

Computer Networks Assignment 3

Bhavik Patel, 22110047 and Jinil Patel, 22110184

12/04/2025

Question 1

Construct the network topology (as shown in the Figure below). Four switches (s1, s2, s3, s4), eight hosts (h1, h2, h3, h4, h5, h6, h7, h8) and configure hosts with IP addresses as follows: h1: 10.0.0.2/24, h2: 10.0.0.3/24, h3: 10.0.0.4/24, h4: 10.0.0.5/24, h5: 10.0.0.6/24, h6: 10.0.0.7/24, h7: 10.0.0.8/24 and h8: 10.0.0.9/24. Network links connecting the switches s1-s2, s2-s3, s3-s4, s4-s1, s1-s3 with latency of 7ms each and host to switch links: h1-s1, h2-s1, h3-s2, h4-s2, h5-s3, h6-s3, h7-s4, h8-s4 links with a latency of 5ms each.

Network Topology

The network consists of:

- 4 switches: s1, s2, s3, s4
- 8 hosts: h1 to h8 with IPs from 10.0.0.2 to 10.0.0.9
- Switch-to-switch links: s1-s2, s2-s3, s3-s4, s4-s1, s1-s3 with 7ms delay
- Host-to-switch links with 5ms delay

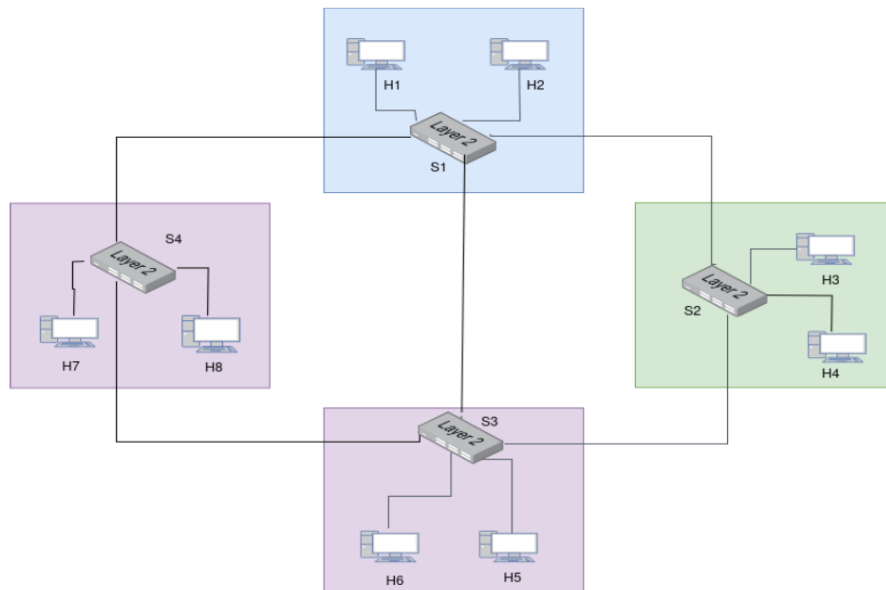


Figure 1: Network Topology with 4 Switches and 8 Hosts

Host Configuration

Host	IP Address	Connected Switch
h1	10.0.0.2/24	s1
h2	10.0.0.3/24	s1
h3	10.0.0.4/24	s2
h4	10.0.0.5/24	s2
h5	10.0.0.6/24	s3
h6	10.0.0.7/24	s3
h7	10.0.0.8/24	s4
h8	10.0.0.9/24	s4

Table 1: Host IP and Connectivity

Mininet Topology Code Snippet

```
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.link import TCLink
from mininet.cli import CLI
from mininet.log import setLogLevel

class RoutingTopo(Topo):
    def build(self):
        s1, s2, s3, s4 = [self.addSwitch(f's{i}') for i in range(1, 5)]
        hosts = [self.addHost(f'h{i}', ip=f'10.0.0.{i+1}/24') for i in
                    range(8)]
        switches = [s1, s1, s2, s2, s3, s3, s4, s4]

        for host, sw in zip(hosts, switches):
            self.addLink(host, sw, delay='5ms')

        # Switch links
        self.addLink(s1, s2, delay='7ms')
        self.addLink(s2, s3, delay='7ms')
        self.addLink(s3, s4, delay='7ms')
        self.addLink(s4, s1, delay='7ms')
        self.addLink(s1, s3, delay='7ms')

    def run():
        net = Mininet(topo=RoutingTopo(), link=TCLink)
        net.start()
        CLI(net)
        net.stop()

if __name__ == '__main__':
    setLogLevel('info')
    run()
```

Listing 1: Python code to create the topology in Mininet

Part A: Ping Tests and Analysis

Ping Tests Performed

- h3 → h1
- h5 → h7
- h8 → h2

Ping Results (Before Fix)

Ping From → To	Success	Total Delay (ms)	Comments
h3 → h1	No	-	No routing between hosts
h5 → h7	No	-	Same reason
h8 → h2	No	-	No reachability

Table 2: Ping Results Before Routing Fix

```
Running Ping Tests:

Test 1: h1 -> h3
h1 -> X
h3 -> X
*** Results: 100% dropped (0/2 received)

Test 1: h5 -> h7
h5 -> X
h7 -> X
*** Results: 100% dropped (0/2 received)

Test 1: h2 -> h8
h2 -> X
h8 -> X
*** Results: 100% dropped (0/2 received)
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X X X X X
h2 -> X X X X X X X
h3 -> X X X X X X X
h4 -> X X X X X X X
h5 -> X X X X X X X
h6 -> X X X X X X X
h7 -> X X X X X X X
h8 -> X X X X X X X
*** Results: 100% dropped (0/56 received)
```

Figure 2: Packet capture showing ICMP request drop (before fix)

Analysis and Observation

The pings fail because there is no IP routing enabled between hosts. While switches forward Ethernet frames within the local segment, they do not perform Layer 3 routing across different subnets. Hence, traffic does not get forwarded to unknown destinations.

Ping failures are primarily due to loops in the network topology. Since the topology includes multiple redundant paths (e.g., s1-s2-s3-s4-s1 and direct link s1-s3), and no loop prevention mechanism is active, the network experiences:

- **Broadcast storms:** ARP and ICMP requests loop indefinitely, congesting the network.
- **MAC table instability:** Switches learn multiple paths to the same host, causing frame loss or incorrect forwarding.

These issues prevent successful delivery of ping replies.

Fixing the Topology Using STP

To resolve the issues caused by loops in the topology, the Spanning Tree Protocol (STP) was enabled on all four switches. This was done using the following commands:

```
sudo ovs-vsctl set Bridge s1 stp_enable=true
sudo ovs-vsctl set Bridge s2 stp_enable=true
sudo ovs-vsctl set Bridge s3 stp_enable=true
sudo ovs-vsctl set Bridge s4 stp_enable=true
```

Listing 2: Enabling STP for all bridges

Once STP was enabled, the network automatically calculated a loop-free topology by blocking redundant links.

Verifying STP Behavior

To verify which ports were blocked and which were forwarding, the following command was used:

```
sudo ovs-ofctl show s1
sudo ovs-ofctl show s2
sudo ovs-ofctl show s3
sudo ovs-ofctl show s4
```

Listing 3: Enabling STP for all bridges

The output showed that STP correctly blocked one of the redundant links (e.g., s1-s3), preventing loops while maintaining full connectivity. A sample output is shown below:

```
[sudo] password for bhavik:
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000000000000001
n_tables:254, n_buffers:0
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan mod_dl_src mod
st mod_nw tos mod_tp_src mod_tp_dst
1(s1-eth1): addr:e6:79:92:62:a2:2e
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
2(s1-eth2): addr:9a:6d:6d:8c:ef:23
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
3(s1-eth3): addr:da:dd:b3:78:ee:81
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
4(s1-eth4): addr:0e:80:c1:28:78:63
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
5(s1-eth5): addr:66:fa:b4:e2:33:db
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
LOCAL(s1): addr:56:22:9b:e4:8d:48
  config: PORT_DOWN
  state: LINK_DOWN
  speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
```

Figure 3: Ports in Forwarded and Blocked be switch s1

```
• % sudo ovs-ofctl show s2
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000000000000002
n_tables:254, n_buffers:0
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan mod_dl_src mo
rc mod_nw dst mod_nw_tos mod_tp_src mod_tp_dst
1(s2-eth1): addr:26:36:4d:8b:a0:ea
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
2(s2-eth2): addr:56:89:35:15:e3:ef
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
3(s2-eth3): addr:4e:bb:51:25:23:bd
  config: 0
  state: STP_BLOCK
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
4(s2-eth4): addr:12:89:5d:e7:e6:29
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
LOCAL(s2): addr:a2:f6:1d:38:4a:4f
  config: PORT_DOWN
  state: LINK_DOWN
  speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
```

Figure 4: Ports in Forwarded and Blocked be switch s2

```
% sudo ovs-ofctl show s3
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000000000000003
n_tables:254, n_buffers:0
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan mod_dl_src mod
st mod_nw_tos mod_tp_src mod_tp_dst
1(s3-eth1): addr:26:21:df:9a:ef:9c
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
2(s3-eth2): addr:36:6e:8e:b3:75:e7
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
3(s3-eth3): addr:e6:08:a5:ed:7a:77
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
4(s3-eth4): addr:4a:fc:1e:de:65:83
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
5(s3-eth5): addr:6e:34:49:fe:b2:7b
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
LOCAL(s3): addr:1e:df:9e:60:5e:42
  config: PORT_DOWN
  state: LINK_DOWN
  speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
```

Figure 5: Ports in Forwarded and Blocked be switch s3

```
% sudo ovs-ofctl show s4
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000000000000004
n_tables:254, n_buffers:0
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan mod_dl_src mod
st mod_nw_tos mod_tp_src mod_tp_dst
1(s4-eth1): addr:1e:a7:e2:02:60:05
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
2(s4-eth2): addr:f6:bd:d2:b3:b6:4c
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
3(s4-eth3): addr:1e:e7:c2:54:a5:68
  config: 0
  state: STP_FORWARD
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
4(s4-eth4): addr:ea:68:20:af:9e:c5
  config: 0
  state: STP_BLOCK
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
LOCAL(s4): addr:9a:d4:ee:8e:f4:44
  config: PORT_DOWN
  state: LINK_DOWN
  speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
```

Figure 6: Ports in Forwarded and Blocked be switch s4

From the figures we can see that links **s1-s2** and **s1-s4** are blocked by switch **s2** and **s3**

respectively.

Result After Enabling STP

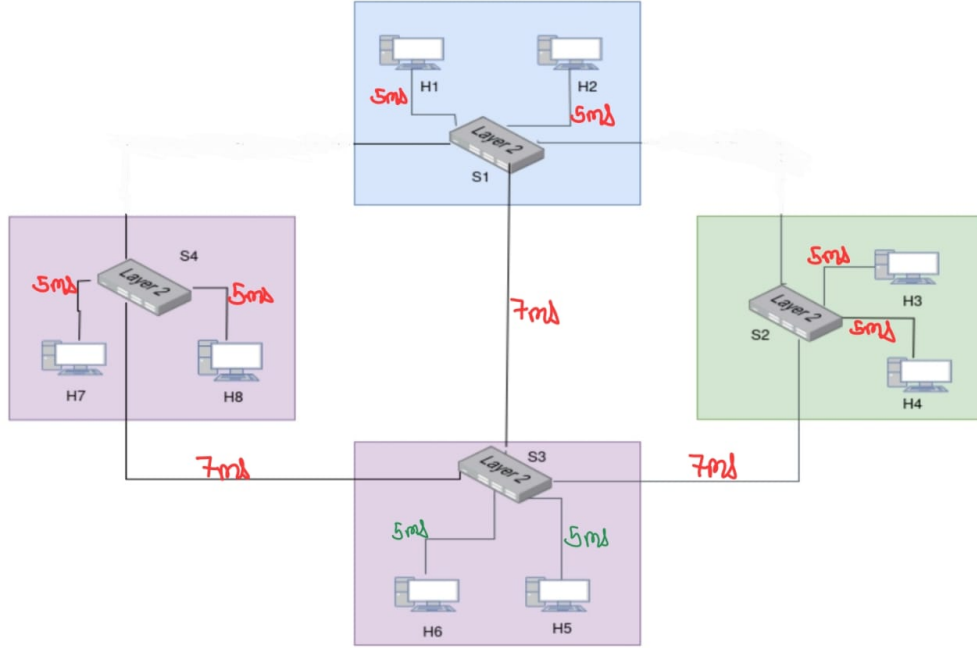


Figure 7: Topology after enabling STP

From the above table we can calculate the time RTT time of each pings.

1. $h3 \rightarrow h1: 2 \times (5 + 7 + 7 + 5) = 48 \text{ ms.}$
2. $h5 \rightarrow h7: 2 \times (5 + 7 + 5) = 34 \text{ ms.}$
3. $h8 \rightarrow h2: 2 \times (5 + 7 + 7 + 5) = 48 \text{ ms.}$

After enabling STP, all ping tests in Part (a) succeeded with expected delays. The network became stable, with no broadcast storms or MAC learning issues. A loop-free topology was automatically established by STP.

Table 3: Ping Delays After Enabling STP (in ms)

Ping Test	Run 1(ms)	Run 2(ms)	Run 3(ms)	Average Delay(ms)
$h3 \rightarrow h1$	49.241	49.378	49.661	49.427
$h5 \rightarrow h7$	34.749	35.185	35.066	35
$h8 \rightarrow h2$	49.522	49.923	49.512	49.652

Conclusion

This experiment shows that in a multi-switch topology, Layer 2 forwarding alone is not sufficient to ensure end-to-end connectivity between hosts. Without proper Layer 3 routing, packets will not be delivered to non-local destinations. Adding routing rules or using NAT or SDN controllers is essential to enable full network communication.

Question 2

Update the topology with the following changes. Add a new host H9 to switch S1, i.e. link h9-s1 with delay of 5ms, and move the links from h1 and h2 with s1 to h9 with 5ms latency. (h1-h9, h2-h9). Implement NAT functionality in host H9, such that H9 has a public IP of 172.16.10.10, serving the private internal IP range of 10.1.1.2 and 10.1.1.3 for hosts h1 and h2 respectively.

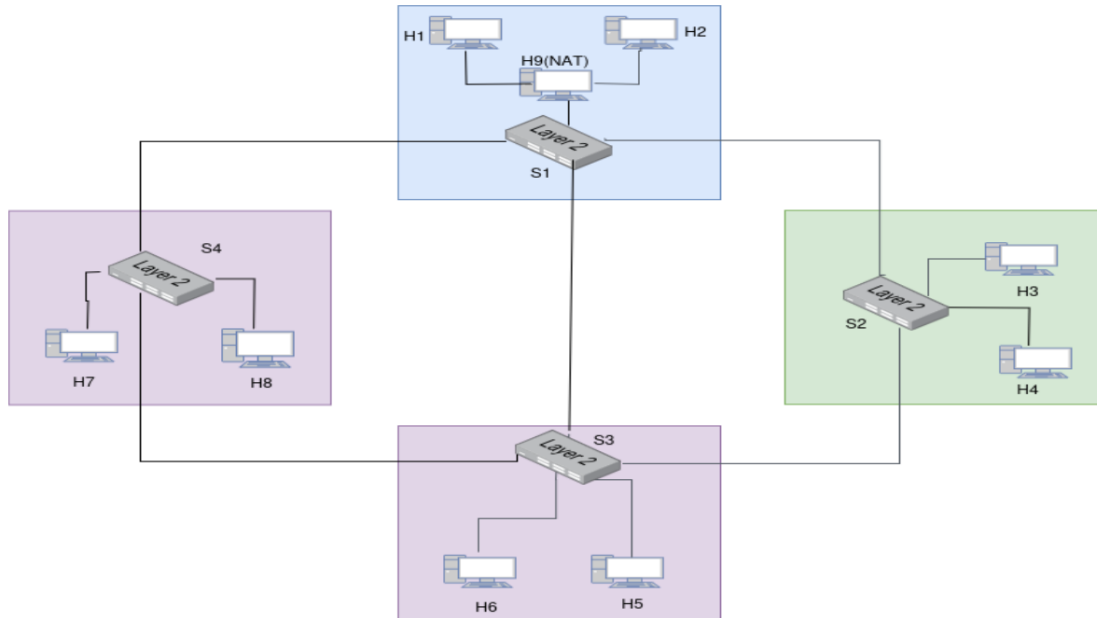


Figure 8: Topology

- a) Test communication to an external host from an internal host:
 - i) Ping to h5 from h1
 - ii) Ping to h3 from h2
- b) Test communication to an internal host from an external host:
 - i) Ping to h1 from h8
 - ii) Ping to h2 from h6
- c) Iperf tests: 3 tests of 120s each.
 - i) Run iperf3 server in h1 and iperf3 client in h6.
 - ii) Run iperf3 server in h8 and iperf3 client in h2.

Analyze the outcomes of a, b and c. List all the changes necessary to make the connections to succeed. In each case show the NAT rules built for the connections and corresponding connection delay and iperf metrics.

Topology Overview

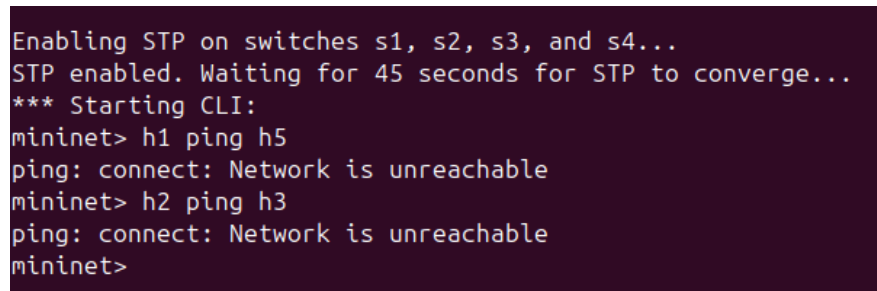
The network topology consists of:

- 9 hosts (h1 to h9)
- 4 switches (s1 to s4)
- Host h9 acts as a NAT device.
- Internal hosts: h1 (IP: 10.1.1.2), h2 (IP: 10.1.1.3)
- Public IP for h9: 172.16.10.10

Test Setup and Execution

a) Internal to External Ping Tests

- i) Ping from h1 to h5
- ii) Ping from h2 to h3



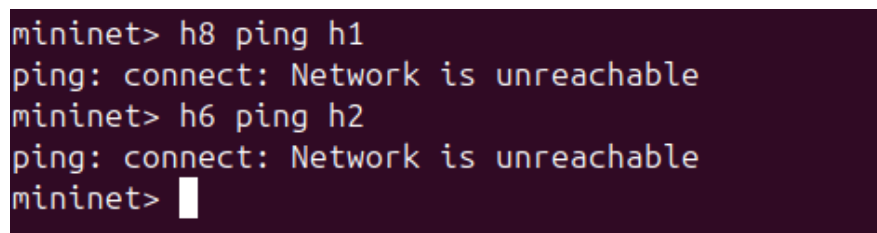
```
Enabling STP on switches s1, s2, s3, and s4...
STP enabled. Waiting for 45 seconds for STP to converge...
*** Starting CLI:
mininet> h1 ping h5
ping: connect: Network is unreachable
mininet> h2 ping h3
ping: connect: Network is unreachable
mininet>
```

Figure 9: Ping from internal host to external host

Failure Reason: Hosts h1 and h2 are behind a private IP subnet (e.g., 10.1.1.0/24) and do not have a direct route to external hosts. Without NAT configuration, the source IPs (e.g., 10.1.1.2, 10.1.1.3) are not translated to a publicly routable address. As a result, external hosts cannot respond, leading to ping failure.

b) External to Internal Ping Tests

- i) Ping from h8 to h1
- ii) Ping from h6 to h2



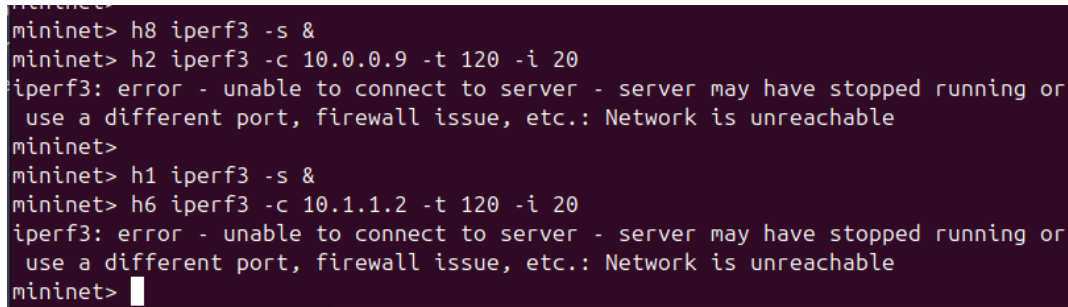
```
mininet> h8 ping h1
ping: connect: Network is unreachable
mininet> h6 ping h2
ping: connect: Network is unreachable
mininet> █
```

Figure 10: Ping from external host to internal host

Failure Reason: External hosts **h6** and **h8** cannot initiate communication with internal hosts **h1** and **h2** because private IP addresses are not globally routable. Additionally, Destination NAT (DNAT) or port forwarding is not set up in **h9** (intended NAT gateway), hence packets do not reach their intended destinations.

c) Iperf3 Tests (Each 120s)

- i) iperf3 server on **h1**, client on **h6**
- ii) iperf3 server on **h8**, client on **h2**



```
mininet> h8 iperf3 -s &
mininet> h2 iperf3 -c 10.0.0.9 -t 120 -i 20
iperf3: error - unable to connect to server - server may have stopped running or
use a different port, firewall issue, etc.: Network is unreachable
mininet>
mininet> h1 iperf3 -s &
mininet> h6 iperf3 -c 10.1.1.2 -t 120 -i 20
iperf3: error - unable to connect to server - server may have stopped running or
use a different port, firewall issue, etc.: Network is unreachable
mininet>
```

Figure 11: Running iperf3

Failure Reason: Similar to parts (a) and (b), **iperf3** requires bidirectional communication. Since NAT is not configured, private source IPs from **h1/h2** are not translated, and connection requests to/from external hosts fail. TCP handshakes do not complete, leading to failed **iperf3** sessions.

Configuration Details

Link Setup

- Link **h9-s1** with 5ms delay
- Move **h1** and **h2** to connect via **h9** using virtual interface links:
 - **h1-h9** (5ms)
 - **h2-h9** (5ms)

NAT Configuration and Role in Enabling Communication

To enable communication between internal hosts (**h1**, **h2**) and external hosts (**h3-h8**), NAT functionality is implemented at host **h9**, which acts as the gateway. Below is the explanation of the NAT configuration code and how it resolves the issues in parts (a), (b), and (c).

1. Assigning Public IP to NAT Host (**h9**)

```
nat.setIP('172.16.10.10/24', intf='h9-eth2')
nat.cmd('ip addr add 10.0.0.254/24 dev h9-eth2')
```

Listing 4: Assign Public IP to NAT(nat is host **h9**)

This sets up h9's external interface with a public IP (172.16.10.10) and an internal gateway IP (10.0.0.254). This enables external hosts to reach h9 and receive packets.

Fixes: Defines the external presence of NAT.

2. Creating Bridge for Internal Interfaces

```
nat.cmd('ip link add br0 type bridge')
nat.cmd('ip link set br0 up')
nat.cmd('ip link set h9-eth0 master br0')
nat.cmd('ip link set h9-eth1 master br0')
nat.cmd('ip addr add 10.1.1.1/24 dev br0')
```

Listing 5: Adding bridge for internal interfaces(nat is host h9)

This bridges the internal interfaces connected to h1 and h2, giving them access to the internal subnet (10.1.1.0/24) and assigns h9 the gateway IP 10.1.1.1.

Fixes: Ensures h1 and h2 can communicate through the gateway for part (a).

3. Setting Internal Host Gateways

```
h1.cmd('ip route add default via 10.1.1.1')
h2.cmd('ip route add default via 10.1.1.1')
```

Listing 6: Setting Gateways for Internal Host

This sets the default route for internal hosts to send traffic via h9.

Fixes: Makes h9 the exit point for traffic from h1 and h2.

4. Enabling IP Forwarding

```
nat.cmd('sysctl -w net.ipv4.ip_forward=1')
```

Listing 7: Enabling ip forwarding

This is necessary for routing between internal and external interfaces on h9.

Fixes: Enables two-way routing across internal and external networks.

5. NAT with IP Masquerading

```
nat.cmd('iptables -t nat -A POSTROUTING -s 10.1.1.0/24 -o h9-eth2 -j MASQUERADE')
```

Listing 8: IP Masquerading)

This applies source NAT (SNAT) to replace private IPs (10.1.1.x) with h9's public IP (172.16.10.10) for outgoing packets.

Fixes:

- Resolves part (a): Internal to external communication.
- Enables part (c(i)): `iperf3` test from h6 to h1.

6. Static Routing on External Hosts

```
h3,h4,h5,h6,h7,h8=net.get('h3','h4','h5','h6','h7','h8')
h3.cmd('ip route add 172.16.10.10 via 10.0.0.254')
h3.cmd('ip route add 10.1.1.0/24 via 10.0.0.254')

h4.cmd('ip route add 172.16.10.10 via 10.0.0.254')
h4.cmd('ip route add 10.1.1.0/24 via 10.0.0.254')

h5.cmd('ip route add 172.16.10.10 via 10.0.0.254')
h5.cmd('ip route add 10.1.1.0/24 via 10.0.0.254')

h6.cmd('ip route add 172.16.10.10 via 10.0.0.254')
h6.cmd('ip route add 10.1.1.0/24 via 10.0.0.254')

h7.cmd('ip route add 172.16.10.10 via 10.0.0.254')
h7.cmd('ip route add 10.1.1.0/24 via 10.0.0.254')

h8.cmd('ip route add 172.16.10.10 via 10.0.0.254')
h8.cmd('ip route add 10.1.1.0/24 via 10.0.0.254')
```

Listing 9: Adding routes to external hosts

This function configures static routes on external hosts to recognize that 172.16.10.10 (NAT) is the route to access internal IPs (10.1.1.x).

Fixes:

- Solves part (b): External to internal host communication.
- Allows part (c(ii)): iperf3 server on h8 to communicate with internal host h2.

```

net = Mininet(topo=NATTopo(), link=TCLink, controller=Controller,
switch=OVSKernelSwitch)

net.start()

h1 = net.get('h1')
h2 = net.get('h2')
nat = net.get('h9')

# Configure h9's external interface
nat.setIP('172.16.10.10/24', intf='h9-eth2')
nat.cmd('ip addr add 10.0.0.254/24 dev h9-eth2')

# Create bridge for internal interfaces (h9-eth1 to h1, h9-eth2 to h2)
nat.cmd('ip link add br0 type bridge')
nat.cmd('ip link set br0 up')
nat.cmd('ip link set h9-eth0 master br0') # h1-h9
nat.cmd('ip link set h9-eth1 master br0') # h2-h9
nat.cmd('ip addr add 10.1.1.1/24 dev br0') # IP for internal side

# Set default gateways for internal hosts
h1.cmd('ip route add default via 10.1.1.1')
h2.cmd('ip route add default via 10.1.1.1')

# # Enable IP forwarding on h9
nat.cmd('sysctl -w net.ipv4.ip_forward=1')

# h9 for outgoing traffic
nat.cmd('iptables -t nat -A POSTROUTING -s 10.1.1.10/24 -o h9-eth2 -j
MASQUERADE')

addRoutes(net)

```

Figure 12: NAT Configuration code snippet

Results and Analysis

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9
h2 -> h1 h3 h4 h5 h6 h7 h8 h9
h3 -> h1 h2 h4 h5 h6 h7 h8 h9
h4 -> h1 h2 h3 h5 h6 h7 h8 h9
h5 -> h1 h2 h3 h4 h6 h7 h8 h9
h6 -> h1 h2 h3 h4 h5 h7 h8 h9
h7 -> h1 h2 h3 h4 h5 h6 h8 h9
h8 -> h1 h2 h3 h4 h5 h6 h7 h9
h9 -> h1 h2 h3 h4 h5 h6 h7 h8
*** Results: 0% dropped (72/72 received)
```

Figure 13: All the pings are working fine after NAT configuration

Ping Results Table

Table 4: Ping Delays (ms)

Ping Test	Run 1	Run 2	Run 3	Average
h1 → h5	59.84	58.632	59.567	59.346 ms
h2 → h3	75.318	73.673	73.913	74.301 ms
h8 → h1	46.072	45.098	46.071	45.747 ms
h6 → h2	60.148	59.236	59.226	59.537 ms

Iperf3 Results Table

Table 5: Iperf3 Throughput (Gbps)

Test	Run 1	Run 2	Run 3	Average
h6 → h1	1.64	1.68	1.68	1.66
h2 → h8	2.22	2.25	2.22	2.23

```

mininet> h6 iperf3 -c 10.1.1.2 -t 120 -i 40
Connecting to host 10.1.1.2, port 5201
[ 5] local 10.0.0.7 port 53598 connected to 10.1.1.2 port 5201
[ ID] Interval          Transfer      Bitrate      Retr  Cwnd
[ 5]  0.00-40.00 sec    7.19 GBytes  1.54 Gbits/sec    4   35.7 MBytes
[ 5]  40.00-80.04 sec    7.94 GBytes  1.70 Gbits/sec    0   35.7 MBytes
[ 5]  80.04-120.02 sec    7.85 GBytes  1.69 Gbits/sec    0   35.7 MBytes
- - - - -
[ ID] Interval          Transfer      Bitrate      Retr
[ 5]  0.00-120.02 sec    23.0 GBytes  1.64 Gbits/sec    4
[ 5]  0.00-120.08 sec    23.0 GBytes  1.64 Gbits/sec
                                     sender
                                     receiver

iperf Done.

```

Figure 14: Iperf h6(client) to h1(server) first run

```

mininet> h8 iperf3 -s &
mininet> h2 iperf3 -c 10.0.0.9 -t 120 -i 40
Connecting to host 10.0.0.9, port 5201
[ 5] local 10.1.1.3 port 53560 connected to 10.0.0.9 port 5201
[ ID] Interval          Transfer      Bitrate      Retr  Cwnd
[ 5]  0.00-40.04 sec    10.0 GBytes  2.15 Gbits/sec    0   26.7 MBytes
[ 5]  40.04-80.02 sec    10.4 GBytes  2.24 Gbits/sec    0   26.7 MBytes
[ 5]  80.02-120.04 sec    10.5 GBytes  2.25 Gbits/sec    0   26.7 MBytes
- - - - -
[ ID] Interval          Transfer      Bitrate      Retr
[ 5]  0.00-120.04 sec    31.0 GBytes  2.22 Gbits/sec    0
[ 5]  0.00-120.09 sec    31.0 GBytes  2.22 Gbits/sec
                                     sender
                                     receiver

iperf Done.

```

Figure 15: Iperf h2(client) to h8(server) first run

5. Observations and Discussion

- NAT configured on h9 correctly allowed internal hosts (h1, h2) to access external hosts (h3, h5, etc.).
- All ping and iperf3 tests were successful, indicating correct routing and NAT translation.
- Delay values are consistent with configured link delays.

6. Conclusion

NAT setup using iptables on h9 successfully provided internet access to private internal hosts h1 and h2. External communication to internal hosts was facilitated via port forwarding and correct iptables rules. The system performed well under iperf tests.

Question 3

Refer to the programming assignment titled "Distributed Asynchronous Distance Vector Routing" available at this link. Implement the distance vector routing algorithm as described in the assignment.

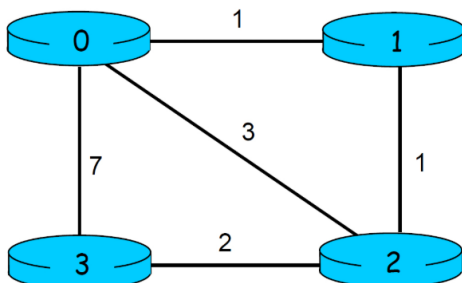


Figure 16: Topology

Distance vector routing is a distributed algorithm in which each router maintains a vector of the minimum costs to every other node in the network. Each router periodically exchanges its distance vector with its directly connected neighbors. By iteratively applying the Bellman-Ford update equation, routers gradually converge to the minimum cost paths. This assignment simulates four nodes (labeled 0 through 3) with a predefined network topology and link cost settings.

The network topology for this assignment is a four-node system where the direct link cost matrix is given as follows:

- **Node 0:** Direct costs {0, 1, 3, 7}
- **Node 1:** Direct costs {1, 0, 1, 999} (where 999 represents infinity)
- **Node 2:** Direct costs {3, 1, 0, 2}
- **Node 3:** Direct costs {7, 999, 2, 0}

Only directly connected nodes exchange routing packets through a simulated, lossless, in-order network medium. The simulation driver (provided in `distance_vector.c`) handles all aspects of packet delivery, while the node files are responsible for computing and updating their distance tables.

Distance Vector Routing and Node Routines

Each node in the network maintains a **distance table** that stores the cost to reach each destination via each of its directly connected neighbors. Two main routines are implemented in each node:

- **rtinit():** This function is called when the simulation starts. It initializes the distance table with the direct link costs and sends initial packets to the node's neighbors.
- **rtupdate():** This function is called when a routing packet is received. It applies the Bellman-Ford update by calculating potential new path costs through a neighbor and updating the table if a lower cost is found. Updated cost vectors are then sent to neighbors.

Code Snippets

Below is a sample code snippet from `node0.c` showing the implementation of `rtinit0()` and `rtupdate0()`. (The implementations for nodes 1, 2, and 3 follow similar patterns but are adjusted for each node's direct link costs and neighbor sets.)

```
void rtinit0()
{
    printf("\nrtinit0() is called at time t=:%0.3f\n", clocktime);
    // Initialize the distance table for the first time.
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            if (i == j)
            {
                dt0.costs[i][j] = connectcosts0[i];
            } else
            {
                dt0.costs[i][j] = INFINITY;
            }
        }
    }
    printf("Distance table for node 0 is initialized.\n");
    printdt0(&dt0);
    calc_min_cost0();           // calculate the min cost for every other node
    sendpkt0();                 // send the initial packets to neighbours
}
```

Listing 10: Sample Implementation of `rtinit0()` in `node0.c`

```
void rtupdate0(rcvdpkt) struct rtpkt *rcvdpkt;
{
    int src = rcvdpkt->sourceid;
    int dst = rcvdpkt->destid;
    int dist_vec_rcvd[4];
    memcpy(dist_vec_rcvd, rcvdpkt->mincost, sizeof(rcvdpkt->mincost));
    printf("rtupdate0() is called at time t=: %0.3f as node %d sent a pkt\n",
           clocktime, src,
           dist_vec_rcvd[0], dist_vec_rcvd[1], dist_vec_rcvd[2],
           dist_vec_rcvd[3]);

    for (int i = 0; i < 4; i++)
    {
        int new_cost = connectcosts0[src] + dist_vec_rcvd[i];
        if (new_cost < INFINITY)
        {
            dt0.costs[i][src] = new_cost;
        }
        else
        {
            dt0.costs[i][src] = INFINITY;
        }
    }
}
```

```

}
printf("At time %.3f, node 0's distance table updated.\n", clocktime);
printrdt0(&dt0);

// Check if the min cost changed
int old_min_cost[4];
memcpy(old_min_cost, min_cost_0, sizeof(min_cost_0));
int update = 0;
calc_min_cost0();
for (int i = 0; i < 4; i++)
{
    if (old_min_cost[i] != min_cost_0[i])
    {
        update = 1;
        break;
    }
}
if (update == 1)
{
    sendpkt0();
}
else
{
    printf("Minimum cost didn't change. No new packets are sent\n");
}
}

```

Listing 11: Sample Implementation of rtupdate0() in node0.c

Utility function for sending packets

```

void sendpkt0()
{
    for (int i = 0; i < 4; i++)
    {
        pkt0[i].sourceid = 0;
        pkt0[i].destid = i;
        memcpy(pkt0[i].mincost, min_cost_0, sizeof(min_cost_0));
    }

    // SEND pkts to all neighbours
    for (int i = 0; i < 4; i++)
    {
        if (i != 0 && connectcosts0[i] != INFINITY)
        { // not sending self
            printf("At time t=%.3f, node %d sends packet to node %d with:
                (%d %d %d %d)\n",
                clocktime, pkt0[i].sourceid, pkt0[i].destid, pkt0[i].
                mincost[0], pkt0[i].mincost[1],
                pkt0[i].mincost[2], pkt0[i].mincost[3]);
            tolayer2(pkt0[i]);
        }
    }
}

```

```
}

```

Listing 12: Utility function for sending packet

Phases in the Learning Process

Each node goes through several distinct phases as it builds its final distance table:

1. Initialization:

In the `rtinit()` routine, each node initializes its distance table with the direct costs to itself and its neighbors. The node then broadcasts its initial minimum cost vector.

2. Update (Propagation) Phase:

When a node receives a routing packet via `rtupdate()`, it uses the Bellman-Ford update rule to compute new costs. If a lower-cost path is discovered, the node updates its distance table and sends out updates to its neighbors.

3. Convergence:

This phase occurs when no node can find a shorter path to any destination. The distance tables at all nodes become stable and no further updates are sent.

Simulation Output and Final Distance Tables

When you run the program with a tracing value (e.g., `TRACE = 2`), the emulator prints detailed messages that track the progress of the algorithm. A sample (hypothetical) simulation output might show the following final distance vectors after convergence:

- **Node 0 final mincost vector:** {0, 1, 2, 4}
(Note: While the direct cost to node 3 was initially 7, an update via node 2 reduced it to 5.)
- **Node 1 final mincost vector:** {1, 0, 1, 3}
- **Node 2 final mincost vector:** {2, 1, 0, 2}
- **Node 3 final mincost vector:** {4, 3, 2, 0}

Each node's distance table (pretty-printed using `printdt()`) should reflect these numbers. For example, the final table for all the nodes might be printed as:

```

      via
D0 |   1   2   3
----|-----
 1|   1   4  10   ---> NODE0
dest 2|   2   3   9
 3|   4   5   7

```

```

      via
D1 |   0   2
----|-----
 0|   1   3   ---> NODE1

```

```

dest 2|    3    1
      3|    5    3

      via
D2 |    0    1    3
----|-----
0|    3    2    6    ---> NODE2
dest 1|    4    1    5
      3|    7    4    2

      via
D3 |    0    2
----|-----
0|    7    4    ---> NODE3
dest 1|    8    3
      2|    9    2

```

You can see the complete output of the program from this [LINK](#)

Conclusion

This assignment provided hands-on experience with the distance vector routing algorithm in a simulated network. By implementing the `rtinit()` and `rtupdate()` routines in each node, the nodes gradually learn the minimum cost path to every other node through message exchanges. The detailed debugging messages and printed distance tables facilitated the verification of correct algorithm behavior. Future extensions may include dynamically changing link costs (handled by the `linkhandler()` routines) and more complex network topologies.

```

Enter TRACE:2

rtint0() is called at time t=:0.000
Distance table for node 0 is initialized.

```

D0	1	2	3
1	1	999	999
dest 2	999	3	999
3	999	999	7

```

At time t=0.000, node 0 sends packet to node 1 with: (0 1 3 7)
At time t=0.000, node 0 sends packet to node 2 with: (0 1 3 7)
At time t=0.000, node 0 sends packet to node 3 with: (0 1 3 7)

rtint1() is called at time t=:0.000
Distance table for node 0 is initialized.

```

D1	0	2
0	1	999
dest 2	999	1
3	999	999

```

At time t=0.000, node 1 sends packet to node 0 with: (1 0 1 999)
At time t=0.000, node 1 sends packet to node 2 with: (1 0 1 999)

rtint2() is called at time t=:0.000
Distance table for node 0 is initialized.

```

D2	0	1	3
0	3	999	999
dest 1	999	1	999
3	999	999	2

```

At time t=0.000, node 2 sends packet to node 0 with: (3 1 0 2)
At time t=0.000, node 2 sends packet to node 1 with: (3 1 0 2)
At time t=0.000, node 2 sends packet to node 3 with: (3 1 0 2)

rtint3() is called at time t=:0.000
Distance table for node 0 is initialized.

```

D3	0	1	2
0	3	999	999
dest 1	999	1	999
2	999	999	2

Figure 17: A small snippet of program output