COL334-Computer Networks

Assignment-3 - Report

Aneeket Yadav(2022CS11116) and Sanyam Garg(2022CS11078)

Part –1

Learning switch:

- We successfully set up mininet and the Ryu application.
- Before considering the network topology given in the problem, we consider a simpler topology to make some key observations.
- Traffic was being monitored by wireshark on the following interfaces : s1-eth1, s1-eth2, s1-eth3, Loopback : lo.
- As long as no ping message was sent, the only trafficobserved on any of s1-eth(i) interfaces were router solicitation packets which were sent on regular intervals. However, on lo interface, we observed packets containing the messages OFPT_PACKET_IN and OFPT_PACKET_OUT.
- These utilised the ICMP protocol. While the dst_address was ff02::2 for all such packets, the prefix of the src_address remained constant at fe80::200:ff:fe00. However, the variation was caused by the input port no. Of the ipv6 address which was one of 1,2 or 3. We observed such messages for each input port indefinitely in a chronologically cyclic order. For OFPT_PACKET_IN messages src_port was 48796(this value was dynamically assigned with different iterations of the experiment) and dst_port was 6653 and vice versa for OFPT_PACKET_OUT messages. This can be explained as follows:
    - The source addresses starting with fe80::200:ff:fe00 are IPv6 link-local addresses. The variation in the last octet (1, 2, or 3) corresponds to the different switch ports.
    - The destination address ff02::2 is the all-routers multicast address in IPv6.
    - Port 6653 is commonly associated with the OpenFlow switch to communicate with the controller.
    - Port 48796 was dynamically assigned to the controller for the current runtime.
    - The OpenFlow switch is systematically checking each of its ports in a round-robin fashion. This ensures that the controller maintains an up-to-date view of the network topology and port status. Regular messages from each port can help in quick detection of port failures or connectivity issues.
    - Since these messages are restricted to the controller and the OpenFlow switch, these do not update the flow table as the hosts are not

communicating with each other which justifies the absence of packets on their interfaces with the switch,

Time	Source	Destination	Protocol	Length	src_port	dst_port	Info
1.140593884	fe80::200:ff:fe00:3	ff02::2	OpenFl…	154	48796	6653	Type: OFPT_PACKET_IN
1.142914919	fe80::200:ff:fe00:3	ff02::2	OpenFl…	160	6653	48796	Type: OFPT_PACKET_OUT
3.188469674	fe80::200:ff:fe00:2	ff02::2	OpenFl…	154	48796	6653	Type: OFPT_PACKET_IN
3.190523969	fe80::200:ff:fe00:2	ff02::2	OpenFl…	160	6653	48796	Type: OFPT_PACKET_OUT
3.700308333	fe80::200:ff:fe00:1	ff02::2	OpenFl…	154	48796	6653	Type: OFPT_PACKET_IN
3.701948784	fe80::200:ff:fe00:1	ff02::2	OpenFl…	160	6653	48796	Type: OFPT_PACKET_OUT
15.989157106	fe80::200:ff:fe00:3	ff02::2	OpenFl…	154	48796	6653	Type: OFPT_PACKET_IN
15.991487722	fe80::200:ff:fe00:3	ff02::2	OpenFl…	160	6653	48796	Type: OFPT_PACKET_OUT
19.060280080	fe80::200:ff:fe00:2	ff02::2	OpenFl…	154	48796	6653	Type: OFPT_PACKET_IN
19.061095339	fe80::200:ff:fe00:2	ff02::2	OpenFl…	160	6653	48796	Type: OFPT_PACKET_OUT

- We then ran the command - h1 ping -c 20 h2 and observed the following:
  - On the lo interface, a single OFPT_PACKET_IN and OFPT_PACKET_OUT message were observed. This indicates that this flow was added to the flow table after the first ping and subsequent pings did not cause a packet_in event.
  - ICMP ping request messages were recorded with src_address = 10.0.0.1 and dst_address = 10.0.0.2, vice versa for ICMP ping reply messages. These messages were observed on both interfaces – s1-eth1 and s1-eth2.
  - Interestingly, ping messages were successfully sent and received even before ARP protocol was completed. The SDN controller (if you're using one) or the Open vSwitch in Mininet might already have the necessary information to forward the packets correctly. The ARP requests and replies you see might be generated as a formality or for maintaining consistency with real network behavior, but they're not actually necessary for the ICMP traffic to flow.

1	0.000000000	00:00:00_00:00:01	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
2	0.005163420	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	0.005174142	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x0269, seq=1/256, ttl=64
4	0.006570804	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0269, seq=1/256, ttl=64
5	1.001663838	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x0269, seq=2/512, ttl=64
6	1.002160655	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0269, seq=2/512, ttl=64
7	2.050893710	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x0269, seq=3/768, ttl=64
8	2.051000614	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0269, seq=3/768, ttl=64
9	3.074707811	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x0269, seq=4/1024, ttl=6
10	3.074793192	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0269, seq=4/1024, ttl=6
11	4.098693580	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x0269, seq=5/1280, ttl=6
12	4.098790878	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0269, seq=5/1280, ttl=6
13	5.123698440	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x0269, seq=6/1536, ttl=6
14	5.123782896	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0269, seq=6/1536, ttl=6
15	5.507197942	00:00:00_00:00:02	00:00:00_00:00:01	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
16	5.507229326	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
17	6.147651576	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x0269, seq=7/1792, ttl=6
18	6.147717905	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0269, seq=7/1792, ttl=6
19	7.171603988	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x0269, seq=8/2048, ttl=6
20	7.171678368	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0269, seq=8/2048, ttl=6
21	8.194657287	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x0269, seq=9/2304, ttl=6

  - After this step, running the command - dpctl dump-flows on Mininet showed that two new flows had been added – one from h2 to h1 and the other from h1 to h2 .

```
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
 cookie=0x0, duration=1761.596s, table=0, n_packets=11, n_bytes=1022, in_port="s
1-eth2",dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01 actions=output:"s1-eth
1"
 cookie=0x0, duration=1761.595s, table=0, n_packets=10, n_bytes=924, in_port="s1
-eth1",dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02 actions=output:"s1-eth2
"
```

- In case of pingall, each host pings every other host. Therefore, for each host, we observe 8 ping requests and replies on its interface with the switch it is directly connected to. The connection between the two switches only contains traffic one of whose src or dst is h4 or h5.
- For each pair of hosts say h_a and h_b, suppose h_a pinged h_b first, then the switches learn the location of h_a through the ping request messages and of h_b through ping reply messages. Therefore, when h_b later pings h_1, these messages do not result in packet_in events as the corresponding flows have already been added to the flow table.
- The two switches are dynamically assigned different port numbers to communicate with the OpenFlow switch(port no. 6633).
- At the end of pingall session,
  - Consider switch s1- s1-eth4(i.e s2)is present in 3x2 = 6 entries as in-port since each of h1,h2,h3 pings h4,h5. Similarly, it is also present as out-port of 6 entries. Hosts h1, h2 and h3 are each present as sources of four flows(since each has to ping 4 other hosts). Thus flow table of s1 contains 6 + 3*4 = 18 entries.
  - Similarly, s2 contains 6 + 2*4 = 14 entries.
- Result of iperf3 test from h1 to h5:
  - H5 was set as the listening server

```
mininet> h5 iperf3 -s &
mininet> h1 iperf3 -c h5
Connecting to host 10.0.0.5, port 5201
[  5] local 10.0.0.1 port 40016 connected to 10.0.0.5 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec   645 MBytes  5.41 Gbits/sec    0   1.06 MBytes
[  5]   1.00-2.00   sec   837 MBytes  7.02 Gbits/sec   52   2.03 MBytes
[  5]   2.00-3.00   sec  1.97 GBytes  16.9 Gbits/sec    0   2.07 MBytes
[  5]   3.00-4.00   sec   755 MBytes  6.33 Gbits/sec    0   2.09 MBytes
[  5]   4.00-5.00   sec   753 MBytes  6.32 Gbits/sec    0   2.10 MBytes
[  5]   5.00-6.00   sec   841 MBytes  7.05 Gbits/sec    0   2.12 MBytes
[  5]   6.00-7.00   sec   794 MBytes  6.66 Gbits/sec    0   2.13 MBytes
[  5]   7.00-8.00   sec   819 MBytes  6.87 Gbits/sec    0   2.15 MBytes
[  5]   8.00-9.00   sec   730 MBytes  6.09 Gbits/sec    0   2.21 MBytes
[  5]   9.00-10.00  sec   762 MBytes  6.42 Gbits/sec    0   2.21 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  9.09 GBytes  7.80 Gbits/sec   52             sender
[  5]   0.00-10.00  sec  9.09 GBytes  7.80 Gbits/sec                  receiver

iperf Done.
```

We observe that bitrate of transfer(throughput) keeps fluctuating over the 10s interval. On the other hand, the size of the congestion window(Cwnd) keeps steadily

increasing indicating that the network is not congested and can support higher throughput. No packet loss was observed.

Hub:

- Unlike a learning switch, using a hub does not add any flow to the OpenFlow flow table.

```
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
*** s2 ------------------------------------------------------------------------
```

- A hub has considerably lower throughput as compared to a learning switch, the latter being approximately 280 to 300 times the former.

```
mininet> h5 iperf3 -s &
mininet> h1 iperf3 -c h5
Connecting to host 10.0.0.5, port 5201
[  5] local 10.0.0.1 port 36084 connected to 10.0.0.5 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  3.38 MBytes  28.3 Mbits/sec    0    167 KBytes
[  5]   1.00-2.00   sec  3.38 MBytes  28.3 Mbits/sec    0    307 KBytes
[  5]   2.00-3.00   sec  3.00 MBytes  25.2 Mbits/sec    0    445 KBytes
[  5]   3.00-4.00   sec  4.12 MBytes  34.6 Mbits/sec    0    584 KBytes
[  5]   4.00-5.00   sec  2.62 MBytes  22.0 Mbits/sec    0    713 KBytes
[  5]   5.00-6.00   sec  3.12 MBytes  26.2 Mbits/sec    0    844 KBytes
[  5]   6.00-7.00   sec  3.62 MBytes  30.4 Mbits/sec    0    973 KBytes
[  5]   7.00-8.00   sec  2.00 MBytes  16.8 Mbits/sec    0   1.08 MBytes
[  5]   8.00-9.00   sec  4.62 MBytes  38.8 Mbits/sec    0   1.23 MBytes
[  5]   9.00-10.00  sec  2.62 MBytes  22.0 Mbits/sec    0   1.36 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  32.5 MBytes  27.3 Mbits/sec    0             sender
[  5]   0.00-10.17  sec  27.9 MBytes  23.0 Mbits/sec                  receiver

iperf Done.
```

- Moreover, packet loss was consistently observed in all iterations of the test for the hub.
- In a hub-based network, all devices share the same collision domain. This means that multiple devices transmitting at the same time can cause collisions, leading to packet loss. In iperf3 tests where a large volume of data is transferred, this can result in high packet loss if the network becomes congested. Unlike hubs, switches create a separate collision domain for each connected device, which isolates devices and prevents collisions.

Part – 2

- When trying to run pingall using the original simple_switch ryu application, the following result was obtained -

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X
h2 -> X X X
h3 -> X X X
h4 -> X X X
*** Results: 100% dropped (0/12 received)
```

- No ping request or reply packet was observed via wireshark From the flow table learnt by each switch, we observe the below behaviour characteristic of a loop in the network topology-
  - Multiple flows observed between the same pair of nodes(though the MAC addresses would be distinct due to difference in port).
  - Flows observed with same nodes as source and destination.
- In a looped network, when packets (like ARP requests) are broadcasted by hosts, they can circulate endlessly among the switches. This causes a broadcast storm, leading to excessive network traffic and dropped packets. In fact, while ascertaining that ping is not possible for every host, htop registered 100% CPU usage for each core.
- It is difficult to explain the exact flows learnt by the SDN as in case of a looped topology, these are a product of timing delays which would vary across runtimes.

Explanation of our approach:

- The spanning tree construction mechanism begins by identifying a starting point (root node) from the network's topology. This starting point is chosen as the node with the smallest identifier. From this node, the algorithm performs a breadth-first exploration of the network.
- The process maintains a list of nodes it has visited to avoid cycles, which are typical in interconnected networks. It begins by adding the root node to a queue and marking it as visited. Then, it checks each neighbor (connected node) of the current node. If a neighbor has not been visited, it is added to the queue, marked as visited, and a connection is established between the current node and the neighbor in the spanning tree.
- The process continues until all nodes in the network have been visited and connected in a loop-free manner, ensuring that the final structure forms a tree. The tree only retains essential connections to maintain connectivity without redundant links, thus preventing loops in the network's forwarding logic.

```
Constructing spanning tree.
Spanning tree: {1: [(2, 2), (4, 3)], 2: [(1, 2), (3, 3)], 4: [(1, 3)], 3: [(2, 2)]}
{2: {1, 3}, 1: {2, 4}, 3: {2, 4}, 4: {1, 3}}
Constructing spanning tree.
Spanning tree: {1: [(2, 2), (4, 3)], 2: [(1, 2), (3, 3)], 4: [(1, 3)], 3: [(2, 2)]}
{2: {1, 3}, 1: {2, 4}, 3: {2, 4}, 4: {1, 3}}
Constructing spanning tree.
Spanning tree: {1: [(2, 2), (4, 3)], 2: [(1, 2), (3, 3)], 4: [(1, 3)], 3: [(2, 2)]}
{2: {1, 3}, 1: {2, 4}, 3: {2, 4}, 4: {1, 3}}
Constructing spanning tree.
Spanning tree: {1: [(2, 2), (4, 3)], 2: [(1, 2), (3, 3)], 4: [(1, 3)], 3: [(2, 2)]}
{2: {1, 3}, 1: {2, 4}, 3: {2, 4}, 4: {1, 3}}
Constructing spanning tree.
Spanning tree: {1: [(2, 2), (4, 3)], 2: [(1, 2), (3, 3)], 4: [(1, 3)], 3: [(2, 2)]}
{2: {1, 3}, 1: {2, 4}, 3: {2, 4}, 4: {1, 3}}
Constructing spanning tree.
Spanning tree: {1: [(2, 2), (4, 3)], 2: [(1, 2), (3, 3)], 4: [(1, 3)], 3: [(2, 2)]}
{2: {1, 3}, 1: {2, 4}, 3: {2, 4}, 4: {1, 3}}
Constructing spanning tree.
Spanning tree: {1: [(2, 2), (4, 3)], 2: [(1, 2), (3, 3)], 4: [(1, 3)], 3: [(2, 2)]}
{2: {1, 3}, 1: {2, 4}, 3: {2, 4}, 4: {1, 3}}
Constructing spanning tree.
Spanning tree: {1: [(2, 2), (4, 3)], 2: [(1, 2), (3, 3)], 4: [(1, 3)], 3: [(2, 2)]}
{2: {1, 3}, 1: {2, 4}, 3: {2, 4}, 4: {1, 3}}
Connected host MAC(06:6f:6c:b3:36:89) to switch 4, port 1
Connected host MAC(fa:a8:98:89:db:08) to switch 2, port 1
Connected host MAC(6a:f5:83:e6:42:f1) to switch 3, port 1
Connected host MAC(3a:69:0e:10:aa:27) to switch 1, port 1
```

Part-3

Algorithm Description-

1. Topology Discovery:
    a. Discover all switches and links in the network.
    b. Create a graph representation of the network topology.
2. Link Delay Measurement:
    a. For each link in the network:
        i. Send custom probe packets between connected switches.
        ii. Measure the round-trip time of these packets.
        iii. Subtract processing delays at switches to estimate link delay.
        iv. Use this delay as the weight for the corresponding link in the graph.
3. Shortest Path Calculation:
    a. Using the weighted graph, calculate the shortest paths between all pairs of switches.
    b. Store the next hop for each source-destination switch pair.
4. Spanning Tree Creation:
    a. Generate a spanning tree of the network using breadth-first search.

    b. This tree will be used for handling broadcast and unknown unicast traffic.
5. L2 Address Learning:
    a. Maintain a table mapping MAC addresses to switch ports.
    b. When a packet is received, update this table with the source MAC and ingress port.
6. Packet Handling:
    a. When a packet arrives at a switch: a. If the destination MAC is known (unicast to known host):
        i. Determine the destination switch.
        ii. Look up the next hop in the shortest path to that switch.
        iii. Forward the packet to the appropriate output port. b. If the destination MAC is unknown or it's a broadcast:
        iv. Forward the packet along the spanning tree (flooding).
7. Flow Rule Installation:
    a. For known unicast traffic:
        i. Install flow rules in switches along the shortest path.
        ii. Match on source and destination MAC addresses.
        iii. Set the output action to the port leading to the next hop.
8. Continuous Operation:
    a. Periodically re-measure link delays and recalculate shortest paths.
    b. Update flow rules as necessary when shortest paths change.

<u>Assumptions:</u>

1. The network topology is static during the operation.
2. Link delays are the primary metric for path selection.
3. Link delays are symmetric (same in both directions).
4. The controller has enough processing power and memory to calculate and store all paths.
5. Switches support OpenFlow 1.3.
6. The network is a single L2 domain (no VLANs or L3 routing considered).
7. Host detection is based on switch ports not connected to other switches.
8. The time taken for topology discovery and initial delay measurements is acceptable.