

CO515: Advances in Computer Networks: Selected Topics – Lab05

(Mininet Lab 02)

E/19/446: Wijerathna I.M.K.D.I.

02/05/2024

Lab Environment Setup

I used the same setup as the previous lab(Mininet Lab1) which is a lightweight ubuntu os setup which runs over vagrant+virtual box. The following is the lab environment structure.

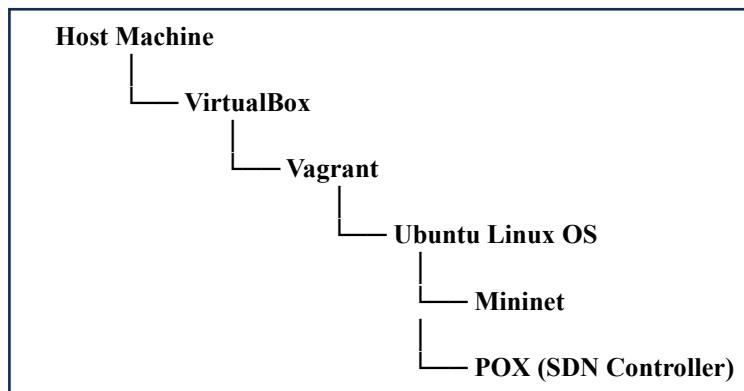


Figure 1: Lab Environment Setup

1. Use the command line to setup the topology

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

```
vagrant@sdn-box:~$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
h3 h3-eth0:s1-eth3
s1 lo:  s1-eth1:h1-eth0  s1-eth2:h2-eth0  s1-eth3:h3-eth0
c0
```

Then using a few verification commands, I checked the network configuration of the created topology.

- IP configurations of the hosts
 - It can be observed that the last hexadecimal value of the mac addresses of hosts 1,2, and 3 are 1,2,3 respectively. This is because the –mac flag usage in the mininet command earlier.

```
mininet> h1 ifconfig
h1-eth0 Link encap:Ethernet HWaddr 00:00:00:00:00:01
      inet addr:10.0.0.1 Bcast:10.255.255.255 Mask:255.0.0.0
      inet6 addr: fe80::200:ff:fe00:1/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
        RX packets:8 errors:0 dropped:0 overruns:0 frame:0
        TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:648 (648.0 B) TX bytes:738 (738.0 B)

lo      Link encap:Local Loopback
      inet addr:127.0.0.1 Mask:255.0.0.0
      inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING MTU:65536 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

mininet> h2 ifconfig
h2-eth0 Link encap:Ethernet HWaddr 00:00:00:00:00:02
      inet addr:10.0.0.2 Bcast:10.255.255.255 Mask:255.0.0.0
      inet6 addr: fe80::200:ff:fe00:2/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
        RX packets:8 errors:0 dropped:0 overruns:0 frame:0
        TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:648 (648.0 B) TX bytes:738 (738.0 B)

lo      Link encap:Local Loopback
      inet addr:127.0.0.1 Mask:255.0.0.0
      inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING MTU:65536 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

mininet>
```

- Network connections and nodes

```
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
h3 h3-eth0:s1-eth3
s1 lo: s1-eth1:h1-eth0 s1-eth2:h2-eth0 s1-eth3:h3-eth0
~o
mininet> nodes
available nodes are:
c0 h1 h2 h3 s1
```

In here, it could be observed that the corresponding network interface cards of each link and how they are connected. The ‘nodes’ command outputs all the nodes in the network. Which are:

- SDN controller – c0
- Openflow Switch – s1
- 3 hosts – h1, h2, h3

Based on the above results from verification commands, the following should be the topology created:

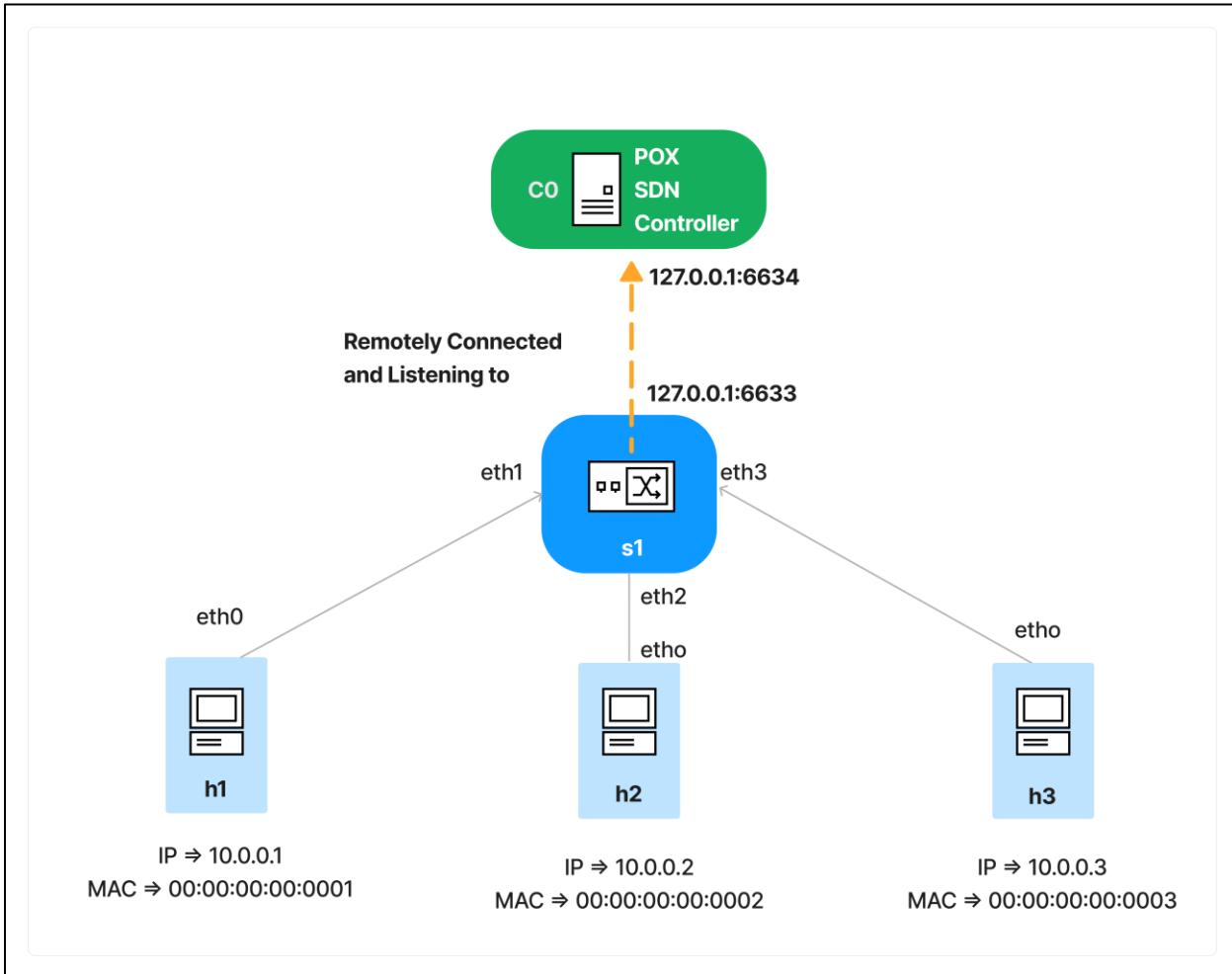


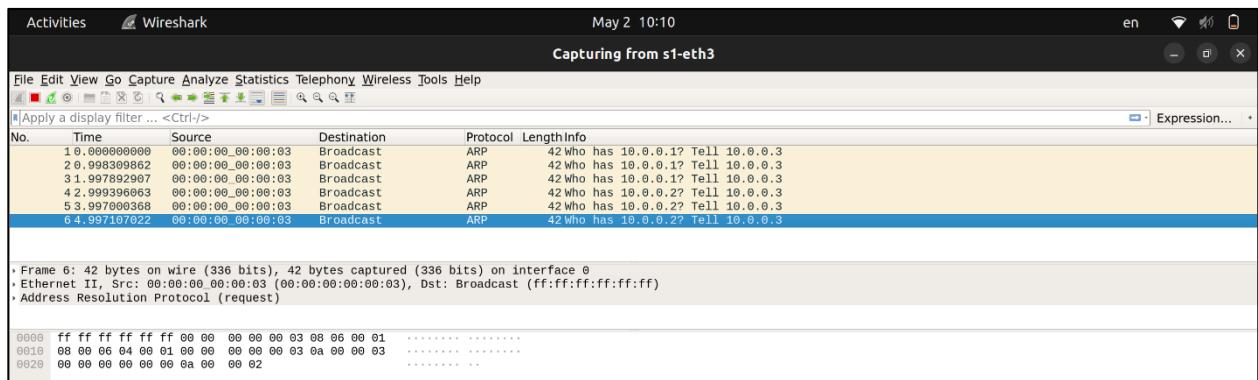
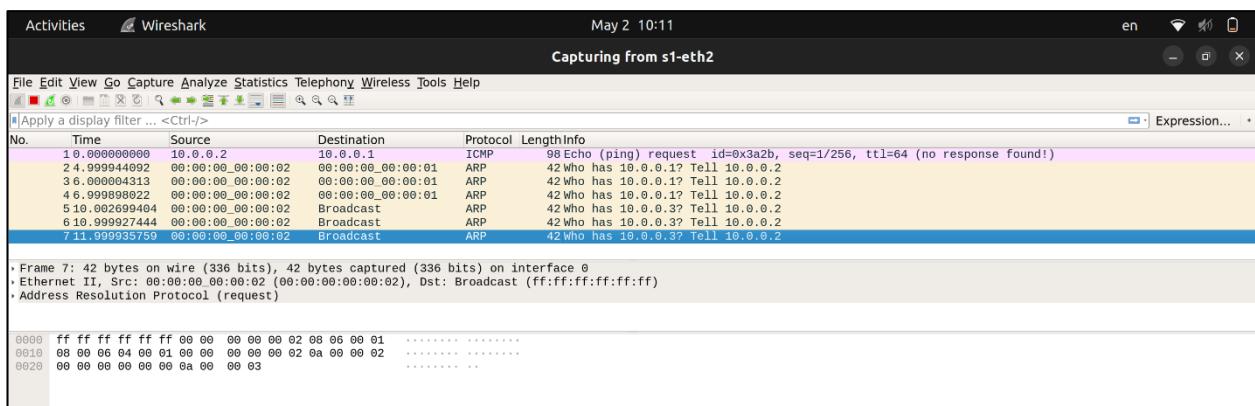
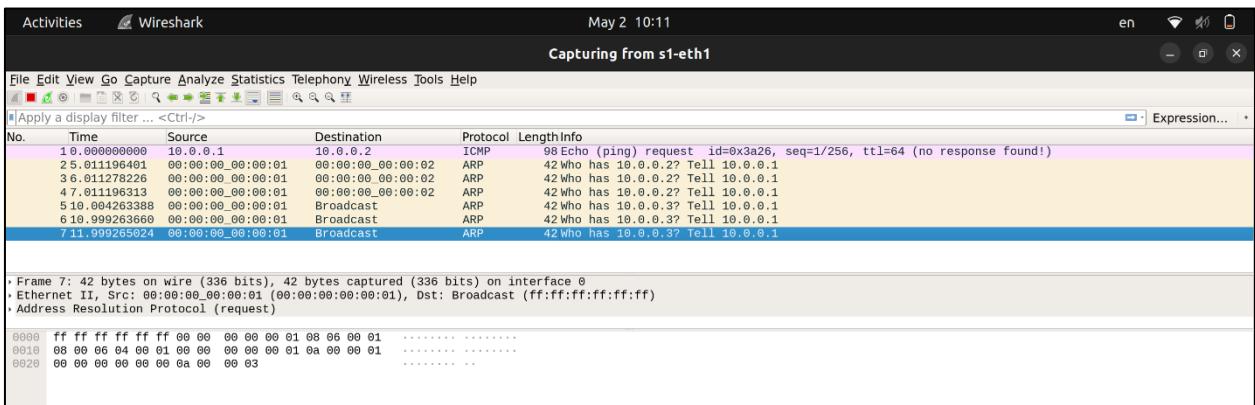
Figure 2: Assumed Topology Upon the Results of Verification Commands

2. Do mininet > pingall

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X
h2 -> X X
h3 -> X X
*** Results: 100% dropped (0/6 received)
mininet> █
```

This failed because no Openflow rules at the moment in the Openflow tables.

The following were the wireshark outputs received at Openflow switch's interfaces (s1-eth1, s1-eth2, s1-eth3) while this 'pingall' command was in operation.



It can be clearly observed that, only a few echo requests have been received along with ARP requests but none of them were replied because connectivity hasn't been configured yet through Openflow rules

3. Start another window do ‘man dpctl’ and ‘man ovs-dpctl’

```
dpctl(8)                               OpenFlow Manual                               dpctl(8)

NAME
    dpctl - administer OpenFlow datapaths

SYNOPSIS
    dpctl [options] command [switch] [args...]

DESCRIPTION
    The dpctl program is a command line tool for monitoring and administering OpenFlow datapaths. It is able to show the current state of a datapath, including features, configuration, and tables entries. When using the OpenFlow kernel module, dpctl is used to add, delete, modify, and monitor datapaths.

    Most dpctl commands take an argument that specifies the method for connecting to an OpenFlow switch. The following connection methods are supported:

    nl:dp_idx
        The local Netlink datapath numbered dp_idx. This form requires that the local host has the OpenFlow kernel module for Linux loaded.

    ssl:host[:port]
        The specified SSL port (default: 6633) on the given remote host. The --private-key, --certificate, and --ca-cert options are mandatory when this form is used.

    tcp:host[:port]
        The specified TCP port (default: 6633) on the given remote host.

    unix:file
        The Unix domain server socket named file.

COMMANDS
    With the dpctl program, datapaths running in the kernel can be created, deleted, and modified. A single machine may host up to 32 datapaths (numbered 0 to 31). In most situations, a machine hosts only one datapath.
    Manual page dpctl(8) line 1 (press h for help or q to quit)
```

This command shows the instructions on how to monitor Openflow data paths using dpctl commands.

```
ovs-dpctl(8)                           Open vSwitch Manual                           ovs-dpctl(8)

NAME
    ovs-dpctl - administer Open vSwitch datapaths

SYNOPSIS
    ovs-dpctl [options] command [switch] [args...]

DESCRIPTION
    The ovs-dpctl program can create, modify, and delete Open vSwitch datapaths. A single machine may host any number of datapaths.

    A newly created datapath is associated with only one network device, a virtual network device sometimes called the datapath's ``local port''. A newly created datapath is not, however, associated with any of the host's other network devices. To intercept and process traffic on a given network device, use the add-if command to explicitly add that network device to the datapath.

    If ovs-vswitchd(8) is in use, use ovs-vsctl(8) instead of ovs-dpctl.

    Most ovs-dpctl commands that work with datapaths take an argument that specifies the name of the datapath. Datapath names take the form [type@]name, where name is the network device associated with the datapath's local port. If type is given, it specifies the datapath provider of name, otherwise the default provider system is assumed.

    The following commands manage datapaths.

    add-dp dp [netdev[,option]...]
        Creates datapath dp, with a local port also named dp. This will fail if a network device dp already exists.

        If netdevs are specified, ovs-dpctl adds them to the new datapath, just as if add-if was specified.

    del-dp dp
        Deletes datapath dp. If dp is associated with any network devices, they are automatically removed.

    add-if dp netdev[,option]...
        Adds each netdev to the set of network devices datapath dp monitors, where dp is the name of an existing datapath, and netdev is the name of one of the host's network devices, e.g. eth0. Once a network device has been added to a datapath,
```

This command shows the instruction on how to configure Openflow rules using ovs-dpctl commands.

Basic functions like add, delete, methods and varies ways to perform new Openflow rules.

4. \$ dpctl show tcp:127.0.0.1:6634

```
vagrant@sdn-box:~$ dpctl show tcp:127.0.0.1:6634
features_reply (xid=0x4ee0137a): ver:0x1, dpid:1
n_tables:254, n_buffers:256
features: capabilities:0xc7, actions:0xffff
  1(s1-eth1): addr:f2:13:58:7e:c6:79, config: 0, state:0
    current: 10GB-FD COPPER
  2(s1-eth2): addr:a2:62:e7:7d:50:18, config: 0, state:0
    current: 10GB-FD COPPER
  3(s1-eth3): addr:4a:a5:46:f6:8d:25, config: 0, state:0
    current: 10GB-FD COPPER
  LOCAL(s1): addr:1e:6e:bf:38:3a:48, config: 0x1, state:0x1
get_config_reply (xid=0xa11821d1): miss_send_len=0
vagrant@sdn-box:~$ █
```

This command shows the possible Openflow data paths in the current topology

5. \$dpctl dump-flows tcp:127.0.0.1:6634

```
vagrant@sdn-box:~$ dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0x219428eb): flags=none type=1(flow)
vagrant@sdn-box:~$ █
```

This command shows the existing rules in the current flow table. Since there are no rules in our current switch, the empty table was returned.

6. Manually configure the OpenFlow switch with dpctl

- \$dpctl add-flow tcp:127.0.0.1:6634 in_port=1,idle_timeout=1000,actions=output:2
- \$dpctl add-flow tcp:127.0.0.1:6634 in_port=2,idle_timeout=1000,actions=output:1

```
vagrant@sdn-box:~$ dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0xd6c694917): flags=none type=1(flow)
vagrant@sdn-box:~$ dpctl add-flow tcp:127.0.0.1:6634 in_port=1,idle_timeout=1000,actions=output:2
vagrant@sdn-box:~$ dpctl add-flow tcp:127.0.0.1:6634 in_port=2,idle_timeout=1000,actions=output:1
vagrant@sdn-box:~$ dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0x4dd5936): flags=none type=1(flow)
  cookie=0, duration_sec=12s, duration_nsec=530000000s, table_id=0, priority=32768, n_packets=0, n_bytes=0, idle_timeout=1000, hard_timeout=0, in_port=1, actions=output:2
  cookie=0, duration_sec=3s, duration_nsec=972000000s, table_id=0, priority=32768, n_packets=0, n_bytes=0, idle_timeout=1000, hard_timeout=0, in_port=2, actions=output:1
vagrant@sdn-box:~$ █
```

After adding the 2 rules, the flow table is now filled with those two new rules, as shown above. These 2 rules should successfully connect h1 and h2.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 X
h2 -> h1 X
h3 -> X X
*** Results: 66% dropped (2/6 received)
mininet> █
```

Since only h1 and h2 are connected 'pingall' was only successful between those two hosts.

When the flow tables are rechecked after the above 'pingall', it can be observed that 7 packets have been matched those rule while the pingall process.

```
mininet> s1 dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0x55c29868): flags=none type=1(flow)
  cookie=0, duration_sec=139s, duration_nsec=586000000s, table_id=0, priority=32768, n_packets=7, n_bytes=406, idle_timeout=1000, hard_timeout=0, in_port=1, actions=output:2
  cookie=0, duration_sec=iisis, duration_nsec=28000000s, table_id=0, priority=32768, n_packets=7, n_bytes=406, idle_timeout=1000, hard_timeout=0, in_port=2, actions=output:1
mininet> █
```

Wireshark outputs of the observed at the Openflow switch's interfaces

No.	Time	Source	Destination	Protocol	Length	Info
1	00:00:00:000	00:00:00:00:00:01	Broadcast	ARP	42	Who has 10.0.0.2 Tell 10.0.0.1
2	00:00:01:44457	00:00:00:00:00:02	00:00:00:00:00:01	ARP	42	10.0.0.2 is at 00:00:00:00:00:02
3	00:00:01:45946	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0xb3be, seq=1/256, ttl=64 (reply in 4)
4	00:00:01:67319	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0xb3be, seq=1/256, ttl=64 (request in 3)
5	00:00:09:33576	00:00:00:00:00:01	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
6	00:00:09:786159	00:00:00:00:00:01	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
7	1.998921278	00:00:00:00:00:01	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
8	3.003859098	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0xb3b73, seq=1/256, ttl=64 (reply in 9)
9	3.003864746	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) reply id=0xb3b73, seq=1/256, ttl=64 (request in 8)
10	3.004579889	00:00:00:00:00:02	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.2
11	4.003016603	00:00:00:00:00:02	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.2
12	5.003866666	00:00:00:00:00:02	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.2
13	8.019174317	00:00:00:00:00:01	00:00:00:00:00:01	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
14	8.019197692	00:00:00:00:00:01	00:00:00:00:00:02	ARP	42	10.0.0.1 is at 00:00:00:00:00:01

No.	Time	Source	Destination	Protocol	Length	Info
1	00:00:00:000	00:00:00:00:00:01	Broadcast	ARP	42	Who has 10.0.0.2 Tell 10.0.0.1
2	00:00:00:08676	00:00:00:00:00:02	00:00:00:00:00:01	ARP	42	10.0.0.2 is at 00:00:00:00:00:02
3	00:00:00:30163	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0xb3be, seq=1/256, ttl=64 (reply in 4)
4	00:00:00:34954	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0xb3be, seq=1/256, ttl=64 (request in 3)
5	00:00:00:81053	00:00:00:00:00:01	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
6	00:00:07:67610	00:00:00:00:00:01	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
7	1.998858692	00:00:00:00:00:01	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
8	3.003663298	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0xb3b73, seq=1/256, ttl=64 (reply in 9)
9	3.003748287	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) reply id=0xb3b73, seq=1/256, ttl=64 (request in 8)
10	3.004434046	00:00:00:00:00:02	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.2
11	4.002815542	00:00:00:00:00:02	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.2
12	5.003677204	00:00:00:00:00:02	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.2
13	8.018825401	00:00:00:00:00:01	00:00:00:00:00:01	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
14	8.019105931	00:00:00:00:00:01	00:00:00:00:00:02	ARP	42	10.0.0.1 is at 00:00:00:00:00:01

No.	Time	Source	Destination	Protocol	Length	Info
1	00:00:00:000	00:00:00:00:00:01	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.3
2	00:00:00:00000	00:00:00:00:00:03	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.3
3	1.996461552	00:00:00:00:00:03	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.3
4	3.1.996499792	00:00:00:00:00:03	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.3
5	4.2.999407939	00:00:00:00:00:03	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.3
6	5.3.999514085	00:00:00:00:00:03	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.3
7	6.4.999509791	00:00:00:00:00:03	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.3

From these outputs, we can clearly observe that how echo request-responses have been exchanged between h1 and h2 while h3 haven't received any.

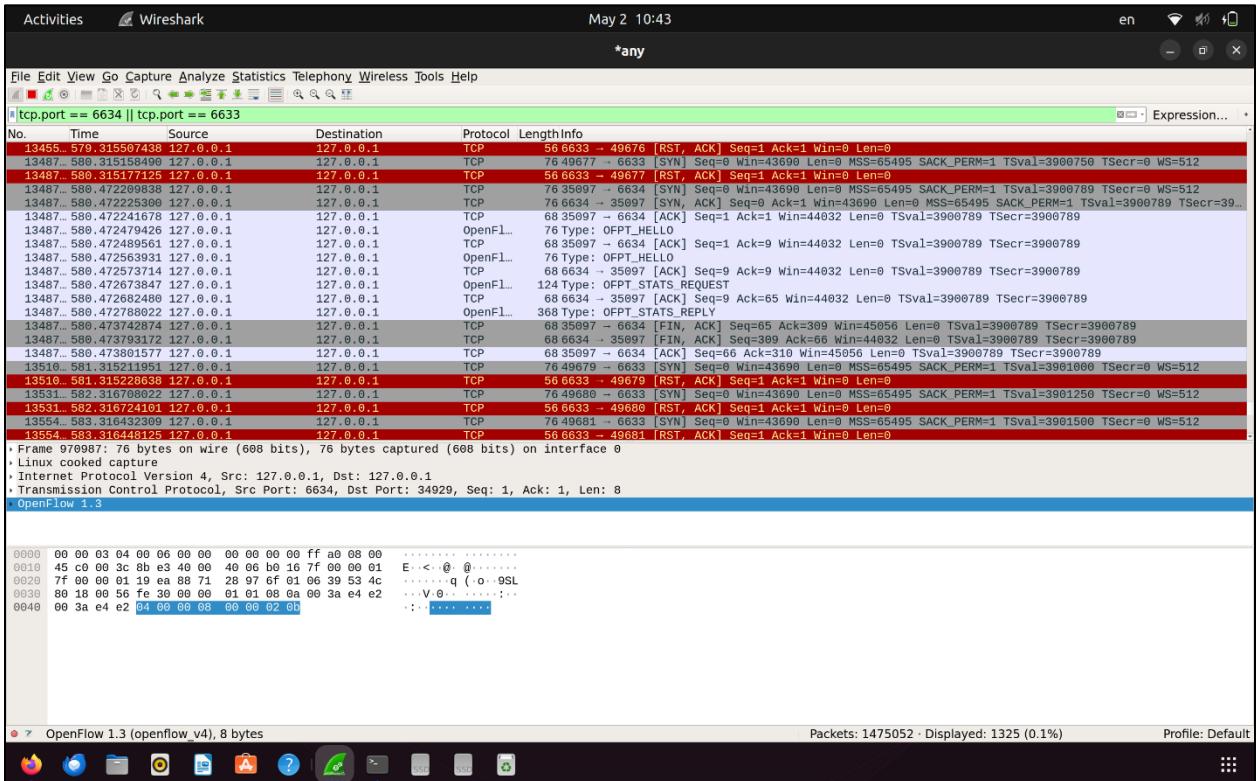
7. Try the following:

- \$dpctl add-flow tcp:127.0.0.1:6634 dl_dst=0:0:0:0:0:1,idle_timeout=1000,actions=output:1
- \$dpctl add-flow tcp:127.0.0.1:6634 dl_dst=0:0:0:0:0:2,idle_timeout=1000,actions=output:2
- \$dpctl add-flow tcp:127.0.0.1:6634 dl_dst=0:0:0:0:0:3,idle_timeout=1000,actions=output:3
- \$dpctl dump-flows tcp:127.0.0.1:6634
- Mininet> pingall

Following is the execution of the above rules and verification through looking up the flow table

```
vagrant@sdn-box:~$ dpctl add-flow tcp:127.0.0.1:6634
dpctl: 'add-flow' command requires at least 2 arguments
vagrant@sdn-box:~$ dpctl add-flow tcp:127.0.0.1:6634 dl_dst:0:0:0:0:0:1,idle_timeout=1000,actions=output:1
vagrant@sdn-box:~$ dpctl add-flow tcp:127.0.0.1:6634 dl_dst:0:0:0:0:0:2,idle_timeout=1000,actions=output:2
vagrant@sdn-box:~$ dpctl add-flow tcp:127.0.0.1:6634 dl_dst:0:0:0:0:0:3,idle_timeout=1000,actions=output:3
vagrant@sdn-box:~$ dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0x7683d91): flags=nom_type=1(flow)
  cookie=0, duration_sec=38s, duration_nsec=396000000s, table_id=0, priority=32768, n_packets=0, n_bytes=0, idle_timeout=1000,hard_timeout=0,dl_dst=00:00:00:00:00:02,actions=output
:2
  cookie=0, duration_sec=26s, duration_nsec=80000000s, table_id=0, priority=32768, n_packets=0, n_bytes=0, idle_timeout=1000,hard_timeout=0,dl_dst=00:00:00:00:00:03,actions=output
:3
  cookie=0, duration_sec=164s, duration_nsec=425000000s, table_id=0, priority=32768, n_packets=0, n_bytes=0, idle_timeout=1000,hard_timeout=0,dl_dst=00:00:00:00:00:01,actions=output
:t:1
vagrant@sdn-box:~$
```

While, executing the “`dpctl dump-flows`” was executed the following Openflow packets were captured at `tcp:127.0.0.1:6633`

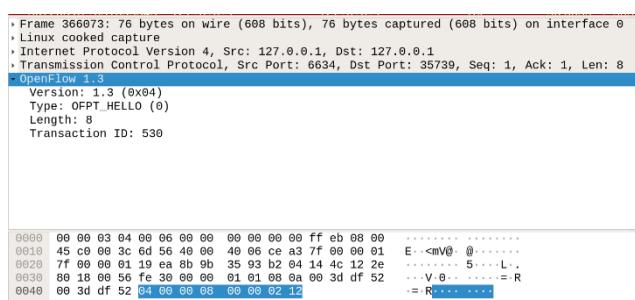


Types of Openflow packets observed:

- **OFPT_HELLO** : Openflow packet to initailize the communication with the Openflow switch (*s1*)
- **OFPT_STATS_REQUEST** : sent to sdn controller (*c0*) requesting the flow table statistics
- **OFPT_STATS_REPLY** : sent by the sdn controller with the current stats(flow rules)

More in-depth view on above Openflow packets

- **OFPT_HELLO** packet



- *OFPT_STATS_REQUEST packet*

```
• Transmission Control Protocol, Src Port: 35739, Dst Port: 6634, Seq: 9, Ack: 9, Len: 56
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_STATS_REQUEST (16)
  Length: 56
  Transaction ID: 1353492294
  Type: OFPST_FLOW (1)
  Flags: 0
  Wildcards: 4194303
  In port: 0
  Ethernet source address: 00:00:00:00:00:00 (00:00:00:00:00:00)
  Ethernet destination address: 00:00:00:00:00:00 (00:00:00:00:00:00)
  Input VLAN id: 0
  Input VLAN priority: 0
  Pad: 00
  DL type: 0
  IP TOS: 0
  IP protocol: 0
  0000 00 00 03 04 00 06 00 00 00 00 00 00 ff eb 08 00 ..... .
  0010 45 00 00 6c cd 83 40 00 40 06 ff 00 00 01 E 1 @ @ o
  0020 7f 00 00 01 8b 9b 19 ea 14 4c 12 36 35 93 b2 0c S .. L 65
  0030 80 18 00 56 fe 60 00 00 01 01 08 0a 00 3d df 52 V . . . = R
  0040 00 3d df 52 01 10 00 38 50 ac a7 46 00 01 00 00 =R . . P . F . .
  0050 00 3f ff ff 00 00 00 00 00 00 00 00 00 00 00 00 ? . . .
  0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
  wireshark_any_20240502052103_15Zwcw.pcapng
```

- *OFPT_STATS_REPLY packet*

```
• Frame 366079: 560 bytes on wire (4480 bits), 560 bytes captured (4480 bits) on interface 0
  • Linux cooked capture
  • Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
  • Transmission Control Protocol, Src Port: 6634, Dst Port: 35739, Seq: 9, Ack: 65, Len: 492
  • OpenFlow 1.0
    .000 0001 = Version: 1.0 (0x01)
    Type: OFPT_STATS_REPLY (17)
    Length: 492
    Transaction ID: 1353492294
    Type: OFPST_FLOW (1)
    - [Expert Info (Warning/Undecoded): Message data not dissected yet]
      [Message data not dissected yet]
      [Severity level: Warning]
      [Group: Undecoded]
    Flags: 0
  0000 00 00 03 04 00 06 00 00 00 00 00 00 ff eb 08 00 ..... .
  0010 45 c0 02 20 6d 59 49 00 40 06 cc bc 7f 00 00 01 E mY@ @ o
  0020 7f 00 00 01 19 ea 8b 9b 35 93 b2 0c 14 4c 12 6e S .. L n
  0030 80 18 00 56 00 15 00 00 01 01 08 0a 00 3d df 52 V . . . = R
  0040 00 3d df 52 01 11 01 ec 50 ac a7 46 00 01 00 00 =R . . P . F . .
  0050 00 60 00 00 00 38 20 fe 00 01 00 00 00 00 00 00 ? . . .
  0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
  wireshark any 20240502052103_15Zwcw.pcapng
```

As the above configured 3 rules connect h2 with h3 now all the connections except for h1-h3 must ping

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 X
h1 -> h1 h3
h3 -> X h2
*** Results: 33% dropped (4/6 received)
mininet>
```

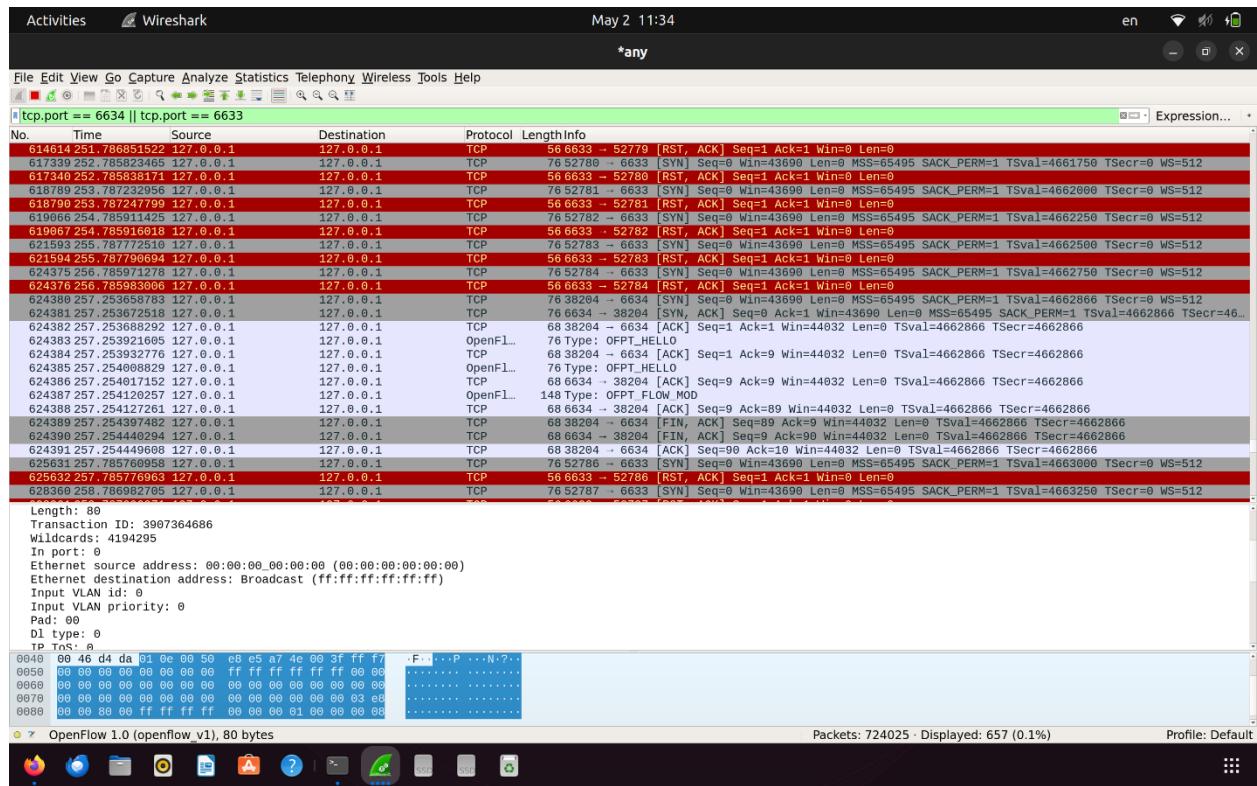
The expected result was achieved.

8. \$dpctl add-flow tcp:127.0.0.1:6634 dl_dst=ff:ff:ff:ff:ff:ff,idle_timeout=1000,actions=flood

```
vagrant@sdn-box:~$ dpctl add-flow tcp:127.0.0.1:6634 idle_timeout=1000,actions=flood
vagrant@sdn-box:~$ dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0x4c2c0e93): flags=none type=1(flow)
  cookie=0, duration_nsec=6s, duration_nsec=898000000s, table_id=0, priority=32768, n_packets=0, n_bytes=0, idle_timeout=1000, hard_timeout=0, actions=FLOOD
  cookie=0, duration_nsec=350s, duration_nsec=195000000s, table_id=0, priority=32768, n_packets=4, n_bytes=168, idle_timeout=1000, hard_timeout=0, in_port=1, actions=output:2
  cookie=0, duration_nsec=341s, duration_nsec=190000000s, table_id=0, priority=32768, n_packets=0, n_bytes=0, idle_timeout=1000, hard_timeout=0, in_port=2, actions=output:1
  cookie=0, duration_sec=939s, duration_nsec=700000000s, table_id=0, priority=32768, n_packets=11, n_bytes=798, idle_timeout=1000, hard_timeout=0, dl_dst=00:00:00:00:02, actions=output:2
  cookie=0, duration_sec=928s, duration_nsec=284000000s, table_id=0, priority=32768, n_packets=8, n_bytes=560, idle_timeout=1000, hard_timeout=0, dl_dst=00:00:00:00:03, actions=output:3
  cookie=0, duration_sec=1065s, duration_nsec=729000000s, table_id=0, priority=32768, n_packets=4, n_bytes=280, idle_timeout=1000, hard_timeout=0, dl_dst=00:00:00:00:01, actions=output:1
  tput:2
  tput:3
  tput:1
vagrant@sdn-box:~$
```

Adding this new Openflow rule.

This was the wireshark output received while adding this new rule:

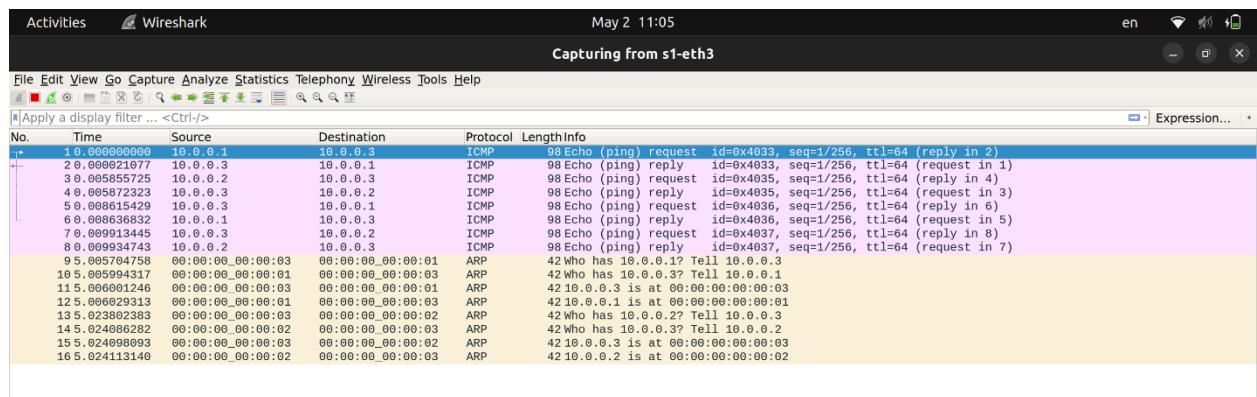


It can be observed that the 'OFPT_FLOW_MOD' packet contained the new rule information. Inside the packet we can see that the FLOOD rule has been sent as allowing from any MAC address to the broadcast mac address(ff:ff:ff:ff:ff:ff).

Now with this rule, the currently isolated h1-h3; must connect.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
mininet> 
```

As expected pingall was successful.



Activities Wireshark May 2 11:06

Capturing from s1-eth1

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	LengthInfo
1	10.0000000000	10.0.0.1	10.0.0.2	ICMP	98 Echo (ping) request id=0x4032, seq=1/256, ttl=64 (reply in 2)
2	0.000251318	10.0.0.2	10.0.0.1	ICMP	98 Echo (ping) reply id=0x4032, seq=1/256, ttl=64 (request in 2)
3	0.002876251	10.0.0.1	10.0.0.3	ICMP	98 Echo (ping) request id=0x4033, seq=1/256, ttl=64 (reply in 4)
4	0.003817626	10.0.0.3	10.0.0.1	ICMP	98 Echo (ping) reply id=0x4033, seq=1/256, ttl=64 (request in 3)
5	0.005795191	10.0.0.2	10.0.0.1	ICMP	98 Echo (ping) request id=0x4033, seq=1/256, ttl=64 (reply in 6)
6	0.00579546993	10.0.0.1	10.0.0.2	ICMP	98 Echo (ping) reply id=0x4034, seq=1/256, ttl=64 (request in 5)
7	0.005802753	10.0.0.3	10.0.0.1	ICMP	98 Echo (ping) request id=0x4035, seq=1/256, ttl=64 (reply in 8)
8	0.005802754	10.0.0.1	10.0.0.3	ICMP	98 Echo (ping) reply id=0x4035, seq=1/256, ttl=64 (request in 7)
9	0.00655029	00:00:00:00:00:01	00:00:00:00:00:02	ARP	42 Who has 10.0.0.2? Tell 10.0.0.1
10	0.0065502983	00:00:00:00:00:01	00:00:00:00:00:03	ARP	42 Who has 10.0.0.3? Tell 10.0.0.1
11.5	0.00932945	00:00:00:00:00:02	00:00:00:00:00:01	ARP	42 Who has 10.0.0.1?
12.5	0.009340336	00:00:00:00:00:01	00:00:00:00:00:02	ARP	42 10.0.0.1 is at 00:00:00:00:00:01
13.5	0.009348956	00:00:00:00:00:03	00:00:00:00:00:01	ARP	42 Who has 10.0.0.17?
14.5	0.009352565	00:00:00:00:00:01	00:00:00:00:00:03	ARP	42 10.0.0.1 is at 00:00:00:00:00:01
15.5	0.009368966	00:00:00:00:00:03	00:00:00:00:00:01	ARP	42 10.0.0.3 is at 00:00:00:00:00:03
16.5	0.00938697	00:00:00:00:00:02	00:00:00:00:00:01	ARP	42 10.0.0.2 is at 00:00:00:00:00:02

No.	Time	Source	Destination	Protocol	LengthInfo
1	10.0000000000	10.0.0.1	10.0.0.2	ICMP	98 Echo (ping) request id=0x4032, seq=1/256, ttl=64 (reply in 2)
2	0.00027115	10.0.0.2	10.0.0.1	ICMP	98 Echo (ping) reply id=0x4032, seq=1/256, ttl=64 (request in 1)
3	0.005593479	10.0.0.2	10.0.0.1	ICMP	98 Echo (ping) request id=0x4034, seq=1/256, ttl=64 (reply in 4)
4	0.005620775	10.0.0.1	10.0.0.2	ICMP	98 Echo (ping) reply id=0x4034, seq=1/256, ttl=64 (request in 3)
5	0.00537786	10.0.0.2	10.0.0.3	ICMP	98 Echo (ping) request id=0x4035, seq=1/256, ttl=64 (reply in 6)
6	0.00678504	10.0.0.3	10.0.0.2	ICMP	98 Echo (ping) reply id=0x4035, seq=1/256, ttl=64 (request in 5)
7	0.012698435	10.0.0.3	10.0.0.2	ICMP	98 Echo (ping) request id=0x4037, seq=1/256, ttl=64 (reply in 8)
8	0.012707667	10.0.0.2	10.0.0.3	ICMP	98 Echo (ping) reply id=0x4037, seq=1/256, ttl=64 (request in 7)
9	0.008472964	00:00:00:00:00:02	00:00:00:00:00:01	ARP	42 Who has 10.0.0.17? Tell 10.0.0.2
10.5	0.008718156	00:00:00:00:00:01	00:00:00:00:00:02	ARP	42 Who has 10.0.0.2?
11.5	0.008734841	00:00:00:00:00:02	00:00:00:00:00:01	ARP	42 10.0.0.2 is at 00:00:00:00:00:02
12.5	0.008752601	00:00:00:00:00:01	00:00:00:00:00:02	ARP	42 10.0.0.1 is at 00:00:00:00:00:01
13.5	0.026487964	00:00:00:00:00:02	00:00:00:00:00:03	ARP	42 Who has 10.0.0.3?
14.5	0.026881765	00:00:00:00:00:03	00:00:00:00:00:02	ARP	42 Who has 10.0.0.2?
15.5	0.026886631	00:00:00:00:00:02	00:00:00:00:00:03	ARP	42 10.0.0.2 is at 00:00:00:00:00:02
16.5	0.026908771	00:00:00:00:00:03	00:00:00:00:00:02	ARP	42 10.0.0.3 is at 00:00:00:00:00:03

It can be clearly observed that all the echo requests and responses were successfully exchanged with all the 3 hosts.

9. How to make the ping successful for one pair of hosts

Lets make ping succefull only for the pair h1-h3

I have used the following commands for this task

- To add the flow path to host-3


```
$ dpctl add-flow tcp:127.0.0.1:6634 dl_dst=0.0.0.3,idle_timeout=1000,actions=output:3
```
- To add flow path to host-1


```
$ dpctl add-flow tcp:127.0.0.1:6634 dl_dst=0.0.0.1,idle_timeout=1000,actions=output:1
```

Flow table Output

```
vagrant@sdn-box:~$ dpctl add-flow tcp:127.0.0.1:6634 dl_dst=0:0:0:0:0:3,idle_timeout=1000,actions=output:3
vagrant@sdn-box:~$ dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0xb911e75e): flags=none type=1(flow)
  cookie=0, duration_sec=28s, duration_nsec=367000000s, table_id=0, priority=32768, n_packets=0, n_bytes=0, idle_timeout=1000,hard_timeout=0,dl_dst=00:00:00:00:00:03,actions=output
:3
  cookie=0, duration_sec=828s, duration_nsec=753000000s, table_id=0, priority=32768, n_packets=4, n_bytes=280, idle_timeout=1000,hard_timeout=0,dl_dst=00:00:00:00:00:01,actions=output
:1
```

Successfully achieved the connectivity between h1 and h3

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X h3
h2 -> X X
h3 -> h1 X
*** Results: 66% dropped (2/6 received)
mininet>
```

Analysis on OpenFlow

OpenFlow Packet Structure

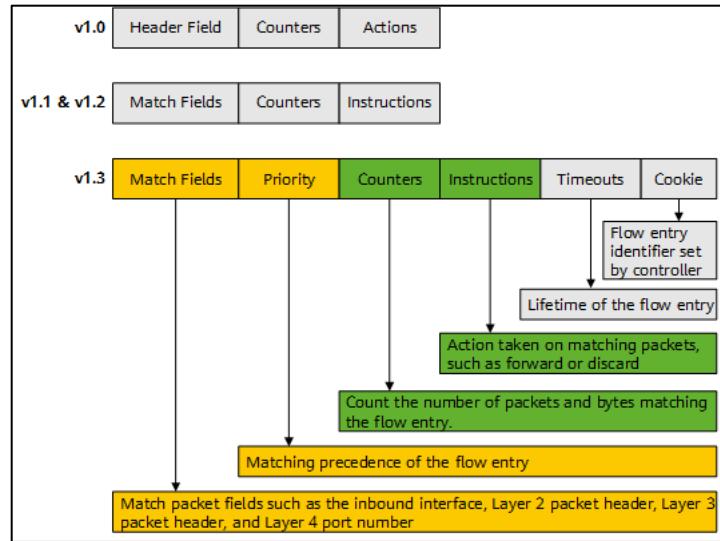


Figure 3: OpenFlow Packet Structure

Fields in an OpenFlow Packet

- **Header:** Contains essential information about the packet, including the version number, type, length, and transaction ID.

Depending on the Openflow header type the message body of that Openflow packet might vary. Common fields in the Openflow body are:

- **Hello:** Used for establishing a connection between OpenFlow controllers and switches.
- **Features Request/Reply:** Requests or provides information about the capabilities of the switch, such as the number of ports and supported features.
 - **Eg:** STATS_REQUEST/REPLY (when executing dpctl dump commands)
- **Flow Mod:** Instructs the switch to add, modify, or delete flow entries in its flow table.
- **Packet Out:** Instructs the switch to forward a packet to a specified port.
- **Packet In:** Sent by the switch to the controller when it encounters a packet for which it has no matching flow entry.
- **Flow Removed:** Indicates that a flow entry has been removed from the switch's flow table.
- **Error:** Indicates an error condition encountered during the processing of a message.

Different OpenFlow message types

- a. *Hello (OFPT_HELLO): Used for establishing a connection between the controller and switch. It includes the version negotiation.*
- b. *Echo Request/Reply (OFPT_ECHO_REQUEST/OFPT_ECHO_REPLY): Used for testing the responsiveness of the switch or controller.*
- c. *Features Request/Reply (OFPT_FEATURES_REQUEST/OFPT_FEATURES_REPLY): Requests or provides information about the switch's capabilities and features.*
- d. *Flow Mod (OFPT_FLOW_MOD): Instructs the switch to add, modify, or delete flow entries in its flow table.*
- e. *Packet Out (OFPT_PACKET_OUT): Instructs the switch to forward a packet to a specified port.*
- f. *Packet In (OFPT_PACKET_IN): Sent by the switch to the controller when it encounters a packet for which it has no matching flow entry.*
- g. *Flow Removed (OFPT_FLOW_REMOVED): Indicates that a flow entry has been removed from the switch's flow table.*
- h. *Error (OFPT_ERROR): Indicates an error condition encountered during the processing of a message.*