Lab4: SDN Open Virtual Switches

Contents

1. Objectives

- Emulate a functional SDN network.
- Understand and get familiar with OVS.
- Understand and get familiar with controllers.

2. Equipment

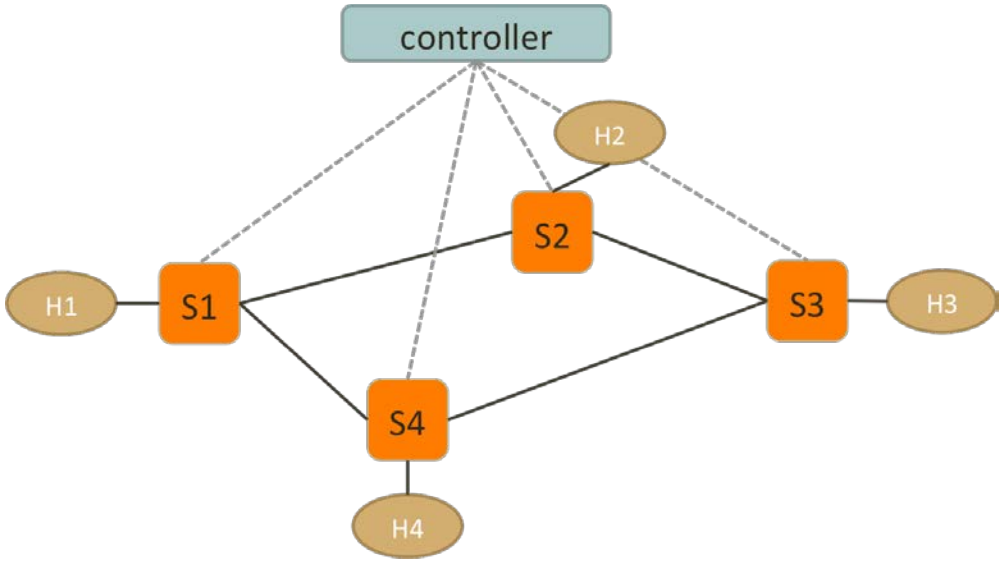
- Computers
- Internet

3. References

- RYU programming guide: http://osrg.github.io/ryu-book/en/html/

4. Experiments

1. Use Mininet to create the following topology: (4 Hosts, 4 OVSes) with a remote controller
2. Use RYU to implement the controller (you can use other controller such as BEACON, POX, etc...)



3. Test Connectivity using ping. (Hint: take care of ARP packets in the controller and install proper rules for them.)
4. Enforce these policies:
   - Everything follows shortest path
   - When there are two shortest paths available
     - ICMP and TCP packets take the lower/left path
       - **S1-S4-S3 and S2-S1-S4**
     - UDP packets take the upper/right path
       - **S1-S2-S3 and S2-S3-S4**
     - H2 and H4 cannot have HTTP traffic (TCP with port:80)
       - New connections are dropped with a TCP RST sent back to **H2 or H4**
       - To be more specific, when the first TCP packet (SYN) arrives **S2 or S4**, forwarded it to controller, controller then create a RST packet and send it back to the host.
     - H1 and H4 cannot have UDP traffic
       - simply drop packets at switches

5. Reports

(a) Write a pseudo code to implement spanning tree in SDN network.

Ideally, in SDN network, the controller can apply any minimal spanning tree algorithm with the view of whole topology. To implement this mechanism, we first use Link Level Discovery Protocol (LLDP) to discover the network topology. Then, we use library “networkx” to compute the spanning tree. Classic minimal spanning tree algorithms are Kruskal’s algorithm, Prim’s algorithm.

Here, we pick Prim's algorithm to answer this problem. The basic idea of Prim's algorithm is to build a single tree by adding edges one by one. In each iteration, it picks the vertex with smallest priority and pick a min-weighted edge connected to it for adding them to the tree. After that, it updates the priority with the weight of the min-weighted edge.

On the other hand, Ryu does provides a STP library to find the spanning tree. In the example "SimpleSwitch13.py", it implements the STP protocol. However, it follows the traditional algorithm in IP network (IEEE 802.1D). Unlike SDN-style design, it requires switches to exchange information with Bridge Protocol Data Unit (BPDU), so we didn't implement spanning tree in this way.

```
Prim's algorithm (G, V, E)
* In SDN, G is given after the controller finish discovering the network.

Q ← All vertices V in G              # Put every vertices into a queue (Q)
for each v in Q:
   v.priority ← inf                  # Set priorities to infinity
Q[0].priority ← 0                    # Set the first vertex's priority to 0 as a starting point
while Q.size() > 0:
   v ← min(Q)                        # Pick a vertex that has minimum priority
   for each u in neighbor(v):        # Iterate through every adjacent vertex to v
      if (v in Q) and weight of (u,v) < v.prioriy:
         v.parent ← u                # Add link (u,v) to the tree
         v.priority ← weight of (u,v)       # Update the priority with the weight of the edge (u,v)
Return Q                             # Return final spanning tree
```

(b) List the advantages of using OpenVSwitch and SDN controller compared to IP networks.

1. **SDN can discover topology faster than IP network.** In SDN, the controller can discover the network through LLDP. The link information is forwarded by the switches to the controller, and the controller will construct the network topology. In contrast, in the IP network, switches discover the network by learning the MAC addresses through ARP.
2. **SDN can find the shortest path easily.** With the whole topology view, the controller can find the shortest path easily through some algorithms (eg. Dijkstra), while in IP network, the shortest path is found by ARP broadcasting and MAC-learning on switches.
3. **SDN can handle link failure faster.** In SDN, when a link is down, the controller can detect it easily and re-allocates a new path for recovery. In IP network, the link is recovered by timeouts on switches and hosts. It takes longer time and need to broadcast again for an alternative path.
4. **SDN can avoid some problem that IP network has.** In SDN, spanning tree is not necessary for the controller, because the loop can be easily avoided with whole network topology.
5. **SDN can provide more flexible packet modification in network.** Switches can modify the header of the packets in SDN. For example, switches can push VLAN tag for virtual LAN, or switches can rewrite address field in IP packets. In IP network, these function needs to be performed either on hosts or special network devices.

6. **SDN can provide flow-based forwarding rules in network.** In SDN, forwarding rules can be setup for some specific flows. Each flow may have different forwarding rules on a switch. In contrast, in IP network, regular switches can only forward the packets with its destination MAC address for shortest path.

(c) Include the controller's code.

The code is uploaded to the NYU new class, so I only go through some function in high level here.

The controller leverages LLDP for network discovery, so we need to add "--observe-links" when runing the code: **"ryu-manager lab4.py --observe-links"**

To find the spanning tree to avoid loop, I directly use the minimal spanning tree algorithm after the topology is discovered by LLDP. By installing blocking flows with lowest priority, spanning tree can be implemented without effecting other flow rules.

Function name	***Description***
switch_features_handler()	Initialize the switches: Add "Table-miss" flows.
add_flow() add_mst_drop_flow() add_drop_flow()	Add flows with actions. Add flows that drop packet for spanning tree. (Use cookie here) Add other flows that drop packets
delete_flow() delete_drop_flow()	Delete flows. Delete flows that drop packet for spanning tree with cookie
_send_packet()	Packet-out a packet created by the controller.
packet_in_handler()	Handle packet-in. (Handle APR, ICMP, TCP, UDP packets)
DumpShortestPathIcmpTCP() DumpShortestPathUDP()	Path-finding algorithm following the policies.
GetTopologyData() PopulateNet() AddHosts()	Update the topology on the controller when it is changed. Install the blocking flows here once a new switch is found. (Including STP)

(d) Include the topology file

The code is uploaded to the NYU new class, so I only go through some introduction in high level here.

Hosts	Assign IP and MAC address for debugging
Switches	No special setting
Links	Assign ports to each links for easier control.
RemoteController	ip='127.0.0.1', port=6633. For RYU running on same machine

(e) Describe how you generate traffic to test your controller and switch behavior

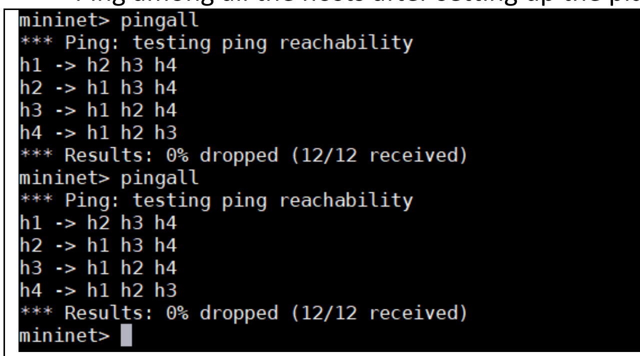
We use both "iperf" and python socket to generate traffic to test our controller. This lab also mentions about ICMP, so we also use "ping" to check the policy about ICMP.

iperf	Python socket

On server:
$ iperf -s -p 80 # To check TCP HTTP packet
$ iperf -s -p 8080 # To check TCP other connection
$ iperf -s -p 8080 –u # To check UDP connection
On client:
$ iperf -c <target IP> -p 80 # To check TCP HTTP
$ iperf -c <target IP> -p 8800 # To check TCP
$ iperf -c <target IP> -p 8800 -u # To check UDP
	On server:
	$ python sock_server.py <ip> <port>
	On client:
	$ python sock_client.py <ip> <port>

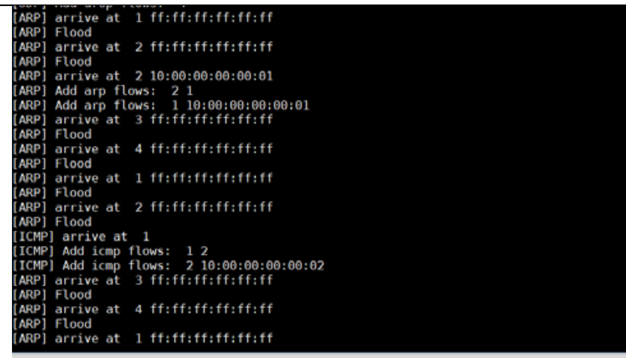
(f) Screenshots:

- Ping among all the hosts after setting up the platform.



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

Mininet: pingall



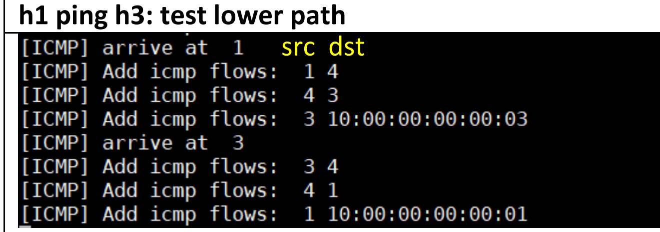
```
[ARP] arrive at  1 ff:ff:ff:ff:ff:ff
[ARP] Flood
[ARP] arrive at  2 ff:ff:ff:ff:ff:ff
[ARP] Flood
[ARP] arrive at  2 10:00:00:00:00:01
[ARP] Add arp flows:  2 1
[ARP] Add arp flows:  1 10:00:00:00:00:01
[ARP] arrive at  3 ff:ff:ff:ff:ff:ff
[ARP] Flood
[ARP] arrive at  4 ff:ff:ff:ff:ff:ff
[ARP] Flood
[ARP] arrive at  1 ff:ff:ff:ff:ff:ff
[ARP] Flood
[ARP] arrive at  2 ff:ff:ff:ff:ff:ff
[ARP] Flood
[ICMP] arrive at  1
[ICMP] Add icmp flows:  1 2
[ICMP] Add icmp flows:  2 10:00:00:00:00:02
[ARP] arrive at  3 ff:ff:ff:ff:ff:ff
[ARP] Flood
[ARP] arrive at  4 ff:ff:ff:ff:ff:ff
[ARP] Flood
[ARP] arrive at  1 ff:ff:ff:ff:ff:ff
```

Ryu: show where rules are installed

- TCP, UDP and ICMP packets on their respective paths.

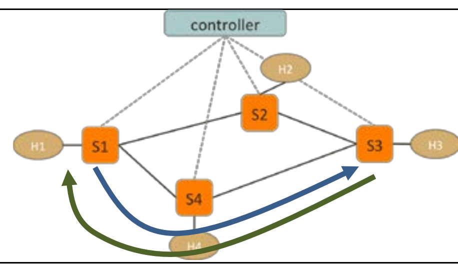
ICMP

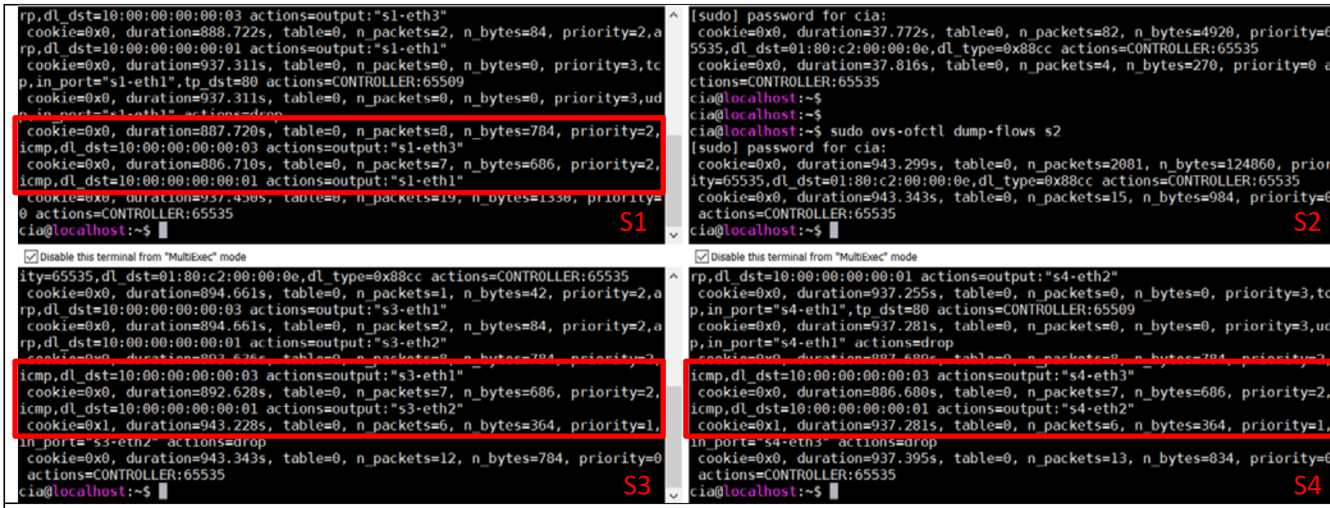
h1 ping h3: test lower path



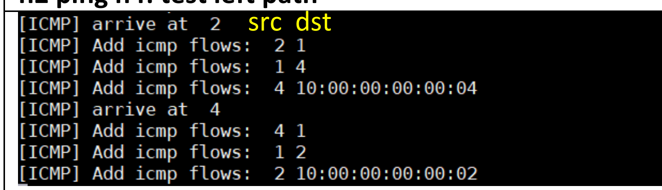
```
[ICMP] arrive at  1    src dst
[ICMP] Add icmp flows:  1 4
[ICMP] Add icmp flows:  4 3
[ICMP] Add icmp flows:  3 10:00:00:00:00:03
[ICMP] arrive at  3
[ICMP] Add icmp flows:  3 4
[ICMP] Add icmp flows:  4 1
[ICMP] Add icmp flows:  1 10:00:00:00:00:01
```

Path install

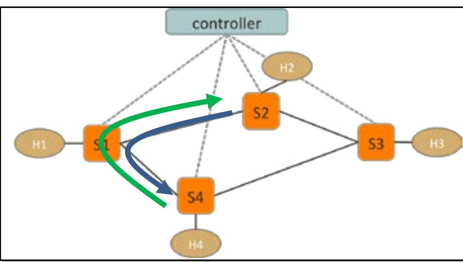


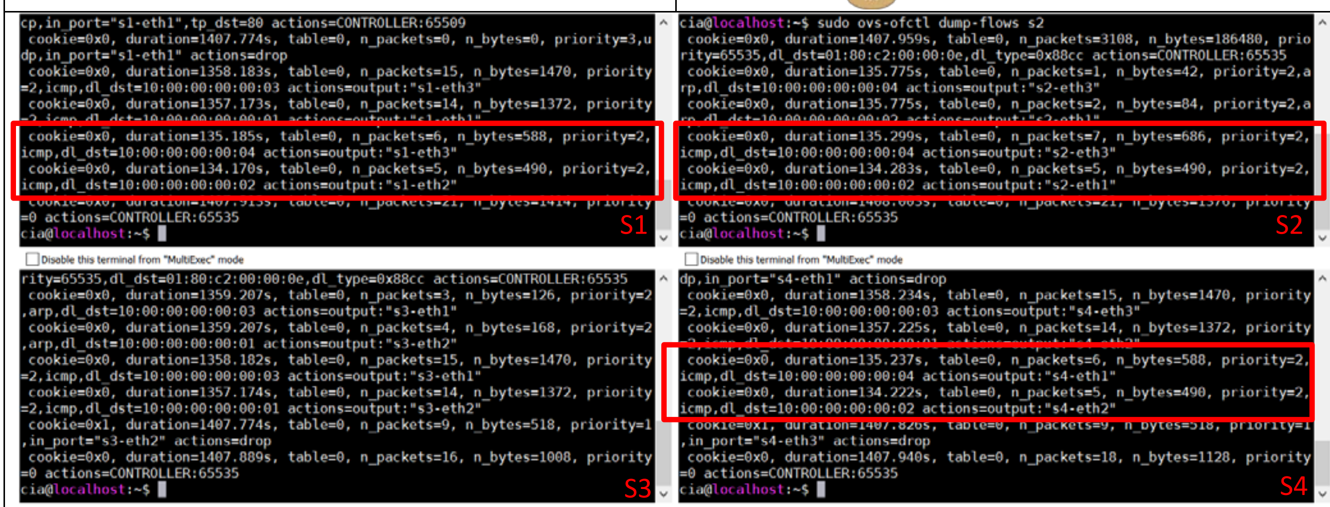


h2 ping h4: test left path

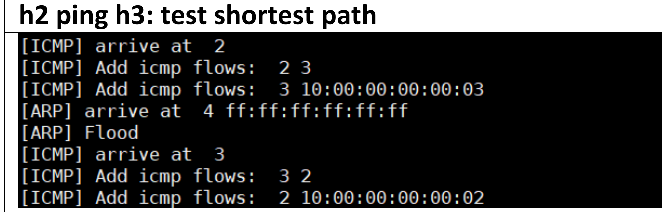


Path install

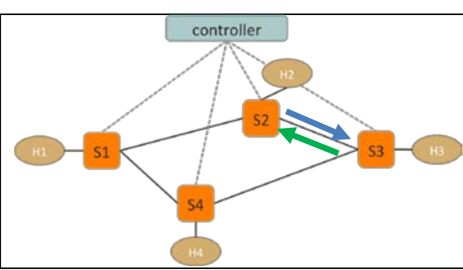


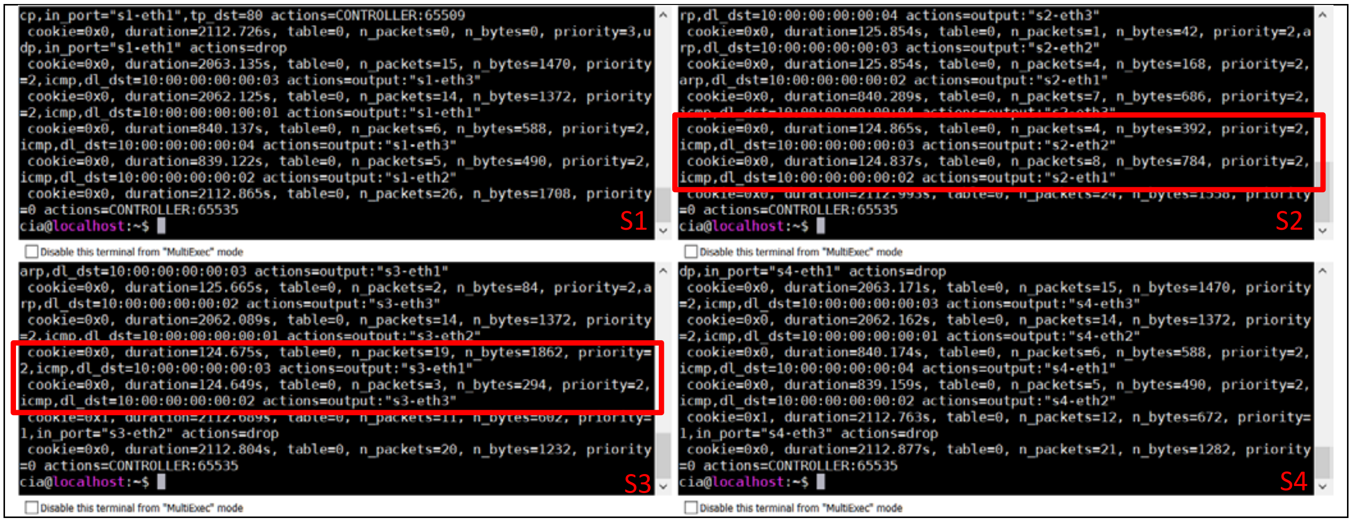


h2 ping h3: test shortest path



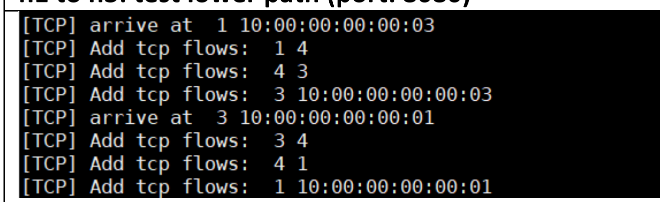
Path install



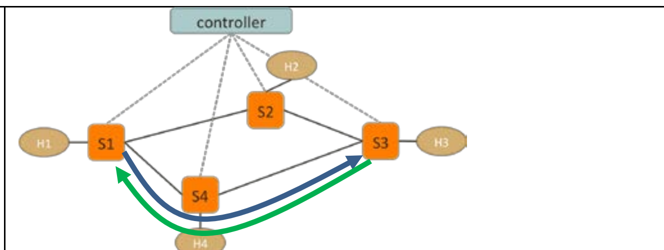


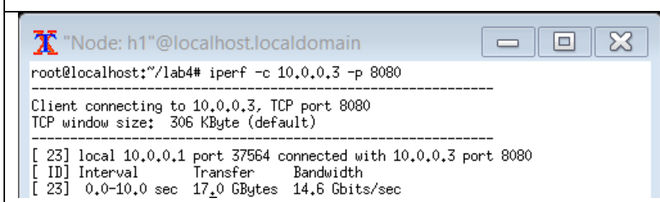
TCP

h1 to h3: test lower path (port: 8080)

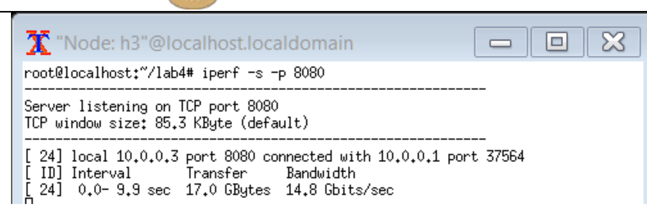


Path install

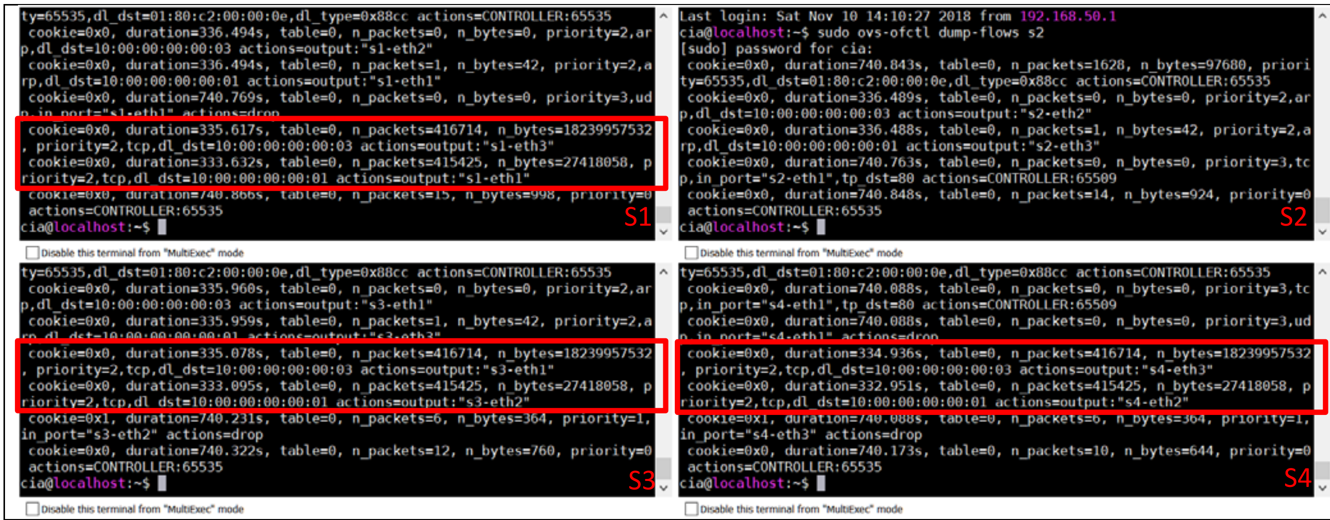




h1: sends the packets

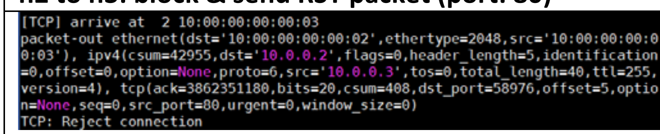


h3: receives the packets



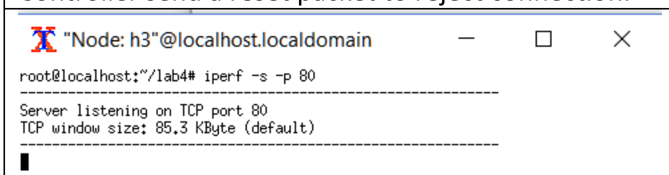
Forwarding rules for sucessful routing is same as ICMP, so I'll skip other cases (left path, shortest path).

h2 to h3: block & send RST packet (port: 80)

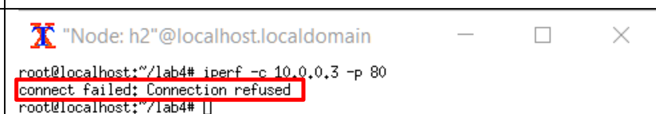


Controller send a reset packet to reject connection.

Wireshark result



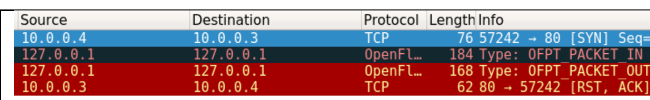
h3 doesn't receive anything. (because h2 is blocked)



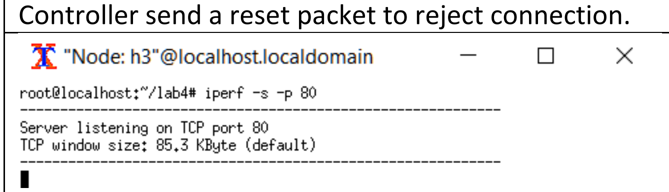
h2 is sucessfully rejected with the TCP RST packet.

h4 to h3: block & send RST packet (port: 80)

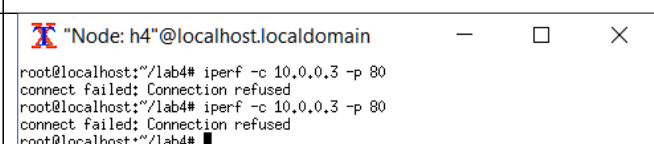
Controller send a reset packet to reject connection.



Wireshark result

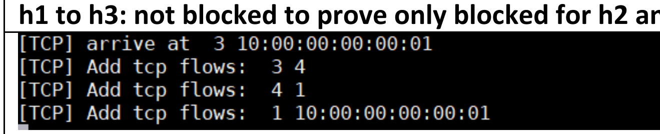


h3 doesn't receive anything. (because h4 is blocked)

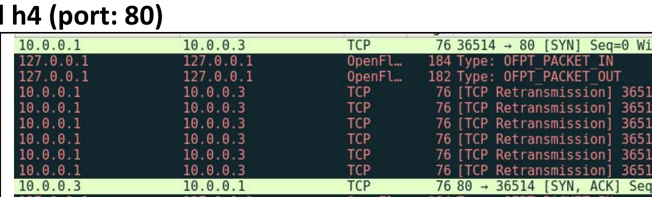


h4 is sucessfully rejected with the TCP RST packet.

h1 to h3: not blocked to prove only blocked for h2 and h4 (port: 80)



Controller set up the path between h1 and h3



Wireshark result

```
"Node: h1"@localhost.localdomain
root@localhost:~/lab4# iperf -c 10.0.0.3 -p 80
------------------------------------------------------------
Client connecting to 10.0.0.3, TCP port 80
TCP window size:  374 KByte (default)
------------------------------------------------------------
[ 23] local 10.0.0.1 port 36514 connected with 10.0.0.3 port 80
[ ID] Interval       Transfer     Bandwidth
[ 23]  0.0-10.0 sec  4.68 GBytes  4.02 Gbits/sec
root@localhost:~/lab4#
```

```
"Node: h3"@localhost.localdomain
root@localhost:~/lab4# iperf -s -p 80
------------------------------------------------------------
Server listening on TCP port 80
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 24] local 10.0.0.3 port 80 connected with 10.0.0.1 port 36514
[ ID] Interval       Transfer     Bandwidth
[ 24]  0.0- 9.7 sec  4.68 GBytes  4.13 Gbits/sec
```

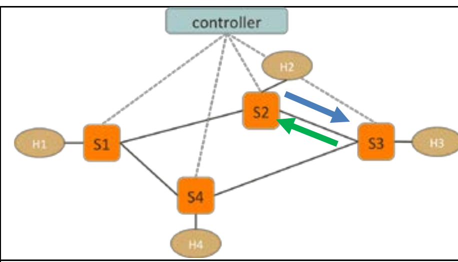
h1 and h3 are sucessfully connected.

UDP

h2 to h3: not blocked to prove only blocked for h1 and h4

```
[UDP] arrive at  2
[UDP] Add udp flows:  2 3
[UDP] Add udp flows:  3 10:00:00:00:00:03
[UDP] arrive at  3
[UDP] Add udp flows:  3 2
[UDP] Add udp flows:  2 10:00:00:00:00:02
```

Controller set up the path between h2 and h3




```
------------------------------------------------------------
Client connecting to 10.0.0.3, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 23] local 10.0.0.2 port 37078 connected with 10.0.0.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 23]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[ 23] Sent 893 datagrams
[ 23] Server Report:
[ 23]  0.0- 6.8 sec   881 KBytes  1.06 Mbits/sec   0.000 ms  279/  893 (0%)
```

h2 is sucessfully connected to h3.

```
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 23] local 10.0.0.3 port 5001 connected with 10.0.0.2 port 37078
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[ 23]  0.0- 6.8 sec   881 KBytes  1.06 Mbits/sec   0.038 ms  279/  893 (31%)
```

h3 sucessfully receives the packets from h2.

h1 to h3: block UDP

These rules are install when the switches connect to the controller

```
root@localhost:~/lab4# iperf -c 10.0.0.3 -u
------------------------------------------------------------
Client connecting to 10.0.0.3, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 23] local 10.0.0.1 port 59293 connected with 10.0.0.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 23]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[ 23] Sent 893 datagrams
[ 23] WARNING: did not receive ack of last datagram after 10 tries.
root@localhost:~/lab4#
```

h1 is sucessfully blocked.

```
^Croot@localhost:~/lab4# iperf -s -u
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
```

h3 doesn’t receive anything. (because h1 is blocked)

h4 to h3: block UDP

```
^Croot@localhost:~/lab4# iperf -s -u
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
```

h3 doesn’t receive anything. (because h4 is blocked)

```
connect failed: connection refused
root@localhost:~/lab4# iperf -c 10.0.0.3 -u
------------------------------------------------------------
Client connecting to 10.0.0.3, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 23] local 10.0.0.4 port 53587 connected with 10.0.0.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 23]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[ 23] Sent 893 datagrams
[ 23] WARNING: did not receive ack of last datagram after 10 tries.
root@localhost:~/lab4#
```

h4 is sucessfully blocked.

- Rules installed at each switch.

S1

```
cia@localhost:~$ sudo ovs-ofctl dump-flows s1
 cookie=0x0, duration=3169.826s, table=0, n_packets=1289, n_bytes=77340, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=2215.925s, table=0, n_packets=5, n_bytes=210, priority=2,arp,dl_dst=10:00:00:00:00:03 actions=output:"s1-eth2"
 cookie=0x0, duration=135.964s, table=0, n_packets=3, n_bytes=126, priority=2,arp,dl_dst=10:00:00:00:00:04 actions=output:"s1-eth3"
 cookie=0x0, duration=120.701s, table=0, n_packets=15, n_bytes=630, priority=2,arp,dl_dst=10:00:00:00:00:01 actions=output:"s1-eth1" ARP
 cookie=0x0, duration=120.700s, table=0, n_packets=3, n_bytes=126, priority=2,arp,dl_dst=10:00:00:00:00:02 actions=output:"s1-eth2"
 cookie=0x0, duration=3169.719s, table=0, n_packets=903, n_bytes=1365336, priority=3,udp,in_port="s1-eth1" actions=drop UDP drop
 cookie=0x0, duration=2215.151s, table=0, n_packets=114132, n_bytes=5037813936, priority=2,tcp,dl_dst=10:00:00:00:00:03 actions=output:"s1-eth3"
 cookie=0x0, duration=2213.180s, table=0, n_packets=109744, n_bytes=7243112, priority=2,tcp,dl_dst=10:00:00:00:00:01 actions=output:"s1-eth1"   TCP
 cookie=0x0, duration=146.450s, table=0, n_packets=2, n_bytes=196, priority=2,icmp,dl_dst=10:00:00:00:00:03 actions=output:"s1-eth3"
 cookie=0x0, duration=104.788s, table=0, n_packets=5, n_bytes=490, priority=2,icmp,dl_dst=10:00:00:00:00:04 actions=output:"s1-eth3" ICMP
 cookie=0x0, duration=95.426s, table=0, n_packets=7, n_bytes=686, priority=2,icmp,dl_dst=10:00:00:00:00:01 actions=output:"s1-eth1"
 cookie=0x0, duration=62.628s, table=0, n_packets=4, n_bytes=392, priority=2,icmp,dl_dst=10:00:00:00:00:02 actions=output:"s1-eth2"
 cookie=0x1, duration=3169.719s, table=0, n_packets=13, n_bytes=602, priority=1,in_port="s1-eth3" actions=drop Spanning tree
 cookie=0x0, duration=3169.995s, table=0, n_packets=22, n_bytes=1272, priority=0 actions=CONTROLLER:65535
```

S2

```
cia@localhost:~$ sudo ovs-ofctl dump-flows s2
 cookie=0x0, duration=3213.447s, table=0, n_packets=1292, n_bytes=77520, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=2259.581s, table=0, n_packets=11, n_bytes=462, priority=2,arp,dl_dst=10:00:00:00:00:03 actions=output:"s2-eth2"
 cookie=0x0, duration=164.119s, table=0, n_packets=11, n_bytes=462, priority=2,arp,dl_dst=10:00:00:00:00:01 actions=output:"s2-eth3"
 cookie=0x0, duration=149.363s, table=0, n_packets=3, n_bytes=126, priority=2,arp,dl_dst=10:00:00:00:00:04 actions=output:"s2-eth2" ARP
 cookie=0x0, duration=149.363s, table=0, n_packets=14, n_bytes=588, priority=2,arp,dl_dst=10:00:00:00:00:02 actions=output:"s2-eth1"
 cookie=0x0, duration=3213.269s, table=0, n_packets=1, n_bytes=74, priority=3,tcp,in_port="s2-eth1",tp_dst=80 actions=CONTROLLER:65509 TCP block
 cookie=0x0, duration=991.293s, table=0, n_packets=646, n_bytes=976752, priority=2,udp,dl_dst=10:00:00:00:00:03 actions=output:"s2-eth2"
 cookie=0x0, duration=985.976s, table=0, n_packets=1, n_bytes=1512, priority=2,udp,dl_dst=10:00:00:00:00:02 actions=output:"s2-eth1" UDP
 cookie=0x0, duration=169.423s, table=0, n_packets=2, n_bytes=196, priority=2,icmp,dl_dst=10:00:00:00:00:01 actions=output:"s2-eth3"
 cookie=0x0, duration=159.407s, table=0, n_packets=2, n_bytes=196, priority=2,icmp,dl_dst=10:00:00:00:00:03 actions=output:"s2-eth2" ICMP
 cookie=0x0, duration=148.337s, table=0, n_packets=2, n_bytes=196, priority=2,icmp,dl_dst=10:00:00:00:00:04 actions=output:"s2-eth3"
 cookie=0x0, duration=106.176s, table=0, n_packets=6, n_bytes=588, priority=2,icmp,dl_dst=10:00:00:00:00:02 actions=output:"s2-eth1"
 cookie=0x0, duration=3213.680s, table=0, n_packets=269, n_bytes=377664, priority=0 actions=CONTROLLER:65535
```

S3

```
cia@localhost:~$ sudo ovs-ofctl dump-flows s3
 cookie=0x0, duration=3234.953s, table=0, n_packets=1291, n_bytes=77460, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=2281.062s, table=0, n_packets=15, n_bytes=630, priority=2,arp,dl_dst=10:00:00:00:00:03 actions=output:"s3-eth1"
 cookie=0x0, duration=2281.007s, table=0, n_packets=6, n_bytes=252, priority=2,arp,dl_dst=10:00:00:00:00:01 actions=output:"s3-eth3"
 cookie=0x0, duration=170.789s, table=0, n_packets=8, n_bytes=336, priority=2,arp,dl_dst=10:00:00:00:00:04 actions=output:"s3-eth2" ARP
 cookie=0x0, duration=170.788s, table=0, n_packets=11, n_bytes=462, priority=2,arp,dl_dst=10:00:00:00:00:02 actions=output:"s3-eth3"
 cookie=0x0, duration=2280.117s, table=0, n_packets=114132, n_bytes=5037813936, priority=2,tcp,dl_dst=10:00:00:00:00:03 actions=output:"s3-eth1"
 cookie=0x0, duration=2278.155s, table=0, n_packets=109744, n_bytes=7243112, priority=2,tcp,dl_dst=10:00:00:00:00:01 actions=output:"s3-eth2" TCP
 cookie=0x0, duration=1012.717s, table=0, n_packets=616, n_bytes=931392, priority=2,udp,dl_dst=10:00:00:00:00:03 actions=output:"s3-eth1"
 cookie=0x0, duration=1007.402s, table=0, n_packets=1, n_bytes=1512, priority=2,udp,dl_dst=10:00:00:00:00:02 actions=output:"s3-eth3" UDP
 cookie=0x0, duration=180.803s, table=0, n_packets=7, n_bytes=686, priority=2,icmp,dl_dst=10:00:00:00:00:03 actions=output:"s3-eth1"
 cookie=0x0, duration=160.401s, table=0, n_packets=2, n_bytes=196, priority=2,icmp,dl_dst=10:00:00:00:00:01 actions=output:"s3-eth2"
 cookie=0x0, duration=150.268s, table=0, n_packets=2, n_bytes=196, priority=2,icmp,dl_dst=10:00:00:00:00:02 actions=output:"s3-eth3" ICMP
 cookie=0x0, duration=140.281s, table=0, n_packets=3, n_bytes=294, priority=2,icmp,dl_dst=10:00:00:00:00:04 actions=output:"s3-eth2"
 cookie=0x0, duration=3235.181s, table=0, n_packets=27, n_bytes=3028, priority=0 actions=CONTROLLER:65535
```

S4

```
cia@localhost:~$ sudo ovs-ofctl dump-flows s4
 cookie=0x0, duration=3264.484s, table=0, n_packets=1292, n_bytes=77520, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=2687.608s, table=0, n_packets=4, n_bytes=168, priority=2,arp,dl_dst=10:00:00:00:00:03 actions=output:"s4-eth3"
 cookie=0x0, duration=230.583s, table=0, n_packets=4, n_bytes=168, priority=2,arp,dl_dst=10:00:00:00:00:01 actions=output:"s4-eth2"
 cookie=0x0, duration=200.402s, table=0, n_packets=11, n_bytes=462, priority=2,arp,dl_dst=10:00:00:00:00:04 actions=output:"s4-eth1" ARP
 cookie=0x0, duration=200.401s, table=0, n_packets=4, n_bytes=168, priority=2,arp,dl_dst=10:00:00:00:00:02 actions=output:"s4-eth3"
 cookie=0x0, duration=3264.306s, table=0, n_packets=2, n_bytes=148, priority=3,tcp,in_port="s4-eth1",tp_dst=80 actions=CONTROLLER:65509 TCP block
 cookie=0x0, duration=3264.306s, table=0, n_packets=903, n_bytes=1365336, priority=3,udp,in_port="s4-eth1" actions=drop UDP drop
 cookie=0x0, duration=2309.730s, table=0, n_packets=114132, n_bytes=5037813936, priority=2,tcp,dl_dst=10:00:00:00:00:03 actions=output:"s4-eth3"
 cookie=0x0, duration=2307.766s, table=0, n_packets=109744, n_bytes=7243112, priority=2,tcp,dl_dst=10:00:00:00:00:01 actions=output:"s4-eth2"   TCP
 cookie=0x0, duration=241.036s, table=0, n_packets=5, n_bytes=490, priority=2,icmp,dl_dst=10:00:00:00:00:03 actions=output:"s4-eth3"
 cookie=0x0, duration=190.012s, table=0, n_packets=5, n_bytes=490, priority=2,icmp,dl_dst=10:00:00:00:00:01 actions=output:"s4-eth2" ICMP
 cookie=0x0, duration=169.892s, table=0, n_packets=8, n_bytes=784, priority=2,icmp,dl_dst=10:00:00:00:00:04 actions=output:"s4-eth1"
 cookie=0x0, duration=157.214s, table=0, n_packets=2, n_bytes=196, priority=2,icmp,dl_dst=10:00:00:00:00:02 actions=output:"s4-eth2"
 cookie=0x1, duration=3264.306s, table=0, n_packets=13, n_bytes=602, priority=1,in_port="s4-eth2" actions=drop Spanning tree
 cookie=0x0, duration=3264.717s, table=0, n_packets=18, n_bytes=924, priority=0 actions=CONTROLLER:65535
```

Here is the design of our flow rules:

Priority: 3	TCP block	match(in_port, tcp_dst_port = 80)	Action(send to controller)
	UDP drop	match(in_port)	Action(drop)
Priority: 2	TCP	match(dst_MAC)	Action(forward)
	UDP	match(dst_MAC)	Action(forward)
	ICMP	match(dst_MAC)	Action(forward)
Priority: 1	Spanning tree	match(in_port)	Action(drop)
Priority: 0	Table miss	match()	Action(send to controller)

(g) Challenges you've encountered while doing this experiment, and explain how you manage to solve them. If you do not experience any problem, simply say no problems.

1. **The header of TCP RST packet.** We spend some time observing how TCP RST packet should be created to correctly reject the source host. These fields should be correctly setup to perform rejections on the source host: MAC src, MAC dst, IP src, IP dst, IP proto, TCP src_port, TCP dst_port. Most importantly, the <u>ACK number</u> should be set as [sequence number + 1], and the <u>control bits</u> should be set as [0b010100] (Enable ACK flag and RST flag).
2. **Discovery of the network topology and spanning tree.** We use LLDP to discovery the topology, and then apply spanning tree algorithm with the whole view of topology. We didn't use STP library provided by RYU.
3. **Handling ARP packets.** We didn't assign fixed rules for ARP packets. Instead, we find the shortest path for ARP packets. However, hosts cannot discovered through LLDP, so we use flooding if the host is not recorded. Otherwise, the controller will find the shortest path.
4. **Policy of lower/left path and upper/right path.** To find either shortest path or following this policy, we cannot adopt the shortest path algorithm in "networkx", so we write our own trivial shortest path algorithm for this special topology.

We have zero tolerance to forged or fabricated data!! A single piece of forged/fabricated data would bring the total score down to zero.