Assignment - 3

Team 3

April 13, 2025

CS 331

Computer Networks

Prof. Sameer G Kulkarni

INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR

Palaj, Gandhinagar - 382355

Network Loops, NAT, and Routing Algorithms

GitHub Repo

Submitted by

Guntas Singh Saran (22110089)

Computer Science and Engineering

Hitesh Kumar (22110098)

Computer Science and Engineering

Contents

Chapter 1

Network Loop

1.1 Creating the Topology

The network topology comprises four switches (**s1** to **s4**) and eight hosts (**h1** to **h8**). Each host is connected to a switch and configured with a unique IP address within the same subnet. The IP configuration for each host can be easily understood from the given python snippet for the custom Topology Class.

Listing 1.1: Sample Python Listing **main.py**

```python
class CustomTopo(Topo):
    def build(self):
        # Switches
        s1 = self.addSwitch('s1')
        s2 = self.addSwitch('s2')
        s3 = self.addSwitch('s3')
        s4 = self.addSwitch('s4')

        h1 = self.addHost('h1', ip='10.0.0.2/24')
        h2 = self.addHost('h2', ip='10.0.0.3/24')
        h3 = self.addHost('h3', ip='10.0.0.4/24')
        h4 = self.addHost('h4', ip='10.0.0.5/24')
        h5 = self.addHost('h5', ip='10.0.0.6/24')
        h6 = self.addHost('h6', ip='10.0.0.7/24')
        h7 = self.addHost('h7', ip='10.0.0.8/24')
        h8 = self.addHost('h8', ip='10.0.0.9/24')

        # Host - Switch with 5ms delay
        self.addLink(h1, s1, delay='5ms')
        self.addLink(h2, s1, delay='5ms')
        self.addLink(h3, s2, delay='5ms')
        self.addLink(h4, s2, delay='5ms')
        self.addLink(h5, s3, delay='5ms')
        self.addLink(h6, s3, delay='5ms')
        self.addLink(h7, s4, delay='5ms')
        self.addLink(h8, s4, delay='5ms')

        # Switch - Switch with 7ms delay
        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')
```

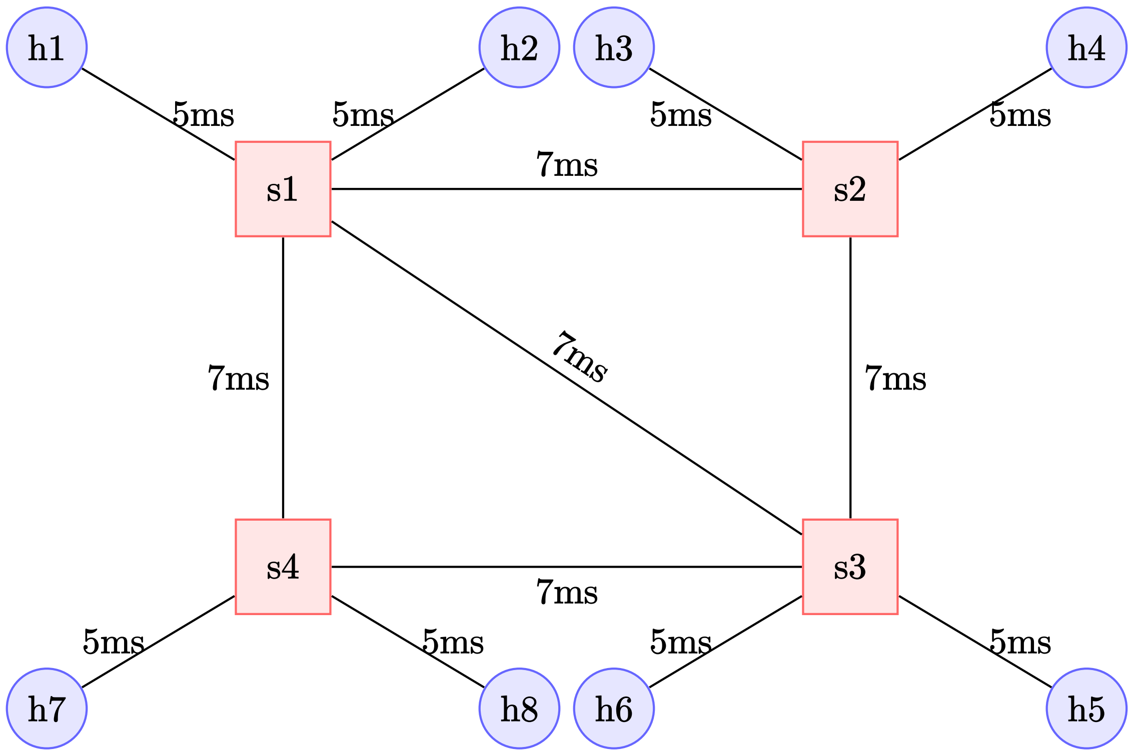


Figure 1.1: Network Topology with Four Switches and Eight Hosts

1.2 Network Behavior on Ping Tests

To evaluate the connectivity and performance of the network, a set of ICMP ping tests were conducted between selected host pairs. Each test was executed three times with a 30-second interval to allow for accurate measurement of latency and to observe consistency.

Ping Tests Performed

- `h3` → `h1`
- `h5` → `h7`
- `h8` → `h2`

Observations

During the initial ping tests, all packets failed to reach their destinations. Each ping resulted in 100% packet loss. This indicated that there was a problem in the network's ability to forward packets correctly between switches.

Analysis

Upon investigation, it was found that the failure was due to the presence of loops in the switch topology. While having redundant paths between switches is generally beneficial for fault tolerance, it can lead to serious network issues if not managed properly.

Switches forward broadcast frames (such as ARP requests) out of all ports except the one they were received on. When there are loops, these broadcast frames keep circulating endlessly through the network. This phenomenon is known as a **broadcast storm**, and it consumes significant bandwidth, leading to degraded performance or complete network failure.

Additionally, Ethernet switches maintain a MAC address table to associate devices with specific ports. In a looped topology, the same frame may arrive at a switch from multiple directions, causing it to constantly update its MAC table with conflicting port information. This leads to **MAC table instability**, where the switch can no longer correctly determine which port to use for a particular destination. As a result, unicast traffic is flooded across the network just like broadcasts, worsening congestion.

Packet Capture Evidence

Packet captures during the ping tests confirmed that ARP requests were being broadcast repeatedly, and no ARP replies were received initially. This behavior is consistent with a switching loop, where broadcast frames circulate endlessly, preventing any meaningful traffic from being delivered.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
2	0.044332369	2e:eb:67:61:27:41	92:57:d9:f4:05:69	ARP	42	10.0.0.2 is at 2e:eb:67:61:27:41
4	0.046193520	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
5	0.046350525	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
6	0.046474564	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
7	0.046599442	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
8	0.047240592	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
9	0.047845424	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
10	0.081102802	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
11	0.081322944	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
12	0.081449777	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
13	0.081694783	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
14	0.081820778	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
15	0.081938951	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
16	0.086589871	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
17	0.086760007	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4

Figure 1.2: First reply to the ARP request from h1 to h3 with h1's MAC Address

No.	Time	Source	Destination	Protocol	Length	Info
43078	80.502314710	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43077	80.502198773	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43076	80.502075851	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43075	80.501954884	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43074	80.501834756	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43073	80.501701777	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43072	80.499761845	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43071	80.499646745	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43070	80.499529690	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43069	80.499412634	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43068	80.499294741	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43067	80.499176848	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43066	80.499059793	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43065	80.498942737	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43064	80.498825682	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
43063	80.498707789	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4

Figure 1.3: Flooding behavior observed as the same ARP packet is repeatedly rebroadcast

No.	Time	Source	Destination	Protocol	Length	Info
18281	19.094436231	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
18282	19.094464448	2e:eb:67:61:27:41	92:57:d9:f4:05:69	ARP	42	10.0.0.2 is at 2e:eb:67:61:27:41
18283	19.094576479	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
18284	19.094604417	2e:eb:67:61:27:41	92:57:d9:f4:05:69	ARP	42	10.0.0.2 is at 2e:eb:67:61:27:41
18285	19.094716447	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
18286	19.094745223	2e:eb:67:61:27:41	92:57:d9:f4:05:69	ARP	42	10.0.0.2 is at 2e:eb:67:61:27:41
18287	19.094857254	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
18288	19.094885471	2e:eb:67:61:27:41	92:57:d9:f4:05:69	ARP	42	10.0.0.2 is at 2e:eb:67:61:27:41
18289	19.094997502	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
18290	19.095026278	2e:eb:67:61:27:41	92:57:d9:f4:05:69	ARP	42	10.0.0.2 is at 2e:eb:67:61:27:41
18291	19.095138308	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
18292	19.095167364	2e:eb:67:61:27:41	92:57:d9:f4:05:69	ARP	42	10.0.0.2 is at 2e:eb:67:61:27:41
18293	19.099686584	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
18294	19.099767604	2e:eb:67:61:27:41	92:57:d9:f4:05:69	ARP	42	10.0.0.2 is at 2e:eb:67:61:27:41
18295	19.099967638	92:57:d9:f4:05:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
18296	19.100000605	2e:eb:67:61:27:41	92:57:d9:f4:05:69	ARP	42	10.0.0.2 is at 2e:eb:67:61:27:41

Figure 1.4: Response from h1 after STP was enabled and loops were eliminated

1.3 Solution: Spanning Tree Protocol (STP)

Spanning Tree Protocol (STP) is a Layer 2 protocol that prevents loops in Ethernet networks by selectively blocking redundant paths while keeping one active path between any two nodes. In our initial configuration, STP was not enabled, which led to broadcast storms and MAC table instability due to the presence of loops in the network. To resolve this, STP was enabled on all switches to ensure a loop-free topology.

Once STP was activated, the resulting topology and port states were examined using Mininet commands. The first observation was the full connectivity of hosts and switches as shown in the output of the `net` command below:

```
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
h3 h3-eth0:s2-eth1
h4 h4-eth0:s2-eth2
h5 h5-eth0:s3-eth1
h6 h6-eth0:s3-eth2
h7 h7-eth0:s4-eth1
h8 h8-eth0:s4-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0 s1-eth3:s2-eth3 s1-eth4:s4-eth4 s1-eth5:s3-eth5
s2 lo:  s2-eth1:h3-eth0 s2-eth2:h4-eth0 s2-eth3:s1-eth3 s2-eth4:s3-eth3
s3 lo:  s3-eth1:h5-eth0 s3-eth2:h6-eth0 s3-eth3:s2-eth4 s3-eth4:s4-eth3 s3-eth5:s1-eth5
s4 lo:  s4-eth1:h7-eth0 s4-eth2:h8-eth0 s4-eth3:s3-eth4 s4-eth4:s1-eth4
c0
```

Next, the status of each switch interface was inspected using the `dpctl show` command. From the output, it was evident that STP had successfully blocked the redundant paths by placing the following ports into `STP_BLOCK` state:

- `s2-eth4`: **s2**'s interface connected with **s3** switch's **s3-eth3** interface.
- `s3-eth5`: **s3**'s interface connected with **s1** switch's **s1-eth5** interface.

NOTE: These blocked interfaces might differ each time we create the topology in mininet and start STP. So, the results for another run may differ.

All remaining interfaces were in the `STP_FORWARD` state, actively handling traffic while preventing any loops in the network. But, STP takes some time to converge and during that time the state of all those interfaces remains `STP_LEARN`.

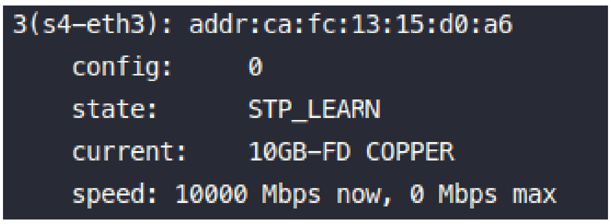

```
3(s4-eth3): addr:ca:fc:13:15:d0:a6
    config:     0
    state:      STP_LEARN
    current:    10GB-FD COPPER
    speed: 10000 Mbps now, 0 Mbps max
```

Figure 1.5: STP Learn state while the Redundant links are to be found

To visually represent the updated, loop-free topology after STP convergence, the TikZ diagram shown below highlights only the active links. Blocked interfaces (s2-eth4 and s3-eth5) have been excluded to reflect the logical topology used for data forwarding:

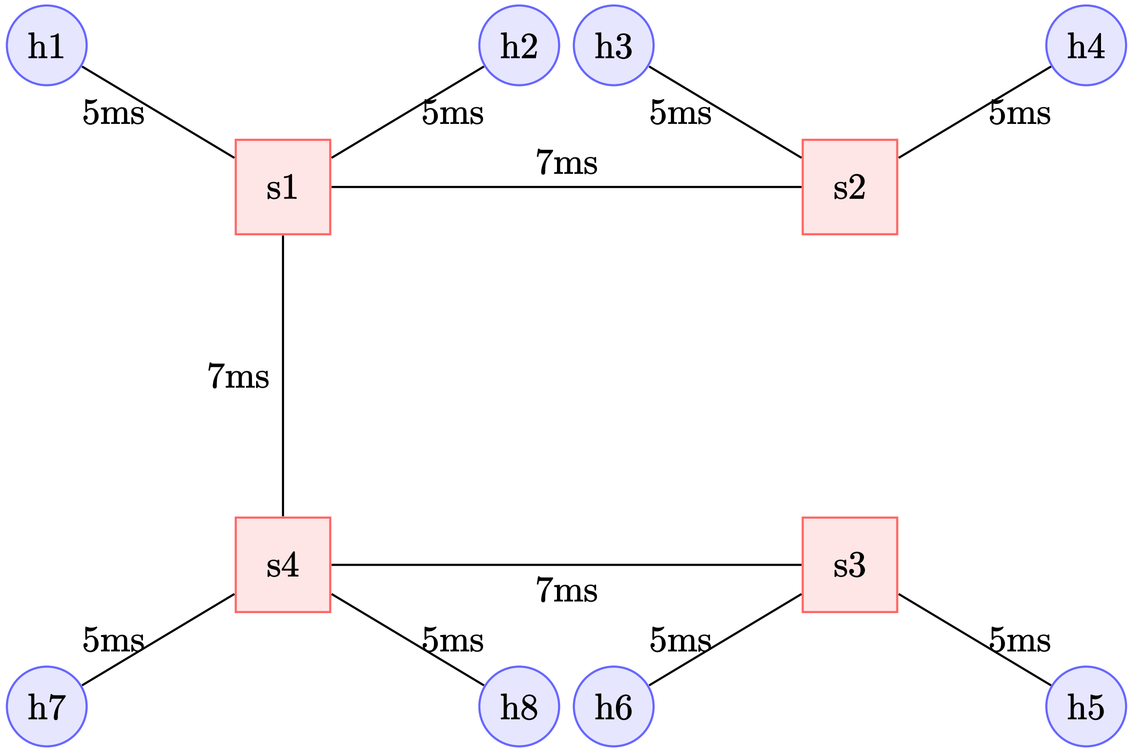


Figure 1.6: Effective Network Topology after STP has Blocked Redundant Interfaces

1.4 Ping Test Results

Table 1.1: Average RTTs for 30s Ping Tests After STP Convergence

Ping Test	Attempt 1 (ms)	Attempt 2 (ms)	Attempt 3 (ms)	Avg. Time (ms)
h3 → h1	34.469	34.327	34.763	**34.520**
h5 → h7	35.326	34.440	34.248	**34.671**
h2 → h8	35.934	34.731	34.818	**35.161**

The ping results observed for all the three ping configurations can also be inferred from the fig. 1.6. To show the correctness of the Network Topology let's try to ping h5 from h4 and for that the expected RTT is about 62 ms through s2 → s1 → s4 → s3. The figure below shows the similar results with RTT having 65 ms value nearly equal to 62 ms.

```
--- 10.0.0.6 ping statistics ---
30 packets transmitted, 30 received, 0% packet loss, time 29037ms
rtt min/avg/max/mdev = 62.125/65.054/130.920/12.235 ms
```

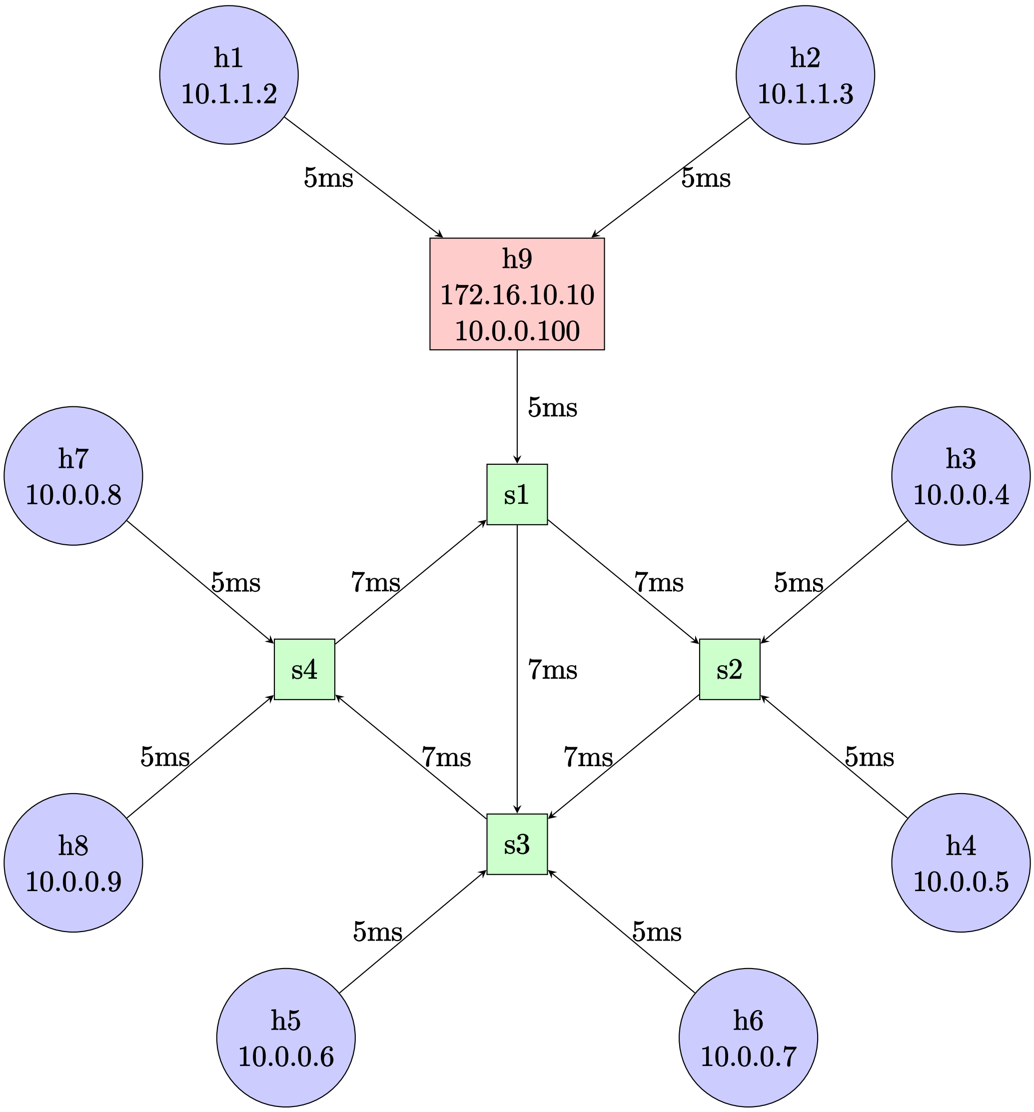
Figure 1.7: Ping Result for h4 → h5

1.5 Conclusion

This experiment highlighted the importance of managing loops in switch-based networks. Without STP, loops caused broadcast storms and disrupted communication. Enabling STP ensured a loop-free topology, stabilized MAC learning, and restored reliable connectivity. Thus, STP is crucial for maintaining robust and efficient layer-2 network operations.

Chapter 2

Configure Host-based NAT



```
mininet> net
h1 h1-eth0:h9-eth1
h2 h2-eth0:h9-eth2
h3 h3-eth0:s2-eth1
h4 h4-eth0:s2-eth2
h5 h5-eth0:s3-eth1
h6 h6-eth0:s3-eth2
h7 h7-eth0:s4-eth1
h8 h8-eth0:s4-eth2
h9 h9-eth0:s1-eth1 h9-eth1:h1-eth0 h9-eth2:h2-eth0
s1 lo:  s1-eth1:h9-eth0 s1-eth2:s2-eth3 s1-eth3:s4-eth4 s1-eth4:s3-eth5
s2 lo:  s2-eth1:h3-eth0 s2-eth2:h4-eth0 s2-eth3:s1-eth2 s2-eth4:s3-eth3
s3 lo:  s3-eth1:h5-eth0 s3-eth2:h6-eth0 s3-eth3:s2-eth4 s3-eth4:s4-eth3 s3-eth5:s1-eth4
s4 lo:  s4-eth1:h7-eth0 s4-eth2:h8-eth0 s4-eth3:s3-eth4 s4-eth4:s1-eth3
c0
```

Figure 2.1: The intial interfaces in the topology.

2.1 Topology and Changes

The original topology featured four Layer 2 switches (S1, S2, S3, S4) with hosts H1-H8 connected directly to switches. H9 was intended as a NAT host. Modifications included:

- Moved H1 and H2 from S1 to H9 with 5ms delay.
- Added H9-S1 link with 5ms delay.
- Configured H9 with internal IP `10.1.1.1/24` (`bridge br0`) and external IP 172.16.10.10/24.

2.2 Observations

2.2.1 Parts A and B Communication

- **Part A (Internal to External)**:
  - Ping from H1 to H5 and H2 to H3 initially failed due to the absence of a default gateway and NAT configuration on H9.
- **Part B (External to Internal)**:
  - Pings from H8 to H1 and H6 to H2 initially failed due to lack of DNAT and routing to the internal network (`10.1.1.0/24`).

```
mininet> h1 ping -c 3 h5
ping: connect: Network is unreachable
mininet> h2 ping -c 3 h3
ping: connect: Network is unreachable
mininet> h8 ping -c 3 h1
ping: connect: Network is unreachable
mininet> h6 ping -c 3 h2
ping: connect: Network is unreachable
```

Figure 2.2: Initially no connection is established across internal or external hosts of respective subnets.

2.2.2 H1 and H2 Communication

- H1 and H2 were not communicating initially because they lacked a common interface and were directly connected to S1 instead of H9. The addition of a bridge interface on H9 resolved this, enabling direct communication within the `10.1.1.0/24` subnet without requiring NAT.

```
mininet> h1 ping -c 3 h2
PING 10.1.1.3 (10.1.1.3) 56(84) bytes of data.
64 bytes from 10.1.1.3: icmp_seq=1 ttl=64 time=45.7 ms
64 bytes from 10.1.1.3: icmp_seq=2 ttl=64 time=23.8 ms
64 bytes from 10.1.1.3: icmp_seq=3 ttl=64 time=23.8 ms

--- 10.1.1.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 23.799/31.106/45.691/10.313 ms
```

Figure 2.3: h1 and h2 now become reachable owing to the bridge interface.

2.3 Issues and Rectifications

2.3.1 Initial Issues

- **Lack of Internal Connectivity**: No bridge or common interface existed for H1 and H2.
- **No Gateway for Internal Hosts**: H1 and H2 lacked default routes to reach external networks.
- **No External Access to Internal Network**: External hosts (H3-H8) could not route to `10.1.1.0/24` or perform reverse NAT.
- **Missing NAT Configuration**: IP forwarding and iptables rules were incomplete.

2.3.2 Rectifications with Code

- **Bridge Interface**:
  - **Issue**: No common interface for H1 and H2 to communicate.
  - **Rectification**: Created bridge `br0` on H9 to connect H1 and H2.
  - **Code**:

```
h1, h2, h9 = net.get("h1", "h2", "h9")
h9.cmd("ip link add br0 type bridge")
h9.cmd("ip link set br0 up")
h9.cmd("ip link set h9-eth1 master br0")  # h1-h9
h9.cmd("ip link set h9-eth2 master br0")  # h2-h9
```

  - **Explanation**: The `bridge (br0)` acts as a virtual switch, connecting H1 and H2 via H9's interfaces (h9-eth1, h9-eth2), allowing Layer 2 communication within `10.1.1.0/24`.

- **Internal IP for NAT**:
  - **Issue**: H9 lacked an internal IP to serve as a gateway.
  - **Rectification**: Assigned `10.1.1.1/24` to `br0`.
  - **Code**:

```
h9.cmd("ip addr add 10.1.1.1/24 dev br0") # IP for internal side
```

  - **Explanation**: This IP serves as the default gateway for H1 (`10.1.1.2`) and H2 (`10.1.1.3`), enabling them to route traffic through H9.

- **Additional IP for Routing**:
  - **Issue**: External hosts needed a route to H9's external network.
  - **Rectification**: Added `10.0.0.100/24` to `h9-eth0`.
  - **Code**:

```
h9.cmd("ip addr add 10.0.0.100/24 dev h9-eth0")
```

  - **Explanation**: This secondary IP on h9-eth0 allows external hosts in the `10.0.0.0/24` network to route traffic to H9, facilitating communication with the internal network.

- **Default Routes**:
  - **Issue**: H1 and H2 had no gateway to external networks.
  - **Rectification**: Set default routes to `10.1.1.1`.
  - **Code**:

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

  - **Explanation**: These routes direct all unknown traffic from H1 and H2 to H9 (`10.1.1.1`), which then handles NAT to external networks.

- **IP Forwarding**:
  - **Issue**: H9 could not forward packets between interfaces.
  - **Rectification**: Enabled IP forwarding.
  - **Code**:

```
h9.cmd("sysctl -w net.ipv4.ip_forward=1")
```

  - **Explanation**: This enables H9 to act as a router, forwarding packets between the internal bridge (`br0`) and external interface (`h9-eth0`).

- **IPTables**:
  - **Issue**: No NAT rules existed for internal-to-external or external-to-internal traffic.
  - **Rectification**: Added `MASQUERADE` and forwarding rules.
  - **Code**:

```
h9.cmd("iptables -t nat -A POSTROUTING -s 10.1.1.0/24 -o h9-eth0
    -j MASQUERADE")
```

  - **Explanation**: The `POSTROUTING` rule translates H1/H2's private IPs (`10.1.1.0/24`) to H9's public IP (`172.16.10.10`) for external access.

```
Chain POSTROUTING (policy ACCEPT 15 packets, 1024 bytes)
 pkts bytes target     prot opt in     out     source               destination

   12   864 MASQUERADE  0    --  *      h9-eth0  10.1.1.0/24          0.0.0.0/0
```

Figure 2.4: POSTROUTING configuration added in NAT's IP Table

- **Route Add for External Hosts**:
  - **Issue**: External hosts could not reach H9 or the internal network.
  - **Rectification**: Added routes to `172.16.10.10` and `10.1.1.0/24` via `10.0.0.100`.
  - **Code**:

```
for host in [h3, h4, h5, h6, h7, h8]:
    host.cmd("ip route add 172.16.10.10 via 10.0.0.100") # Route
    to NAT host
    host.cmd("ip route add 10.1.1.0/24 via 10.0.0.100") # Route
    to internal network
```

  - **Explanation**: These routes allow H3-H8 to send traffic to H9's external IP and the internal subnet, enabling reverse communication.

- **Enable Spanning Tree Protocol**:
  - **Issue**: External hosts will be unreachable due to redundant links present in the topology.
  - **Rectification**: Enable STP same as did in Question 1.

– **Code**:

```
for switch in ['s1', 's2', 's3', 's4']:
    net.get(switch).cmd('ovs-vsctl set bridge {} stp_enable=true'
    .format(switch))
print("Waiting for STP to converge...")
sleep(30)
```

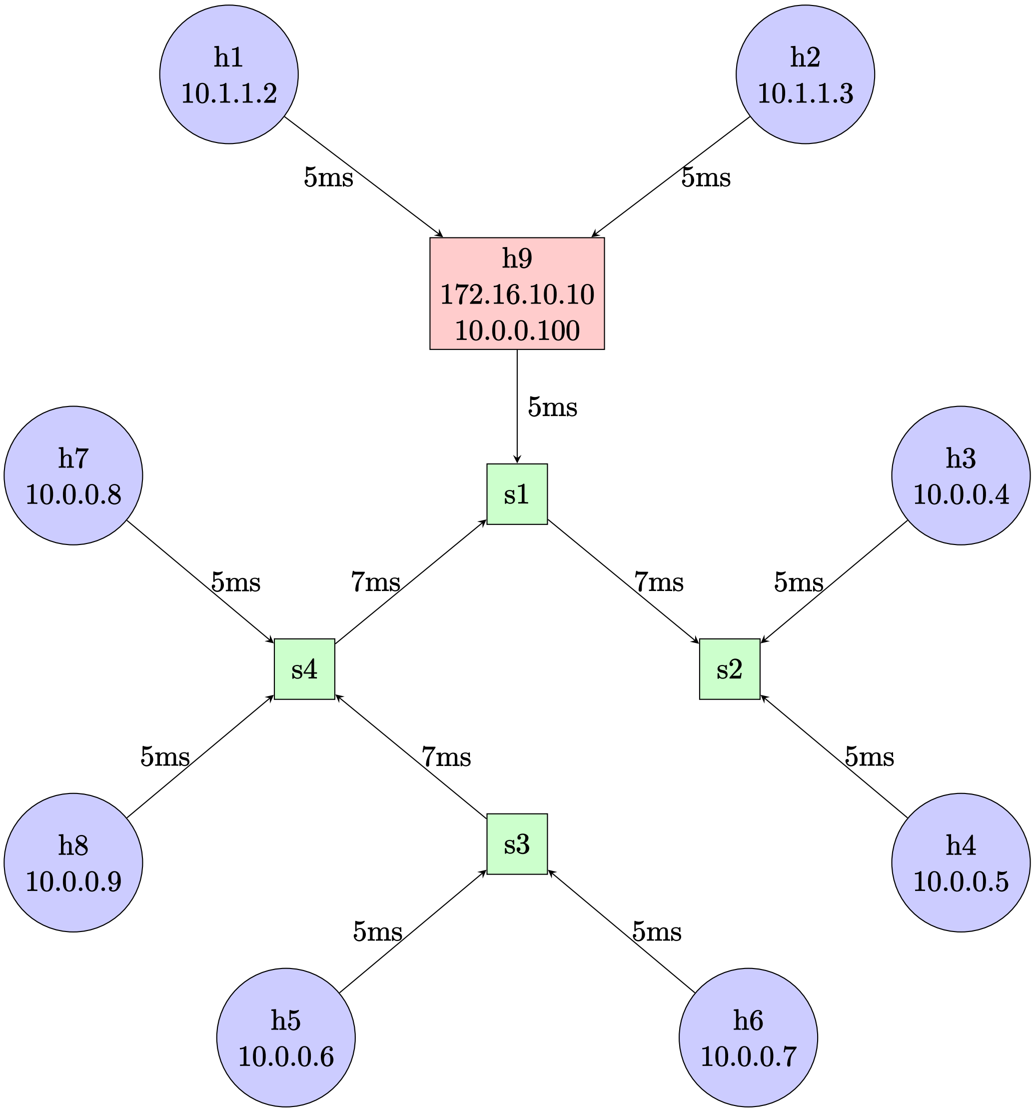


Figure 2.5: The links **s1-s3** and **s2-s3** have been disconnected as a result of blocking of redundant interfaces **s1-eth4** and **s2-eth4** by STP. Their identification is done as done in Ques 1.

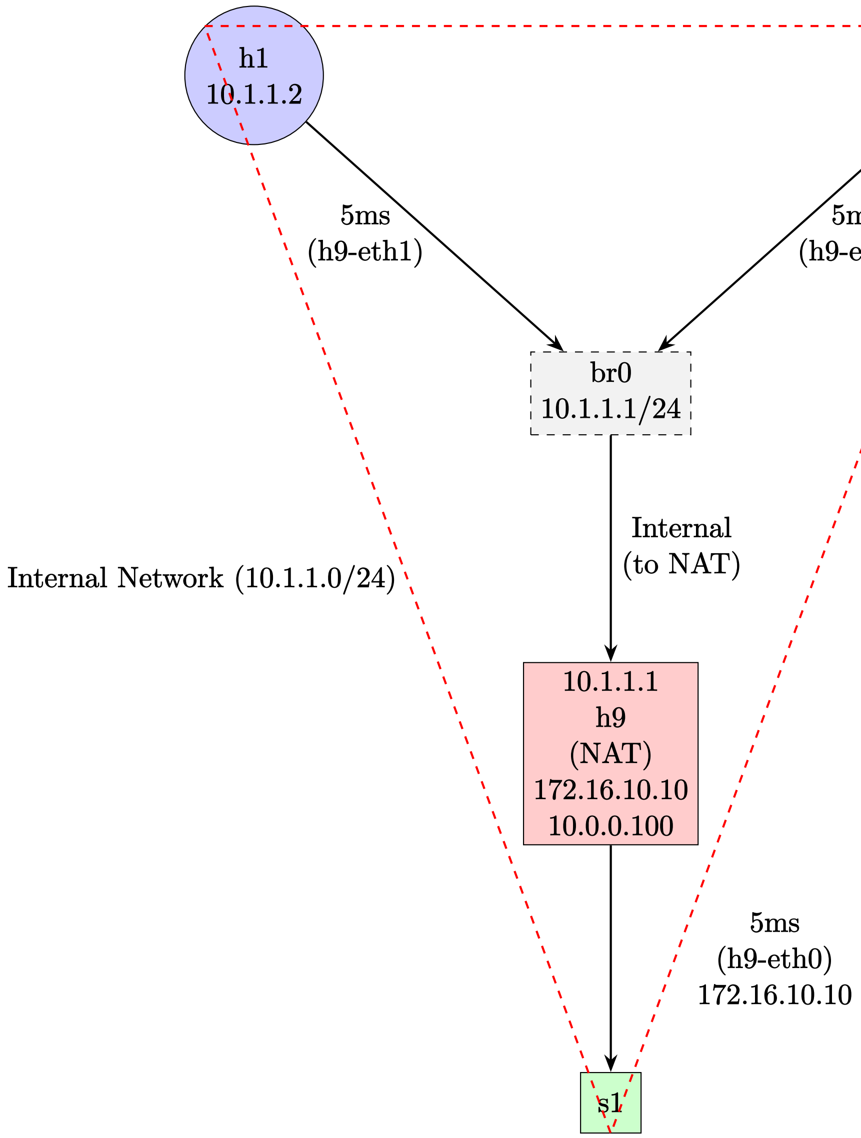
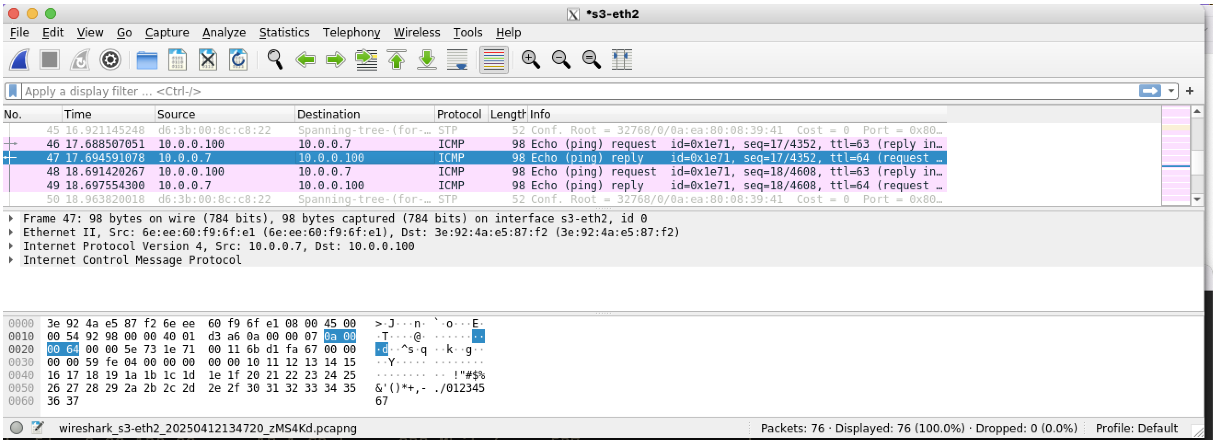


Figure 2.6: Figure showing NAT having aliases for both internal subnet (10.1.1./24) and to the external subnet (10.0.0./24) with the internal **`bridge br0`** interface

```
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 2.7: All hosts being reachable to each other.




```
[  5]   0.00-120.08 sec  13.1 GBytes   938 Mbits/sec                  receiver

iperf Done.
mininet> h1 ping h6
PING 10.0.0.7 (10.0.0.7) 56(84) bytes of data.
64 bytes from 10.0.0.7: icmp_seq=1 ttl=63 time=68.3 ms
64 bytes from 10.0.0.7: icmp_seq=2 ttl=63 time=60.3 ms
64 bytes from 10.0.0.7: icmp_seq=3 ttl=63 time=62.6 ms
64 bytes from 10.0.0.7: icmp_seq=4 ttl=63 time=58.2 ms
64 bytes from 10.0.0.7: icmp_seq=5 ttl=63 time=55.3 ms
64 bytes from 10.0.0.7: icmp_seq=6 ttl=63 time=58.2 ms
64 bytes from 10.0.0.7: icmp_seq=7 ttl=63 time=48.9 ms
64 bytes from 10.0.0.7: icmp_seq=8 ttl=63 time=51.7 ms
64 bytes from 10.0.0.7: icmp_seq=9 ttl=63 time=53.1 ms
64 bytes from 10.0.0.7: icmp_seq=10 ttl=63 time=53.7 ms
64 bytes from 10.0.0.7: icmp_seq=11 ttl=63 time=52.3 ms
```

Figure 2.8: **h1 ping h6**: wireshark packet capture shows h6 (`10.0.0.7`) reaches the NAT's same subnet (`10.0.0.100`) rather than h1 directly.

2.4 Finally all the Communication Tests

2.4.1 Communication to an external host from an internal host

1. Ping to h5 from h1: (Expected RTT = 58 ms)
2. Ping to h3 from h2: (Expected RTT = 44 ms)

Table 2.1: Average RTTs for 30s Ping Tests After STP Convergence

Ping Test	Attempt 1 (ms)	Attempt 2 (ms)	Attempt 3 (ms)	Avg. Time (ms)
h1 → h5	60.677	58.969	58.746	**59.464**
h2 → h3	46.068	44.703	45.023	**45.265**

2.4.2 Test communication to an internal host from an external host

1. Ping to h1 from h8: (Expected RTT = 44 ms)
2. Ping to h2 from h6: (Expected RTT = 58 ms)

Table 2.2: Average RTTs for 30s Ping Tests After STP Convergence

Ping Test	Attempt 1 (ms)	Attempt 2 (ms)	Attempt 3 (ms)	Avg. Time (ms)
h8 → h1	46.005	44.980	44.350	**45.112**
h6 → h2	60.786	58.717	58.814	**59.439**

2.4.3 `iperf` tests: 3 tests of 120s each.

1. Run iperf3 server in h1 and iperf3 client in h6 at port 1212.

```
h1 iperf3 -s -p 1212 -D
h6 iperf3 -c h1 -p 1212 -t 120
```

2. Run iperf3 server in h8 and iperf3 client in h2 at port 1212.

```
h8 iperf3 -s -p 1212 -D
h2 iperf3 -c h8 -p 1212 -t 120
```

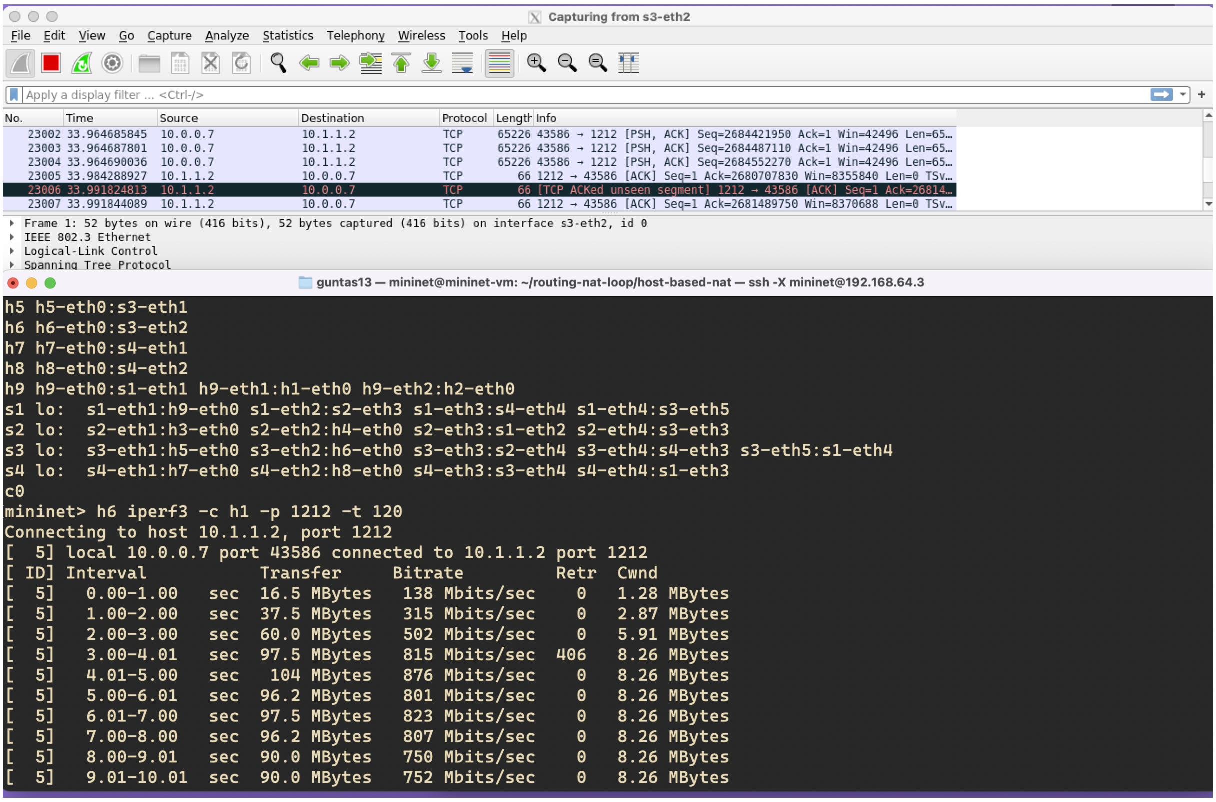


Figure 2.9: Server in h1 and iperf3 Client in h6. Also, see packets in the wireshark.

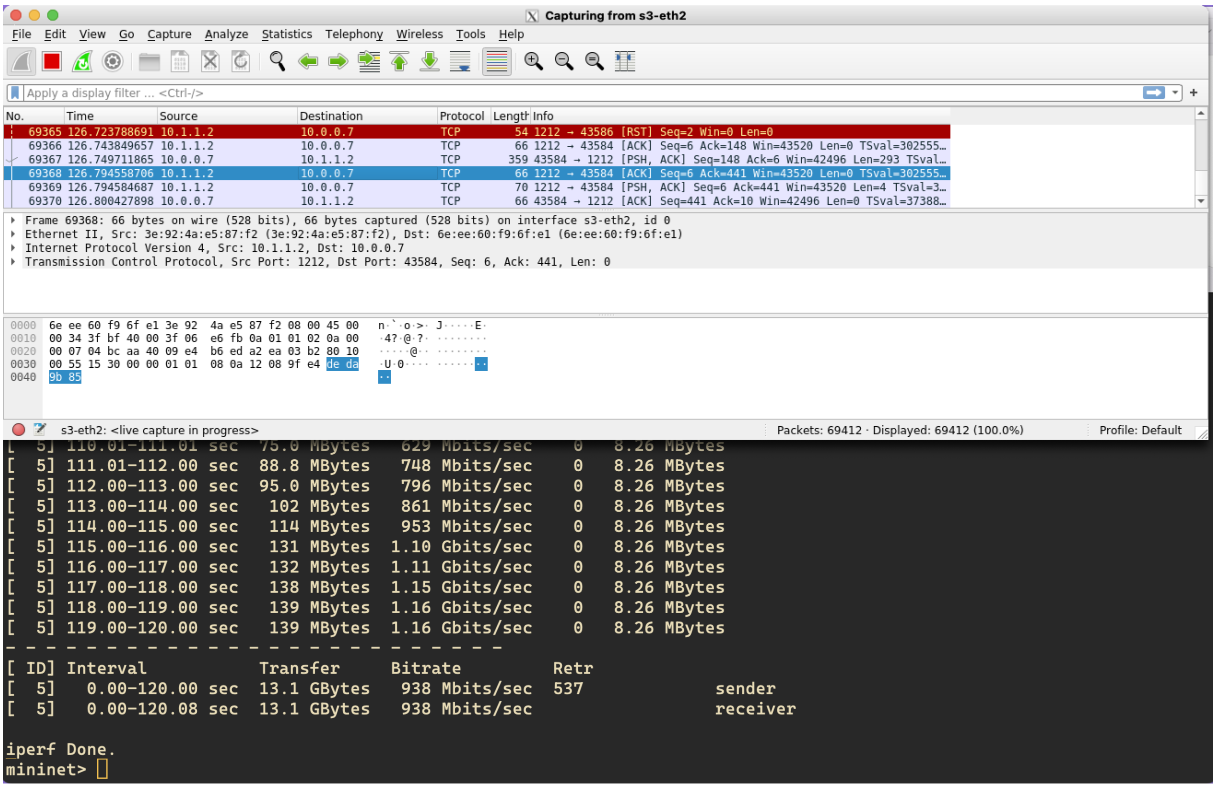


Figure 2.10: Final Result of running iperf: can see the completed packets in wireshark.

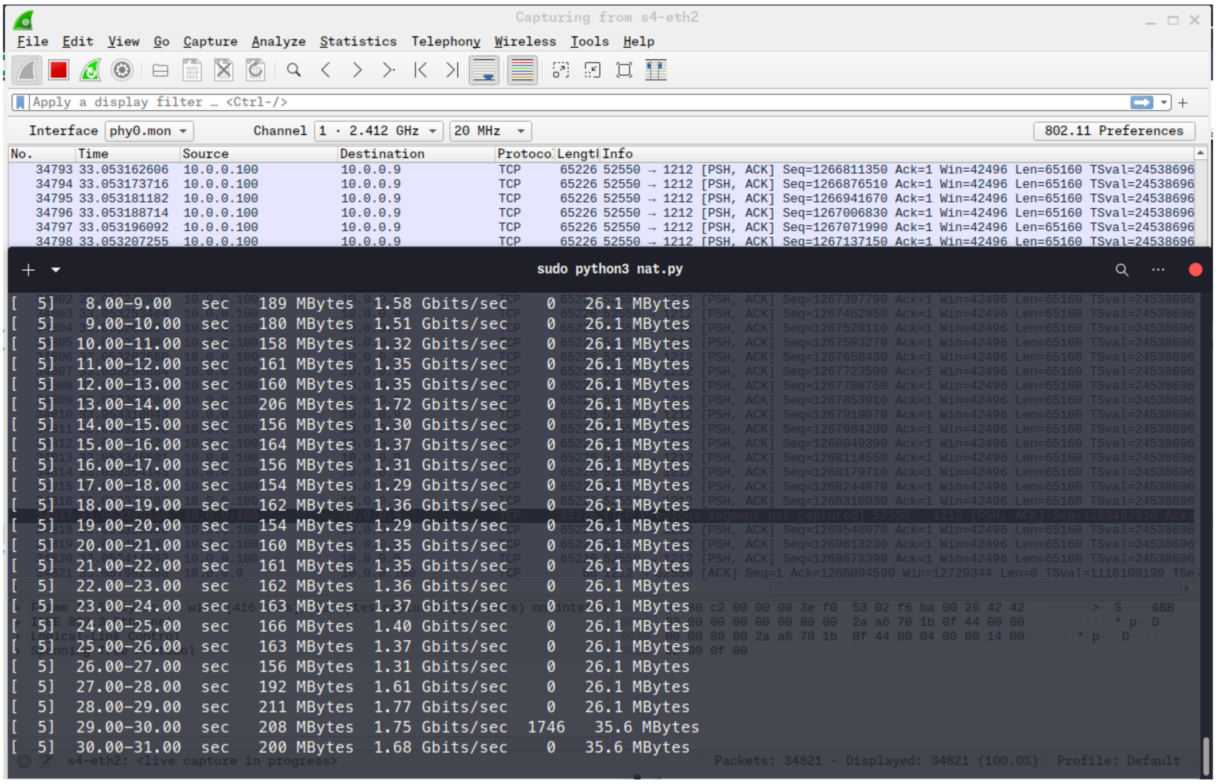


Figure 2.11: Server in h8 and iperf3 Client in h2. Also, see packets in the wireshark.

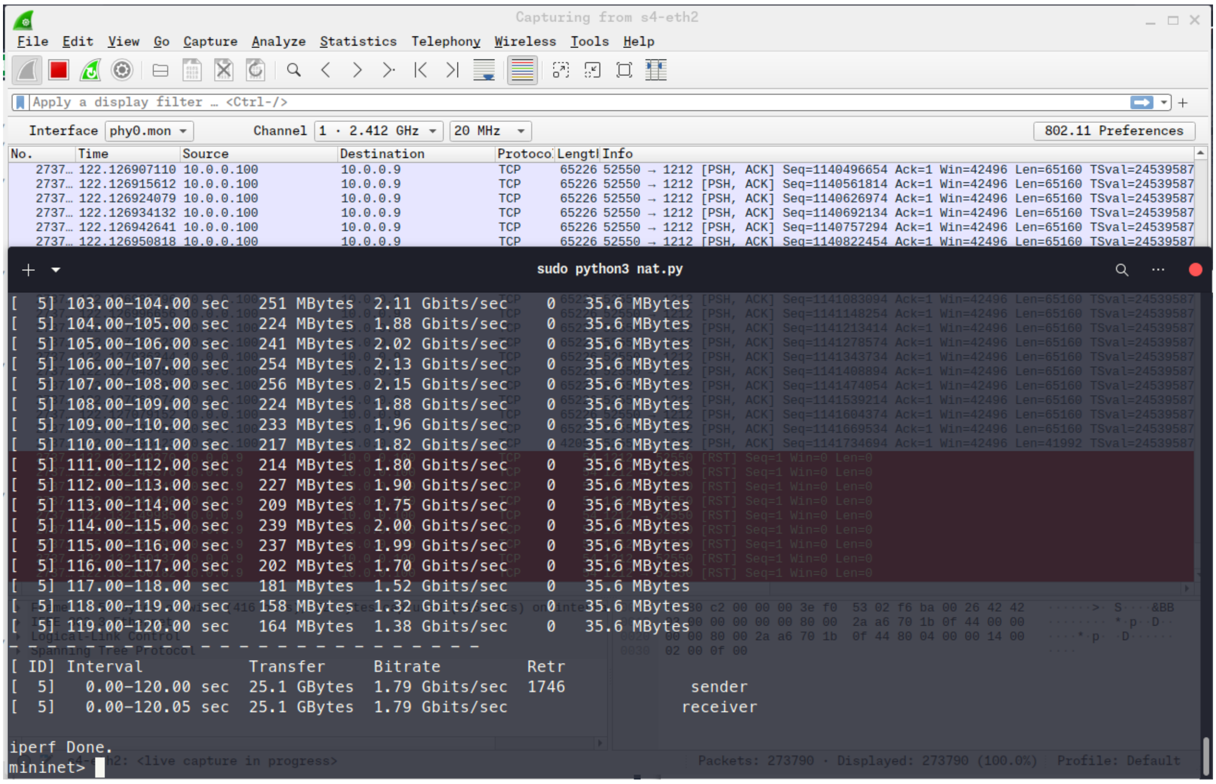


Figure 2.12: Final Result of running iperf: can see the completed packets in wireshark.

Chapter 3

Distributed Asynchronous Distance Vector Routing

3.1 Distance Vector Routing Algorithm

The Distance Vector Routing Algorithm, also known as the Bellman-Ford algorithm, operates on the principle that each node maintains a vector of distances to all other nodes in the network and periodically shares this vector with its directly connected neighbors. The key steps are:

1. **Initialization**: Each node initializes its distance table with the costs to its directly connected neighbors and sets the cost to non-neighbors as infinity (e.g., 999). The distance to itself is zero.

2. **Information Exchange**: Nodes send their distance vectors to neighbors, containing the minimum cost to reach each node.

3. **Update**: Upon receiving a distance vector from a neighbor, a node updates its distance table using the Bellman-Ford equation:

$$D_x(y) = \min_v [c(x, v) + D_v(y)]$$

   where $D_x(y)$ is the cost from node x to node y, $c(x, v)$ is the cost of the link from x to neighbor v, and $D_v(y)$ is the cost reported by v to y.

4. **Convergence**: If the minimum cost to any destination changes, the node sends an updated distance vector to its neighbors, continuing until all tables stabilize.

This process is asynchronous, meaning updates occur independently as packets are received, and the network emulator ensures in-order delivery without loss.

3.2 Implementation Approach

The implementation involves four source files (**`node0.c`**, **`node1.c`**, **`node2.c`**, **`node3.c`**), each corresponding to a node, and a main emulator file (**`distance_vector.c`**). The approach includes:

3.2.1 Data Structures

- **Distance Table**: A 4x4 matrix (**`costs[i][j]`**) where **`costs[i][j]`** is the cost from node i to node j via neighbor j. Initialized with direct costs and infinity for non-neighbors.

- **Minimum Cost Array**: A vector (**`min_costsX`**) storing the minimum cost to each node, updated after table changes.

- **Routing Packet**: A `struct rtpkt` containing `sourceid`, `destid`, and `mincost[4]` fields, used to exchange distance vectors.

3.2.2 Code Organization

Each node file implements:

- **Initialization Function (`rtinitX`)**: Sets up the distance table with direct costs (e.g., `wts0 = {0, 1, 3, 7}` for node 0) and sends the initial distance vector to neighbors.
- **Update Function (`rtupdateX`)**: Updates the distance table based on received packets, recalculates minimum costs, and sends updates if changes occur.
- **Helper Functions**: `make_distance_vectorX` computes the minimum costs, and `send_packetX` sends the distance vector to neighbors.

3.2.3 Key Logic

- **Initialization**: For node 0, `dt0.costs[1][1] = 1`, `dt0.costs[2][2] = 3`, `dt0.costs[3][3] = 7`, with others set to 999.
- **Update**: For a packet from node s, `dtX.costs[i][s] = wtsX[s] + rcvdpkt->mincost[i]`, and the minimum cost is recomputed. If it changes, neighbors are notified.
- **Error Handling**: The code uses `INF (999)` for unreachable nodes and includes tracing via `TRACE`.

3.2.4 Refinements

- Removed global `src_node` to avoid duplicate symbol errors, hardcoding node IDs in `send_packetX`.
- Used local `src_node` in `rtupdateX` for the packet source ID.
- Made a global `make_distance_vector()` function for each of the node modules.
- Created a `send_packetX()` for each of the modules to send the packets to only the neighbours.

Listing 3.1: Example: rtinit0 in `node0.c`

```c
void rtinit0() {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            if (i == j) dt0.costs[i][j] = wts0[i];
            else dt0.costs[i][j] = INF;
        }
    }
    if (TRACE > 0) printdt0(&dt0);
    make_distance_vector0();
    send_packet0();
}
```

Listing 3.2: **make_distance_vector0** and **send_packet0()** in **node0.c**

```c
int min_macro0(int a, int b)
{
    return (a < b) ? a : b;
}

int min_cost0(int* a)
{
    return min_macro0(min_macro0(a[0], a[1]), min_macro0(a[2], a[3]));
}

// Send the distance vector to all directly connected neighbors
// min_costs[i] = min_cost(dt0.costs[i]);
// i.e. For destinations marked on columns, minimum cost to reach them
// from node 0 is the minimum of the costs to reach them via all other nodes
int min_costs0[4];
void make_distance_vector0()
{
    for (int i = 0; i < 4; i++)
        min_costs0[i] = min_cost0(dt0.costs[i]);
}

void send_packet0()
{
    struct rtpkt packet;
    packet.sourceid = 0;
    for (int j = 0; j < 4; j++)
        packet.mincost[j] = min_costs0[j];

    for (int i = 0; i < 4; i++)
    {
        if (i == 0) continue;
        packet.destid = i;
        tolayer2(packet);
    }
}
```

3.3 Role of `distance_vector.c`

The **distance_vector.c** file serves as the network emulator and main routine, which students should not modify. Its key functions include:

- **Initialization (init)**: Calls **rtinitX** for all nodes and sets up the event list for link changes (disabled with **LINKCHANGES=0**).
- **Event Handling (main)**: Processes events (e.g., packet arrivals from **FROM_LAYER2**) by invoking **rtupdateX** and handles link changes via **linkhandlerX**.
- **Packet Transmission (tolayer2)**: Simulates packet delivery to neighbors with variable delay (1-10 time units), checking connectivity via a **connectcosts** matrix.
- **Utilities**: Provides **jimsrand()** for random delays and **insertevent()** for event scheduling.

Listing 3.3: **rtupdate0** in **node0.c**

```c
void rtupdate0(struct rtpkt *rcvdpkt)
{
    int src = rcvdpkt->sourceid;

    int changed = 0;
    int old_min_costs[4];
    for (int i = 0; i < 4; i++)
        old_min_costs[i] = min_costs0[i];

    // Update the distance table for node 0
    // i.e. if node 0's own minimum cost to another node changes as a result of the update
    // node 0 informs its directly connected neighbors of this change in minimum cost
    // hence node 0's own minimum cost to another node src_node = dt0.costs[src_node][src_node] + rcvdpkt->mincost[i]

    for (int i = 0; i < 4; i++)
    {
        int new_cost = dt0.costs[src][src] + rcvdpkt->mincost[i];
        if (new_cost < INF)
            dt0.costs[i][src] = new_cost;
        else
            dt0.costs[i][src] = INF;
    }

    if (TRACE > 0) printdt0(&dt0);

    make_distance_vector0();
    for (int i = 0; i < 4; i++)
    {
        if (min_costs0[i] != old_min_costs[i])
        {
            changed = 1;
            break;
        }
    }

    if (changed)
        send_packet0();
}
```

```
MAIN: rcv event, t=11.022, at 2 src: 0, dest: 2, contents:   0   1   2   7
                via
   D2 |    0     1    3
  ----|-----------------
     0|    3     2     7
dest 1|    4     1     5
     3|   10     9     2
MAIN: rcv event, t=12.792, at 2 src: 1, dest: 2, contents:   1   0   1   3
                via
   D2 |    0     1    3
  ----|-----------------
     0|    3     2     7
dest 1|    4     1     5
     3|   10     4     2
MAIN: rcv event, t=13.977, at 2 src: 0, dest: 2, contents:   0   1   2   5
                via
   D2 |    0     1    3
  ----|-----------------
     0|    3     2     7
dest 1|    4     1     5
     3|    8     4     2
MAIN: rcv event, t=15.855, at 2 src: 3, dest: 2, contents:   4   3   2   0
                via
   D2 |    0     1    3
  ----|-----------------
     0|    3     2     6
dest 1|    4     1     5
     3|    8     4     2
MAIN: rcv event, t=17.443, at 2 src: 0, dest: 2, contents:   0   1   2   4
                via
   D2 |    0     1    3
  ----|-----------------
     0|    3     2     6
dest 1|    4     1     5
     3|    7     4     2

Simulator terminated at t=17.443218, no packets in medium
(base)  guntas13  JetBrains Projects/routing-nat-loop  master
```

Figure 3.1: Running the routing algorithm.