes and centralization of the controller, (3) reduction in maintenance costs.

So, when SDN and NFV combined, form the perfect recipe for 5G [17]. With SDN contributing to better management of the resources and NFVs allowing the introduction of devices anywhere almost immediately with some lines of code. This makes 5G network more scalable. Also having a central controller benefits such a network as the controller gathers information of all the network

traffic, routes and functionality. This helps it implement policies based on statistics.

## 6.2 Using Mininet for Testing

A number of studies have been performed by researchers to test various network scenarios and analyse the performances of various tools. In [17] we see the comparison between the performances of POX and ONOS controllers in their experiments involving vRouter, IPRAN and MCORD.

In such experiments, Mininet can be used to insert routers, switches and hosts into the network topology. It can also help connect to the remote controllers. For experimenting with IPRAN and MCORD Mininet-wifi was used, which is an implementation of Mininet for management and orchestration of Wireless SDNs.

### 6.2.1 vRouter: Virtual Router

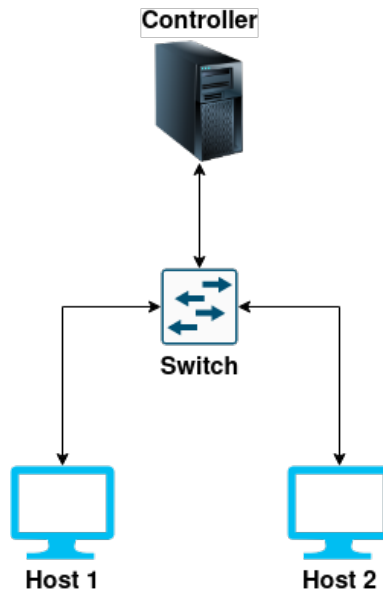


Figure 6.1: vRouter

vRouter, or the virtual router, is as the name suggests a software virtualization of router. That is, a vRouter provides the functionality of a router through programmable logic on a bare metal switch. The controller communicates with the switch to handle routing. In the experiment performed by [cite], they built a topology containing 2 hosts connected to the vRouter switch and it communicated with the SDN controller using Border Gateway Protocol (BGP). When used with ONOS, it has a holistic view of the network and can depict all changes that occur in the topology. When used with POX, such features can not be expected. Instead, POX displays messages about the state of the connection. For POX, Mininet takes care of the debugging messages. Whereas for ONOS is adept at displaying messages to the users by itself.



### 6.2.2 IPRAN: Internet Protocol Radio Access Network

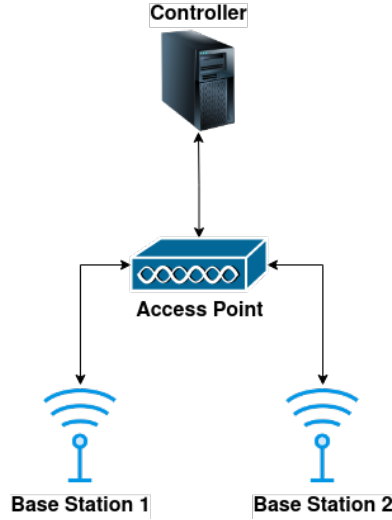


Figure 6.2: IP RAN

Internet Protocol Radio Access Network, or IPRAN, is a term used to describe a Radio Access Network [22] (such as 2G, 3G, 4G or 5G) which uses Internet Protocol (IP). An IPRAN network is built with packet-switched routers that can provide high bandwidth, low delay, low costs and high reliability. Since IPRAN is used for mobile networking, for this experiment - Mininet-wifi [7] is used. Mininet-wifi provides the virtualization for the base stations and access points that are needed. Again both POX and ONOS are used as controllers. The topology created contains two base stations which communicate with an access point. POX works with Mininet just as expected - it displays only the connection state messages whereas ONOS, with its holistic view of the network, depicts all changes in the topology in detail. With ONOS, several important issues such as mobility and handovers could be tested. ONOS allows complex topologies to be tested and to derive important conclusions from those tests.

### 6.2.3 MCORD: Mobile-CORD

M-CORD or Mobile Central Office Re-architected as a Data center (Mobile-CORD), is an open source reference solution for carriers deploying 5G mobile wireless networks. It is a cloud native solution which is built on SDN, NFV and cloud technologies. It is a mix of both virtualized RAN functions and a virtualized mobile core to enable mobile edge applications and innovative services using a micro services architecture. It offers several VM copies to coexist. To test MCORD functionalities, an experimental environment consisting of open software, such as OpenStack, ONOS, vRouter, Virtual Terminal Network (VTN), is required. MCORD requires Ubuntu LTS to run. With ONOS, the tests were possible but MCORD is not offered as a use case by POX controller. There are no special interfaces that you could build to experiment with CORD or MCORD. So POX and Mininet-wifi is not sufficient for this test.

The strengths of Mininet lies in its simplicity and convenience. As we saw in the experiments, Mininet was used as a tool to help test the POX controller. This is the best use of such a lightweight virtualization tool. We can not expect it to be robust and secure. It does not come with a ton of features either. Mininet should not be relied upon with complex networks or systems. In the previous chapter, we discussed about the shortcomings of Mininet as a testbed for an SDN system. 5G is a much more complex network which happens to be implemented with SDN and NFV. So, it is not advisable to use Mininet solely as a testbed for 5G.

## Chapter 7

# Conclusion and Future Work

### 7.1 Conclusion

From our study, we can conclude that Mininet is a useful tool for emulating and orchestrating networks. It is particularly helpful for learning the concepts of and experimenting with SDN. It also allows users to get accustomed to the different SDN controllers. A lightweight tool like Mininet allows its users to set up their own custom network topology from the command-line itself or with only a few lines of Python code - in a matter of a few seconds.

As simple and intuitive as Mininet is, there exists some major concerns. Mininet does its job well for simple networks. But when emulating complex networks, Mininet does not perform so well. The kind of process virtualization that Mininet uses, is not performance efficient. Running only two instances of Mininet on a single physical machine, reduces the throughput of the virtual devices in Mininet significantly - their CPU speeds are limited. There also exists major security concerns. By default, all Mininet virtual devices have root privileges and since those processes share the same process space, there is a major breach of security.

Apart from the resource management and isolation issues, Mininet also does not allow accurate analysis of data. Which makes it really difficult to test a system. Using Mininet as a TestBed for SDN is not advised. Although Mininet is being developed and features are being added constantly by developers in the open-source community, it is still not ready for being used as a robust and secure SDN TestBed.

Mininet now offers experimental features such as cluster for distributing Mininet across multiple servers, but this feature is not yet fully developed to be used for testing a production level system. Software like Distrinet (which is a re-implementation of Mininet), although complex, are a better choice to implement and test distributed SDN systems.

Finally, for 5G, the extension of Mininet: Mininet-Wifi suffers from many of the same issues that were discussed. Although it is possible to implement a simplified IP RAN with Mininet-Wifi and a controller such as POX, it doesn't let you analyse the data flows accurately. Platforms such as BlueArch [21] are better suited to be used as a TestBed for 5G networks.

## 7.2 Future Work

Future works could include research on Mininet and the various open-source extensions of Mininet, which can be used to test SDN networks. Developers could also modify Mininet and extend it in ways that would deal with the issues regarding isolation, resource management and accurate data analysis; while maintaining the essence of Mininet, i.e., a lightweight virtualization tool.

Mininet-wifi can be further researched and compared with the other popular emulation and orchestration tools. A comparative study would give a better idea of the merits, and demerits, of a simple tool like Mininet-wifi over a robust platform such as BlueArch.

Using Mininet, further researches could also be conducted into SDN and NFV. Although it is not suitable for complex networks, it still serves the purpose of rapid prototyping.

# Appendix A

## Mininet Experiments

### A.1 Experiment 1

POX, a component based OpenFlow controller, provides an extensive API which can be used to model the control logic of networking devices. For a better understanding of the POX API, let us build our own component. POX comes with a handful components out of the box, and it is convenient to start by modifying an already existing component. For this task, we shall modify the code for a hub to make the switch act like an L2 learning switch.

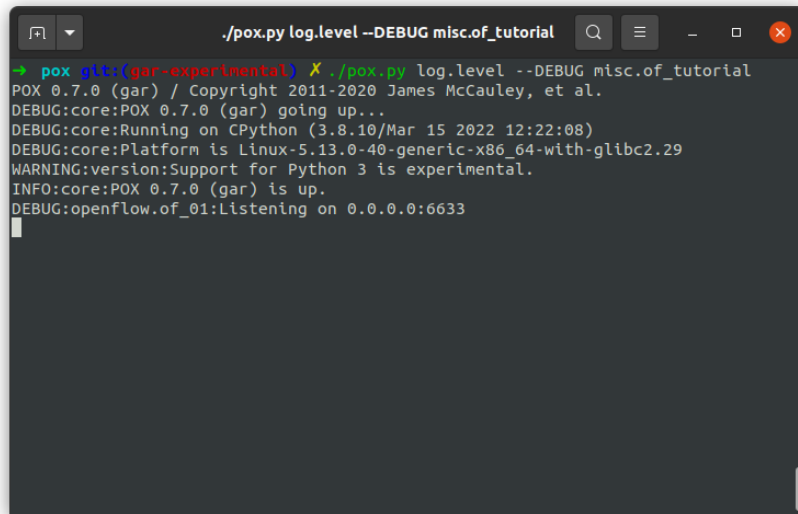
There are a couple of ways we can go about it. That is, we can start by editing the `pox/fowarding/hub.py` file or the `pox/misc/of_tutorial.py` file.

We chose the latter as the existing comments make the code more intuitive. Now before we begin to modify the code, let's look at what this component does. To execute it, run the following:

Terminal (POX)

```
$ cd ~/pox
$ ./pox.py log.level --DEBUG misc.of_tutorial
```

On running this, you will find that POX loads up and starts listening for OpenFlow connections on "0.0.0.0:6633".



```
./pox.py log.level --DEBUG misc.of_tutorial
+ pox git:(gar-experimental) X ./pox.py log.level --DEBUG misc.of_tutorial
POX 0.7.0 (gar) / Copyright 2011-2020 James McCauley, et al.
DEBUG:core:POX 0.7.0 (gar) going up...
DEBUG:core:Running on CPython (3.8.10/Mar 15 2022 12:22:08)
DEBUG:core:Platform is Linux-5.13.0-40-generic-x86_64-with-glibc2.29
WARNING:version:Support for Python 3 is experimental.
INFO:core:POX 0.7.0 (gar) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
```

Figure A.1: Running of tutorial.py provided by POX

The reason why it says that the IP is 0.0.0.0 is that we have not explicitly mentioned 127.0.0.1 (localhost) on the command line while running POX. That isn't a problem, since we are running it locally, i.e., on the same machine. This will be different for you if the POX controller is running on a different machine (physical or virtual).

On a separate terminal, let's run Mininet with three hosts connected to a single switch. We shall also use the --mac option as this will automatically set the MAC addresses of the hosts to a sequential order. We set the switch type to to an OpenFlow virtual switch and connect to the remote POX controller which is already running.

```
Terminal (Mininet)
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
```

```
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

The POX controller that is running, in this experiment, turns the complex switch into a dumb hub. We can confirm that by monitoring TCP traffic on the hosts.

For that, let's open xterm for the three hosts h1, h2, h3.

```
Terminal (Mininet)
mininet> xterm h1 h2 h3
```

Now, on terminals to h2 and h3. Run **tcpdump** to monitor all tcp traffic on the hx-eth0 interfaces.

```
Node2: h2
# tcpdump -XX -n -i h2-eth0
```

```
Node3: h3
# tcpdump -XX -n -i h3-eth0
```

You might see some router solicitation messages received by both the hosts. But to actually confirm that the switch is acting like a hub, use an xterm to host h1 and ping either of h2 or h3. You will notice that the same packets, be it ARP or ICMP, are received on both h2-eth0 and h3-eth0.

```
Node1: h1
# ping -c1 10.0.0.3
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=1.74 ms

--- 10.0.0.3 ping statistics ---
1 packets transmitted,1 received,0% packet loss,time 0ms
rtt min/avg/max/mdev = 1.736/1.736/1.736/0.000 ms
```

```

"Node: h2"
02:46:02.185768 0x0020: 0000 0000 0003 0a00 0003 .....
ARP, Reply 10.0.0.3 is-at 00:00:00:00:00:03, length 28
0x0000: 0000 0000 0001 0000 0000 0003 0806 0001 .....
0x0010: 0800 0604 0002 0000 0000 0003 0a00 0003 .....
0x0020: 0000 0000 0001 0a00 0001 .....
02:46:27.329425 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 9224, seq 1, length 64
0x0000: 0000 0000 0003 0000 0000 0001 0800 4500 .....E.
0x0010: 0054 163a 4000 4001 106c 0a00 0001 0a00 .T.:@.@.l.....
0x0020: 0003 0800 2510 2408 0001 abb3 8662 0000 ....%,$.....b..
0x0030: 0000 b9fd 0400 0000 0000 1011 1213 1415 .....
0x0040: 1617 1819 1a1b 1c1d 1e1f 2021 2223 2425 .....!""#$%
0x0050: 2627 2829 2a2b 2c2d 2e2f 3031 3233 3435 &'()*+,-./012345
0x0060: 3637 67
02:46:27.331407 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 9224, seq 1, length 64
0x0000: 0000 0000 0001 0000 0000 0003 0800 4500 .....E.
0x0010: 0054 416d 0000 4001 2539 0a00 0003 0a00 .TAm..@.%9.....
0x0020: 0001 0000 2d10 2408 0001 abb3 8662 0000 ....-,$.....b..
0x0030: 0000 b9fd 0400 0000 0000 1011 1213 1415 .....
0x0040: 1617 1819 1a1b 1c1d 1e1f 2021 2223 2425 .....!""#$%
0x0050: 2627 2829 2a2b 2c2d 2e2f 3031 3233 3435 &'()*+,-./012345
0x0060: 3637 67

```

Figure A.2: tcpdump on terminal to Host h2

```

"Node: h3"
02:46:02.184374 0x0020: 0000 0000 0001 0a00 0001 .....
ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01, length 28
0x0000: 0000 0000 0003 0000 0000 0001 0806 0001 .....
0x0010: 0800 0604 0002 0000 0000 0001 0a00 0001 .....
0x0020: 0000 0000 0003 0a00 0003 .....
02:46:27.329429 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 9224, seq 1, length 64
0x0000: 0000 0000 0003 0000 0000 0001 0800 4500 .....E.
0x0010: 0054 163a 4000 4001 106c 0a00 0001 0a00 .T.:@.@.l.....
0x0020: 0003 0800 2510 2408 0001 abb3 8662 0000 ....%,$.....b..
0x0030: 0000 b9fd 0400 0000 0000 1011 1213 1415 .....
0x0040: 1617 1819 1a1b 1c1d 1e1f 2021 2223 2425 .....!""#$%
0x0050: 2627 2829 2a2b 2c2d 2e2f 3031 3233 3435 &'()*+,-./012345
0x0060: 3637 67
02:46:27.329521 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 9224, seq 1, length 64
0x0000: 0000 0000 0001 0000 0000 0003 0800 4500 .....E.
0x0010: 0054 416d 0000 4001 2539 0a00 0003 0a00 .TAm..@.%9.....
0x0020: 0001 0000 2d10 2408 0001 abb3 8662 0000 ....-,$.....b..
0x0030: 0000 b9fd 0400 0000 0000 1011 1213 1415 .....
0x0040: 1617 1819 1a1b 1c1d 1e1f 2021 2223 2425 .....!""#$%
0x0050: 2627 2829 2a2b 2c2d 2e2f 3031 3233 3435 &'()*+,-./012345
0x0060: 3637 67

```

Figure A.3: tcpdump on terminal to Host h3



We can see that both h2 and h3 received the same packets. Hence we can now be sure that the Open vSwitch is acting like a hub. Also, if we run **iperf**, we will find that the bandwidth is quite low. The reason for this is that - all the incoming packets are sent to the controller due to table miss. The reason for table miss is that unlike **forwarding.hub**, this module does not install a default flow entry to flood all packets. Instead all packets are sent to the controller first. This packet-in/packet-out events which occur for every incoming packet, slows down the network.

Terminal (Mininet)

```
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['19.5 Mbits/sec', '21.2 Mbits/sec']
mininet>
```

You can also open an xterm to the switch s1 and run: **dpctl dump-flows tcp:127.0.0.1:6654**. You will not find any flow entries in the table.

Now let's take a look at the code. Open the file `~/pox/pox/misc/of_tutorial.py` on your preferred text editor.

of\_tutorial.py

```
from pox.core import core
import pox.openflow.libopenflow_01 as of

log = core.getLogger()
```

The POX core serves as a central point for POX's API. The components are registered to the core, which acts as the meeting point between the components. Apart from the core, `openflow.libopenflow_01` is imported. This module contains classes and constants corresponding to elements of the OpenFlow protocol. `log` stores a reference to the logger object which is used to print out debug/info/warning messages to the terminal screen.

of\_tutorial.py

```
def launch():
    """
    Starts the component
    """
    def start_switch(event):
        log.debug("Controlling %s" % (event.connection))
        Tutorial(event.connection)
    core.openflow.addListenerByName("ConnectionUp", start_switch)
```

When POX is executed, the `launch()` functions of all the components are invoked. Here, we can see that an event listener is added. The event that is being listened to is named "ConnectionUp".

This event fires when a connection is established between the controller and the switch, i.e., after the HELLO messages are exchanged and an OpenFlow version negotiated.

The function that will be triggered on ConnectionUp event is `start_switch()`. This function logs a connection success message and then passes a reference of the event's connection object to an instance of the Tutorial class. This class defines the control logic. Let's look at its contents now:

```
_____ of_tutorial.py (class Tutorial) _____  
def __init__(self, connection):  
    # Keep track of the connection to the switch so that  
    # we can send it messages!  
    self.connection = connection  
  
    # This binds our PacketIn event listener  
    connection.addListener(self)  
  
    # Use this table to keep track of which ethernet  
    # address is on which switch port (keys are MACs,  
    # values are ports).  
    self.mac_to_port = {}
```

As you can see, the code on this file is pretty self-explanatory. The connection object that was passed while instantiating Tutorial, is used to add an event listener for the event of Packet-In. The handler is defined in the class itself. There is also an empty dictionary defined: `mac_to_port`. This is of no use for a hub, but we will be using this later when we modify the code to make the switch act like an L2 learning switch.

```
_____ of_tutorial.py (class Tutorial) _____  
def resend_packet(self, packet_in, out_port):  
    """  
    Instructs the switch to resend a packet that it had  
    sent to us. "packet_in" is the ofp_packet_in object  
    the switch had sent to the controller due to a  
    table-miss.  
    """  
  
    msg = of.ofp_packet_out()  
    msg.data = packet_in  
  
    # Add an action to send to the specified port  
    action = of.ofp_action_output(port = out_port)  
    msg.actions.append(action)  
  
    # Send message to switch  
    self.connection.send(msg)
```

The `resend_packet()` function is used to send messages to the switch. This is the messenger part

of the controller. `of.ofp_packet_out()` returns an instance of the packet-out message type. Note that only packet-out is used and is not accompanied by flow-mod here. The data of the message is replicated from the `packet_in` object. Lastly, before sending the message, an action is created and appended to the message. The action used here is output via the `out_port`.

Now you can see why the **iperf** results seen earlier, are so poor. Since there are no flow entries installed in the switch's flow-table, all packets have to go to the controller and only after the switch receives the corresponding packet-out message, it floods the ports.

```
_____ of_tutorial.py (class Tutorial) _____
def act_like_hub (self, packet, packet_in):
    """
    Implement hub-like behavior -- send all packets to
    all ports besides the input port.
    """

    self.resend_packet(packet_in, of.OFPP_ALL)
```

This function is self-explanatory. Go through the `resend_packet()` explanation to understand how it works. ( Note: We can use `of.OFPP_FLOOD` instead of `of.OFPP_ALL`. These are just constants which hold special port numbers. )

Now let's look at the handler for packet-in event.

```
_____ of_tutorial.py (class Tutorial) _____
def _handle_PacketIn (self, event):
    """
    Handles packet in messages from the switch.
    """

    packet = event.parsed # Parsed packet data.
    if not packet.parsed:
        log.warning("Ignoring incomplete packet")
        return

    packet_in = event.ofp # Actual ofp_packet_in message

    # Comment out the following line and uncomment the
    # one after when starting the exercise.
    self.act_like_hub(packet, packet_in)
    #self.act_like_switch(packet, packet_in)
```

Firstly, we did not see any `addListener()` listening to a `PacketIn` event - then how is the listener added to core? There is a shortcut provided, so that you do not have to add in every event listener by yourself. The trick is to start the method names with an underscore. In this instance, `_handle_PacketIn()` is added as the handler for the `PacketIn` event.

All this function does here is that it sends the parsed packet and the actual `packet_in` message to `act_like_hub()` for processing.

You must have noticed another function defined in the source code named `act_like_switch()`. This is what we will be making functional by adding in necessary code.

It is better not to work on the original file, instead make a copy of it in the same location. We have named ours: `of_tutorial_modified.py`.

Let us list the things we need to take care of, to make the switch act like an L2 learning switch.

1. In `_handle_PacketIn()`, comment out or remove the following:

```
self.act_like_hub(packet, packet_in)
```

and un-comment the following:

```
self.act_like_switch(packet, packet_in)
```

2. `act_like_switch` is where we need to add the main logic. There are a number of things we need to address here.
  - (a) First, un-comment the function
  - (b) Map the source MAC address of the packet to the `in_port`, making use of the `mac_to_port` dictionary. Also, update the dictionary if network configuration changes and port to MAC mappings change.
  - (c) *If* the port associated with the destination is known, send a packet-out message to the switch with the action set to output via concerned port.  
*Else*, flood the ports with the packet.
  - (d) Test the code with `pingall`. If no errors occur, change the message type from `packet_out` to `flow_mod`. This is essential because an L2 learning switch needs to have its flow tables populated, this will improve the bandwidth as the number of table-miss will be significantly less
3. Finally, test the code. Use `tcpdump` to note that not all packets are flooded every time. Use `dpctl` on the switch to take note of the flow rules being installed in the switch.

So, let us start. We already have the modified file opened. First let's remove `act_like_hub` and un-comment `act_like_switch`. So, the handler looks like this:

```

(1) of_tutorial_modified.py
def _handle_PacketIn (self, event):
    """
    Handles packet in messages from the switch.
    """

    packet = event.parsed # Parsed packet data.
    if not packet.parsed:
        log.warning("Ignoring incomplete packet")
        return

    packet_in = event.ofp # Actual ofp_packet_in message

    self.act_like_switch(packet, packet_in)

```

Now, whenever a PacketIn event occurs, the handler will delegate the processing to `act_like_switch` function.

Next, we will remove the two lines which comment out the function. (2a)

To facilitate the MAC learning part, we need to store the address and in-port as key-value pairs in the dictionary. Before that, we can extract the source & destination MAC addresses and the in-port as well.

```

(2b) of_tutorial_modified.py
def act_like_switch (self, packet, packet_in):
    source_mac = str(packet.src)
    destination_mac = str(packet.dst)
    in_port = packet_in.in_port
    out_port = of.OFPP_FLOOD #since, default is to flood

    if source_mac not in self.mac_to_port or \
        self.mac_to_port[source_mac] != in_port:

        self.mac_to_port[source_mac] = in_port
        log.debug("Updating mac_to_port: %s -> %s", \
            source_mac, self.mac_to_port[source_mac])

```

Here, we first check whether or not the source MAC address is already present in the dictionary. If it is not present or the mapping of the MAC to port-number has changed, we add/update the value in the dictionary. Along with it, we log a debug message stating the change in the dictionary.

Next, we need to check whether the destination MAC is present in the dictionary. If it is, we will set `out_port` to the value found in the dict.

```
_____ (2c) of_tutorial_modified.py _____  
if destination_mac in self.mac_to_port:  
    out_port = self.mac_to_port[destination_mac]
```

For now we do not need to add the else clause, that is because we had initialized the `out_port` with `OFPP_FLOOD`. Nevertheless, we can add a log message in the else clause stating that a new MAC address is found and that the packet will now be flooded.

```
_____ (2c) of_tutorial_modified.py _____  
if destination_mac in self.mac_to_port:  
    out_port = self.mac_to_port[destination_mac]  
else:  
    log.debug("New MAC (%s) detected, hence flooding", \  
            destination_mac)  
  
self.resend_packet(packet_in, out_port)
```

This program should now be functional. It is best to test the program out at this point.

We run POX with the modified python module now.

```
_____ Terminal (POX) _____  
$ cd ~/pox  
$ ./pox.py log.level --DEBUG misc.of_tutorial_modified
```

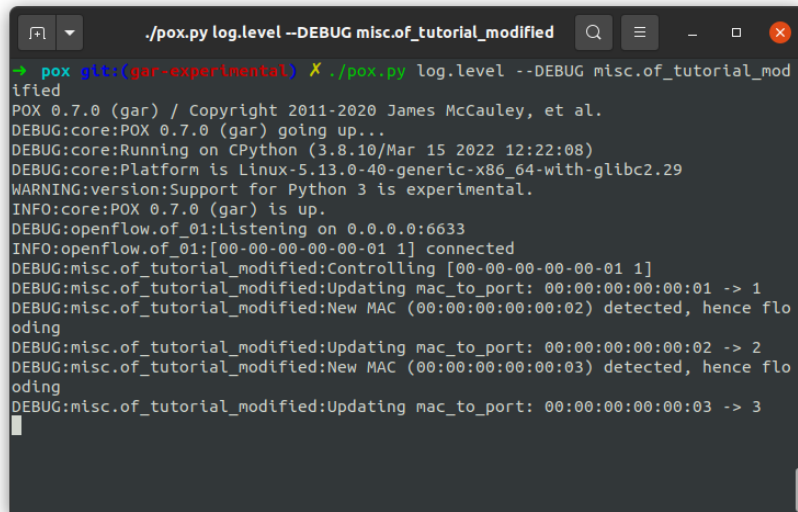
Wait for sometime before Mininet connects to the controller. It should automatically connect, but if it does not, then exit out of Mininet and re-run Mininet with the same topology and configurations as before.

Once Mininet is running and has connected to the controller, we can perform **pingall** and **iperf** to test the controller.

A terminal window titled 'sudo mn --topo single,3 --mac --switch ovsk --controller r...' showing the execution of Mininet commands. The user runs 'pingall', which tests reachability between hosts h1, h2, and h3, resulting in 0% dropped packets. Then, the user runs 'iperf' three times, testing TCP bandwidth between h1 and h3, with results ranging from approximately 20.8 to 30.6 Mbits/sec.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['26.7 Mbits/sec', '30.6 Mbits/sec']
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['24.2 Mbits/sec', '27.4 Mbits/sec']
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['20.8 Mbits/sec', '23.4 Mbits/sec']
mininet>
```

Figure A.4: Mininet connected to modified python module, running tests

A terminal window titled './pox.py log.level --DEBUG misc.of\_tutorial\_modified' showing the output of the POX controller. The output includes version information, platform details, and a series of debug messages indicating that the controller is listening on port 6633, has connected to a switch, and is updating MAC addresses for three hosts (00:00:00:00:00:01, 00:00:00:00:00:02, and 00:00:00:00:00:03).

```
POX 0.7.0 (gar) / Copyright 2011-2020 James McCauley, et al.
DEBUG:core:POX 0.7.0 (gar) going up...
DEBUG:core:Running on CPython (3.8.10/Mar 15 2022 12:22:08)
DEBUG:core:Platform is Linux-5.13.0-40-generic-x86_64-with-glibc2.29
WARNING:version:Support for Python 3 is experimental.
INFO:core:POX 0.7.0 (gar) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
DEBUG:misc.of_tutorial_modified:Controlling [00-00-00-00-00-01 1]
DEBUG:misc.of_tutorial_modified:Updating mac_to_port: 00:00:00:00:00:01 -> 1
DEBUG:misc.of_tutorial_modified:New MAC (00:00:00:00:00:02) detected, hence flooding
DEBUG:misc.of_tutorial_modified:Updating mac_to_port: 00:00:00:00:00:02 -> 2
DEBUG:misc.of_tutorial_modified:New MAC (00:00:00:00:00:03) detected, hence flooding
DEBUG:misc.of_tutorial_modified:Updating mac_to_port: 00:00:00:00:00:03 -> 3
```

Figure A.5: POX controller logging debug messages

We can see that the switch is working just as expected. The switch is not flooding packets once the dictionary is populated. To confirm this behaviour, repeat the experiment where we had opened xterm to h1, h2 and h3. Use `tcpdump` on h2 and h3 and ping from h1. You will see that, unlike

a hub, the packets aren't being flooded to all the hosts on the network. Only initially does that happen (when ARP is done), after that only the targeted host receives the packet.

This code makes the switch act like an L2 learning switch. But there is still something that we need to take care of, and that is, we need to install flows in the flow-table. The problem with the current program is that it makes the switch send all the incoming packets to the controller for processing. This takes up a lot more time and causes wastage of bandwidth. You can see that on performing **iperf** the results are still underwhelming.

Let's fix this.

To install flow-entries, we need to use the message type: flow-mod. To do this we need to call the `of.ofp_flow_mod()` function. After we get an object of the desired type, we can add the match fields from the packet that we have. We will also add some additional fields such as the timeout values and the buffer-id (if present). One of the most important parameters needs to be added to the message, i.e., the action list. We need to create and append an action to output the packet received, through the port on which the destination MAC is mapped.

```
(2d) of_tutorial_modified.py
if destination_mac in self.mac_to_port:
    out_port = self.mac_to_port[destination_mac]

    log.debug("Installing flow for %s.%d -> %s.%d", \
        source_mac, in_port, destination_mac, out_port)

    msg = of.ofp_flow_mod()

    msg.match = of.ofp_match.from_packet(packet)
    msg.idle_timeout = 30
    msg.hard_timeout = 60
    msg.buffer_id = packet_in.buffer_id
    action = of.ofp_action_output(port = out_port)
    msg.actions.append(action)
    msg.data = packet_in
    self.connection.send(msg)
else:
    log.debug("New MAC (%s) detected, hence flooding", \
        destination_mac)
    self.resend_packet(packet_in, out_port)
```

We have added all the necessary fields in the message and appended the action which outputs through the desired port. One other thing is changed, that is, the `resend_packet()` is moved inside the else clause, that is because we are already sending a flow-mod for the packets whose destination port is known.

Let's look at the complete function now.



```

(2d) of_tutorial_modified.py
def act_like_switch (self, packet, packet_in):
    source_mac = str(packet.src)
    destination_mac = str(packet.dst)
    in_port = packet_in.in_port
    out_port = of.OFPP_FLOOD #since, default is to flood

    if source_mac not in self.mac_to_port or \
        self.mac_to_port[source_mac] != in_port:

        self.mac_to_port[source_mac] = in_port
        log.debug("Updating mac_to_port: %s -> %s", \
            source_mac, self.mac_to_port[source_mac])

    if destination_mac in self.mac_to_port:
        out_port = self.mac_to_port[destination_mac]

    log.debug("Installing flow for %s.%d -> %s.%d", \
        source_mac, in_port, destination_mac, out_port)

    msg = of.ofp_flow_mod()

    msg.match = of.ofp_match.from_packet(packet)
    msg.idle_timeout = 30
    msg.hard_timeout = 60
    msg.buffer_id = packet_in.buffer_id
    action = of.ofp_action_output(port = out_port)
    msg.actions.append(action)
    msg.data = packet_in
    self.connection.send(msg)
    else:
        log.debug("New MAC (%s) detected, hence flooding", \
            destination_mac)
        self.resend_packet(packet_in, out_port)

```

Now the program is complete. To test the program, perform **pingall** and **iperf**.

```
"Node: s1" (root)
root@casper:/home/shubham#
root@casper:/home/shubham#
root@casper:/home/shubham#
root@casper:/home/shubham# dpctl dump-flows tcp:127.0.0.1:6654
stats_reply (xid=0xad88f8fb): flags=none type=1(flow)
  cookie=0, duration_sec=25s, duration_nsec=369000000s, table_id=0, priority=327
  68, n_packets=1, n_bytes=42, idle_timeout=30,hard_timeout=60,arp,d1_vlan=0xffff,
  d1_vlan_pcp=0x00,d1_src=00:00:00:00:00:02,d1_dst=00:00:00:00:00:03,nw_src=10.0.0
  .2,nw_dst=10.0.0.3,nw_proto=1,actions=output:3
  cookie=0, duration_sec=25s, duration_nsec=366000000s, table_id=0, priority=327
  68, n_packets=1, n_bytes=42, idle_timeout=30,hard_timeout=60,arp,d1_vlan=0xffff,
  d1_vlan_pcp=0x00,d1_src=00:00:00:00:00:03,d1_dst=00:00:00:00:00:01,nw_src=10.0.0
  .3,nw_dst=10.0.0.1,nw_proto=1,actions=output:1
  cookie=0, duration_sec=25s, duration_nsec=363000000s, table_id=0, priority=327
  68, n_packets=1, n_bytes=42, idle_timeout=30,hard_timeout=60,arp,d1_vlan=0xffff,
  d1_vlan_pcp=0x00,d1_src=00:00:00:00:00:01,d1_dst=00:00:00:00:00:02,nw_src=10.0.0
  .1,nw_dst=10.0.0.2,nw_proto=1,actions=output:2
  cookie=0, duration_sec=25s, duration_nsec=361000000s, table_id=0, priority=327
  68, n_packets=1, n_bytes=42, idle_timeout=30,hard_timeout=60,arp,d1_vlan=0xffff,
  d1_vlan_pcp=0x00,d1_src=00:00:00:00:00:02,d1_dst=00:00:00:00:00:01,nw_src=10.0.0
  .2,nw_dst=10.0.0.1,nw_proto=1,actions=output:1
  cookie=0, duration_sec=25s, duration_nsec=360000000s, table_id=0, priority=327
  68, n_packets=1, n_bytes=42, idle_timeout=30,hard_timeout=60,arp,d1_vlan=0xffff,
  d1_vlan_pcp=0x00,d1_src=00:00:00:00:00:03,d1_dst=00:00:00:00:00:02,nw_src=10.0.0
```

Figure A.6: Flow entries found on switch S1

After performing **pingall**, we find that the switch's flow-table has been populated with entries.

```
sudo mn --topo single,3 --mac --switch ovsk --controller r...
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['26.8 Gbits/sec', '26.8 Gbits/sec']
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['24.8 Gbits/sec', '24.9 Gbits/sec']
mininet>
```

Figure A.7: Much improved bandwidth

Upon performing **iperf**, we can see that the performance has improved significantly. Earlier the bandwidth would reach a maximum of 50 Mbits/sec but now, after installing flow entries, they reach upto 25 Gbits/sec.

## A.2 Experiment 2

For this experiment, let's recreate David Mahler's example of a full-mesh topology [25]. For a custom topology: you can either build from scratch, or you can create a custom topology file which will then be used in the `--custom` option of `mn`.

### A.2.1 Custom Topology Script for `mn`

First, let's have a look at the custom topology script provided by Mininet. All custom topology scripts are conventionally stored in the `~/mininet/custom` directory.

An example is provided by Mininet. You will find a file named: `topo-2sw-2host.py`

```
topo-2sw-2host.py
"""Custom topology example

Two directly connected switches plus a host for each
switch:

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

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

from mininet.topo import Topo

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

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

        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's3' )
        rightSwitch = self.addSwitch( 's4' )

        # Add links
        self.addLink( leftHost, leftSwitch )
        self.addLink( leftSwitch, rightSwitch )
        self.addLink( rightSwitch, rightHost )
```

```
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

The code is self-explanatory. To execute this script we run Mininet with the following options:

```
sudo mn --custom=/home/username/mininet/custom/topo-2sw-2host.py --topo=mytopo
```

We will use this script as the base to create the mesh topology.

First, create a file in the custom directory named: *mesh.py*

```
mesh.py
"""Mesh topology

A full mesh of four switches plus two hosts, one
connected to s1 and the other connected to s3:

          switch(s2) --- switch(s3) --- host(h2)
              |         x         |
          host(h1) --- switch(s1) --- switch(s4)

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

from mininet.topo import Topo
from mininet.node import OVSSwitch

class MeshTopo( Topo ):
    "Create a mesh topology."

    def addLinksFullMesh( self ):
        "Add links between all pairs of switches."

        switches = self.switches()

        for i in range(len(switches)):
            for j in range(i+1, len(switches)):
                self.addLink(switches[i], switches[j])

    def build( self ):
        "Create custom topo."
```

```

# Add hosts and switches
h1 = self.addHost( name='h1' )
h2 = self.addHost( name='h2' )
s1 = self.addSwitch( name='s1', \
    protocols='OpenFlow13' )
s2 = self.addSwitch( name='s2', \
    protocols='OpenFlow13' )
s3 = self.addSwitch( name='s3', \
    protocols='OpenFlow13' )
s4 = self.addSwitch( name='s4', \
    protocols='OpenFlow13' )

# Add links between switches
self.addLinksFullMesh()

# Add links between hosts and switches
self.addLink( node1=h1, node2=s1 )
self.addLink( node1=h2, node2=s3 )

topos = { 'meshtopo': ( lambda: MeshTopo() ) }

```

To create the topology, we modified the file to add four switches in a fully connected mesh and added two hosts.

Before we run the network scenario, we need to make sure that we use a topology aware SDN controller. Just because we have an SDN controlled network doesn't mean that we don't have the same old concerns regarding switch network loops and broadcast storms. Floodlight controller's topology and forwarding modules together detect and prevent packets from travelling in a loop. Floodlight controller discovers links and how the nodes are connected. As a result it gets to store the full picture of the topology. Floodlight then builds a loop free tree from the topology graph for broadcast, as a result it deals with the problem of looping.

So in order to do that, you need to download <https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview> floodlight controller either locally and build it from source or download it's <https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/8650780/Floodlight+VMVM> disk image. The latter is used for this example.

Open a terminal and SSH into the floodlight VM [*username: floodlight; password: floodlight*]. Note the IP address of interface connected to the host machine [you can use **ifconfig** on the VM]. Now run the floodlight controller.

```

Terminal I
floodlight@floodlight:~$ cd floodlight
floodlight@floodlight:~$ java -jar target/floodlight.jar

```

**Note:** If you get any errors about target/floodlight.jar missing, run the command: **ant**. This will build the project and you will have target/floodlight.jar created.

Once the controller is up and running, use another terminal to run Mininet. Make sure you use a remote controller, with the IP address of the interface of the floodlight VM connected to the host machine and the port number (which is 6653 by default). Also, it is essential to pass *mesh.py* in the `--custom` option and pass *meshtopo* in the `--topo` option.

```
Terminal II
$ sudo mn --controller=remote,ip=192.168.56.3,port=6653
--custom mesh.py --topo meshtopo

# Make sure you use the IP address of your floodlight vm
```

You will now have Mininet running. You can check the network for the nodes and their links (using **dump**, **links** and **net**). To test for connectivity, try using **ping** between the two hosts. You will initially see requests getting timed-out, that is because the floodlight controller takes some time to figure out the network and create a spanning tree. But once it has figured it out, it will do away with the looping issue and you will see that h1 and h2 can communicate with one another.

## A.2.2 Standalone Python Script

We can create the same topology in an independent python script, i.e., without using it as a custom topology script with **sudo mn**.

```
mesh_topo.py
#!/usr/bin/env python3

from mininet.net import Mininet
from mininet.node import RemoteController, OVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info

def createMeshNetwork():

    'Create a mesh of switches and use remote controller'

    info( '*** Creating network\n' )
    net = Mininet( controller=RemoteController, switch=OVSSwitch )

    # Replace ip with the IP of your floodlight vm
    info( '*** Connecting to remote controller at: 192.168.56.3:6653\n' )
    c0 = net.addController(name='c0',ip='192.168.56.3', port=6653)
```

```

info( '*** Adding hosts:\n' )
h1 = net.addHost( name='h1', ip='10.0.0.1')
h2 = net.addHost( name='h2', ip='10.0.0.2')
info ( h1.name, h2.name, '\n' )

info( '*** Adding switches:\n' )
s1 = net.addSwitch(name='s1',protocols='OpenFlow13')
s2 = net.addSwitch(name='s2',protocols='OpenFlow13')
s3 = net.addSwitch(name='s3',protocols='OpenFlow13')
s4 = net.addSwitch(name='s4',protocols='OpenFlow13')
info ( s1.name, s2.name, s3.name, s4.name, '\n')

switches = [s1,s2,s3,s4]

info( '*** Adding links:\n' )
for i in range(len(switches)):
    for j in range(i+1, len(switches)):
        link = net.addLink(switches[i], switches[j])
        info( '({}, {}) '.format(switches[i].name, switches[j].name) )

net.addLink(h1, s1)
net.addLink(h2, s3)
info( '({}, {}) ( {}, {})\n'.format(h1.name,s1.name,h2.name,s3.name) )

net.start()
CLI( net )
net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    createMeshNetwork()

```

Make sure that the floodlight controller is running. You can now run the script using: **sudo ./mesh\_topo.py**



# Bibliography

- [1] Floodlight Controller - Confluence. <https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview>. (Accessed: 2022-04-21).
- [2] Linux Network Namespace. <https://man7.org/linux/man-pages/man8/ip-netns.8.html>. (Accessed: 2022-05-17).
- [3] MaxiNet: Distributed Network Emulation. <https://maxinet.github.io/>. (Accessed: 2022-06-02).
- [4] Mininet: An Instant Virtual Network. <http://mininet.org/>. (Accessed: 2022-02-15).
- [5] Mininet: How Does It Work. <https://github.com/mininet/mininet#how-does-it-work>. (Accessed: 2022-02-15).
- [6] Mininet Python API: Reference Manual. <http://mininet.org/api/hierarchy.html>. (Accessed: 2022-03-28).
- [7] Mininet-WiFi - Emulator For Software Defined Wireless Networks. <https://mininet-wifi.github.io/>. (Accessed: 2022-06-03).
- [8] NOX Network Control Platform. <https://github.com/noxrepo/nox>. (Accessed: 2022-04-21).
- [9] ONOS Documentation. <https://wiki.onosproject.org/display/ONOS/ONOS>. (Accessed: 2022-04-21).
- [10] OpenFlow. <https://www.opennetworking.org/sdn-resources/openflow>. (Accessed: 2022-03-11).
- [11] OpenFlow v1.4 Switch Specification. <https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.4.0.pdf>. (Accessed: 2022-03-19).
- [12] OpenStack Documentation. <https://docs.openstack.org/>. (Accessed: 2022-04-10).
- [13] Overlay File System. <https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html>. (Accessed: 2022-05-17).
- [14] POX Documentation. <https://noxrepo.github.io/pox-doc/html/>. (Accessed: 2022-04-21).

- [15] Ryu Controller Documentation. <https://ryu.readthedocs.io/en/latest/>. (Accessed: 2022-04-21).
- [16] What is SDN and where software-defined networking is going. <https://www.networkworld.com/article/3209131/what-sdn-is-and-where-its-going.html>. (Accessed: 2022-04-10).
- [17] C. Bouras, A. Kollia, and A. Papazois. Exploring sdn & nfv in 5g using onos & pox controllers. *Int. J. Interdiscip. Telecommun. Netw.*, 10(4):46–60, oct 2018.
- [18] G. Di Lena, A. Tomassilli, D. Saucez, F. Giroire, T. Turetti, and C. Lac. DistriNet: A mininet implementation for the cloud. *SIGCOMM Comput. Commun. Rev.*, 51(1):2–9, mar 2021.
- [19] N. Feamster, J. Rexford, and E. Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, apr 2014.
- [20] O. Flauzac, E. M. Gallegos Robledo, and F. Nolot. Is mininet the right solution for an sdn testbed? In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2019.
- [21] S. Ghosh. Bluearch—an implementation of 5g testbed. *Journal of Communications*, 14:1110–1118, 12 2019.
- [22] A. Gudipati, D. Perry, L. E. Li, and S. Katti. Softran: Software defined radio access network. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN '13*, page 25–30, New York, NY, USA, 2013. Association for Computing Machinery.
- [23] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.
- [24] B. Lantz and B. O'Connor. A mininet-based virtual testbed for distributed sdn development. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15*, page 365–366, New York, NY, USA, 2015. Association for Computing Machinery.
- [25] D. Mahler. Mininet custom topologies. <https://www.youtube.com/watch?v=yHUNeyAQKWY>, November 2013. Accessed: 2022-02-02.
- [26] R. Shubbar, M. Alhisnawi, A. Abdulhassan, and M. Ahamdi. A comprehensive survey on software-defined network controllers. In R. Kumar, B. K. Mishra, and P. K. Pattnaik, editors, *Next Generation of Internet of Things*, pages 199–231, Singapore, 2021. Springer Singapore.
- [27] C. Tipantuña and P. Yanchapaxi. Network functions virtualization: An overview and open-source projects. In *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6, 2017.