University of Southern California

EE597 Project – Software Defined Network

Broadband Network Architectures

by

Wanwiset Peerapatanapokin

Tingyang Sun

Table of Contents

Project Overview

The objective of this project is to build a Software Defined Network (SDN). SDN works differently from traditional networks. Traditional networks usually consist of switches and routers. Each device is programmed separately and processes control plane and data plane with the same hardware. In SDN there is no distinction between router and switch, the forwarding device is a programmable switch where instructions can be installed based on layer 2, 3 or 4, information in a packet. So it can potentially perform the function of a switch, router and firewall in one device. The SDN also introduced the separation of control plane and data plane. The SDN controller is a device that's responsible for the control plane. Every forwarding device in SDN must have a connection to the controller. In SDN, every unknown packet arriving at a programmable switch will be forwarded to the controller. The controller acts as a central point of the network, it inspects the packet, and sends instructions to forwarding devices. The programmable switches' only task is forwarding in the data plane by installing flows as instructed by the controller. The packet forwarding switch and the controller interact with each other using a protocol called OpenFlow.

In this project, we are to emulate a traditional network by building a controller that has capabilities of both the router and switch. The controller is able to identify devices in the network and instruct them to perform the functionalities of switch or router based on a pre-defined topology. The SDN is implemented in Mininet, a linux-based virtual machine tool that can emulate a realistic virtual network. Inside mininet, we run a python-based controller called POX. MobaXterm also deserves a mention as a SSH client tool, it provides a convenient interface and multiple quality of life features, including: enabling usage of external file editor, interface based file explorer, direct FTP, and supporting multiple sessions to the virtual machine

High-level description of switch design

The switch's function is simple in nature, it forwards frames based on the MAC table, if the destination is unknown, floods the frame to all ports.

In the switch handler, we use the mac_port table we created in the controller file. Every time the frame goes into switch, switch saves the source mac address and corresponding port.

If the destination MAC address is not in the table, we flood the frame to every port except the incoming one. If it is found, we get the out-port number and install the flow rule.

In the simulation scenario, when A starts sending frames to B through the switch, the switch will not find a rule and forwards the packet to the controller. The controller records the source MAC address of A, floods the packet out the switch's ports and instructs the switch to install the flow of A to B. The switch will forward the next frame that it receives immediately without sending it to the controller since it already knows the rule for this type of frame. Hence, the first few packets from A to B will have higher latency, since the forwarding rule hasn't been installed and the packets must flow through the controller.

High-level description of router design

In SDN, when an unknown packet arrives, the programmable switch forwards it to the controller. The controller is also pre-defined with the IP and MAC addresses of router interfaces and the routing table based on the topology. The router's main function is to forward packets based on the routing table. Additionally, it must also be able to handle ARP, otherwise, layer 3 connection can never be established. We also implemented ICMP reply function, as the router's interfaces have IP addresses and can be pinged. Finally, the router is able to send an ICMP unreachable message back to the source if it cannot find any matches in the routing table for an ICMP packet.

For every packet going into the router, we define a router handler function, which can distinguish the type of packet payload. If there are many routers, an identifier is assigned to each device where each of them can have different IP addresses and routing tables.

If it is an arp packet, we import a help function called arp handler that is used to deal with both the arp request and reply case. If it is an IP packet, we import another help function called ip handler. It can deal with icmp echo requests whose destination ip address is one of the router interfaces. Furthermore, it also decides forwarding ports based on the longest prefix match between the destination ip address and routing table.

After the first packet forward, it also installs the flow rules so that the next packet does not need to go through the same control process again.

Finally, we add icmp_unreach function to deal with the case in which the packet destination address cannot be found in the routing table.

Data structures

The data structures we used in this project are mainly for initializing all the variables we need in the controller. In the switch, we use a dictionary called *mac_to_port* for the MAC table to match the MAC address to the port number. In the router, we used two 2-D lists called *route_table* and *port_info* for saving the routing table and interface port assignments. Another dictionary is used in the router called *arp_table* to match the IP address to the MAC address.

Another thing to be careful of when dealing with IP and MAC addresses is the POX address data type. They are not simply strings. They have their own data type, which can be assigned by *IPAddr(ip_string)* and *EthAddr(mac_string)*.

To explain how the files interact with one another, in each scenario the required files include controller.py, topology.py, router.py and switch.py. Topology.py is used to initialize the topology using the Topo subclass which is imported from Mininet. Here, we create new switches, routers and hosts which can be referenced by variable names such as r1, s4, or h9 once initialized in Mininet. After generating devices, they are connected together by the *addLink* method, we are also able to specify the ports of the link.

Controller.py is the code that represents the controller. Every switch and router that is created in Mininet will create a connection to the controller.py's Tutorial class. Each connection is unique and can be referenced by the DPID to control each device separately. The forwarding devices' IP addresses, MAC addresses, and routing table are stored in the controller. The controller has a built in listener, when a packet arrives at the controller it calls the *_handle_PacketIn* function. Here, we can check which device the packet arrives from by checking the DPID. Then, we can select to handle the packet as a switch or router accordingly. This is where switch.py and router.py are used, the controller imports switch.py and router.py functionalities.

To summarize, topology.py is used for the initialization of the network and controller.py is used to process packets that arrive at the forwarding devices. When a packet arrives at the controller, it checks the DPID to see if the device is a switch or router, then it calls the switch_handler or router_handler from switch.py and router.py.

Running the SDN and Results

After installing Mininet VM, configure host-only network setup and boot the virtual system. No further installation is required. Next, transfer the project files to the VM via FTP. There are eight files in total, three controller files, three topology files, router.py and switch.py. Then, open two SSH sessions. One session for running mininet and another for running POX controller. To start scenario 1. Enter the directory that the files are stored and issue the commands:

Controller: ./pox.py log.level --DEBUG misc.controller_1

Mininet topology: sudo mn --custom topology_1.py --topo mytopo --mac --controller remote

Change the controller and topology filename as necessary for different scenarios. Switch.py and router.py serve as general switch and router functions where all scenarios utilize. After exiting Mininet, always issue the command *'sudo mn -c'* to clear out the topology and remaining data.

We investigate three scenarios in this project. Scenario 1 consists of a single switch and multiple hosts. Scenario 2 consists of a single router and multiple hosts. And Scenario 3 contains multiple switches, routers, and hosts. The scenarios' topology, IP assignments and routing tables are shown in Fig. 1,2 and 3 below.

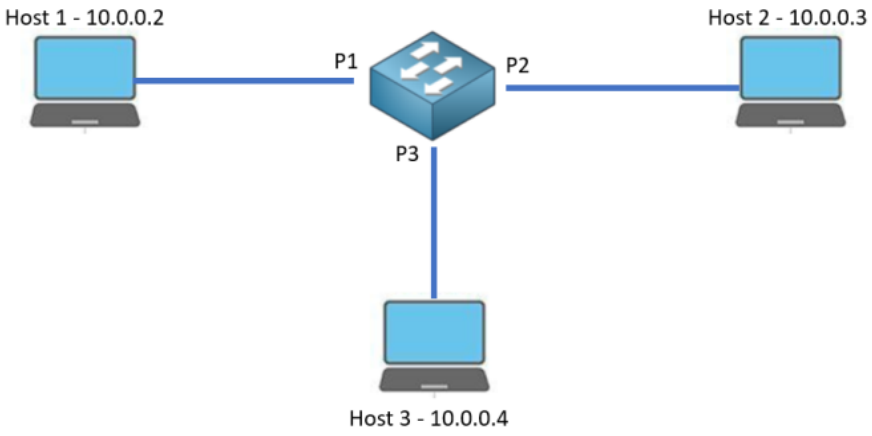

Figure 1 Scenario 1 Topology

Host Details				
Host	IP Address	Default GW	Network	Subnet Mask
1	10.0.0.2	10.0.0.1	10.0.0.0	255.255.255.0
2	10.0.0.3	10.0.0.1	10.0.0.0	255.255.255.0
3	10.0.0.4	10.0.0.1	10.0.0.0	255.255.255.0

Table 1 Host details for scenario 1

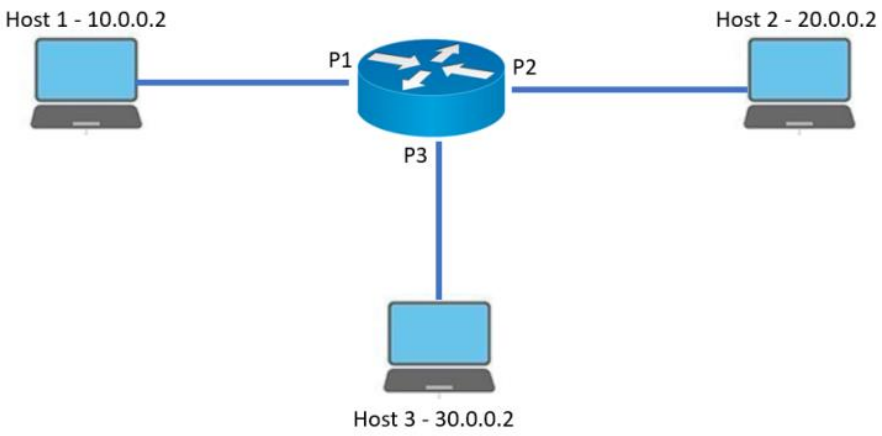

Figure 2 Scenario 2 Topology

Host Details				
Host	IP Address	Default GW	Network	Subnet Mask
1	10.0.0.2	10.0.0.1	10.0.0.0/24	255.255.255.0
2	20.0.0.2	20.0.0.1	20.0.0.0/24	255.255.255.0
3	30.0.0.2	30.0.0.1	30.0.0.0/24	255.255.255.0

Table 2 Host details for scenario 2

Router Interface Details			
Interface	Network Attached	Interface IP	Interface MAC
1	10.0.0.0/24	10.0.0.1	02:00:DE:AD:BE:11
2	20.0.0.0/24	20.0.0.1	02:00:DE:AD:BE:12
3	30.0.0.0/24	30.0.0.1	02:00:DE:AD:BE:13

Table 3 Router Interface details for scenario 2

Routing Table Info					
Entry no	Network Address	Subnet Mask	Prefix Length	Next Hop Router	Output Interface
1	10.0.0.0	255.255.255.0	24	0.0.0.0	1
2	20.0.0.0	255.255.255.0	24	0.0.0.0	2
3	30.0.0.0	255.255.255.0	24	0.0.0.0	3

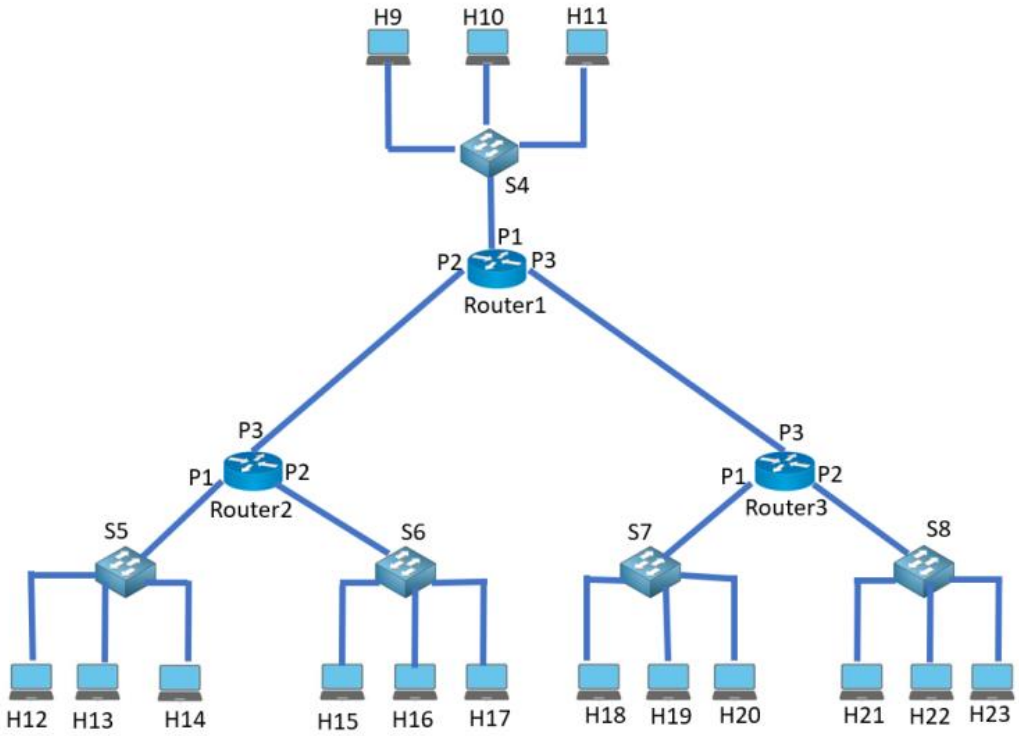

Figure 3 Scenario 3 Topology

Host Details				
Host	IP Address	Default GW	Network	Subnet Mask
h9	172.17.16.2	172.17.16.1	172.17.16.0	255.255.255.0
h10	172.17.16.3	172.17.16.1	172.17.16.0	255.255.255.0
h11	172.17.16.4	172.17.16.1	172.17.16.0	255.255.255.0
h12	10.0.0.2	10.0.0.1	10.0.0.0	255.255.255.128
h13	10.0.0.3	10.0.0.1	10.0.0.0	255.255.255.128
h14	10.0.0.4	10.0.0.1	10.0.0.0	255.255.255.128
h15	10.0.0.130	10.0.0.129	10.0.0.128	255.255.255.128
h16	10.0.0.131	10.0.0.129	10.0.0.128	255.255.255.128
h17	10.0.0.132	10.0.0.129	10.0.0.128	255.255.255.128
h18	20.0.0.2	20.0.0.1	20.0.0.0	255.255.255.128
h19	20.0.0.3	20.0.0.1	20.0.0.0	255.255.255.128
h20	20.0.0.4	20.0.0.1	20.0.0.0	255.255.255.128
h21	20.0.0.130	20.0.0.129	20.0.0.128	255.255.255.128
h22	20.0.0.131	20.0.0.129	20.0.0.128	255.255.255.128
h23	20.0.0.132	20.0.0.129	20.0.0.128	255.255.255.128

Table 5 Host details for Scenario 3

Interface Details for Router 1			
Interface	Network Attached	Interface IP	Interface MAC
1	172.17.16.0/24	172.17.16.1	02:00:DE:AD:BE:11
2	192.168.0.0/30	192.168.0.1	02:00:DE:AD:BE:12
3	192.168.0.4/30	192.168.0.5	02:00:DE:AD:BE:13
Interface Details for Router 2			
Interface	Network Attached	Interface IP	Interface MAC
1	10.0.0.0/25	10.0.0.1	02:00:DE:AD:BE:21
2	10.0.0.128/25	10.0.0.129	02:00:DE:AD:BE:22
3	192.168.0.0/30	192.168.0.2	02:00:DE:AD:BE:23
Interface Details for Router 3			
Interface	Network Attached	Interface IP	Interface MAC
1	20.0.0.0/25	20.0.0.1	02:00:DE:AD:BE:31
2	20.0.0.128/25	20.0.0.129	02:00:DE:AD:BE:32
3	192.168.0.4/30	192.168.0.6	02:00:DE:AD:BE:33

Table 6 Router Interface details for scenario 3

Routing Table Info for Router 1					
Entry no	Network Address	Subnet Mask	Prefix Length	Next Hop Router	Output Interface
1	172.17.16.0	255.255.255.0	24	0.0.0.0	1
2	10.0.0.0	255.255.255.0	24	192.168.0.2	2
3	20.0.0.0	255.255.255.0	24	192.168.0.6	3
Routing Table Info for Router 2					
Entry no	Network Address	Subnet Mask	Prefix Length	Next Hop Router	Output Interface
1	172.17.16.0	255.255.255.0	24	192.168.0.1	3
2	0.0.0.0	0.0.0.0	0	192.168.0.1	3
3	10.0.0.0	255.255.255.128	25	0.0.0.0	1
4	10.0.0.128	255.255.255.128	25	0.0.0.0	2
Routing Table Info for Router 3					
Entry no	Network Address	Subnet Mask	Prefix Length	Next Hop Router	Output Interface
1	172.17.16.0	255.255.255.0	24	192.168.0.5	3
2	10.0.0.0	255.255.255.0	24	192.168.0.5	3
3	20.0.0.0	255.255.255.128	25	0.0.0.0	1
4	20.0.0.128	255.255.255.128	25	0.0.0.0	2

Table 7 Routing Table Info for Scenario 3

In scenario 1, we verify our SDN with a few commands as shown in the figure below. First, *h1 ping h2* is successful with no dropped packets. We also observe higher latency in the first packet phenomenon. Second, *pingall*, a mininet command where each host pings all other hosts, is issued. Mininet returns a successful result with zero drop. Finally we issue *iperf h1 h2*, a TCP based bandwidth tester between two hosts. We did not include a speed limit in the topology virtualization, so the bandwidth is only limited by the processing speed, returning the result of 34 Gbps.

```
mininet>
mininet> h1 ping h2
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=105 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.867 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.055 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=0.089 ms
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=0.148 ms
^C
--- 10.0.0.3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4003ms
rtt min/avg/max/mdev = 0.055/21.405/105.869/42.233 ms
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
.*** Results: ['34.1 Gbits/sec', '34.1 Gbits/sec']
```

We can also verify the flows installed in the switch by using the command *dpctl dump-flows*, the result is as shown below.

```
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=26.843s, table=0, n_packets=1, n_bytes=42, idle_age=21, priority=25,arp,in_port=3,vlan_tci=0x0000,dl_src=ae:ae:7e:ee:0f:af,d
l_dst=b2:bd:bb:f8:ac:bb actions=output:2
 cookie=0x0, duration=21.74s, table=0, n_packets=0, n_bytes=0, idle_age=21, priority=25,arp,in_port=2,vlan_tci=0x0000,dl_src=b2:bd:bb:f8:ac:bb,dl_
dst=ae:ae:7e:ee:0f:af actions=output:3
 cookie=0x0, duration=26.996s, table=0, n_packets=1, n_bytes=42, idle_age=21, priority=25,arp,in_port=3,vlan_tci=0x0000,dl_src=ae:ae:7e:ee:0f:af,d
l_dst=b2:5e:af:4d:c6:61 actions=output:1
 cookie=0x0, duration=35.443s, table=0, n_packets=1, n_bytes=42, idle_age=30, priority=25,arp,in_port=2,vlan_tci=0x0000,dl_src=b2:bd:bb:f8:ac:bb,d
l_dst=b2:5e:af:4d:c6:61 actions=output:1
 cookie=0x0, duration=30.378s, table=0, n_packets=0, n_bytes=0, idle_age=30, priority=25,arp,in_port=1,vlan_tci=0x0000,dl_src=b2:5e:af:4d:c6:61,dl
_dst=b2:bd:bb:f8:ac:bb actions=output:2
 cookie=0x0, duration=21.909s, table=0, n_packets=0, n_bytes=0, idle_age=21, priority=25,arp,in_port=1,vlan_tci=0x0000,dl_src=b2:5e:af:4d:c6:61,dl
_dst=ae:ae:7e:ee:0f:af actions=output:3
 cookie=0x0, duration=35.337s, table=0, n_packets=108671, n_bytes=7172494, idle_age=12, priority=25,ip,in_port=2,vlan_tci=0x0000,dl_src=b2:bd:bb:f
8:ac:bb,dl_dst=b2:5e:af:4d:c6:61,nw_tos=0 actions=output:1
 cookie=0x0, duration=18.158s, table=0, n_packets=4, n_bytes=272, idle_age=17, priority=25,ip,in_port=1,vlan_tci=0x0000,dl_src=b2:5e:af:4d:c6:61,d
l_dst=b2:bd:bb:f8:ac:bb,nw_tos=16 actions=output:2
 cookie=0x0, duration=26.917s, table=0, n_packets=1, n_bytes=98, idle_age=26, priority=25,ip,in_port=3,vlan_tci=0x0000,dl_src=ae:ae:7e:ee:0f:af,dl
_dst=b2:5e:af:4d:c6:61,nw_tos=0 actions=output:1
 cookie=0x0, duration=18.121s, table=0, n_packets=0, n_bytes=0, idle_age=18, priority=25,ip,in_port=2,vlan_tci=0x0000,dl_src=b2:bd:bb:f8:ac:bb,dl_
dst=b2:5e:af:4d:c6:61,nw_tos=16 actions=output:1
 cookie=0x0, duration=26.957s, table=0, n_packets=1, n_bytes=98, idle_age=26, priority=25,ip,in_port=1,vlan_tci=0x0000,dl_src=b2:5e:af:4d:c6:61,dl
_dst=ae:ae:7e:ee:0f:af,nw_tos=0 actions=output:3
 cookie=0x0, duration=26.721s, table=0, n_packets=0, n_bytes=0, idle_age=26, priority=25,ip,in_port=3,vlan_tci=0x0000,dl_src=ae:ae:7e:ee:0f:af,dl_
dst=b2:bd:bb:f8:ac:bb,nw_tos=0 actions=output:2
 cookie=0x0, duration=35.405s, table=0, n_packets=384032, n_bytes=21368717600, idle_age=12, priority=25,ip,in_port=1,vlan_tci=0x0000,dl_src=b2:5e:
af:4d:c6:61,dl_dst=b2:bd:bb:f8:ac:bb,nw_tos=0 actions=output:2
 cookie=0x0, duration=26.805s, table=0, n_packets=1, n_bytes=98, idle_age=26, priority=25,ip,in_port=2,vlan_tci=0x0000,dl_src=b2:bd:bb:f8:ac:bb,dl
_dst=ae:ae:7e:ee:0f:af,nw_tos=0 actions=output:3
mininet>
```

In scenario 2, we begin the verification of the SDN with similar commands to scenario 1 as shown below. After that, two extra commands are used to verify the router functionality. *H1 ping 30.0.0.1* is used, 30.0.0.1 is one of the router's interfaces. Here, we see that the router is able to process ICMP requests and generate ICMP replies to the host. *H1 ping 8.8.8.8* is issued, where 8.8.8.8 is Google's DNS server public IP address. We did not set up an internet connection for Mininet, therefore, we were not able to connect with 8.8.8.8. The purpose is to demonstrate the function of the router, where it replies with ICMP message 'Destination Net Unreachable' when the router cannot find a match in the routing table.

```
mininet> h1 ping h2
PING 20.0.0.2 (20.0.0.2) 56(84) bytes of data.
64 bytes from 20.0.0.2: icmp_seq=1 ttl=64 time=50.5 ms
64 bytes from 20.0.0.2: icmp_seq=2 ttl=64 time=23.7 ms
64 bytes from 20.0.0.2: icmp_seq=3 ttl=64 time=0.304 ms
64 bytes from 20.0.0.2: icmp_seq=4 ttl=64 time=0.080 ms
64 bytes from 20.0.0.2: icmp_seq=5 ttl=64 time=0.083 ms
^C
--- 20.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4003ms
rtt min/avg/max/mdev = 0.080/14.960/50.546/20.009 ms
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['34.4 Gbits/sec', '34.5 Gbits/sec']
mininet> h1 ping 30.0.0.1
PING 30.0.0.1 (30.0.0.1) 56(84) bytes of data.
64 bytes from 30.0.0.1: icmp_seq=1 ttl=64 time=41.3 ms
64 bytes from 30.0.0.1: icmp_seq=2 ttl=64 time=16.1 ms
64 bytes from 30.0.0.1: icmp_seq=3 ttl=64 time=43.6 ms
64 bytes from 30.0.0.1: icmp_seq=4 ttl=64 time=23.3 ms
64 bytes from 30.0.0.1: icmp_seq=5 ttl=64 time=3.03 ms
^C
--- 30.0.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 3.037/25.507/43.660/15.345 ms
mininet> h1 ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Net Unreachable
From 10.0.0.1 icmp_seq=2 Destination Net Unreachable
From 10.0.0.1 icmp_seq=3 Destination Net Unreachable
From 10.0.0.1 icmp_seq=4 Destination Net Unreachable
From 10.0.0.1 icmp_seq=5 Destination Net Unreachable
^C
--- 8.8.8.8 ping statistics ---
5 packets transmitted, 0 received, +5 errors, 100% packet loss, time 4007ms

mininet>
```

The flows installed in the router of scenario 2 is as shown below.

```
mininet> dpctl dump-flows
*** r1 ------------------------------------------------------------------------
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=658.364s, table=0, n_packets=382432, n_bytes=21563778176, idle_age=641, priority=25,ip,in_port=1,vlan_tci=0x0000,dl_src=72:0e
:f7:5e:a3:29,dl_dst=02:00:de:ad:be:11,nw_src=10.0.0.2,nw_dst=20.0.0.2,nw_tos=0 actions=mod_dl_src:02:00:de:ad:be:12,mod_dl_dst:ba:12:e7:8c:25:df,ou
tput:2
 cookie=0x0, duration=650.481s, table=0, n_packets=1, n_bytes=98, idle_age=650, priority=25,ip,in_port=3,vlan_tci=0x0000,dl_src=d2:00:bf:73:ca:34,d
l_dst=02:00:de:ad:be:13,nw_src=30.0.0.2,nw_dst=10.0.0.2,nw_tos=0 actions=mod_dl_src:02:00:de:ad:be:11,mod_dl_dst:72:0e:f7:5e:a3:29,output:1
 cookie=0x0, duration=650.443s, table=0, n_packets=1, n_bytes=98, idle_age=650, priority=25,ip,in_port=2,vlan_tci=0x0000,dl_src=ba:12:e7:8c:25:df,d
l_dst=02:00:de:ad:be:12,nw_src=20.0.0.2,nw_dst=30.0.0.2,nw_tos=0 actions=mod_dl_src:02:00:de:ad:be:13,mod_dl_dst:d2:00:bf:73:ca:34,output:3
 cookie=0x0, duration=646.611s, table=0, n_packets=0, n_bytes=0, idle_age=646, priority=25,ip,in_port=2,vlan_tci=0x0000,dl_src=ba:12:e7:8c:25:df,dl
_dst=02:00:de:ad:be:12,nw_src=20.0.0.2,nw_dst=10.0.0.2,nw_tos=16 actions=mod_dl_src:02:00:de:ad:be:11,mod_dl_dst:72:0e:f7:5e:a3:29,output:1
 cookie=0x0, duration=659.311s, table=0, n_packets=97597, n_bytes=6441610, idle_age=641, priority=25,ip,in_port=2,vlan_tci=0x0000,dl_src=ba:12:e7:8
c:25:df,dl_dst=02:00:de:ad:be:12,nw_src=20.0.0.2,nw_dst=10.0.0.2,nw_tos=0 actions=mod_dl_src:02:00:de:ad:be:11,mod_dl_dst:72:0e:f7:5e:a3:29,output:
1
 cookie=0x0, duration=650.406s, table=0, n_packets=1, n_bytes=98, idle_age=650, priority=25,ip,in_port=3,vlan_tci=0x0000,dl_src=d2:00:bf:73:ca:34,d
l_dst=02:00:de:ad:be:13,nw_src=30.0.0.2,nw_dst=20.0.0.2,nw_tos=0 actions=mod_dl_src:02:00:de:ad:be:12,mod_dl_dst:ba:12:e7:8c:25:df,output:2
 cookie=0x0, duration=650.367s, table=0, n_packets=1, n_bytes=98, idle_age=650, priority=25,ip,in_port=1,vlan_tci=0x0000,dl_src=72:0e:f7:5e:a3:29,d
l_dst=02:00:de:ad:be:11,nw_src=10.0.0.2,nw_dst=30.0.0.2,nw_tos=0 actions=mod_dl_src:02:00:de:ad:be:13,mod_dl_dst:d2:00:bf:73:ca:34,output:3
 cookie=0x0, duration=646.679s, table=0, n_packets=1, n_bytes=66, idle_age=646, priority=25,ip,in_port=1,vlan_tci=0x0000,dl_src=72:0e:f7:5e:a3:29,d
l_dst=02:00:de:ad:be:11,nw_src=10.0.0.2,nw_dst=20.0.0.2,nw_tos=16 actions=mod_dl_src:02:00:de:ad:be:12,mod_dl_dst:ba:12:e7:8c:25:df,output:2
mininet>
```

Finally in scenario 3, we select to show only *pingall* command to verify the connectivity of the whole network. Other functionalities work as expected and are similar to what is shown in previous scenarios. We observe a successful result as shown below, all hosts are able to connect with each other across the network of routers and switches.

```
mininet> pingall
*** Ping: testing ping reachability
h9 -> h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23
h10 -> h9 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23
h11 -> h9 h10 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23
h12 -> h9 h10 h11 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23
h13 -> h9 h10 h11 h12 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23
h14 -> h9 h10 h11 h12 h13 h15 h16 h17 h18 h19 h20 h21 h22 h23
h15 -> h9 h10 h11 h12 h13 h14 h16 h17 h18 h19 h20 h21 h22 h23
h16 -> h9 h10 h11 h12 h13 h14 h15 h17 h18 h19 h20 h21 h22 h23
h17 -> h9 h10 h11 h12 h13 h14 h15 h16 h18 h19 h20 h21 h22 h23
h18 -> h9 h10 h11 h12 h13 h14 h15 h16 h17 h19 h20 h21 h22 h23
h19 -> h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h20 h21 h22 h23
h20 -> h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h21 h22 h23
h21 -> h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h22 h23
h22 -> h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h23
h23 -> h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
*** Results: 0% dropped (210/210 received)
mininet>
```

References

1. Mininet installation and tutorial:

http://mininet.org/walkthrough/

2. POX controller documentation:

https://noxrepo.github.io/pox-doc/html/

3. David Mahler's youtube guide on Mininet topologies:

https://www.youtube.com/watch?v=yHUNeyaQKWY

4. Mininet commands and topology changes:

https://kiranvemuri.info/dynamic-topology-changes-in-mininet-advanced-users-5c452f7f302a