Exercise 5b – First Steps in Mininet

1. Start a Network (10P)

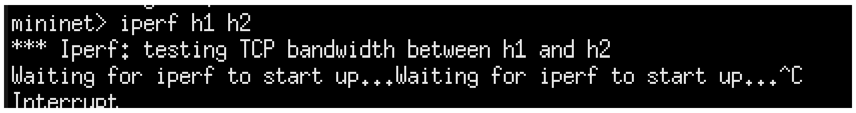
Let’s get back to our basic OpenFlow network from the lecture:
Assume you want to realize this topology in Python. What would you probably do? Define classes and methods for hosts, controllers, switches, link these entities, implement a specific controller, etc. In sum, hundreds or thousands of lines of code. In Mininet, you can do this in one line as shown in the lecture:

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

This command creates a topology with a single switch and three hosts, sets the MAC address of the hosts to be easily readable and defines that the controller instance will be running remotely. In one line, Mininet thus created three virtual hosts (each has a separate IP address), one software switch (with three ports), connected the virtual hosts to the
software switch (note: the switch is running in the kernel) with a ‘virtual Ethernet cable’ and set the MAC address of each host appropriately. Consider if you had to do this on your own.

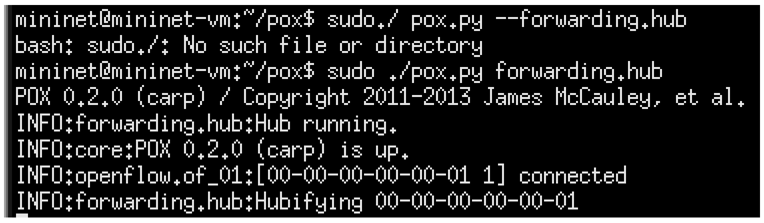
a. (10P) In the mininet console, check the ability to reach the other hosts in the network for each host h1, h2, h3. What do you observe? What is the reason for the result?

h1,h2, and h3 cannot connect to each other:



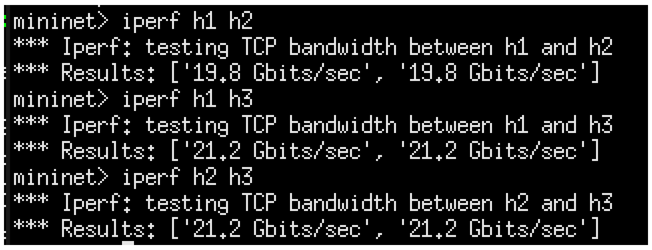
```
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
Waiting for iperf to start up...Waiting for iperf to start up...^C
Interrupt
```

Open a new terminal input   cd pox -> ./pox.py forwarding.hub. then the remote controller is connected.



```
mininet@mininet-vm:~/pox$ sudo./ pox.py --forwarding.hub
bash: sudo./: No such file or directory
mininet@mininet-vm:~/pox$ sudo ./pox.py forwarding.hub
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
INFO:forwarding.hub:Hub running.
INFO:core:POX 0.2.0 (carp) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
INFO:forwarding.hub:Hubifying 00-00-00-00-00-01
```

Then, h1, h2, and h3 can connect to each other:



```
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['19.8 Gbits/sec', '19.8 Gbits/sec']
mininet> iperf h1 h3
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['21.2 Gbits/sec', '21.2 Gbits/sec']
mininet> iperf h2 h3
*** Iperf: testing TCP bandwidth between h2 and h3
*** Results: ['21.2 Gbits/sec', '21.2 Gbits/sec']
```

b. (Optional) Play around with some commands introduced in the lecture, for example: *help, nodes, ifconfig*, etc.

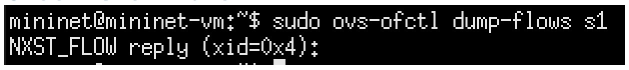
c. (Optional) Start xterm for the different devices as shown in the lecture. Play around with some commands here as well. What is the difference you observe when executing *ifconfig* at a host compared to executing the same command on the switch?

2. Flow tables (50P)

Currently, the network is not attached to a controller yet and the switch is not getting any instructions from any source. Have a look at the switches flow tables:

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

Check the flows



```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
```

What you are doing here is using the Open vSwitch (ovs) OpenFlow (of) Control (ctl) command to dump the flow rules currently installed at s1. You will observe that there are currently no rules installed, so let's change that:

```
$ sudo ovs-ofctl add-flow <switch> <match>,<actions>
```

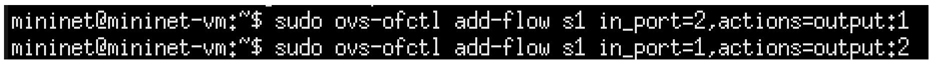
Will install the flow table entry <match>,<actions> into <switch>.

a. (20P) Add the following flow rules to s1:

i. Any packet that arrives on port 1 should be forwarded via port 2

ii. Any packet that arrives on port 2 should be forwarded via port 1

Add two flows.

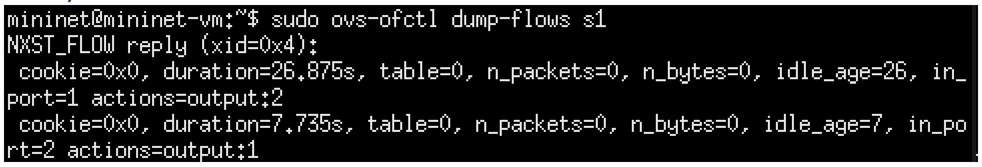


```
mininet@mininet-vm:~$ sudo ovs-ofctl add-flow s1 in_port=2,actions=output:1
mininet@mininet-vm:~$ sudo ovs-ofctl add-flow s1 in_port=1,actions=output:2
```

To be able to correctly express these rules, have a look at the man-page of ovs-ofctl.

To verify that your flows have been installed correctly, execute dump-flows for s1 again.

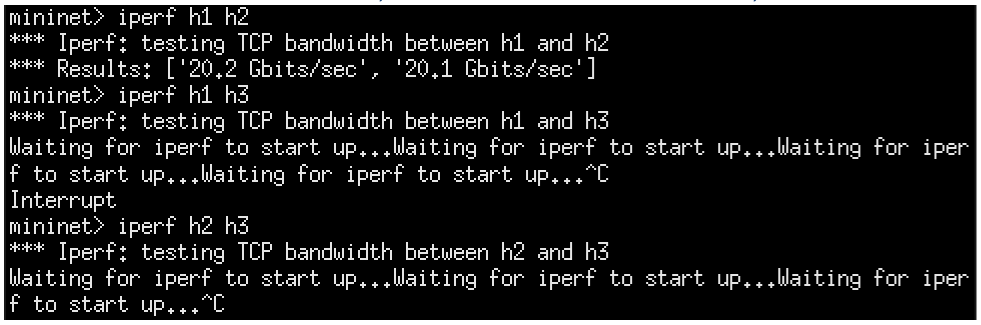
Verify the flows. There are two flows.



```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=26.875s, table=0, n_packets=0, n_bytes=0, idle_age=26, in_
port=1 actions=output:2
 cookie=0x0, duration=7.735s, table=0, n_packets=0, n_bytes=0, idle_age=7, in_po
rt=2 actions=output:1
```

(25P) Now test your network for connectivity among the hosts again. What do you observe? Please describe the OpenFlow messages exchanged within the network that have led to the current connectivity situation.

Host h1 and h2 are connected, h1 and h3 are not connected, h2 and h3 are not connected:



```
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['20.2 Gbits/sec', '20.1 Gbits/sec']
mininet> iperf h1 h3
*** Iperf: testing TCP bandwidth between h1 and h3
Waiting for iperf to start up...Waiting for iperf to start up...Waiting for iper
f to start up...Waiting for iperf to start up...^C
Interrupt
mininet> iperf h2 h3
*** Iperf: testing TCP bandwidth between h2 and h3
Waiting for iperf to start up...Waiting for iperf to start up...Waiting for iper
f to start up...^C
```

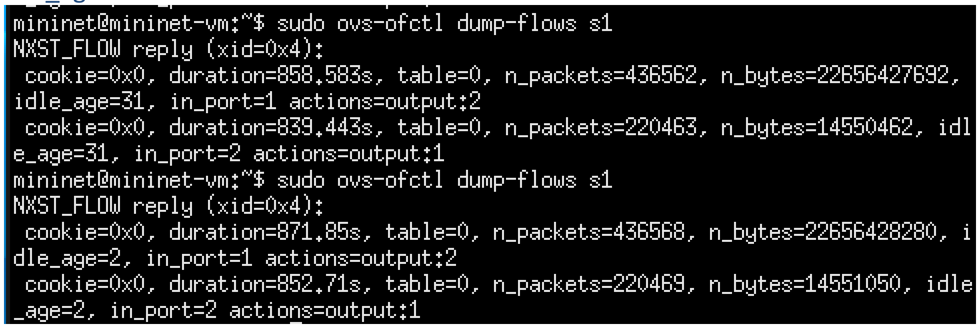
H1 and h2 can communicate, because when h1 send packets to switch, it send the packet to port 1 of the switch, and there is a rule that all packets from port 1 forward to port 2 which connects h2. In the same way, when h2 send packets to port 2 of the switch, there is a rule that all packets from port 2 forward to port 1 which connects h1.

For other situation, such as h1 want to send packet to h3, when the switch receive the packet from port 1 then it forward the packet to port 2 which connects the host h2, when h2 receive that packet, h2 will drop it, and h3 will never receive the packet from h1.

(5P) Dump the flows in s1 again. What has changed since the last dump-flows command? Why?

The duration, n_packets, n_bytes , and idle_age changed. The duration records the existing time of the rule, so it changes every time dump the flows. The n_packets and n_bytes record the data flow of the rules, when we use ping command, there are data exchange between h1 and h2 via those rules, so they change because the ping command was used. If there is no ping command or other data exchange between h1 and h2, the n_packets and n_bytes will remain the same between two dump flows.

The idle_age increase if there is no data exchange via those rules. If there is data change via those rules, the idle_age will be reset to 0.



```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=858.583s, table=0, n_packets=436562, n_bytes=22656427692,
idle_age=31, in_port=1 actions=output:2
 cookie=0x0, duration=839.443s, table=0, n_packets=220463, n_bytes=14550462, idl
e_age=31, in_port=2 actions=output:1
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=871.85s, table=0, n_packets=436568, n_bytes=22656428280, i
dle_age=2, in_port=1 actions=output:2
 cookie=0x0, duration=852.71s, table=0, n_packets=220469, n_bytes=14551050, idle
_age=2, in_port=2 actions=output:1
```

3. Let's analyze the network! (40P)

One popular tool to analyze networks in general is the protocol analyzer Wireshark. We can also analyze the behavior of OpenFlow with this tool. Start Wireshark by connecting to the VM with X forwarding enabled and then entering:

```
$ sudo wireshark &
```

A new window should show up. In Wireshark, we can now dissect our traffic. What we want to do is to get only the OpenFlow traffic to show up. In Wireshark, go to "Capture->Interfaces" and select the "lo" interface. This will cause Wireshark to capture the traffic that it sees on the loopback interface.

a. (10P) Why do we need to look at the loopback interface, rather than checking eth0 or eth1 for packets?

Because all virtual network built in this virtual machine, they use local loop IP address 127.0.0.1. For example, h1 is in this machine and h2 is in this machine, so they communicate via 127.0.0.1.

What we need to do next is to tell Wireshark that we only want to see the OpenFlow related traffic. For this, Wireshark offers the "filter" functionality. In the main Wireshark window, you should see a "filter" field. Into that field, simply enter:

```
of
```

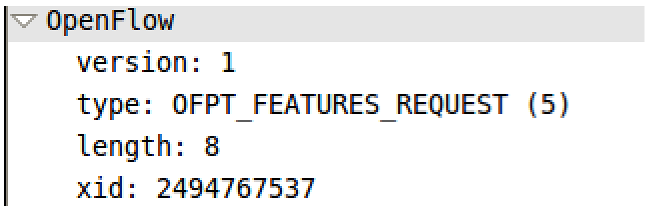
To see some action, we will now start a controller. In your SSH console, enter:

```
$ controller ptcp:
```

What this does is to start a controller that acts as a simple learning switch (remember the Computer Networks lecture?). Now you should see some packets being passed through Wireshark. You can select single packets by clicking on them. Try this with, for instance, the **of_features_reply** packet. If you click on it, you will see some information about the packet in the section below. Here, you can peak into the packet headers (Ethernet, IP, TCP) and OpenFlow message contents.

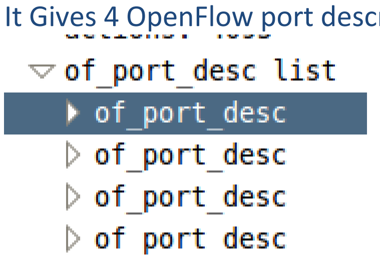
b. (10P) Click on "OpenFlow" and have a look at the information you get. Given the information that a controller asks a switch for available ports with an **of_features_request** message, describe what you see in the **of_features_reply** packet.

Of_features_request:



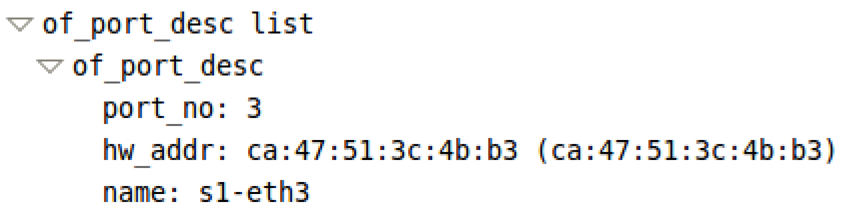
```
▽ OpenFlow
    version: 1
    type: OFPT_FEATURES_REQUEST (5)
    length: 8
    xid: 2494767537
```

Of_features_reply:
It Gives 4 OpenFlow port descriptions.



```
▽ of_port_desc list
   ▶ of_port_desc
   ▷ of_port_desc
   ▷ of_port_desc
   ▷ of_port_desc
```

Each port has a port number, a hardware address, and a connection name, in the figure below, it shows this port connects between switch 1 and Ethernet 3, the port number is 3, and the hardware address is ca:47…4b:b3.



```
▽ of_port_desc list
   ▽ of_port_desc
       port_no: 3
       hw_addr: ca:47:51:3c:4b:b3 (ca:47:51:3c:4b:b3)
       name: s1-eth3
```

One of the nice features of Mininet is that we can simply setup a new topology any time. To do that properly, we first have to shut down Mininet and clean-up the environment (e.g., remove external controllers like the one we started above). Stop the controller you have just started. Afterwards, proceed by:

```
mininet> exit
$ sudo mn -c
```

Now, we can start a new network. Start the most simple topology (two hosts, one switch, one OF reference controller) with:

```
$ sudo mn
```

Note that this time, in contrast to our example before, the controller runs inside the VM and is not located remotely. Our controller now is the OF reference controller. We now want to see how this controller automatically configures the switch.
Switch to Wireshark. You are probably seeing many echo-request/reply messages (do you have a guess what these are for?). We want to filter out these messages to be able to focus on the things that really happen in the network. For that, in the filter field of Wireshark, replace the “of” with:

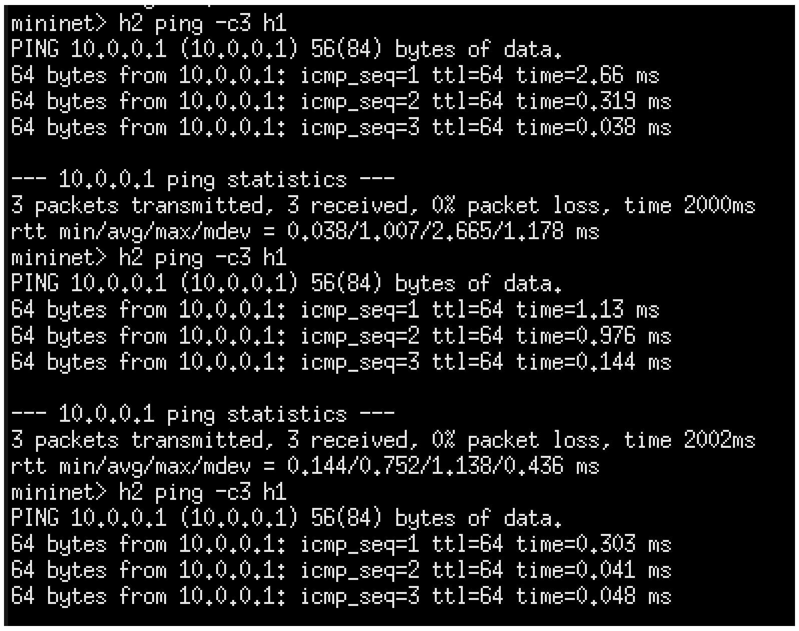
```
of and not (of10.echo_request.type or of10.echo_reply.type)
```

In some (outdated) versions of Wireshark you may have to use the following instead:

```
of && (of.type != 3) && (of.type != 2)
```

c. (20P) Ping host h2 from host h1 at least three times (ping –c3). Have a close look at the latency of the ping command. What do you observe? Explain this phenomenon with the help of Wireshark.

The ping latency decreased every time, but after 3 times, it will not decrease.



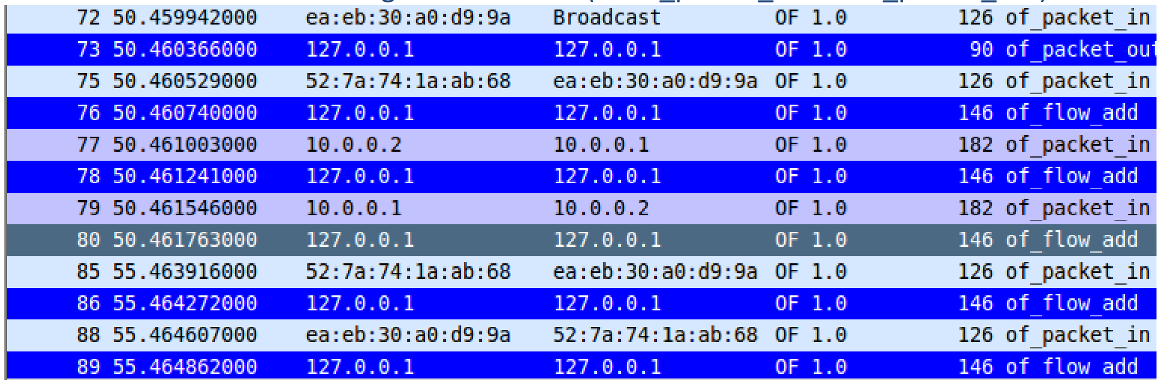
```
mininet> h2 ping -c3 h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=2.66 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.319 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.038 ms

--- 10.0.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.038/1.007/2.665/1.178 ms
mininet> h2 ping -c3 h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=1.13 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.976 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.144 ms

--- 10.0.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 0.144/0.752/1.138/0.436 ms
mininet> h2 ping -c3 h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.303 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.041 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.048 ms
```

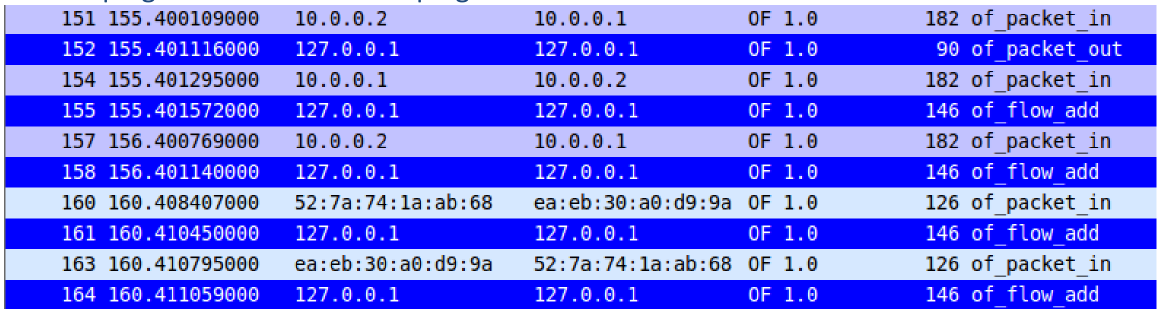
The third ping did not increase the records (packet-in and packet-out) in Wireshark, which means it did not have communication with controllers(no packet-in and packet-out), that is why it is the fastest.

The fact is that the first ping between h1 and h2 need to one broadcast of_packet_in and one hardware address of_packet_in to "learn" the network, then they add the rules(h1-h2, h2-h1) corresponding to the IP address, then they add the rules(h1-h2, h2-h1) corresponding to hardware address to communicate without asking the controllers(no of_packet_in and of_packet_out).



```
 72 50.459942000   ea:eb:30:a0:d9:9a   Broadcast           OF 1.0        126 of_packet_in
 73 50.460366000   127.0.0.1           127.0.0.1           OF 1.0         90 of_packet_out
 75 50.460529000   52:7a:74:1a:ab:68   ea:eb:30:a0:d9:9a   OF 1.0        126 of_packet_in
 76 50.460740000   127.0.0.1           127.0.0.1           OF 1.0        146 of_flow_add
 77 50.461003000   10.0.0.2            10.0.0.1            OF 1.0        182 of_packet_in
 78 50.461241000   127.0.0.1           127.0.0.1           OF 1.0        146 of_flow_add
 79 50.461546000   10.0.0.1            10.0.0.2            OF 1.0        182 of_packet_in
 80 50.461763000   127.0.0.1           127.0.0.1           OF 1.0        146 of_flow_add
 85 55.463916000   52:7a:74:1a:ab:68   ea:eb:30:a0:d9:9a   OF 1.0        126 of_packet_in
 86 55.464272000   127.0.0.1           127.0.0.1           OF 1.0        146 of_flow_add
 88 55.464607000   ea:eb:30:a0:d9:9a   52:7a:74:1a:ab:68   OF 1.0        126 of_packet_in
 89 55.464862000   127.0.0.1           127.0.0.1           OF 1.0        146 of_flow_add
```

We can confirm that theory by observing the record of second ping command, in the figure below, we see that the h1 and h2 already know each other, so they only need to add the rules corresponding to IP address and hardware address communication. That is why the second ping latency higher than third ping and lesser than first ping.



```
151 155.400109000   10.0.0.2            10.0.0.1            OF 1.0        182 of_packet_in
152 155.401116000   127.0.0.1           127.0.0.1           OF 1.0         90 of_packet_out
154 155.401295000   10.0.0.1            10.0.0.2            OF 1.0        182 of_packet_in
155 155.401572000   127.0.0.1           127.0.0.1           OF 1.0        146 of_flow_add
157 156.400769000   10.0.0.2            10.0.0.1            OF 1.0        182 of_packet_in
158 156.401140000   127.0.0.1           127.0.0.1           OF 1.0        146 of_flow_add
160 160.408407000   52:7a:74:1a:ab:68   ea:eb:30:a0:d9:9a   OF 1.0        126 of_packet_in
161 160.410450000   127.0.0.1           127.0.0.1           OF 1.0        146 of_flow_add
163 160.410795000   ea:eb:30:a0:d9:9a   52:7a:74:1a:ab:68   OF 1.0        126 of_packet_in
164 160.411059000   127.0.0.1           127.0.0.1           OF 1.0        146 of_flow_add
```

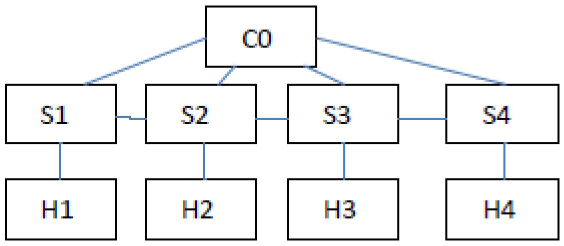
4. Play around with some topologies! (50P)

Here are some more commands for creating a topology. Play around with them, do not forget to exit and cleanup mininet after each experiment:

```
$ sudo mn --topo linear,4
```

a. (10P) Which topology does this command create?

It create this topology in the following figure:

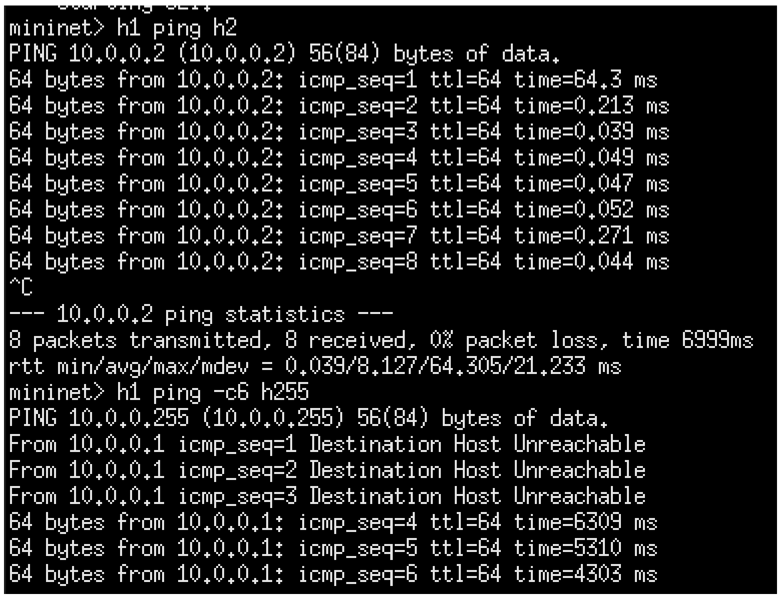


b. (20P) Start a linear topology with 256 switches. What can you observe when you try to ping host h2 from host h1, and then host h255 from host h1? Can you explain what has happened?

As the figure below, when h1 ping h2, the first latency is higher than a simple network (with 3 switches), but the following latency is acceptable.

When h1 ping h255, the first 3 ping latencies are so high and the result is unreachable. But following latencies decreases until it reach an acceptable latency, it eventually reach the latency between h1 and h2.

Which means in a big topology, the it takes a lot of time to "learn" the network, then it can work properly.



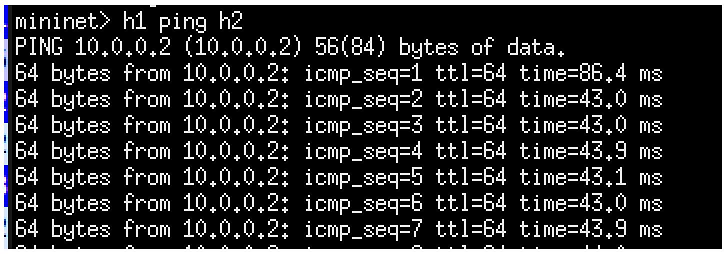
```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=64.3 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.213 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.039 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.049 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.047 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.052 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.271 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.044 ms
^C
--- 10.0.0.2 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 6999ms
rtt min/avg/max/mdev = 0.039/8.127/64.305/21.233 ms
mininet> h1 ping -c6 h255
PING 10.0.0.255 (10.0.0.255) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=6309 ms
64 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=5310 ms
64 bytes from 10.0.0.1: icmp_seq=6 ttl=64 time=4303 ms
```

Try this topology:

```
$ sudo mn --link tc,delay=10ms
```

c. (20P) What do you observe when you ping host h2 from host h1 now? Explain your observations.

The result is shown below:



```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=86.4 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=43.0 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=43.0 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=43.9 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=43.1 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=43.0 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=43.9 ms
```

The first ping takes 80+ms, because the trip path is h1-s1-c1-s1-h2-s1-c1-s2-h1, in every step it has 10ms delay, so it takes 80ms

Ping h2 from h1 is a round trip. The trip path is h1-s1-h2-s1-h1, every step takes 10ms delay, so the total time is 40+ms. When we use traffic control the real round trip delay is 4 times of the set delay.