Fall 2019

Modeling of OpenFlow based software defined networks using Mininet

Aishwarya kamal Maddala
amaddala@iastate.edu

Modeling of OpenFlow-Based Software Defined Networks using Mininet

By

Aishwarya kamal Maddala

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Electrical and Computer Engineering

Program of Study Committee:

Ahmed E. Kamal, Major Professor

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2019

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

ACKNOWLEDGMENTS

I would like to thank my major professor, Ahmed Kamal for his constant guidance and support throughout the course of this research. I would also like to thank him for all the resources that he provided for this research.

In addition, I would also like to thank my friends, colleagues, the department faculty and staff for making my time at Iowa State University a wonderful and a great learning experience.

I would also like to thank my family for their encouragement and support, without whom I couldn’t have accomplished my goals.

ABSTRACT

Software-defined networking (SDN) has become one of the most important architectures for the management of largescale complex networks, which may require re-policing or reconfigurations from time to time. By decoupling the control plane from data plane, SDN achieves easy re-policing. Hence, the network switches or routers forward packets simply by following the flow table rules which are set by the control plane. OpenFlow is the most popular SDN protocol or standard and has a set of design specifications. SDN or OpenFlow is a relatively new area, and it has attracted both academia and industry. This research work uses a network emulation tool called 'Mininet', for the implementation of SDN using OpenFlow. This tool allows to create the virtual switches, hosts, controllers and links. Using all these components, a network of the desired topology and scale can be created. The focus of this research is to update the labs created for the coursework in Electrical and Computer Engineering department. The tools used in the labs are outdated and are required to be changed, tested and presented. In addition, new labs with new concepts have also been added.

CHAPTER 1. INTRODUCTION

SDN has gained a lot of attention in recent years, because it enables easier and faster network innovation and addresses the lack of programmability in existing networking architectures. It is an emerging network architecture where network control is decoupled from forwarding and is directly programmable. The migration has taken place from formerly tightly bound in individual network devices, into accessible computing devices. This enables the underlying infrastructure to be abstracted for applications and network services, which can treat the network as a virtual or logical entity.

Network intelligence is logically centralized in software based SDN controllers, which maintain a global view of the network. This allows the applications and policy engines to view the network as a single and logical switch. With SDN, enterprises and carriers gain vendor-independent control over the entire network from a single logical point, which greatly simplifies the network design and operation. The network devices are also greatly simplified with SDN, since they no longer need to understand and process thousands of protocol standards but merely accept instructions from the SDN controllers.

1.1 OVERVIEW

1.1.1 SOFTWARE DEFINED NETWORKING

Figure 1.1 illustrates the SDN framework, which consists of three layers. The bottom layer is the infrastructure layer, and is also called the data plane. It comprises the forwarding network elements. The responsibilities of the forwarding plane are mainly monitoring local information, as well as data forwarding and gathering statistics. One layer above is the control layer, also called the control plane. It is responsible for programming and managing the forwarding plane. To that

end, it makes use of the information provided by the forwarding plane and defines network operation and routing. It comprises one or more software controllers that communicate with the forwarding network elements through standardized interfaces, which are referred to as southbound interfaces.

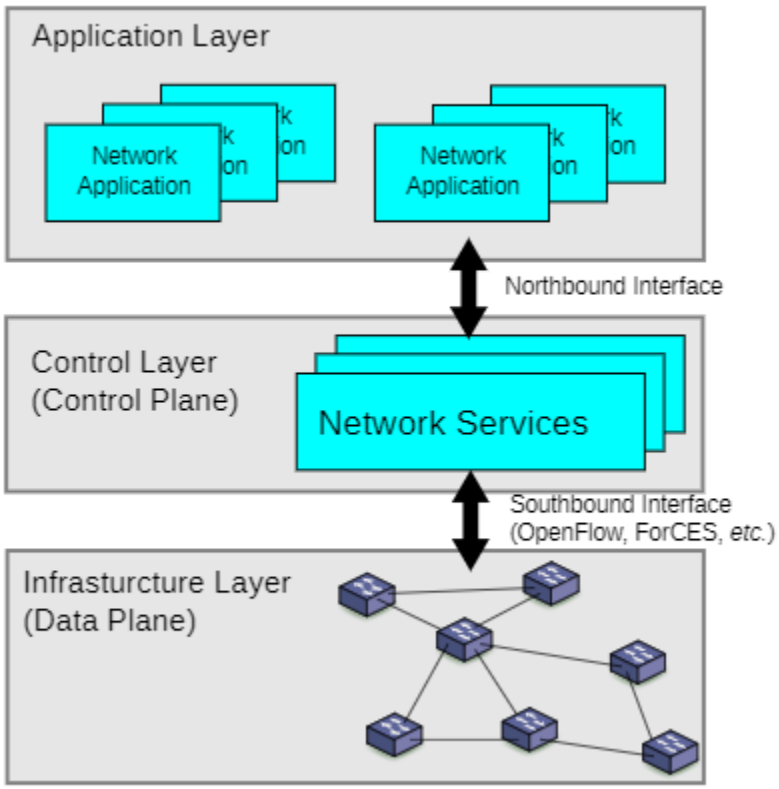


Figure 1.1: A three-layer SDN architecture

1.1.2 OPENFLOW PROTOCOL

For the southbound interface of SDN, the OpenFlow protocol is the most commonly used protocol which separates the data plane from the control plane. The OpenFlow architecture consists of three basic concepts. The first one is that the network is built up by OpenFlow-compliant switches that

compose the data plane; Then, the control plane consists of one or more OpenFlow controllers; The last one is a secure control channel connects the switches with the control plane. An OpenFlow-compliant switch is a basic forwarding device that forwards packets according to its flow table. This flow table holds a set of table entries, each of which consists of match fields, instructions and counters. These entries are also called flow rules or flow entries. The “header fields” in a flow table entry describe to which packets this entry will be applicable. These consist of a wildcard-capable match over specified header fields of packets. To allow fast packet forwarding with OpenFlow, the switch requires ternary content addressable memory (TCAM) that allows the fast lookup of wildcard matches.

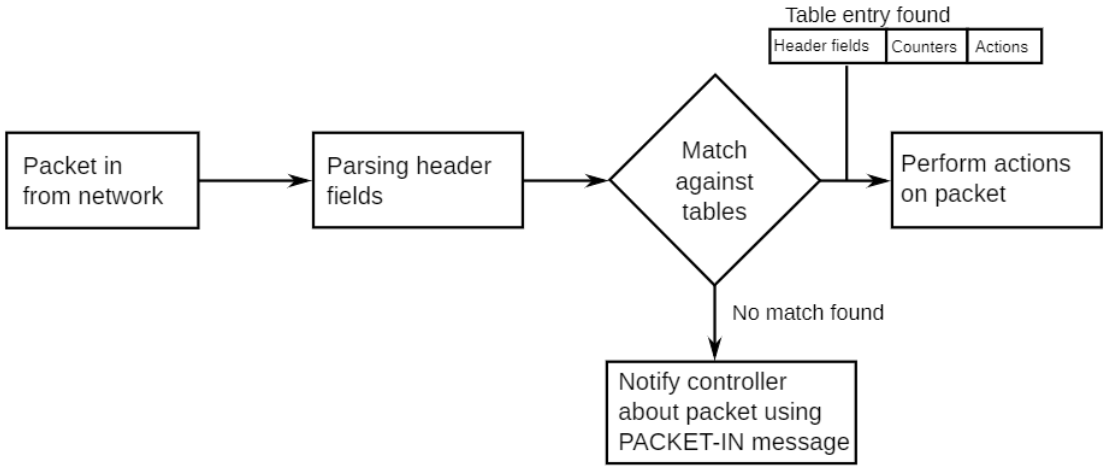


Figure 1.2: Basic packet forwarding with openflow in a switch

The basic packet forwarding mechanism with OpenFlow is illustrated in Figure 1.2. When a switch receives a packet, it analyzes the packet header, which is then matched against the flow table. If there is a match found with the header field wildcard, then the flow table entry is considered. If several such entries are found, packets are matched based on prioritization, i.e., the most specific entry or the wildcard with the highest priority is selected. Then, the switch updates the counters of

that particular flow table entry. Finally, the switch performs the actions specified by the flow table entry on the packet, e.g., the switch forwards the packet to a port. Otherwise, if no flow table entry matches the packet header, the switch generally notifies its controller about the packet, which is buffered when the switch is capable of buffering. To that end, it encapsulates either the unbuffered packet or the first bytes of the buffered packet using a PACKET-IN message and sends it to the controller; it is common to encapsulate the packet header and the number of bytes defaults to 128. The controller that receives the PACKET-IN notification identifies the correct action for the packet and installs one or more appropriate entries in the requesting switch. Buffered packets are then forwarded according to these rules; this is triggered by setting the buffer ID in the flow insertion message or in explicit PACKET-OUT messages. Most commonly, the controller sets up the whole path for the packet in the network by modifying the flow tables of all switches on the path.

1.2 REFERENCES

[1]. Lara, A.; Kolasani, A.; Ramamurthy, B. Network Innovation Using OpenFlow: A Survey.IEEE Commun. Surv. Tutor. 2013,16, 1–20.

[2]. Astuto, B.N.; Mendonça, M.; Nguyen, X.N.; Obraczka, K.; Turletti, T.A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. IEEE Commun. Surv. Tutor. 2014 , doi:10.1109/SURV.2014.012214.00180.

[3]. Jain, R.; Paul, S. Network Virtualization and Software Defined Networking for Cloud Computing: A Survey. IEEE Commun. Mag. 2013, 51, 24–31.

[4]. Rexford, J.; Freedman, M.J.; Foster, N.; Harrison, R.; Monsanto, C.; Reitblatt, M.; Guha, A.; Katta, N.P.; Reich, J.; Schlesinger, C. Languages for Software-Defined Networks. IEEE Commun. Mag.2013,51, 128–134.

CHAPTER 2. INTRODUCTION TO MININET AND OPENFLOW

2.1 ABSTRACT

SDN (Software Defined Networking) is the future in networking evolution. The more commonly utilized protocol in SDN is OpenFlow protocol. It is proposed to make the way that a controller communicates characteristics of SDN to be consistent. The light-weight approach of using OS-level virtualization features, including processes and network namespaces, allows it to scale to hundreds with network devices in an SDN architecture. It is a system for rapidly prototyping large networks on the constrained resources. Mininet is a network modelling tool used to investigate the behavioral of nodes for software defined networks. This paper provides a tutorial on network emulation using Mininet and its support for OpenFlow.

2.2 INTRODUCTION

In recent years, the design of software defined data centers has received increased attention as a means of increasing the deployment speed for new applications. Since both storage virtualization and server are relatively mature, a key facilitator for this perspective is the creation of software-defined networks. The objectives of SDN include the ability to introduce network innovations faster and to radically simplify and automate the management of large networks. The notions provided by SDN can standardize traffic routing mechanisms and potentially reduce management complexity, which contribute to better network reliability, serviceability, and availability. Moreover, introducing a centralized SDN network controller enables end-to-end network visibility

and faster reconfiguration in response to disasters. This paper considers the use of Mininet to model the behavior of SDN networks. It provides a brief tutorial on the installation and use of Mininet in a virtual machine (VM) environment, run several applications and use Wireshark to look at packet flows including OpenFlow packets. Scalability is discussed, both in terms of host server limitations and virtual network response times. Also, the structure for using this approach to create and simulate SDN networks for both academic and research purposes is discussed.

2.3 MININET

Mininet is an emulation platform for networks, including hosts, switches, controllers, and network applications. It reproduces a network by running the network code on virtual hardware. That is, it creates a realistic network by running real kernel, switch and application code. Figure 2.1 shows the emulating hardware network using Mininet. It was created in python and a Python API for user customization is provided. The Mininet hosts run standard Linux operating systems, and Mininet switches running Linux can support OpenFlow. It can run in a VM based on Linux Ubuntu server, and using process-based virtualization it can create several virtual host and network instances on a single operating system. Mininet includes an OpenFlow-aware command line interface, which supports a basic set of network topologies and allows highly flexible custom topologies.

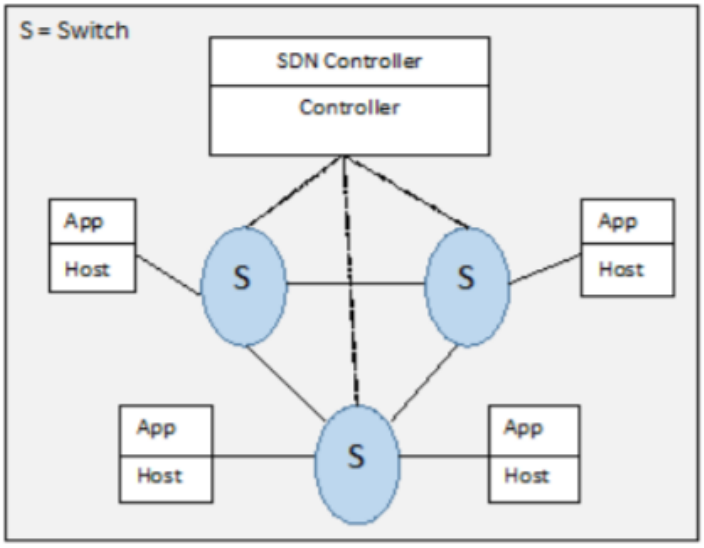


Figure 2.1: Emulating hardware network using Mininet

This paper experimented with Mininet 2.2.2 running on Ubuntu Linux on Virtual box VM. Also, an X server called Xming is installed to run Wireshark.

2.3.1 WALKTHROUGH

To run Mininet as root it must be run using the sudo command to run the mininet. This command is used to see the help menu available on the Mininet

$ sudo mn -h

To view the control traffic using the OpenFlow Wireshark dissector, firstly wireshark is opened in the background using the command:

$ sudo wireshark &

Wireshark captures packets on the Mininet VM's loopback interface.

The following command starts the minimal topology and enters the command line interface. It is the default topology and includes OpenFlow switch which is connected to two hosts and the OpenFlow controller.

$ sudo mn

The following command displays the nodes available in the current network which are available for the mininet default minimal topology.

mininet > nodes

The following command displays the dump information about all nodes available in the current mininet network.

mininet > dump

The below command tests the connectivity between host h1 and h2. This command will keep checking the connectivity between hosts until we stop the command.

mininet > h1 ping h2

The following command checks for the connection between host h1 and h2 for one packet.

mininet > h1 ping –c 1 h2

The below command is used in order to clear mininet or previously used command.

$ sudo mn –c

Mininet consists of default topologies such as minimal, single, reversed, linear and tree. Network controller is denoted by C0, switches are denoted by S1 to Sn and hosts are denoted by H1 to Hn.

2.3.2 VIRTUAL NETWORKS IN MININET

Custom topologies can be easily defined in Mininet using a simple Python API, for example; "custom/topo-2sw-2host.py" is a python script which connects two switches directly, with a single host off each switch as shown below in Figure 2.2.

```
from mininet.topo import Topo

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

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

        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's1' )

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

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

Figure 2.2: Text file for virtual network written in Python

In this case, components are added using the script:

leftHost = self.addHost('h1')

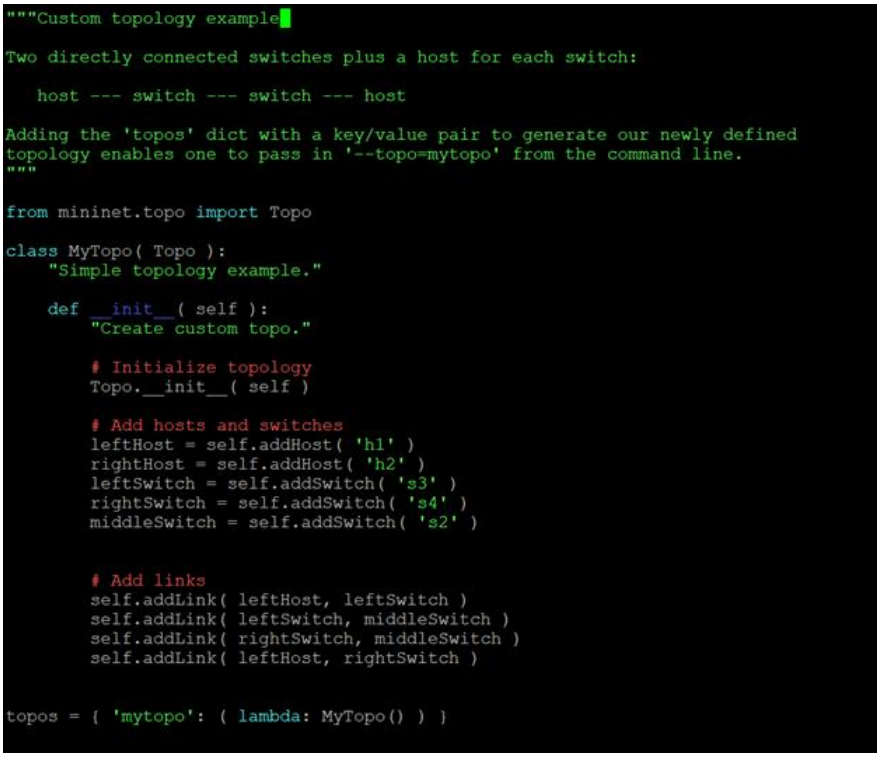
leftSwitch = self.addSwitch('s1')

And connected to each other by a link using the script:

self.addLink(leftHost, leftSwitch)

This text file is then saved and runs using the command:

$ sudo mn –custom ~/mininet/custom/topo-2sw-2host.py –topo my topo –test pingall



```
"""Custom topology example

Two directly connected switches plus a host for each switch:

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

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

from mininet.topo import Topo

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

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

        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's3' )
        rightSwitch = self.addSwitch( 's4' )
        middleSwitch = self.addSwitch( 's2' )


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


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

Figure 2.3: Custom topology with 3 switches

This custom topology is experimented with 3 switches connected in linear topology, and one host is connected to each of the two switches at the end as shown in above Figure 2.3.

2.4 CONCLUSION

The growing popularity of SDN within cloud computing environments and related applications has led to an interest in modeling the behavior of SDN network configurations. The increase of security, mobile devices, virtualization, efficient BigData management and high quality of services has lead SDN as a promising architecture. It magnifies the scope of development in networking. This paper has demonstrated the use of Mininet to simulate an SDN network. It discussed how to install Mininet in a virtual machine, and provided some common useful commands, and discussed how to capture OpenFlow message exchanges on the network.

2.5 REFERENCES

[1] F. Alam, I. Katib and A. S. Alzahrani. "New Networking Era: Software Defined Networking". International Journal of Advanced Research in Computer Science and Software Engineering.Volume 3, Issue 11, November 2013

[2] Open Networking Foundation. "OpenFlow /Software Defined-networking (SDN)". Available at: http://www. www.opennetworking.org/.

[3] Introduction to Mininet at https://github.com/ mininet/mininet.wiki.git/

[4] D. Kumar and M. Sood. "Software Defined Networking: A Concept and Related Issues" in Int. J. Advanced Networking and Applications. Volume: 6 Issue: 2 Pages: 2233-2239 (2014).

[5] Mininet Commands at http://mininet.org/

[6] B. Lantz, B. Heller, N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks". In HotNets. ACM, 2010.

[7] SDN Hub. (2016, April 20). Useful Mininet setup. [Online]. Available: http://sdnhub.org/resources/useful-mininetsetups/

[8] D. Drutskoy, E. Keller, and J. Rexford. "Scalable Network Virtualization in Software-Defined Networks". IEEE Internet Computing, 2013

[9] Mininet Topologies at http://www.routereflector.com/2013/11/ mini netas-an-sdn-testplatform

[10] Wireshark at https://www.wireshark.org/

CHAPTER 3. OPENFLOW CONTROLLERS AND MINIEDIT

3.1 ABSTRACT

Software defined networking (SDN) promises a way to more flexible networks that can adapt to changing demands. At the same time these networks should also benefit from simpler management mechanisms. In the OpenFlow architecture the operating system is represented by the OpenFlow controller. As the key component of the OpenFlow ecosystem, the behavior and performance of the controller are significant for the entire network. This paper explains in brief the OpenFlow controllers and discuss in detail the working of Open vSwitch (ovs) controller and its use in Mininet. This paper also discusses about a network simulator's graphical user interface called MiniEdit.

3.2 INTRODUCTION

SDN has emerged as a new standard of networking that enables network operators, vendors, and even third parties to establish and create new capabilities at a faster pace. This SDN prototype shows potential for all domains of users. It played an important role in increasing the capabilities of traditional networking system. Software Defined Networking provides some great features that allow the network providers and administrators to act as fast as possible to access, interchange and update any system easily. Mininet is the most widely used open source networking Simulator. It can construct a vast network with the collection of networking elements such as switches, end-hosts, routers based on Linux kernel. Complicated network topology can be designed to virtualize using Mininet.

An OpenFlow Controller is a variety of SDN Controller that uses the OpenFlow Protocol. It uses the OpenFlow protocol to connect and configure the network devices like routers, switches, etc., to determine the best path for application traffic. To meet the changing needs, SDN controllers can simplify the network management, modify the network flows and handle all the communications between applications and devices. Administrators can manage network traffic at a more granular level and more dynamically by implementing network control plane in software rather than in firmware. An SDN Controller relays information to the switches/routers (via southbound APIs) and the applications and business logic (via northbound APIs). Most popular examples for developing SDN controllers are OVS, Open Daylight, NOX, POX, Floodlight, and Beacon, Ryu and Pyretic. Ryu Controller is an open, software-defined networking (SDN) Controller designed to increase the alertness of the network by making it easy to manage and adapt how traffic is handled. Pyretic controller is Python based controller that works on the control layer of SDN. Controllers are distinct from the switches in SDN. This separation of the control from the forwarding allows more sophisticated traffic management. OpenFlow communication protocol gives access to the forwarding plane of a network switch or router over the network.

This paper discusses the Open vSwitch (ovs) OpenFlow controller and its use to control networks emulated by Mininet. It also discusses about the web-based graphical tool called MiniEdit, which is used to build custom networks.

3.3 OVS CONTROLLER

Open vSwitch is a software implementation of a virtual multilayer network switch, designed to enable effective network automation through programmatic extensions. In addition to this, it is also designed to assist transparent distribution over several physical servers by authorizing creation

of cross-server switches in a way that withdraws out the underlying server architecture, which is similar to the VMware vNetwork distributed vswitch or Cisco Nexus 1000V. This controller can operate both as a control stack for dedicated switching firmware and also as an software based network switch running within a virtual machine hypervisor; as a result, it has been ported to multiple virtualization platforms, networking hardware accelerators, and switching chipsets. Open vSwitch has also been desegregated into several cloud computing software platforms and virtualization management systems, which include OpenStack, openQRM, OpenNebula and oVirt. The attributes provided by ovs controller include: complete IPv6 support, remote configuration protocol with existing bindings for the C and Python programming languages, implementation of packet forwarding engine in user space or kernel space, allowing additional flexibility as well as providing performance improvements by processing the most number of forwarded packets without leaving the kernel space and by using multithreaded kernel space and user space components, etc.

3.3.1 MININET AND OVS CONTROLLER

One of Mininet's key features is that it makes it very simple to establish a complete virtual network including switches, hosts, links and OpenFlow controllers. By default, Mininet runs Open vSwitch in OpenFlow mode, which requires an OpenFlow controller. Mininet comes with built-in Controller () classes to support various controllers, including the OpenFlow reference controller (**controller**), Open vSwitch's **ovs-controller**, and the now-deprecated NOX Classic. This paper allows to inspect flow tables entries in the switches with ovs-controller.

The network using ovs-controller with 3 switches and 3 hosts, each host connected to one switch can be started in Mininet using the following command;

$ sudo mn –mac –controller=ovsc, ip=Controller-IP –topo=linear,3

With Controller-IP being the vboxnet() interface IP number, e.g., 192.168.56.3

In Mininet, ‘net’ command can be used to find out which ports from switches are connected to which ports in hosts. The flow tables can be looked up using the following commands:

Mininet > dpctl dump-flows

Mininet > sh ovs-ofctl dump-flows s1

The OF packets can be seen in the middle pane of Wireshark and contents of the OF packets can also be observed. Mininet can be exited and VM can be cleared using the following command:

$ sudo mn –c

3.4 MINIEDIT

The Mininet network simulator includes MiniEdit, a simple GUI editor for Mininet. MiniEdit is an experimental tool created to indicate how Mininet can be enlarged. It is an illustrative GUI-based application written with Mininet's Python API. It is a simple network editor that allows you to drag and drop switches and hosts, wire them up, and create a live, usable network by pressing the "run" button. This paper discusses on how to create a network using MiniEdit and how to observe the flow tables in Mininet. MiniEdit opens in an xterm like Wireshark.

3.4.1 MINIEDIT USER INTERFACE

The MiniEdit script is located in Mininet’s examples folder, and to run it, the following command can be executed:

$ sudo ~/mininet/examples/miniedit.py

Mininet needs to run in root privileges so that, MiniEdit is started with the sudo command. MiniEdit has a very simple user interface that shows a canvas with the row of tool icons on the left side of the window, and a menu bar along the top of the window as shown in Figure 3.1.

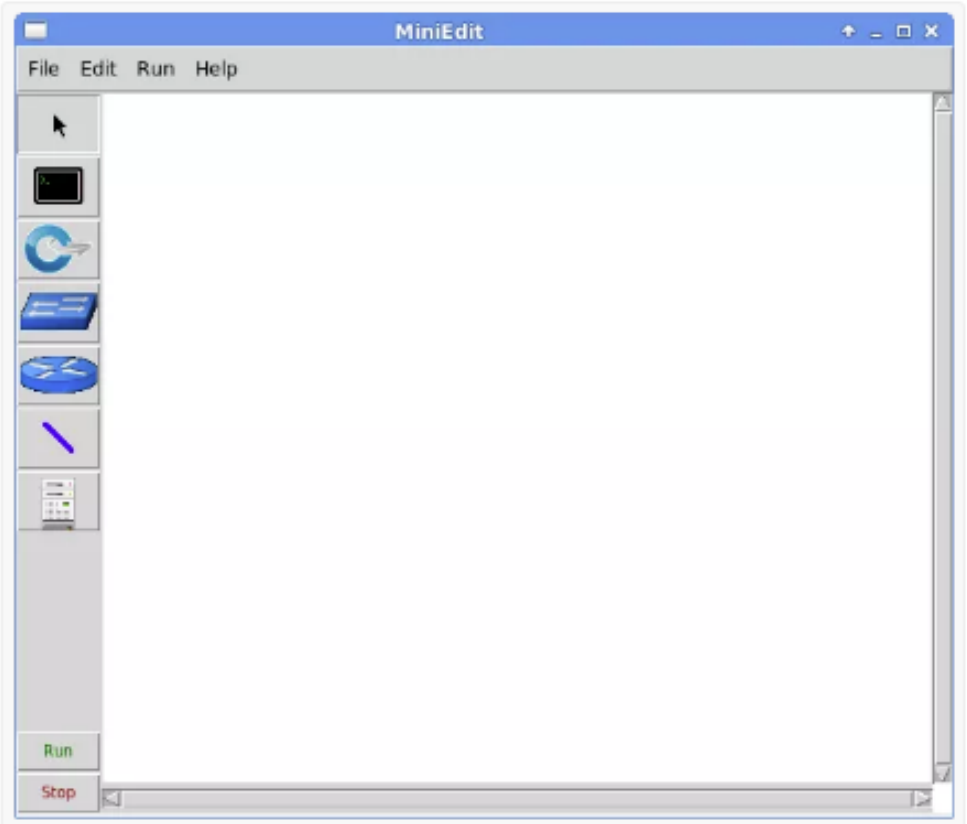


Figure 3.1: MiniEdit canvas

The icons represent the following tools:


This *Select* tool is used to move nodes around on the canvas. Click and drag any existing node. Also, this tool is not needed to select a node or link on the canvas. To select an existing node

or link, just moving the mouse pointer over it works regardless of the tool that is currently active and then right-click to reveal a configuration menu for the selected element or the *Delete* key is used to remove the selected element.

 The *Host* tool creates nodes on the canvas that will perform the function of host computers. Click on the tool, then click anywhere on the canvas in order to place a node. As long as the tool remains selected, hosts can be added by clicking anywhere on the canvas. The user may configure each host by right-clicking on it and choosing *Properties* from the menu.

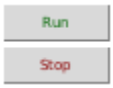
 The *Switch* tool creates OpenFlow-enabled switches on the canvas. These switches are expected to be connected to a controller. The tool operates the same way as the *Hosts* tool above. The user may configure each switch by right-clicking on it and choosing *Properties* from the menu.

 The *Legacy Switch* tool creates a learning Ethernet switch with default settings. The switch will operate independently, without a controller. The *legacy switch* cannot be configured and is set up with Spanning Tree disabled, so legacy switches cannot be connected in a loop.

 The *Legacy Router* tool creates a basic router that will operate independently, without a controller. It is basically just a host with IP Forwarding enabled. The *legacy router* cannot be configured from the MiniEdit GUI.

 The *NetLink* tool creates links between nodes on the canvas. Links can be created by selecting the *NetLink* tool, then clicking on one node and dragging the link to the target node. The user may configure the properties of each link by right-clicking on it and choosing *Properties* from the menu.

 The *Controller* tool creates a controller. Multiple controllers can be added. By default, the MiniEdit creates a Mininet OpenFlow reference controller, which implements the behavior of a learning switch. Other controller types can be configured. The user may configure the properties of each controller by right-clicking on it and choosing *Properties* from the menu.



The *Run* starts Mininet simulation scenario currently displayed in the MiniEdit canvas. The *Stop* button stops it. When MininEdit simulation is in the "Run" state, right-clicking on network elements reveals operational functions such as opening a terminal window, viewing switch configuration, or setting the status of a link to "up" or "down".

3.4.2 CUSTOM NETWORK TOPOLOGY USING MINIEDIT

The hosts, switches and controllers can be selected from the left plane and can be dropped to the right plane in order to form the required network. The links are added between nodes in the canvas. This can be done by connecting a host to a switch, or a switch to switch and connecting every host to at least one switch. Also, the switches are connected to create a network and each switch is connected to one of the controllers. The screenshot of these connections together is shown below in Figure 3.2. MiniEdit allows to create this complex custom network topology in a few minutes,

however, manually writing a Mininet Custom topology script to create this scenario can take a lot longer.

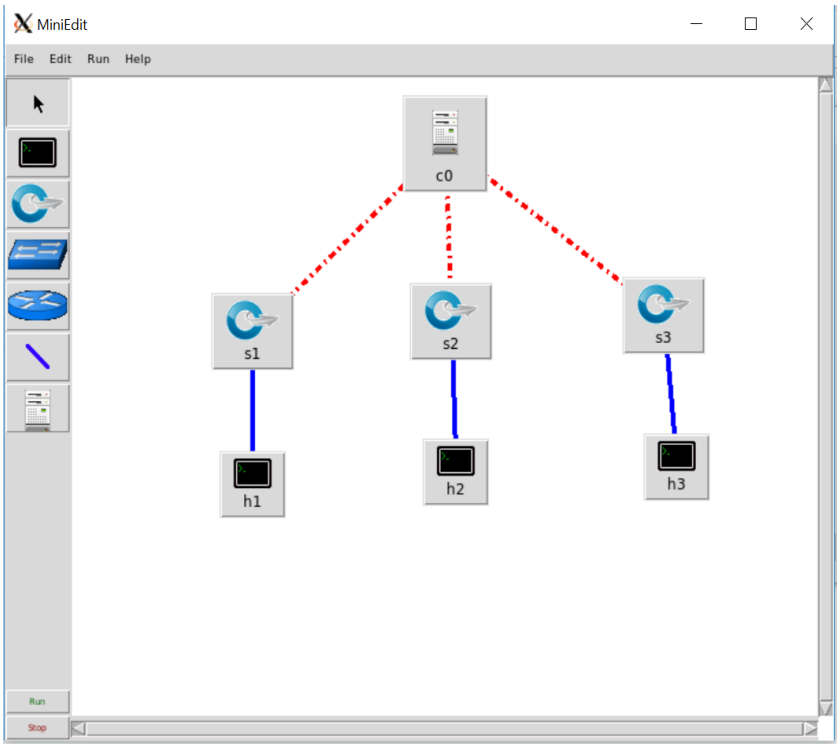


Figure 3.2: Connections between hosts, switches and controllers in MiniEdit

3.4.3 CONFIGURE CONTROLLERS

The controllers built in the network need to be arranged. By default, OpenFlow reference controllers that comes built in to Mininet can be used. Each of the controllers need to be configured, so it uses a different port. To change the ports, right click on the controller and properties are to be selected, which is as shown in Figure 3.3.

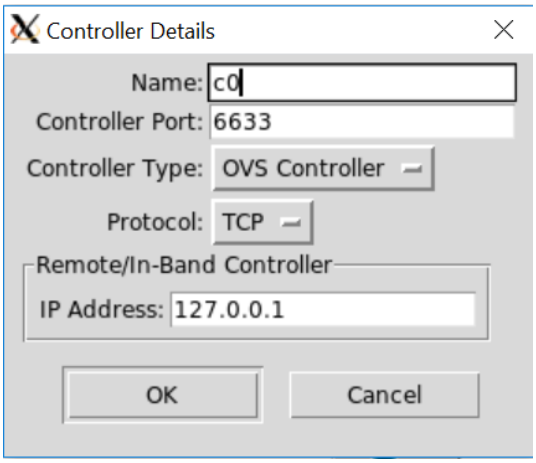


Figure 3.3: configuring a controller

In this paper, a network with three switches, two hosts and one controller are built. It relates to the Ethernet links. The controller is configured with controller type as OVS-controller. After the network is built and configured, it is saved as a python file for example; mytopology.py. The simulation scenario is started by clicking the Run button on the MiniEdit GUI. In the terminal window, where the Mininet is started, messages showing the progress of simulation with startup and MiniEdit CLI prompt. There is an option called 'show OVS summary' under the run command in MiniEdit which allows us to see the listing of the switch configurations. Also, if 'Root terminal' option is selected under run command, it allows us view the flow tables of the switches. The Mininet prompt in MiniEdit console window can be stopped by typing exit command and the simulation can be stopped in the MiniEdit GUI by stop command.

3.5 CONCLUSION

This paper familiarized about the Open vSwitch (ovs) OpenFlow controller and its use to control networks emulated by Mininet. Also, this paper discussed about how flow tables are managed by ovs-controller and how to handle flow tables manually. In addition, it introduced to a graphical

tool called MiniEdit. This tool demonstrates how Mininet can be extended and seems to be a very useful in tool for creating custom based software-defined network simulation scenarios.

3.6 REFERENCES

[1] Open Networking Foundation. OpenFlow Switch Specication https://www.opennetworking.org/sdn-resources/onf-specifications/openflow

[2] Amin Tootoonchian, Sergey Gorbunov, Martin Casado, Rob Sherwood. On Controller Performance in Software-Defined Networks. In Proc. Of HotIce, April, 2012

[3] http://www.openvswitch.org/support/dist-docs/ovs-testcontroller.8.txt

[4] B. Pfaff and B. Davie. The Open vSwitch Database Management Protocol. RFC 7047 (Informational), Dec. 2013.

[5] Open vSwitch – An Open Virtual Switch. http://www.openvswitch.org , September 2014.

[6] https://www.brianlinkletter.com/how-to-use-miniedit-mininets-graphical-user-interface/

CHAPTER 4. LEARNING SWITCH USING MININET

4.1 ABSTRACT

SDN has become apparent as promising network architecture. SDN implements division of network data and control planes, combined with centralized management. This is becoming an important feature of cloud computing environments and telecommunication service providers, all of whom are implementing software defined data centers. Mininet is a cost effective and an efficient way to emulate and study software defined networks. This paper deals with network emulation using Mininet, and its support for OpenFlow and it shows how a switch acts as a hub and a learning switch. This paper also explains how path failures are handled using MiniEdit network simulator's GUI.

4.2 INTRODUCTION

SDN Controllers in a software-defined network is the important part of the network. It is the application that acts as a calculated control point in the SDN network, manage flow control to the switches/routers below (via southbound APIs) and the applications and business logic above (via northbound APIs) to employ intelligent networks. In recent days, as organizations deploy more SDN networks, the Controllers have been charged with combination between SDN Controller domains, using common application intersections, such as OpenFlow and open virtual switch database (OVSDB).

An SDN Controller program typically contains a collection of "pluggable" elements that can execute different network tasks. Some of the common tasks including inventorying what devices are within the network and the capacities of each, assembling network statistics, etc. Extensions

can be enclosed that improves the functionality and support more advanced capabilities, such as running algorithms to discharge analytics and organizing new orders throughout the network.

Two of the most well-known protocols used by SDN Controllers to communicate with the switches/routers is OpenFlow and OVSDB. Other SDN Controller protocols are being evolved, while more demonstrated networking protocols are determining ways to run in an SDN environment. For example, the Internet Engineering Task Force (IETF) working group – the Interface to the Routing System (i2rs)– developed an SDN standard that qualifies an SDN Controller to hold proven, traditional protocols, such as OSPF, MPLS, BGP, and IS-IS. The type of protocols assisted can influence the overall architecture of the network – for example, while OpenFlow experiments to completely centralize packet-forwarding decisions, i2rs divides the decision making by leveraging conventional routing protocols to accomplish distributed routing and allowing applications to change routing decisions.

POX is an SDN OpenFlow controller that also can execute as a switch and is a networking software platform written in Python. This paper mainly discusses about creating a learning switch. It uses POX controller to show how a switch acts as a hub and a learning switch. Also, in addition to this the paper focuses on how to handle path failures in a network in MiniEdit GUI.

4.3 POX CONTROLLER

POX provides a substructure for communicating with SDN switches using either the OpenFlow or OVSDB protocol. Developers can use POX to design an SDN controller using the Python programming language. It is a most approved tool for teaching and researching

software defined networks and network applications programming. Developers may invent a more difficult SDN controller by creating new POX elements. Or, developers may create network applications that refers to POX's API. POX officially requires Python 2.7, and should run under Linux, Mac OS, and Windows. POX presently communicates with OpenFlow switches and involves special assistance for the Open vSwitch/Nicira extensions. pox.py boots up POX. It takes a number of module names on the command line, finds the modules, invokes their launch () function (if it exists), and then changes to the "up" state. Modules are looked for everywhere that Python normally looks, plus the "pox" and "ext" directories. POX elements are additional Python programs that can be called when POX is started from the command line. These elements execute the network functionality in the software defined network. POX comes with some stock elements already available.

4.3.1 SWITCH ACTS AS A HUB

An Ethernet hub, active hub, network hub, repeater hub, multiport repeater, or simply hub is a network hardware device for attaching various Ethernet devices together and making them perform as a single network segment. The POX stock elements are recorded in the POX Wiki and the code for each element can be found in the *~/pox/pox* directory on the *Mininet 2.2* VM image. The most recent mininet releases have POX preinstalled. If it's missing on the VM, it can be downloaded from the POX repository on github into the VM:

$ git clone http://github.com/noxrepo/pox

$ cd pox

The POX controller can activate verbose logging and start the of_tutorial component which currently acts like a hub, by following the below command:

$ ./pox.py log.level –DEBUG misc.of_tutorial

The switches might take some time to connect. When an OpenFlow switch no longer have its connection to a controller, it will generally increase the period between which it aims to contact the controller, up to a maximum of 15 seconds. Since the OpenFlow switch has not connected yet, this delay may be anything between 0 and 15 seconds. If this is long wait, the switch can be programmed to wait no more than N seconds using the --max-backoff parameter.

In mininet CLI a network is created with 3 hosts with the following command:

$ sudo mn –topo single,3 –switch ovsk – controller remote

Now we verify that hosts can ping each other, and that all hosts see the exact same traffic - the behavior of a hub. To do this, xterms will be created for each host and traffic is viewed in each. In the Mininet console, start up three xterms:

mininet > xterm h1 h2 h3

In the xterms for h2 and h3, run tcpdump, a utility to print the packets seen by a host:

tcpdump –XX –n –I h2-eth0

tcpdump –XX –n –I h3-eth0

In the xterm for h1, send a ping:

ping –c1 10.0.0.2

The ping packets are now going up to the controller, which then floods them out all interfaces except to the sending one. Identical ARP and ICMP packets are seen corresponding to the ping in both xterms running tcpdump. This is how a hub works; it sends all packets to every port on the network.

The code of_tutorial .py can be opened using an editor called vim as following:

$ vi of_tutorial.py

The current code calls act_like_hub() from the handler for packet_in messages to implement switch behavior. The flooding action, which is the action of the hub is implemented using:

out_action = of.ofp_action_output(port= of.OFPP_FLOOD)

The above is executed in the resend_packet() method which is invoked from act_like_hub () using:

self.resend_packet (packet_in, of.OFPP_ALL)

The resend_packet() function sends received packets to all ports and creates an OpenFlow Packet Out to be sent from the controller to the switch. Its data is the Packet In :

msg = of.ofp_packet_out()

msg.data = packet_in

Then it includes the actions in the Packet Out, including the action,

define action to send to out_port

action = of.ofp_action_output(port = out_port)

append the defined action to the list of actions

msg.actions.append(action)

Then, the message (Packet Out) is sent to the switch

Send message to switch

self.connection.send(msg)

The above is used by the ofp_packet_out() from the controller to the switch.

4.3.2 SWITCH ACTS AS A LEARNING ETHERNET SWITCH

The learning switch will scrutinize each packet and acquire the knowledge of the source-port mapping. Then, the source MAC address will be correalted with the port. If the destination of the packet is already related to some port, the packet will be sent to the given port, else it will be flooded on all ports of the switch. The switch acts as a hub and alterations are made to the of_tutorial.py which makes the switch act as a leaning Ethernet switch. The of_tutorial.py file is duplicated to of_tutorial-switch.py. Only the of_tutorial-switch.py file is modified. The *act_like_hub()* function call is commented.

The port information is learned from the input MAC frame:

self.mac_to_port[packet.src] = packet_in.in_port

Checked if the destination MAC is in the MAC to Port table (which is dictionary in Python) or not:

if packet.dst in mac_to_port.keys()

then, the packet is sent on the output port by calling resend_packet:

self.resend_packet(packet_in, mac_to_port(packet.dst))

Otherwise, it is sent on all outgoing ports

self.resend_packet(packet_in, of.OFPP_ALL)

Now we need to modify the flow table

Create flow modification packet

msg = of.ofp_flow_mod()

```
# Set fields to match received packet

msg.match = of.ofp_match.from_packet(packet)

#< Set other fields of flow_mod (timeouts? buffer_id?) >

#< Add an output action, and send -- similar to resend_packet() >

msg.actions.append(of.ofp_action_output(port=self.mac_to_port[packet.dst]))

self.connection.send(msg)
```

Now, this program is saved and called on the Mininet VM using the following command:

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

The single switch network with 3 hosts is called using:

```
$ sudo mn –topo single,3 –switch ovsk –controller remote
```

The ping operations are executed in the mininet CLI and flow of packets is captured from Wireshark.

4.4 HANDLING PATH FAILURES USING MINIEDIT GUI

Network discrepancies in software defined networking occurs mainly because of faults, errors or discards in network elements like switches, hosts, hubs, bridges, repeaters, human errors and malfunctioning of devices, failed software and firmware, power failures, software or hardware failures, ongoing changes between hardware and firm wares, cryptographic attacks such as Dos attack, and natural disasters like OS crashes and system failures. Link failures mainly occur when link capacity gets decreased or fiber cable cuts or network congestion. These link failures can be observed by knowing the difference in the flow tables in a network during the path failure. They can be observed using the MiniEdit GUI. The working of this GUI is discussed earlier in chapter-

3, where we created a custom topology network and observed the flow tables. In this chapter, flow tables are observed in the network when there is a path failure.

The MiniEdit script is located in Mininet's *examples* folder. To run MiniEdit, the following command is executed:

$ sudo ~/mininet/examples/miniedit.py

Mininet needs to run with *root* privileges so MiniEdit is started using the *sudo* command. A network, similar to the one in Chapter-3 is created using MiniEdit, and then, run the network. The network consists of 3 switches, 2 hosts and a controller which is configured to be OVS controller.

In the MiniEdit menu command, *Run → Show OVS Summary* can be used to see an listing of switch configurations. In this case, each switch is listening to the correct controller on the correct port or not can be verified.

To open an xterm connected to the host computer by using the MiniEdit menu command, *Run → Root Terminal* can be used.

In the root terminal window, first the userid from *root* to *mininet* is modified, so the switches running in the *mininet* account can be seen.

su mininet

Then, check the flow table on switch *s1* using the commands below

$ sudo ovs-ofctl dump-flows s1

A xterm window on hosts *h1* and *h2* are opened. Right-click on each host in the MiniEdit GUI and select *Terminal* from the menu that appears. In the *h1* xterm window, *Wireshark* is started with the command, `wireshark &`. In the *h2* xterm window, a packet trace is started with the

command `tcpdump`. This is done to illustrate two different methods of viewing traffic on the virtual Ethernet ports of each host.

Then, a *ping* command is executed to send traffic between host *h1* and *h2*. On the MiniEdit console window, the following command is executed:

```
mininet> h1 ping h2
```

In the MiniEdit console, the results of the *ping* command are shown. In the Wireshark window and in the host *h2* xterm window — which is running *tcpdump* — ICMP packets successfully sent and responses received.

To simulate a broken link in the network, move the mouse pointer over one of the blue links in the network and right-click. The *Link Down* from the menu is chosen that appears. The link will turn into a dashed blue line, indicating it is not connected. It is seen that no more traffic is received at host *h2* and that the *ping* command shows packets sent from host *h1* are not being acknowledged to. Now, restore the link operation by right-clicking on the dashed line and choosing *Link Up* from the menu. The link will again appear as a solid blue line, traffic will again be received at host *h2*, and the *ping* command running on *h1* will show it is receiving responses from *h2*. The flow table on switches are observed again. The flows are installed for ICMP packets and ARP packets.
On the *root* terminal window, the following command is entered:

```
$ sudo ovs-ofctl dump-flows s1
```

Then, quit the Mininet CLI by typing `exit` at the `mininet>` prompt, and stop the MiniEdit GUI.

4.5 CONCLUSION

The Growing popularity of SDN within cloud computing environments and related applications has led to an engrossment in modeling the behavior of SDN network configurations. This paper demonstrated how the switch acts as a hub and as a learning switch using POX controller. The switch examined each packet and learned the source port mapping. Also, this paper discussed how path failures are handled in Mininet using network simulator's GUI, MiniEdit.

4.6 REFERENCES

[1] Sdnhub.org, 'OpenFlow version 1.3 tutorial | SDN Hub', 2015. [Online]. Available: http://sdnhub.org/tutorials/openflow-1-3/. [Accessed: 10- May- 2015].

[2] F. Keti and S. Askar. "Emulation of Software Defined Networks Using Mininet in Different Simulation Environments," in 6 th International Conference on Intelligent Systems, Modelling and Simulation, 2015.

[3] D. Kumar and M. Sood. "Software Defined Networking: A Concept and Related Issues" in Int. J. Advanced Networking and Applications. Volume: 6 Issue: 2 Pages: 2233-2239 (2014).

[4] https://github.com/mininet/openflow-tutorial/wiki/Create-a-Learning-Switch

[5] K. K. Sharma and M. Sood. "Mininet as a Container Based Emulator for Software Defined Networks" in International Journal of Advanced Research in Computer Science and Software Engineering. Volume 4, Issue 12, December 2014

[6] Mininet Topologies at http://www.routereflector.com/2013/11/ mini netas-an-sdn-testplatform

CHAPTER 5. FIREWALL USING OPENFLOW

5.1 ABSTRACT

SDN is a networking conception that focuses to centralize networks and model network flows programmable. OpenFlow is the first paradigm communications interface defined between the control and forwarding layers of an SDN architecture. OpenFlow authorizes direct permissions to and for change of the forwarding plane of network devices such as switches and routers, both physical and virtual. Mininet designs a realistic virtual network, running real kernel, switch and application code, on a single machine with just one command. This paper describes the actions of a Firewall using OpenFlow and Mininet. The firewall designed allows the TCP traffic based on the rules defined by the user.

5.2 INTRODUCTION

Software-Defined Networking is the most recent proposed networking prototype in which the data and the control planes are differentiated from one another. The control plane as being the network's important element, i.e., it is accountable for taking all decisions, for example, how to forward data, while the data plane is what moves the data. In conventional networks, both the control and data planes are tightly consolidated and executed in the forwarding devices that makes a network. The SDN control plane is executed by the "controller" and the data plane by "switches". The controller performs as the "brain" of the network and transmits commands to the switches on how to handle traffic. OpenFlow has evolved as the de facto SDN prototype and indicates how the controller and the switches communicate as well as the tasks controllers install on switches.

In computing and networking, a firewall is a network security system that observes and controls incoming and outgoing network traffic established on preestablished security rules. A firewall typically determines a barrier between a believed internal network and untrusted external network, such as the Internet. Firewalls are often differentiated as either network firewalls or host-based firewalls. Network firewalls filter traffic between two or more networks and run on network hardware. Host-based firewalls run on host computers and control network traffic in and out of those machines.

A firewall is a network security device that observes incoming and outgoing network traffic and determines whether to allow or block specific traffic based on a predefined set of security rules. Firewalls have been a first line of defense in network security for over 25 years. They establish a barrier between secured and controlled internal networks that can be trusted and untrusted outside networks, such as the Internet. A firewall can be hardware, software, or both. This paper discusses how to create a firewall for a tree topology using Mininet and POX controller and how to define rules for the firewall.

5.3 FIREWALL USING MININET

A Firewall can be defined as something that blocks traffic coming its way and filters it according to some defined *rules*. A typical Firewall can be implemented to protect a network from the internet. For an SDN based Firewall, the OpenFlow controller is used to filter traffic between hosts according to some rules and accordingly let it pass through or not. This is done by using the POX controller to determine the required *policies* or *rules* and filter traffic between hosts using the switches. An SDN firewall with two hosts is shown in Figure 5.1.

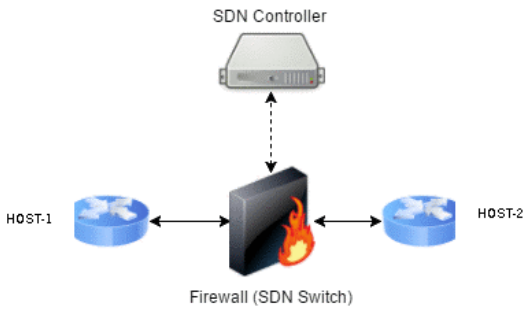


Figure 5.1: SDN Firewall with two hosts

5.3.1 TOPOLOGY SETUP IN MININET

The tree topology network called tree.py with 4 hosts, 3 switches and a POX controller is designed.

```
from mininet.net import Mininet

from mininet.node import RemoteController

from mininet.cli import CLI

from mininet.log import setLogLevel, info
```

Now, a function is created in which a code is written to add MAC addresses, hosts and switches and inbuilt info function is used instead of print statement to print or log out information.

```
def treeTopo():

    net = Mininet( controller=RemoteController )

    info( '*** Adding controller\n' )

    net.addController('c0')
```

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

    info( '*** Adding switches\n' )
    s1 = net.addSwitch( 's1' )
    s2 = net.addSwitch( 's2' )
    s3 = net.addSwitch( 's3' )
```

Now, links between hosts and switches are created and then between the switches themselves.

```
info( '*** Creating links\n' )
    net.addLink( h1, s2 )
    net.addLink( h2, s2 )
    net.addLink( h3, s3 )
    net.addLink( h4, s3 )
```

```
    net.addLink( s2, s1 )
    net.addLink( s3, s1 )


    root = s1
    layer1 = [s2,s3]
    for idx,l1 in enumerate(layer1):
        net.addLink( root,l1 )
```

Lastly, the network and Mininet CLI will be started and stopped when the process is complete.

```
    info( '*** Starting network\n')
    net.start()
   info( '*** Running CLI\n' )
    CLI( net )
    info( '*** Stopping network' )
    net.stop()
if __name__ == '__main__':
    setLogLevel( 'info' )
    treeTopo()
```

5.3.2 FIREWALL RULES AND POX IMPLEMENTATION

It should also be noted that it will be a bi-directional block, and if a rule exists, the controller will make sure that switches neither let incoming nor outgoing traffic from either host reach the other. So, for Firewall, the following 2 rules are developed:

- h1 and h2 are mutually blocked
- h2 and h4 are mutually blocked

The code for setting up the rules of the firewall can be programmed in the POX directory. A new file called firewall.py is designed in which the code is written.

```
from pox.core import core

import pox.openflow.libopenflow_01 as of

from pox.lib.revent import *

from pox.lib.addresses import EthAddr
```

The rules of the firewall are specified by specifying the addresses of the hosts as follows:

```
rules = [['10.0.0.1','10.0.0.2'], ['10.0.0.2', '10.0.0.4']]
```

Next, SDNFirewall class is created in which the actual controller is going to be accessing and ensuring flows and changing flow tables accordingly. The first function __init__ is just a constructor. The second and more interesting function, _handle_ConnectionUp will invoke each time a host tries to reach another host through the switches. When this occurs, each rule in our list of rules will be iterated. Here match fields are created by providing the two hosts in the rule,

specified by rule[0] and rule[1] to block. Then an OpenFlow flow_mod message is created using of.ofp_flow_mod() and its match field is set to our blocking rule. At the end, event.connection.send(flow_mod) is used to send blocking rule to the switch so that it can be enforced. Lastly, the launch function is created, that POX requires and pass the SDNFirewall class to it. The code for this is as follows:

```
class SDNFirewall (EventMixin):


    def __init__ (self):
        self.listenTo(core.openflow)


    def _handle_ConnectionUp (self, event):
        for rule in rules:
            block = of.ofp_match()
            block.dl_src = EthAddr(rule[0])
            block.dl_dst = EthAddr(rule[1])
            flow_mod = of.ofp_flow_mod()
            flow_mod.match = block
            event.connection.send(flow_mod)
```

```
def launch ():
    core.registerNew(SDNFirewall)
```

In order to start the Mininet emulation, the following can be executed

```
$ sudo python tree.py
```

The hosts and switches are initialized and Mininet CLI is started. The controller can be started by first cd into the pox directory and run:

```
$ ./pox.py log.level –DEBUG openflow.of_01 forwarding.l2_learning misc.firewall
```

Then run pingall in the mininet CLI and the results are observed as shown in Figure 5.2.

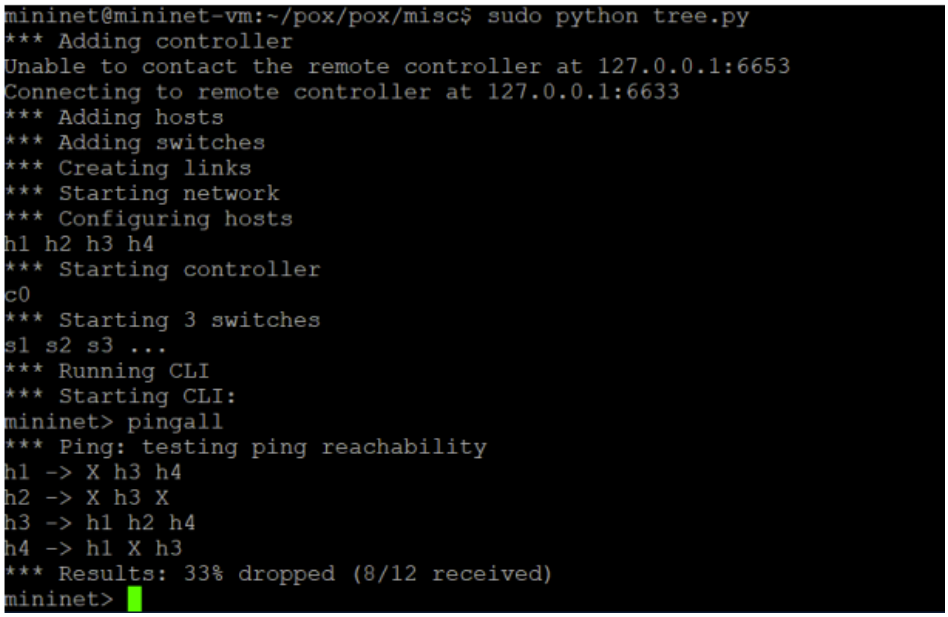


Figure 5.2: Ping reachability according to firewall rules

According to the firewall rules, h1 and h2 were never able to connect and also h2 and h4 were never connected.

5.4 CONCLUSION

SDN has introduced a radical change in network architecture which simplified the control of networks, but on the other hand many challenges came to light. One of the fundamental issues which exposed due to the new architecture of SDN is the security risks. Network firewall is an essential component for securing traffic by imposing security policies. This paper implemented some firewall functionalities on SDN through writing some firewall applications that run on the top of the SDN POX controller. The firewall application, which is working at layer2, layer3 and layer4 capable of detecting the traffic and enforcing specified rules. Using rules specifying firewall characteristics the switches are controlled using POX to either permit traffic between hosts or restrict it.

5.4 REFERENCES

[1] Javid, Tariq, Tehseen Riaz, and Asad Rasheed. "A layer2 firewall for software defined network." In Information Assurance and Cyber Security (CIACS), 2014 Conference on, pp. 39-42. IEEE, 2014.

[2] Suh, Michelle, Sae Hyong Park, Byungjoon Lee, and Sunhee Yang. "Building firewall over the software-defined network controller." In Advanced Communication Technology (ICACT), 2014 16th International Conference on, pp. 744-748. IEEE, 2014.

[3] https://github.com/pratiklotia/SDN-Firewall

[4]https://cmpe150winter1701.courses.soe.ucsc.edu/system/files/attachments/Lab3SimpleFirewallusingOpenFlow.pdf

[5] http://academicscience.co.in/admin/resources/project/paper/f201706191497888502.pdf

[6] Kaur, Karamjeet, Japinder Singh, and Navtej Singh Ghumman. "Mininet as Software Defined Networking Testing Platform." In International Conference on Communication, Computing & Systems (ICCCS. 2014.

CHAPTER 6. FUTURE WORK SUMMARY AND DISCUSSION

Emerging technologies, such as SDN and network virtualization, are gaining increased attention in the industrial sector. Despite years of work on SDN, the technology is still in its infancy. The future work on SDN include creating better SDN infrastructure, applying SDN in new settings, and increasing the scope of SDN beyond network switches.

SDN reveals various classes of difficulties related to security, few of which are improving languages and control frameworks that can implement specified security policies, collaborating network control with security appliances, and securing the SDN infrastructure. Eventually, the formal approaches that improving programming languages present may help researchers improve solutions to all these problems.

Designing reliable software defined networks needs ways to make sure controller applications, the controller platform, and the underlying switches perform correctly. SDN elements are simpler to verify (i.e., to prove properties about the system), test (i.e., to generate synthetic inputs that exercise different parts of the system) and debug (i.e., pinpoint the cause of a misbehaving system) than traditional networks because of the easy, open interface to the data plane.

SDN and programmable networks in general, face a traction between programmability and performance. Ultimately, hardware data planes can forward traffic at higher rates but are less flexible; on the other hand, software data planes are flexible but have been unable to forward traffic at line rates.

Nowadays networks have many middleboxes (e.g., firewalls, load balancers, intrusion detection systems, transcoders, proxy caches) that can execute arbitrarily complex packet processing operations. Although these middleboxes have historically been dedicated appliances that reside at

fixed locations in the network, industry has lately introduced an effort around Network Functions Virtualization (NFV), where middlebox functionality runs on top of product servers and switches. NFV allows the programmer to see the network as a bunch of resources, creating flexible placement and duplication of middlebox responsibilities in acknowledgment to changing traffic demands.

Software Defined Networking provides a sufficient domain for research and investigation, as it is still a new and diversified technology.

REFERENCES

[1] N. Handigol, B. Heller, V. Jeyakumar, D. Mazieres, and ` N. McKeown. Where is the debugger for my software-defined network? In Proc. Workshop on Hot Topics in Software Defined Networks, pages 55–60, 2012.

[2] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov. Designing extensible IP router software. In Proc. Networked Systems Design and Implementation, May 2005.

[3] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare active network architecture. IEEE Network, 12(3):29–36, 1998.

[4] B. Anwer, M. Motiwala, M. bin Tariq, and N. Feamster. SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware. In Proc. ACM SIGCOMM, New Delhi, India, Aug. 2010.

[5] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In Proc. ACM SIGCOMM, Aug. 2013.

[6] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge. Smart packets for active networks. In IEEE Conference on Open Architectures and Network Programming, pages 90–97. IEEE, 1999.