University of Southern California

EE597 Project – Software Defined Network

Broadband Network Architectures

by

Wanwiset Peerapatanapokin

Tingyang Sun

# **Table of Contents**

**Project Overview………………………………………………………………………………....3**

**Part 1 – High-level description of router design…………………………....………………….4**

**Part 2 – High-level description of switch design ……………………………………………....5**

**Part 3 – Data structures...……………………………………………………………………….6**

**Running the SDN and Results....…………………………………………………….………….7**

**References ………………………………………………………………………….………......14**

## **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**

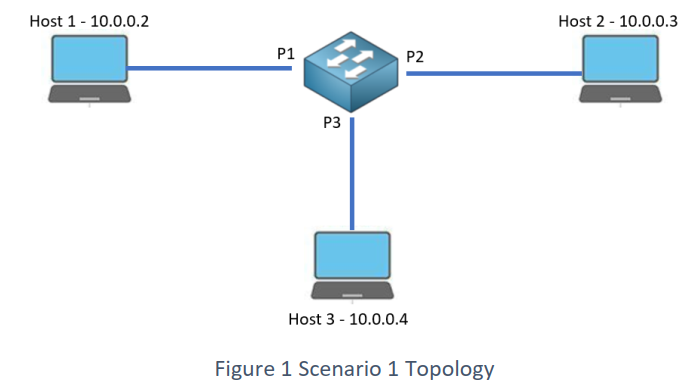
After installing Mininet VM, configure host-only network setup and boot the virtual system. No further installation is required. Next, transfer the project files to the VM via FTP. There are eight files in total, three controller files, three topology files, router.py and switch.py. Then, open two SSH 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:

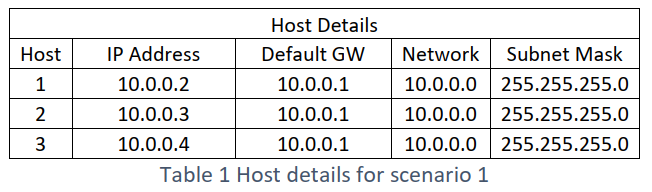
Controller: ./pox.py log.level --DEBUG misc.controller\_1

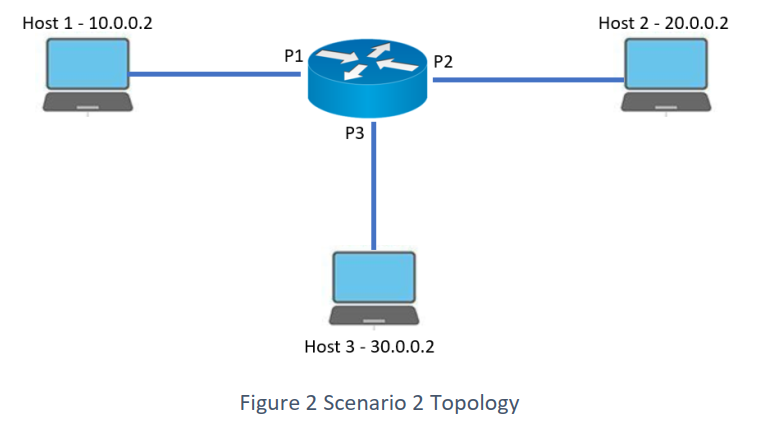
Mininet topology: sudo mn --custom topology\_1.py --topo mytopo --mac --controller remote

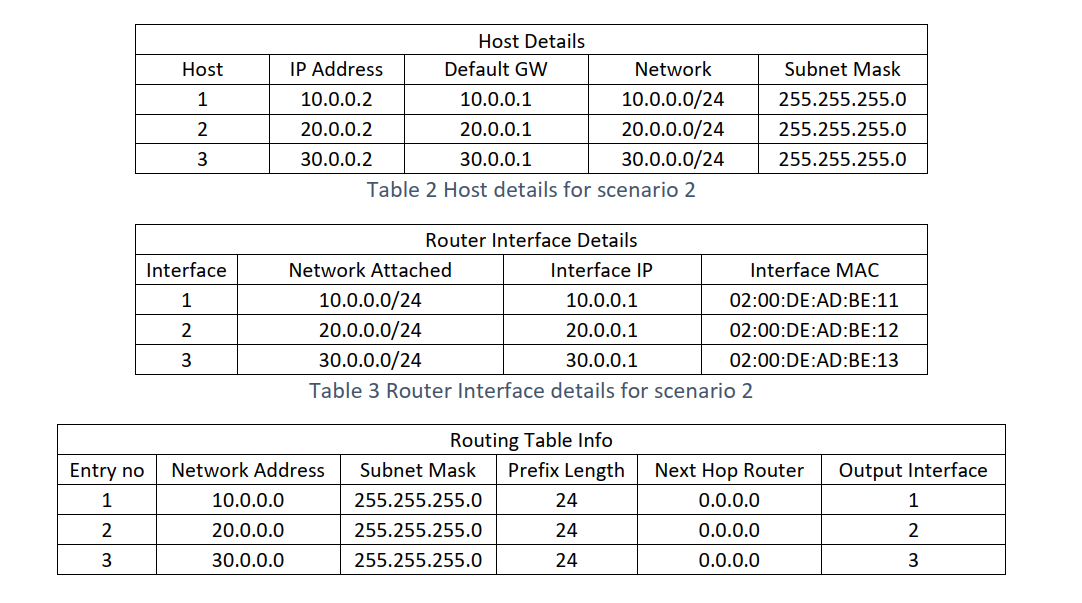
Change the controller and topology filename as necessary for different scenarios. Switch.py and router.py serve as general switch and router functions where all scenarios utilize. After exiting Mininet, always issue the command *‘sudo mn -c’* to clear out the topology and remaining data.

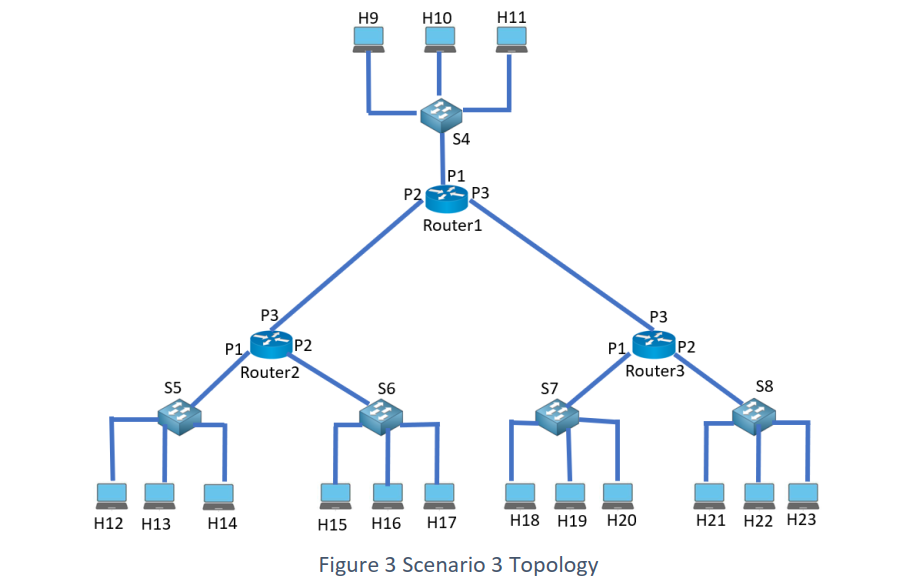
We investigate three scenarios in this project. Scenario 1 consists of a single switch and multiple hosts. Scenario 2 consists of a single router and multiple hosts. And Scenario 3 contains multiple switches, routers, and hosts. The scenarios’ topology, IP assignments and routing tables are shown in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

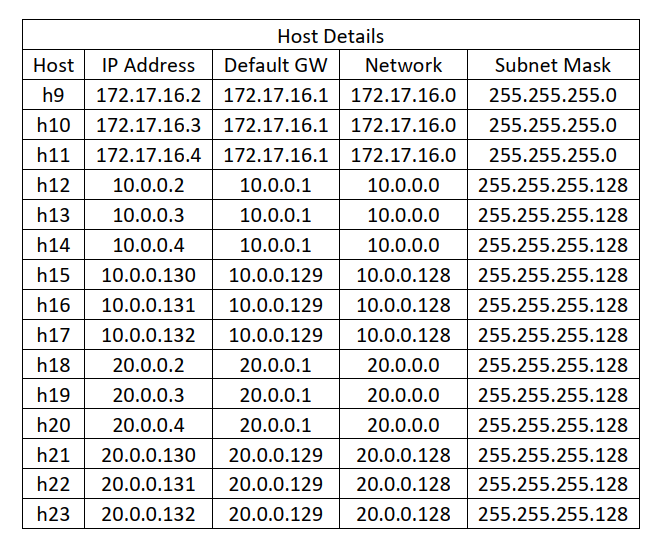


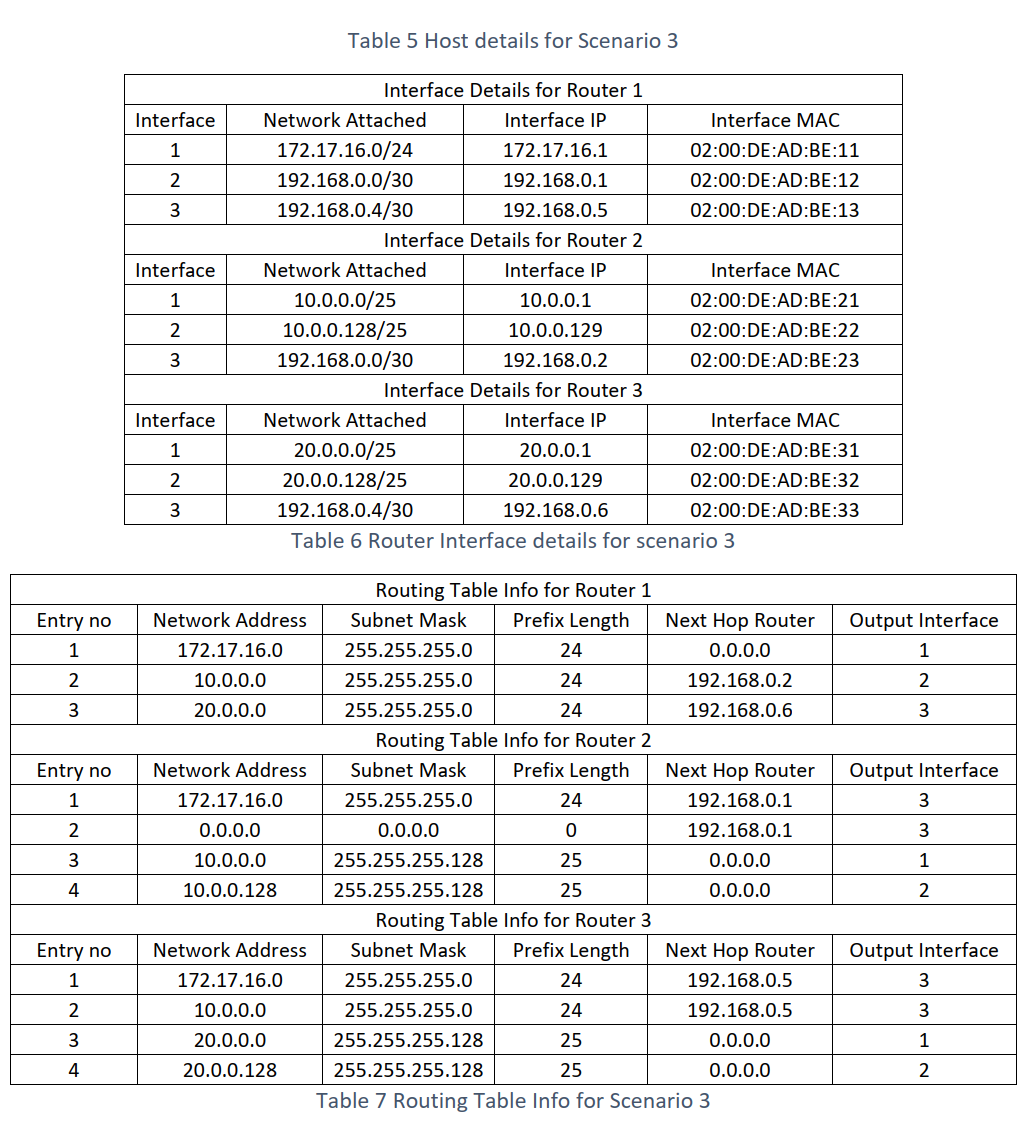


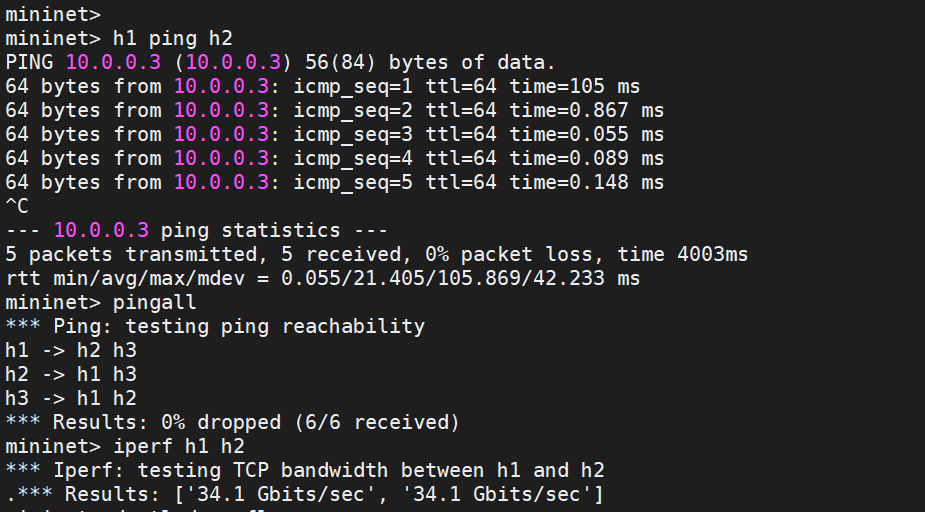


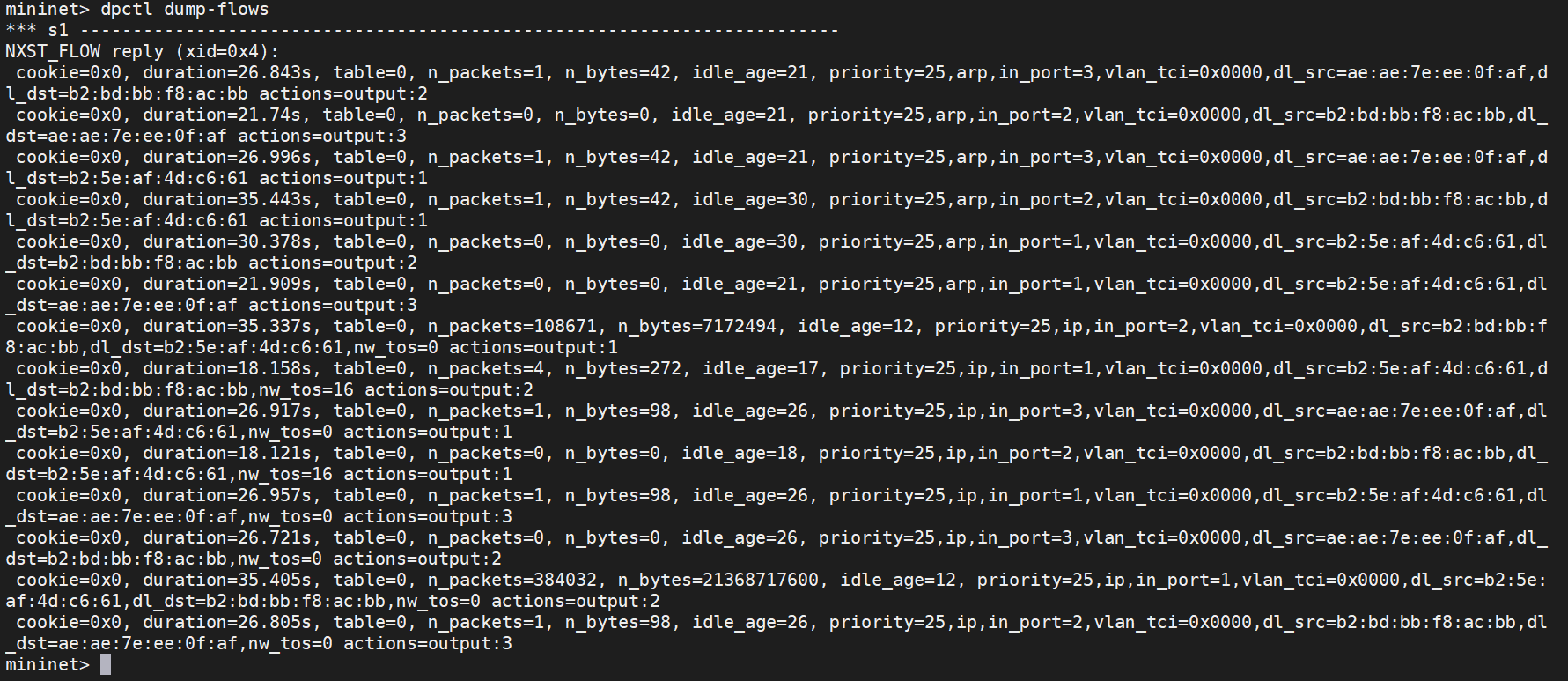




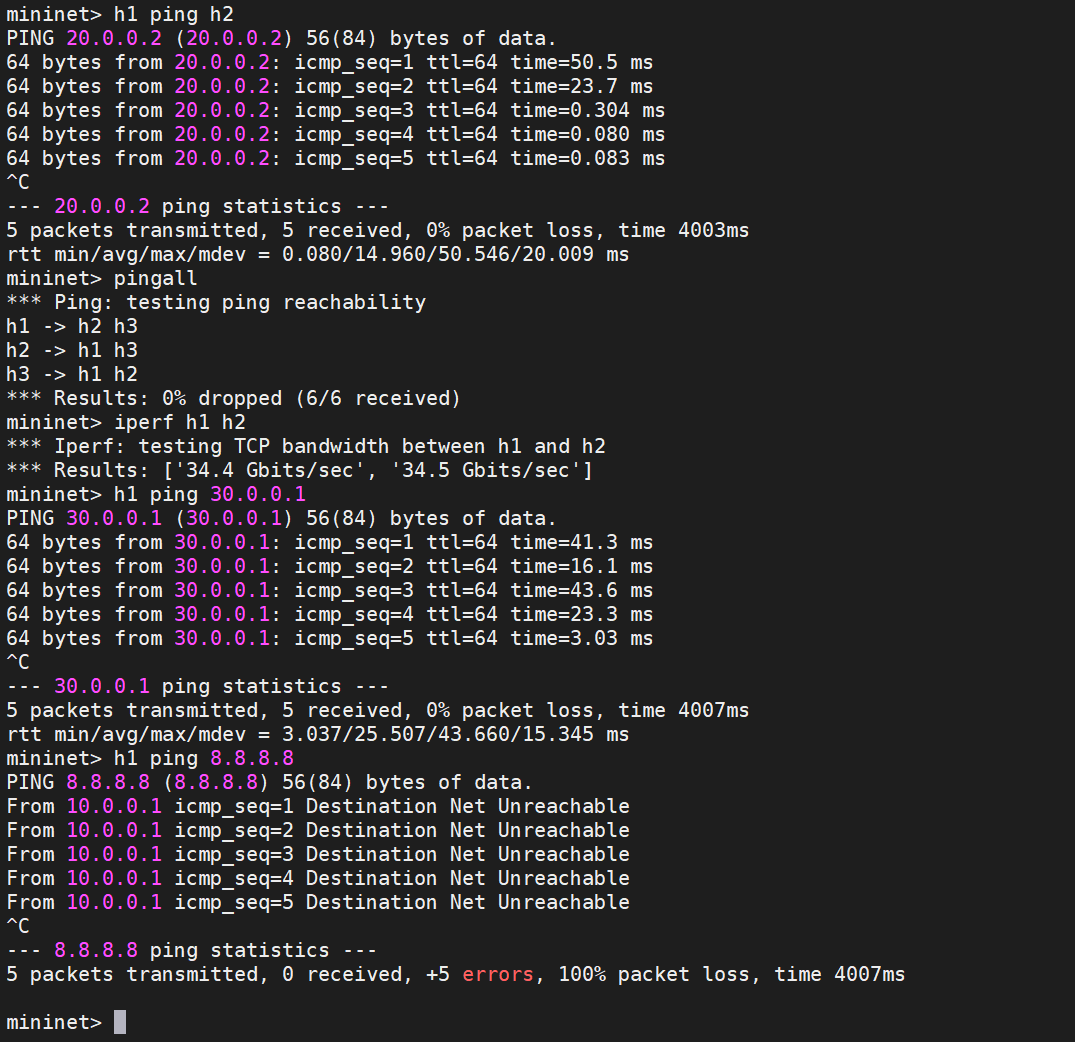


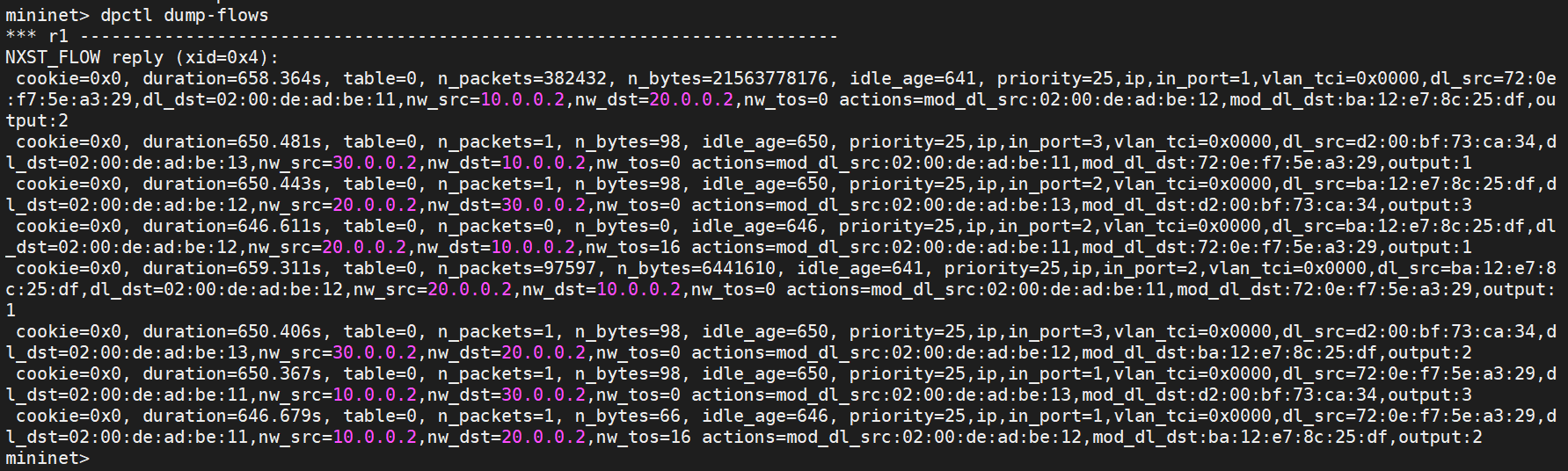


In scenario 1, we verify our SDN with a few commands as shown in the figure below. First, *h1 ping h2* is successful with no dropped packets. We also observe higher latency in the first packet phenomenon. Second, *pingall*, a mininet command where each host pings all other hosts, is issued. Mininet returns 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.

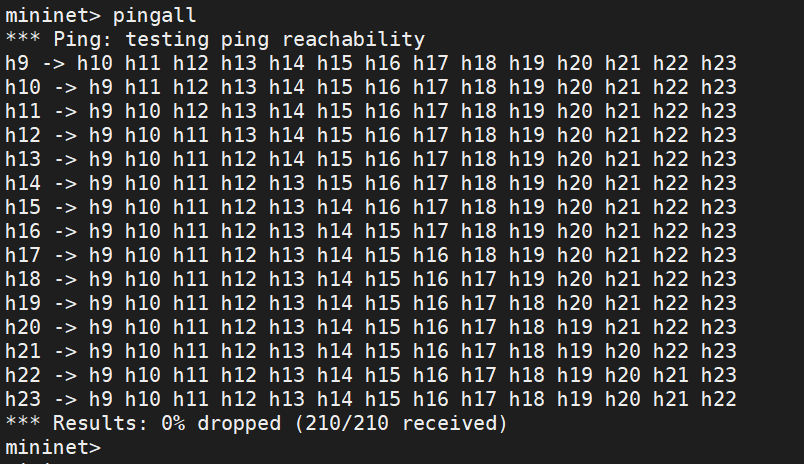
We can also verify the flows installed in the switch by using the command *dpctl dump-flows*, the result is as shown below.

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.



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

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.



## **References**

1. Mininet installation and tutorial:

<http://mininet.org/walkthrough/>

1. POX controller documentation:

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

1. David Mahler’s youtube guide on Mininet topologies:

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

1. Mininet commands and topology changes:

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