Networked Systems Coursework 1

Task 1

1. Modify topo1.py to create four hosts (h1-h4) connected to the same switch.
2. Run `h1 ping h4`, `iperf h1 h4`, `dump`, and `pingall`.

Solutions

1. `topo1.py` can be found in the `.zip` file
2. Below are images evidencing the running of the commands in the topo1 network topology

```
vagrant@mnvm:~$ sudo python3 topo1.py
mininet> host1 ping host4
PING 192.168.0.4 (192.168.0.4) 56(84) bytes of data.
64 bytes from 192.168.0.4: icmp_seq=1 ttl=64 time=0.000 ms
64 bytes from 192.168.0.4: icmp_seq=2 ttl=64 time=0.000 ms
64 bytes from 192.168.0.4: icmp_seq=3 ttl=64 time=3.71 ms
64 bytes from 192.168.0.4: icmp_seq=4 ttl=64 time=0.000 ms
64 bytes from 192.168.0.4: icmp_seq=5 ttl=64 time=0.000 ms
64 bytes from 192.168.0.4: icmp_seq=6 ttl=64 time=7.10 ms
64 bytes from 192.168.0.4: icmp_seq=7 ttl=64 time=4.01 ms
64 bytes from 192.168.0.4: icmp_seq=8 ttl=64 time=3.59 ms
64 bytes from 192.168.0.4: icmp_seq=9 ttl=64 time=3.51 ms
64 bytes from 192.168.0.4: icmp_seq=10 ttl=64 time=3.48 ms
64 bytes from 192.168.0.4: icmp_seq=11 ttl=64 time=0.310 ms
64 bytes from 192.168.0.4: icmp_seq=12 ttl=64 time=3.89 ms
64 bytes from 192.168.0.4: icmp_seq=13 ttl=64 time=0.999 ms
64 bytes from 192.168.0.4: icmp_seq=14 ttl=64 time=0.662 ms
^C
--- 192.168.0.4 ping statistics ---
14 packets transmitted, 14 received, 0% packet loss, time 48592ms
rtt min/avg/max/mdev = 0.000/2.232/7.102/2.145 ms
```

Fig 1.1: Running `h1 ping h4`

```
vagrant@mnvm:~$ sudo python3 topo1.py
mininet> iperf host1 host4
*** Iperf: testing TCP bandwidth between host1 and host4
*** Results: ['224 Mbits/sec', '248 Mbits/sec']
```

Fig 1.2: Running `iperf h1 h4`

```
vagrant@mnvm:~$ sudo python3 topo1.py
mininet> iperf host1 host4
*** Iperf: testing TCP bandwidth between host1 and host4
*** Results: ['224 Mbits/sec', '248 Mbits/sec']
mininet> dump
<Host host1: host1-eth0:192.168.0.1 pid=33734>
<Host host2: host2-eth0:192.168.0.2 pid=33736>
<Host host3: host3-eth0:192.168.0.3 pid=33738>
<Host host4: host4-eth0:192.168.0.4 pid=33740>
<OVSSwitch switch1: lo:127.0.0.1,switch1-eth1:None,switch1-eth2:None,switch1-eth3:None,switch1-eth4:None pid=33745>
<OVSController c0: 127.0.0.1:6653 pid=33727>
```

Fig 1.3: Running `dump`

```
mininet> pingall
*** Ping: testing ping reachability
host1 -> host2 host3 host4
host2 -> host1 host3 host4
host3 -> host1 host2 host4
host4 -> host1 host2 host3
*** Results: 0% dropped (12/12 received)
```

Fig 1.4: Running `pingall`

Task 2

1. Run `h1 ping h4`.
2. Run `dpctl dump-flows` and inspect the rules the controller installed. Provide screenshots in your report.
3. Briefly explain how `l2_learning.py` works.
4. When you run h1 ping h4 is there any chance you receive an ICMP packet in h2? Explain your answer.

Solutions

```
vagrant@mnvm:~$ sudo python3 topo2.py
mininet> h1 ping h4
PING 10.0.1.3 (10.0.1.3) 56(84) bytes of data.
64 bytes from 10.0.1.3: icmp_seq=3 ttl=64 time=0.000 ms
64 bytes from 10.0.1.3: icmp_seq=4 ttl=64 time=0.000 ms
64 bytes from 10.0.1.3: icmp_seq=5 ttl=64 time=3.46 ms
64 bytes from 10.0.1.3: icmp_seq=6 ttl=64 time=0.000 ms
64 bytes from 10.0.1.3: icmp_seq=7 ttl=64 time=0.000 ms
64 bytes from 10.0.1.3: icmp_seq=8 ttl=64 time=0.000 ms
64 bytes from 10.0.1.3: icmp_seq=9 ttl=64 time=5.28 ms
64 bytes from 10.0.1.3: icmp_seq=10 ttl=64 time=31.6 ms
64 bytes from 10.0.1.3: icmp_seq=11 ttl=64 time=1.60 ms
64 bytes from 10.0.1.3: icmp_seq=12 ttl=64 time=1.95 ms
64 bytes from 10.0.1.3: icmp_seq=13 ttl=64 time=0.801 ms
64 bytes from 10.0.1.3: icmp_seq=14 ttl=64 time=1.43 ms
^C
--- 10.0.1.3 ping statistics ---
14 packets transmitted, 12 received, 14.2857% packet loss, time 63315ms
rtt min/avg/max/mdev = 0.000/3.838/31.552/8.502 ms
```

Fig 2.1: Running `h1 ping h4`

1. Above is the console log when `h1 ping h4` is run.
2. Below is console log when `dpctl dump-flows` is run.

```
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
 cookie=0x0, duration=6.177s, table=0, n_packets=7, n_bytes=686, idle_timeout=10, hard_timeout=30,
 priority=65535,icmp,in_port="s1-eth4",vlan_tci=0x0000,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00
:00:01,nw_src=10.0.1.3,nw_dst=10.0.1.2,nw_tos=0,icmp_type=0,icmp_code=0 actions=output:"s1-eth1"
 cookie=0x0, duration=5.148s, table=0, n_packets=5, n_bytes=490, idle_timeout=10, hard_timeout=30,
 priority=65535,icmp,in_port="s1-eth1",vlan_tci=0x0000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00
:00:04,nw_src=10.0.1.2,nw_dst=10.0.1.3,nw_tos=0,icmp_type=8,icmp_code=0 actions=output:"s1-eth4"
```

Fig 2.2: Running `dpctl dump-flows`

3. The general idea of the learning switch is to dynamically 'learn' and build up a MAC address-to-port mapping table to decide where to forward incoming packets based on a source and destination MAC address.

The process can be split into several phases: * **Learning Phase**: The switch observes which port a source MAC address is received at and populates the mapping table with this relation.

- **Forwarding Logic**: If the destination MAC address is present in the mapping table, the switch will forward the packet through the associated port. However, if the destination MAC address is unknown, the switch will flood the packet across all channels to ensure the packet has reached its destination.

- **Dropping Rules**: For certain types of traffic (link-local or if the destination address is Bridge Filtered), the switch will drop the packets associated. The switch also conducts a sanity check ensuring that packets with the same source and destination addresses are dropped temporarily as it suggests there to be an error or loop in the network.

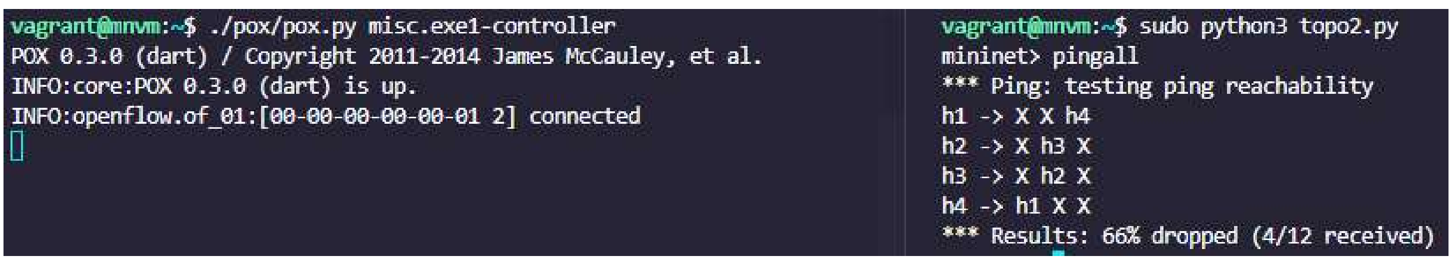
- **OpenFlow Rule Set-Up**: Once a packet flow (Source MAC <-> Input Port <-> Destination MAC) is learnt by the switch, the program leverages OpenFlow table by installing a flow rule for that specific flow. This allows for traffic to be automatically forwarded without the involvement of the controller. If the rule is unused for a period of 10s, the rule is discarded. Similarly, after a fixed period of 30s, the installed rule is removed.

4. There is no chance for `h1 ping h4` to cause ICMP packets to reach `h2`. The reasoning behind this is that as `h1` pings `h4`, once the initial ARP request is complete and `h4`'s MAC address is known, the controller ensures that subsequent ICMP packets will only be directed to `h4`'s port. Therefore, `h2` will never receive any ICMP packets destined for `h4` from `h1`

Task 3

1. Run `pingall`. `h1` and `h4` should be able to ping each other and `h2` and `h3` as well. Provide screenshots in your report.
2. Run `dpctl dump-flows` to see the rules you inserted. Provide the equivalent screenshot.

Solutions



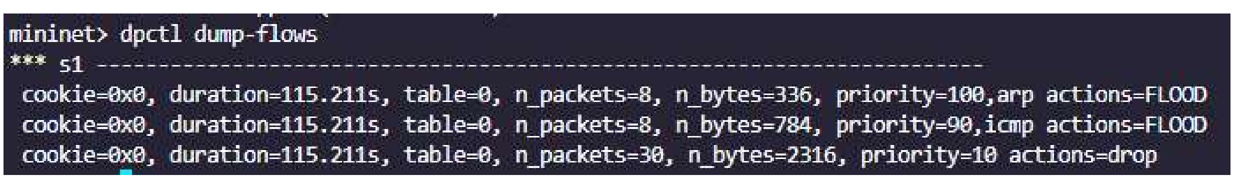
```
vagrant@mnvm:~$ ./pox/pox.py misc.exe1-controller
POX 0.3.0 (dart) / Copyright 2011-2014 James McCauley, et al.
INFO:core:POX 0.3.0 (dart) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
```

```
vagrant@mnvm:~$ sudo python3 topo2.py
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X h4
h2 -> X h3 X
h3 -> X h2 X
h4 -> h1 X X
*** Results: 66% dropped (4/12 received)
```

Fig 3.1: Running `pingall`

1. Above is the result of runing `pingall` with the new OpenFlow rules. We can observe that the new rules enable `h1` to reach `h4` and `h2` to reach `h3`.

2. Below is the result of running `dpctl_dump-flows` which shows the rules inserted in the switch. We can observe the three separate rules for ARP, ICMP and the rest of the traffic. Noting that ARP is given the highest priority to flood the network, followed by ICMP and then the catch-all rule to drop all other packets.



```
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
 cookie=0x0, duration=115.211s, table=0, n_packets=8, n_bytes=336, priority=100,arp actions=FLOOD
 cookie=0x0, duration=115.211s, table=0, n_packets=8, n_bytes=784, priority=90,icmp actions=FLOOD
 cookie=0x0, duration=115.211s, table=0, n_packets=30, n_bytes=2316, priority=10 actions=drop
```

Fig 3.2: Running `dpctl_dump-flows`