Assignment 2 : Mininet and POX

Submitted By: Abhishek Ayachit (862188073)
ACN : Assignment 2 : Mininet and POX
Date: 06/07/2020

Part A: Hub Controller

1. **Have h1 ping h2 and h1 ping h5. How long did it take to ping? What is the difference? Which of the hosts and switches observe traffic?**

   1.
   h1 ping h2: 24.02 ms
   h1 ping h5: 30.62 ms

   2.
   Here, h1 ping h2 is much faster than h1 ping h5 as in the case of h1 ping h5, the packets are at a 4-hop distance and in case of h1 ping h2, the packets are at 2-hop distance.

   3.
   After running tcpdump, it was observed that the traffic is observed on all the hosts and switches as the controller acts as a hub, it sends arp packets to all.

2. **Run iperf h1 h2 and iperf h1 h5. What is the throughput? What is the difference?**

   1.

```
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['15.4 Mbits/sec', '15.9 Mbits/sec']
```

```
mininet> iperf h1 h5
*** Iperf: testing TCP bandwidth between h1 and h5
*** Results: ['2.87 Mbits/sec', '3.10 Mbits/sec']
```

   2.
   Similar to above explanation, h1-h2 has better bandwidth than h1-h5

3. **Run pingall to verify connectivity and dump the output.**

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5
h2 -> h1 h3 h4 h5
h3 -> h1 h2 h4 h5
h4 -> h1 h2 h3 h5
h5 -> h1 h2 h3 h4
*** Results: 0% dropped (20/20 received)
```

Part B: MAC learning controller

1. **Have h1 ping h2 and h1 ping h5. How long did it take to ping? What is the difference? Which of the hosts and switches observe traffic? How does this compare to the hub controller?**

   1.
   h1 ping h2: 17.02 ms
   h1 ping h5: 27.95 ms

   2.
   Similar to Part A, h1 ping h2 is faster than h1 ping h5

   3.
   After running tcpdump, all hosts and switched observed traffic for the first time as the controller is mapping mac to port. After the first time, no traffic was observed on not used hosts and switches.
   Then the traffic was seen on s1, h2 in case of h1 ping h2.
   In the case of h1 ping h5, traffic was seen on s1, s2, s3 and h5.

   4.
   Compared to Part A, the time required for ping in both cases(h2 and h5) is much less as each time the controller doesn't need to send an arp packet in order to choose the port. The mac to port translation is already stored in the controller for the first time.

2. **Run iperf h1 h2 and iperf h1 h5. What is the throughput? What is the difference? How does this compare to the hub controller?**

1.

```
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['8.85 Mbits/sec', '9.22 Mbits/sec']
```

```
mininet> iperf h1 h5
*** Iperf: testing TCP bandwidth between h1 and h5
*** Results: ['2.46 Mbits/sec', '2.65 Mbits/sec']
```

2.

Similar to Part A, h1-h2 has better bandwidth than h1-h5

3.

The bandwidth is almost similar for part A and part B.

3. **Run pingall to verify connectivity and dump the output.**

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5
h2 -> h1 h3 h4 h5
h3 -> h1 h2 h4 h5
h4 -> h1 h2 h3 h5
h5 -> h1 h2 h3 h4
*** Results: 0% dropped (20/20 received)
```

Part C: MAC learning switch

1. **Have h1 ping h2 and h1 ping h5. How long did it take to ping? What is the difference? Which of the hosts and switches observe traffic? How does this compare to Part A, and why?**

   1.
   h1 ping h2: 0.07 ms
   h1 ping h5: 0.08 ms

   2.
   Similar to Part A and B.

   3.
   Similar to Part B, all the hosts and switches observe traffic for the first time as the controller will install rules to the mac learning switches after that. After installing the rules, no traffic was observed on unrelated hosts and switches.

   4.
   Compared to Part A, the time required for ping is much less in both the cases(h2 and h5) as the controller is installing the newly learned rules on the switches. Therefore, the switch doesn't need to go to the controller in order to know the port on which the packet is to be transferred. The switch has already installed rules from the first packet. Therefore, it takes much less time to transfer the packet.

2. **Run iperf h1 h2 and iperf h1 h5. What is the throughput? What is the difference? How does this compare to Part A?**

   1.

```
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['1.17 Gbits/sec', '1.18 Gbits/sec']
```

```
mininet> iperf h1 h5
*** Iperf: testing TCP bandwidth between h1 and h5
*** Results: ['1.01 Gbits/sec', '1.01 Gbits/sec']
```

   2.
   Similar to Part A and B, h1-h2 has better bandwidth than h1-h5

3.
Compared to Part A, the bandwidth is much higher in case of this controller for both the cases(h2 and h5).

3. Run pingall to verify connectivity and dump the output.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5
h2 -> h1 h3 h4 h5
h3 -> h1 h2 h4 h5
h4 -> h1 h2 h3 h5
h5 -> h1 h2 h3 h4
*** Results: 0% dropped (20/20 received)
```

4. Dump the output of the flow rules using ovs-ofctl dump-flows. How many rules are there, and why?

There are many rules when *of.ofp_match.from_packet(packet)* is used
The reason there are so many rules, because we are installing the rules each time the traffic is flowing through the switch. So every time there are packets transferred through the switch, the rules are appending to it. This can be avoided by not installing the same rules more than once.

After Updating the code a little bit: (Posted in Appendix)

```
mininet> sh ovs-ofctl dump-flows s1
 cookie=0x0, duration=13.381s, table=0, n_packets=15, n_bytes=1078, idle_timeout=60, hard_timeout=600, dl_dst=c2:3e:36:7a:c0:85 actions=output:"s1-eth1"
 cookie=0x0, duration=13.340s, table=0, n_packets=14, n_bytes=980, idle_timeout=60, hard_timeout=600, dl_dst=1a:ca:09:2e:3f:3a actions=output:"s1-eth2"
 cookie=0x0, duration=13.296s, table=0, n_packets=13, n_bytes=938, idle_timeout=60, hard_timeout=600, dl_dst=4a:20:e9:8e:2a:78 actions=output:"s1-eth3"
 cookie=0x0, duration=13.188s, table=0, n_packets=8, n_bytes=616, idle_timeout=60, hard_timeout=600, dl_dst=ea:e0:a5:d1:8c:aa actions=output:"s1-eth4"
 cookie=0x0, duration=13.098s, table=0, n_packets=8, n_bytes=616, idle_timeout=60, hard_timeout=600, dl_dst=a6:97:62:12:0f:c2 actions=output:"s1-eth4"
```

Therefore, there are only 4 rules for s1 now. We have avoided the installation of rules again and again here. Similarly for s2 and s3, 5 rules can be seen as following:

```
mininet> sh ovs-ofctl dump-flows s2
 cookie=0x0, duration=40.263s, table=0, n_packets=7, n_bytes=518, idle_timeout=60, hard_timeout=600, dl_dst=c2:3e:36:7a:c0:85 actions=output:"s2-eth1"
 cookie=0x0, duration=40.232s, table=0, n_packets=8, n_bytes=616, idle_timeout=60, hard_timeout=600, dl_dst=ea:e0:a5:d1:8c:aa actions=output:"s2-eth2"
 cookie=0x0, duration=40.135s, table=0, n_packets=8, n_bytes=616, idle_timeout=60, hard_timeout=600, dl_dst=a6:97:62:12:0f:c2 actions=output:"s2-eth2"
 cookie=0x0, duration=40.034s, table=0, n_packets=7, n_bytes=518, idle_timeout=60, hard_timeout=600, dl_dst=1a:ca:09:2e:3f:3a actions=output:"s2-eth1"
 cookie=0x0, duration=39.861s, table=0, n_packets=6, n_bytes=420, idle_timeout=60, hard_timeout=600, dl_dst=4a:20:e9:8e:2a:78 actions=output:"s2-eth1"
mininet> sh ovs-ofctl dump-flows s3
 cookie=0x0, duration=42.388s, table=0, n_packets=7, n_bytes=518, idle_timeout=60, hard_timeout=600, dl_dst=c2:3e:36:7a:c0:85 actions=output:"s3-eth3"
 cookie=0x0, duration=42.373s, table=0, n_packets=12, n_bytes=896, idle_timeout=60, hard_timeout=600, dl_dst=ea:e0:a5:d1:8c:aa actions=output:"s3-eth1"
 cookie=0x0, duration=42.271s, table=0, n_packets=11, n_bytes=854, idle_timeout=60, hard_timeout=600, dl_dst=a6:97:62:12:0f:c2 actions=output:"s3-eth2"
 cookie=0x0, duration=42.152s, table=0, n_packets=6, n_bytes=420, idle_timeout=60, hard_timeout=600, dl_dst=1a:ca:09:2e:3f:3a actions=output:"s3-eth3"
 cookie=0x0, duration=42.056s, table=0, n_packets=6, n_bytes=420, idle_timeout=60, hard_timeout=600, dl_dst=4a:20:e9:8e:2a:78 actions=output:"s3-eth3"
```

Part D: Simplified IP Router

1. **Have h1 ping h2 and h1 ping h5. How long did it take to ping? What is the difference? Which of the hosts and switches observe traffic? How does this compare to the previous controllers?**

   1.
   h1 ping h2: 0.08 ms
   h1 ping h5: 0.11 ms

   2.
   Similar to Part A,B and C.

   3.
   After running tcpdump, in the case of h1 ping h2, it was seen that traffic was observed on h2,h3,s1 and s2. The reason for it was s1 is a mac learning switch and we have added rules on s2. So s1 will send arp packets to all the hosts and switches connected to it. Therefore, h1,h2,h3,s2 shows traffic. S2 doesn't forward it as the rules are already installed on it.
   In case of h1 ping h5, traffic single arp packet was observed on h2, h3. As rules are already installed on s2, it directly sends the traffic to h5. Therefore, all other traffic is shown on s1,s2,s3,h5.

   4.
   Compared to previous controllers, the time taken in this case is much faster than Part A and Part B. The time taken is almost similar to that taken for Part C. As per the explanation above, time taken should be a little less than Part B, but that time is almost negligible.

2. **Run iperf h1 h2 and iperf h1 h5. What is the throughput? What is the difference? How does this compare to the previous controllers?**

   1.

```
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['1.01 Gbits/sec', '1.01 Gbits/sec']
```

```
mininet> iperf h1 h5
*** Iperf: testing TCP bandwidth between h1 and h5

*** Results: ['902 Mbits/sec', '903 Mbits/sec']
```

2.
Similar to Part A,B and C, h1-h2 has better bandwidth than h1-h5

3.
Compared to Part A and Part B, the bandwidth is much higher for Part D in both the cases(h2 and h5). Compared to Part C, the bandwidth is almost similar.

3. Run pingall to verify connectivity and dump the output.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5
h2 -> h1 h3 h4 h5
h3 -> h1 h2 h4 h5
h4 -> h1 h2 h3 h5
h5 -> h1 h2 h3 h4
*** Results: 0% dropped (20/20 received)
```

4. Dump the output of the flow rules using ovs-ofctl dump-flows. How many rules are there, and why?

```
mininet> sh ovs-ofctl dump-flows s2
 cookie=0x0, duration=988.833s, table=0, n_packets=8, n_bytes=784, icmp,nw_dst=10.0.1.0/24 actions=output:"s2-eth2"
 cookie=0x0, duration=988.826s, table=0, n_packets=6, n_bytes=588, icmp,nw_dst=10.0.0.0/24 actions=output:"s2-eth1"
 cookie=0x0, duration=988.819s, table=0, n_packets=7, n_bytes=294, arp,arp_tpa=10.0.1.0/24 actions=output:"s2-eth2"
 cookie=0x0, duration=988.812s, table=0, n_packets=9, n_bytes=378, arp,arp_tpa=10.0.0.0/24 actions=output:"s2-eth1"
```

s1 and s3 have the same rules installed as in the last questions.
There are only 4 rules for s2 as we have installed these manually. These rules are sufficient for the traffic that is going to be transferred from the switch as there are only 2 interfaces from which the packets might travel.

5. How does this network compare to your previous controllers? Which is better, and why?

In the part C controller, the controller sends arp packet to all in order to learn and install the rules on switches whereas in the case of this controller, as we have already installed rules on s2, there is no need to send arp packets to all the hosts. This reduces the overall traffic in the network. Therefore, this controller performs better.

Appendix
Part A:

```python
from mininet.topo import Topo

class MyTopo( Topo ):
    "Simple topology example."

    def build( self ):
        "Create custom topo."

        # Add hosts and switches
        h1 = self.addHost( 'h1' )
        h2 = self.addHost( 'h2' )
        h3 = self.addHost( 'h3 ')
        h4 = self.addHost( 'h4 ')
        h5 = self.addHost( 'h5 ')

        #h1 = self.addHost( 'h1', ip='10.0.0.1/16')
        #h2 = self.addHost( 'h2', ip='10.0.0.2/16')
        #h3 = self.addHost( 'h3', ip='10.0.0.3/16')
        #h4 = self.addHost( 'h4', ip='10.0.1.2/16')
        #h5 = self.addHost( 'h5', ip='10.0.1.3/16')

        s1 = self.addSwitch( 's1' )
        s2 = self.addSwitch( 's2' )
        s3 = self.addSwitch( 's3' )


        # Add links
        self.addLink( h1, s1 )
        self.addLink( h2, s1 )
        self.addLink( h3, s1 )

        self.addLink( s1, s2 )
        self.addLink( s2, s3 )

        self.addLink( h4, s3 )
        self.addLink( h5, s3 )


topos = { 'mytopo': ( lambda: MyTopo() ) }
```

Part B:

```python
def act_like_switch (self, packet, packet_in):
  """
  Implement switch-like behavior.
  """

  # Here's some psuedocode to start you off implementing a learning
  # switch.  You'll need to rewrite it as real Python code.

  # Learn the port for the source MAC
  if packet.src not in self.mac_to_port:
    log.debug("Learning: MAC: %s from Port: %d" % (packet.src, packet_in.in_port))
    #self.mac_to_port ... <add or update entry>
    self.mac_to_port[packet.src] = packet_in.in_port


  #if the port associated with the destination MAC of the packet is known:
  if packet.dst in self.mac_to_port:
    # Send packet out the associated port
    log.debug("HIT: ")
    log.debug("Sending packet to Port %d" % self.mac_to_port[packet.dst])
    self.resend_packet(packet_in, self.mac_to_port[packet.dst])

  else:
    # Flood the packet out everything but the input port
    self.resend_packet(packet_in, of.OFPP_ALL)
```

Part C:

```
def act_like_switch (self, packet, packet_in):
  """
  Implement switch-like behavior.
  """
  # Here's some psuedocode to start you off implementing a learning
  # switch.  You'll need to rewrite it as real Python code.

  # Learn the port for the source MAC
  if packet.src not in self.mac_to_port:
    log.debug("Learning: MAC: %s from Port: %d" % (packet.src, packet_in.in_port))
    #self.mac_to_port ... <add or update entry>
    self.mac_to_port[packet.src] = packet_in.in_port

  #if the port associated with the destination MAC of the packet is known:
  if packet.dst in self.mac_to_port:
    # Send packet out the associated port
    log.debug("HIT: ")
    log.debug("Sending packet to Port %d" % self.mac_to_port[packet.dst])
    self.resend_packet(packet_in, self.mac_to_port[packet.dst])

    # Once you have the above working, try pushing a flow entry
    # instead of resending the packet (comment out the above and
    # uncomment and complete the below.)

    log.debug("Installing flow...")
    # Maybe the log statement should have source/destination/port?
    log.debug("Source MAC: %s   Destination MAC: %s   Output Port: %s" % (packet.src, packet.dst, self.mac_to_port[packet.dst]))
    msg = of.ofp_flow_mod()
    #
    ## Set fields to match received packet
    #msg.match = of.ofp_match.from_packet(packet)
    #
    #< Set other fields of flow_mod (timeouts? buffer_id?) >
    #
    #< Add an output action, and send -- similar to resend_packet() >

    msg.match.dl_dst = packet.dst
    action = of.ofp_action_output(port=self.mac_to_port[packet.dst])
    msg.actions.append(action)
    msg.idle_timeout = 60
    msg.hard_timeout = 600
    self.connection.send(msg)
  else:
    # Flood the packet out everything but the input port
    self.resend_packet(packet_in, of.OFPP_ALL)
```

Part D:

```
from mininet.topo import Topo

class MyTopo( Topo ):
    "Simple topology example."

    def build( self ):
        "Create custom topo."

        # Add hosts and switches
        #h1 = self.addHost( 'h1' )
        #h2 = self.addHost( 'h2' )
        #h3 = self.addHost( 'h3 ')
        #h4 = self.addHost( 'h4 ')
        # h5 = self.addHost( 'h5 ')

        h1 = self.addHost( 'h1', ip='10.0.0.1/16')
        h2 = self.addHost( 'h2', ip='10.0.0.2/16')
        h3 = self.addHost( 'h3', ip='10.0.0.3/16')
        h4 = self.addHost( 'h4', ip='10.0.1.2/16')
        h5 = self.addHost( 'h5', ip='10.0.1.3/16')

        s1 = self.addSwitch( 's1' )
        s2 = self.addSwitch( 's2' )
        s3 = self.addSwitch( 's3' )


        # Add links
        self.addLink( h1, s1 )
        self.addLink( h2, s1 )
        self.addLink( h3, s1 )

        self.addLink( s1, s2 )
        self.addLink( s2, s3 )

        self.addLink( h4, s3 )
        self.addLink( h5, s3 )


topos = { 'mytopo': ( lambda: MyTopo() ) }
```

```
sudo ovs-ofctl add-flow s2 icmp,nw_dst=10.0.1.0/24,actions=output:2
sudo ovs-ofctl add-flow s2 icmp,nw_dst=10.0.0.0/24,actions=output:1
sudo ovs-ofctl add-flow s2 arp,arp_tpa=10.0.1.0/24,actions=output:2
sudo ovs-ofctl add-flow s2 arp,arp_tpa=10.0.0.0/24,actions=output:1
```