LAB 2: Mininet

Network Architecture

Universitat Pompeu Fabra

Martí Esquius Arnau - 267444

Yihan Jin - 253297

Zihao Zhou - 285879

2024-2025

Universitat Pompeu Fabra Barcelona

Table of Contents

1. Introduction

This lab session provides practical experience with Mininet, a powerful network emulation tool that allows us to create and test virtual network topologies. Building on our previous work, we'll explore two key aspects of network configuration and analysis:

Part 1.
We'll begin by enhancing our Mininet script to enable direct command-line interaction with virtual hosts. This will allow us to:

- Test connectivity between nodes using various ping commands
- Measure network performance with iperf bandwidth tests
- Set up and access web servers across hosts
- Experiment with different network conditions (bandwidth limits, latency, packet loss)

Part 2.
Using MiniEdit's graphical interface, we'll:

- Design a dual-LAN topology connected via routers
- Configure IP addressing and subnet masks
- Implement static routing between networks
- Verify connectivity using traceroute

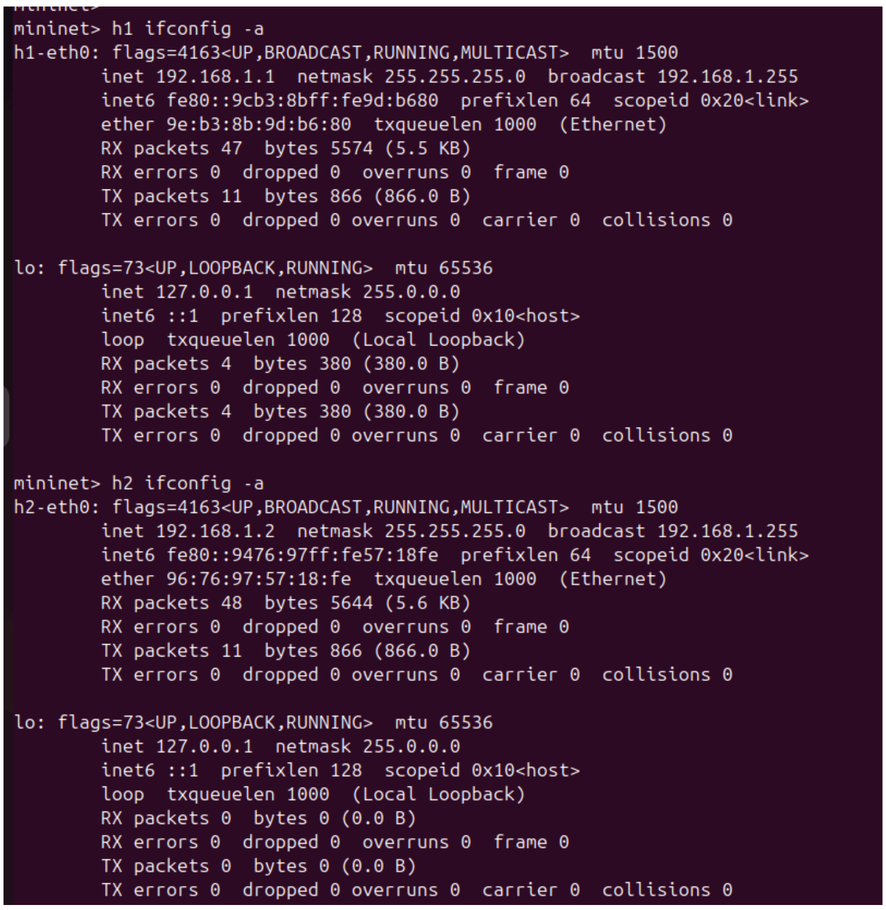
Through these exercises, we'll gain practical insights into fundamental networking concepts while developing skills in network emulation and troubleshooting. The structured approach moves from basic connectivity tests to more complex routing scenarios, mirroring real-world network administration tasks.

2. Part 1: Mininet CLI and Network Tools

In Part 1, we will extend the Mininet script to open a CLI session, run commands like *pingall* and *iperf*, verify connectivity, and capture traffic with *tcpdump* to check the OpenVPN handshake and virtual IP assignment. We will also run tests using *wget* and analyze bandwidth, latency, jitter, and packet loss using a provided script.

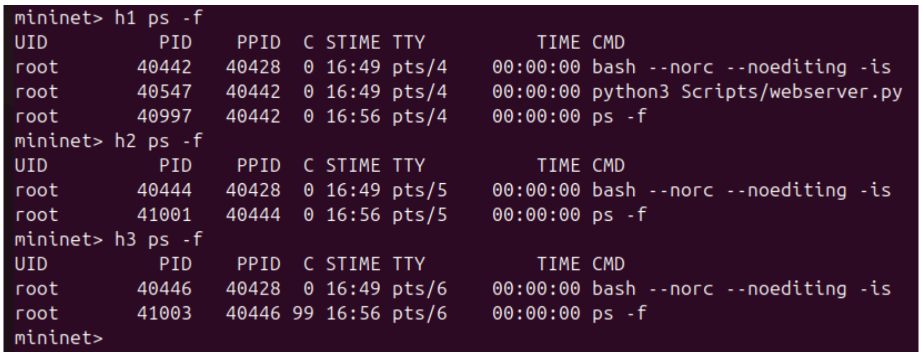
2.1. Verify the running processes

After executing the script, we executed the command *ifconfig -a* in order to test the execution on other running hosts by just using the host name (e.g., h1, h2):



```
mininet> h1 ifconfig -a
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.1  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::9cb3:8bff:fe9d:b680  prefixlen 64  scopeid 0x20<link>
        ether 9e:b3:8b:9d:b6:80  txqueuelen 1000  (Ethernet)
        RX packets 47  bytes 5574 (5.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 11  bytes 866 (866.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 4  bytes 380 (380.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4  bytes 380 (380.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

mininet> h2 ifconfig -a
h2-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.2  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::9476:97ff:fe57:18fe  prefixlen 64  scopeid 0x20<link>
        ether 96:76:97:57:18:fe  txqueuelen 1000  (Ethernet)
        RX packets 48  bytes 5644 (5.6 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 11  bytes 866 (866.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

We have also verified the running processes in each host with *ps -f* :

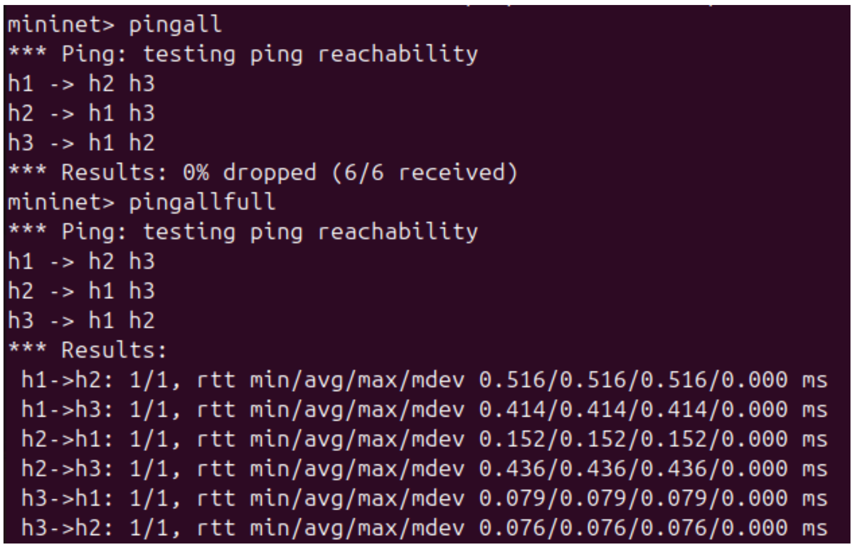


```
mininet> h1 ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
root       40442   40428  0 16:49 pts/4    00:00:00 bash --norc --noediting -is
root       40547   40442  0 16:49 pts/4    00:00:00 python3 Scripts/webserver.py
root       40997   40442  0 16:56 pts/4    00:00:00 ps -f
mininet> h2 ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
root       40444   40428  0 16:49 pts/5    00:00:00 bash --norc --noediting -is
root       41001   40444  0 16:56 pts/5    00:00:00 ps -f
mininet> h3 ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
root       40446   40428  0 16:49 pts/6    00:00:00 bash --norc --noediting -is
root       41003   40446 99 16:56 pts/6    00:00:00 ps -f
mininet>
```

2.2. Commands ‘pingall’, ‘pingallfull’, ‘pingpair’ and ‘pingpairfull’

- *pingall* is a command that sends a ping from every host to every other host. It is often used to test the connectivity between all the hosts.
- *pingallfull* is a command that also sends a ping from every host to every other host but in addition, it offers more details about the pings, like the RTT, the number of retries or the average time it took to reach a host from another. It is also used to test the connectivity and its quality between all hosts.

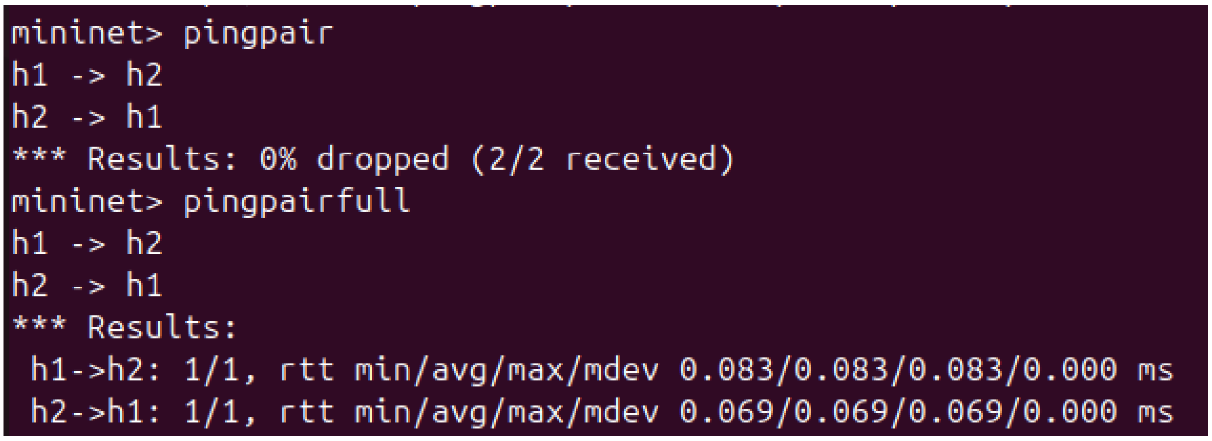
In the following image, we can see how this commands behave in our setup:



```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
mininet> pingallfull
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results:
 h1->h2: 1/1, rtt min/avg/max/mdev 0.516/0.516/0.516/0.000 ms
 h1->h3: 1/1, rtt min/avg/max/mdev 0.414/0.414/0.414/0.000 ms
 h2->h1: 1/1, rtt min/avg/max/mdev 0.152/0.152/0.152/0.000 ms
 h2->h3: 1/1, rtt min/avg/max/mdev 0.436/0.436/0.436/0.000 ms
 h3->h1: 1/1, rtt min/avg/max/mdev 0.079/0.079/0.079/0.000 ms
 h3->h2: 1/1, rtt min/avg/max/mdev 0.076/0.076/0.076/0.000 ms
```

- *pingpair* is a command that performs a ping between two hosts. It allows us to choose the sender and the receiver and pass with it. By default,without arguments, Mininet will choose two hosts to realize the ping. It is often used to test the connectivity between two hosts.
- *pingpairfull* is a command that also performs a ping between two hosts but in addition, it offers more details about the pings, like the average RTT the number of retries. It is also used to test the connectivity and its quality between two hosts.

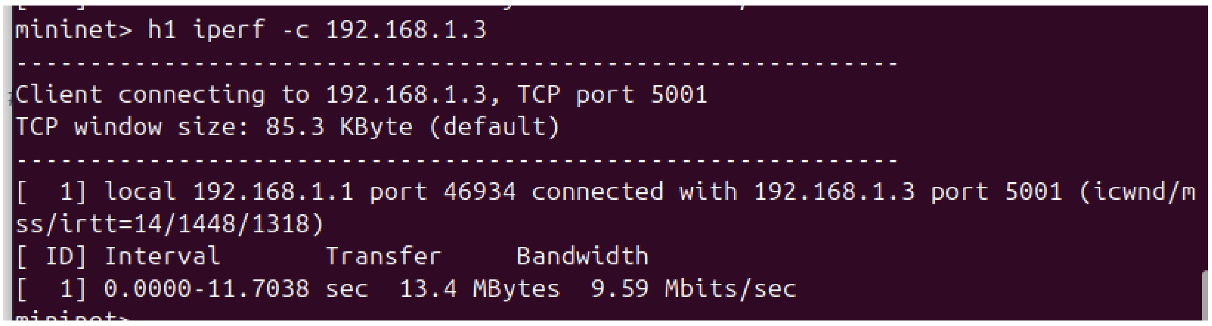
In the following image, we can see how this commands behave:



```
mininet> pingpair
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
mininet> pingpairfull
h1 -> h2
h2 -> h1
*** Results:
 h1->h2: 1/1, rtt min/avg/max/mdev 0.083/0.083/0.083/0.000 ms
 h2->h1: 1/1, rtt min/avg/max/mdev 0.069/0.069/0.069/0.000 ms
```

2.3. The “iperf” command

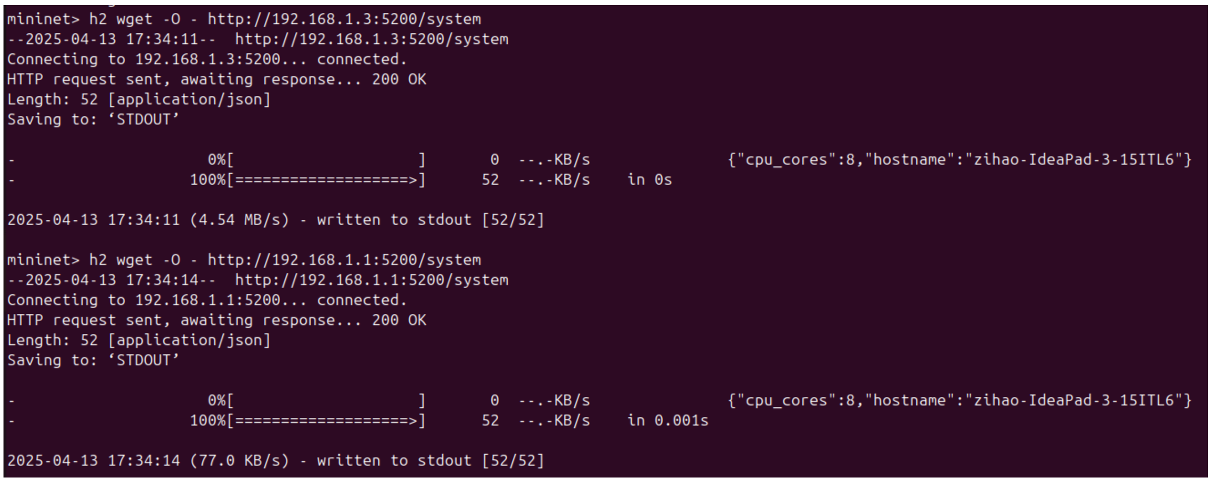
The iperf is used to measure the bandwidth of the TCP or UDP connection between two hosts. To make it work, it requires a server and a client. It operates at the Transport Layer (Layer 4) of the OSI model.
In Mininet, we will run the command *h1 iperf -s* in the server host h1 and then run the following command for the client *h2 iperf -c <server_IP>*, as we can see in the following image:



```
mininet> h1 iperf -c 192.168.1.3
------------------------------------------------------------
Client connecting to 192.168.1.3, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 192.168.1.1 port 46934 connected with 192.168.1.3 port 5001 (icwnd/m
ss/irtt=14/1448/1318)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-11.7038 sec  13.4 MBytes  9.59 Mbits/sec
```

2.4. Start Flask HTTP server

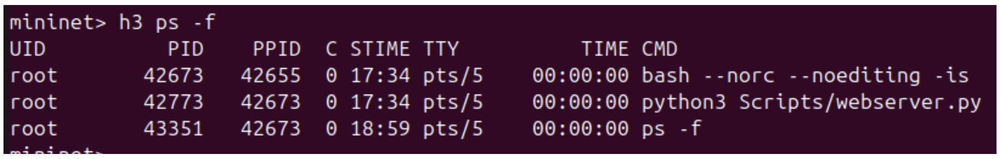
After starting the Flask HTTP server on h3, we noticed that now we have the same HTTP server running in h1 and h3. We realised that after running *wget test* from h2, running a http request on h1 and h3 ips in the port 5200 and the endpoint /system. In both cases the server status was code 200 OK and the response itself had length 52.



```
mininet> h2 wget -O - http://192.168.1.3:5200/system
--2025-04-13 17:34:11--  http://192.168.1.3:5200/system
Connecting to 192.168.1.3:5200... connected.
HTTP request sent, awaiting response... 200 OK
Length: 52 [application/json]
Saving to: ‘STDOUT’

-                     0%[                    ]       0  --.-KB/s               {"cpu_cores":8,"hostname":"zihao-IdeaPad-3-15ITL6"}
-                   100%[===================>]      52  --.-KB/s    in 0s

2025-04-13 17:34:11 (4.54 MB/s) - written to stdout [52/52]

mininet> h2 wget -O - http://192.168.1.1:5200/system
--2025-04-13 17:34:14--  http://192.168.1.1:5200/system
Connecting to 192.168.1.1:5200... connected.
HTTP request sent, awaiting response... 200 OK
Length: 52 [application/json]
Saving to: ‘STDOUT’

-                     0%[                    ]       0  --.-KB/s               {"cpu_cores":8,"hostname":"zihao-IdeaPad-3-15ITL6"}
-                   100%[===================>]      52  --.-KB/s    in 0.001s

2025-04-13 17:34:14 (77.0 KB/s) - written to stdout [52/52]
```

2.5. Running processes in h3

We have seen that, after starting the HTTP Flask server on h3 a new process with PID 42773 has appeared. This is the process that runs the web server.



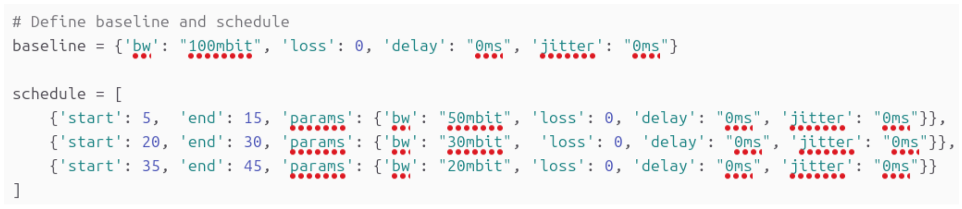
```
mininet> h3 ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
root       42673   42655  0 17:34 pts/5    00:00:00 bash --norc --noediting -is
root       42773   42673  0 17:34 pts/5    00:00:00 python3 Scripts/webserver.py
root       43351   42673  0 18:59 pts/5    00:00:00 ps -f
```

After thinking about it, since a web server must be able to serve requests from other hosts it must be always working, so it's a perpetual process that we don’t expect to end (if everything works nice).

2.6. Bandwidth, Jitter, and Packet Loss Emulation

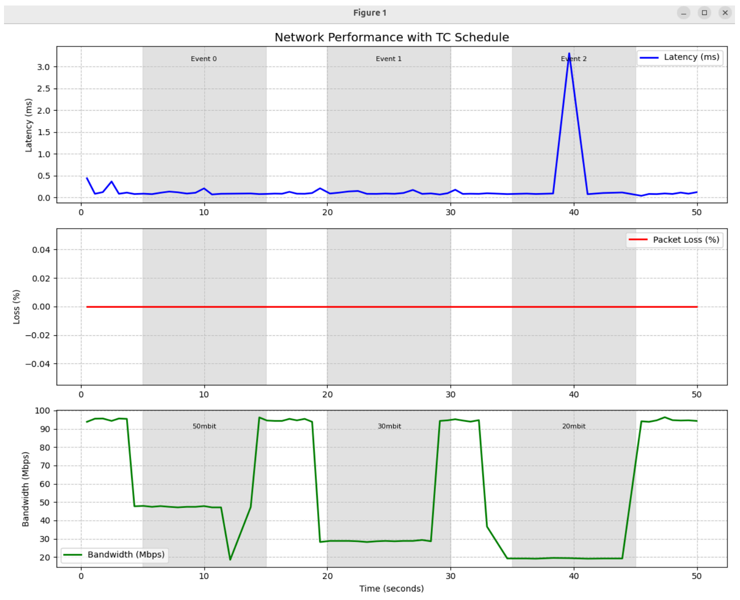
2.6.1. Bandwidth variation

Changing the value of the bandwidth to values of 50, 30 and 20 Mbits in the `metrics.py` script as we can see in the following figure:



```
# Define baseline and schedule
baseline = {'bw': "100mbit", 'loss': 0, 'delay': "0ms", 'jitter': "0ms"}

schedule = [
    {'start': 5,  'end': 15, 'params': {'bw': "50mbit", 'loss': 0, 'delay': "0ms", 'jitter': "0ms"}},
    {'start': 20, 'end': 30, 'params': {'bw': "30mbit",  'loss': 0, 'delay': "0ms", 'jitter': "0ms"}},
    {'start': 35, 'end': 45, 'params': {'bw': "20mbit", 'loss': 0, 'delay': "0ms", 'jitter': "0ms"}}
]
```

The plots we obtained with this new configuration are:



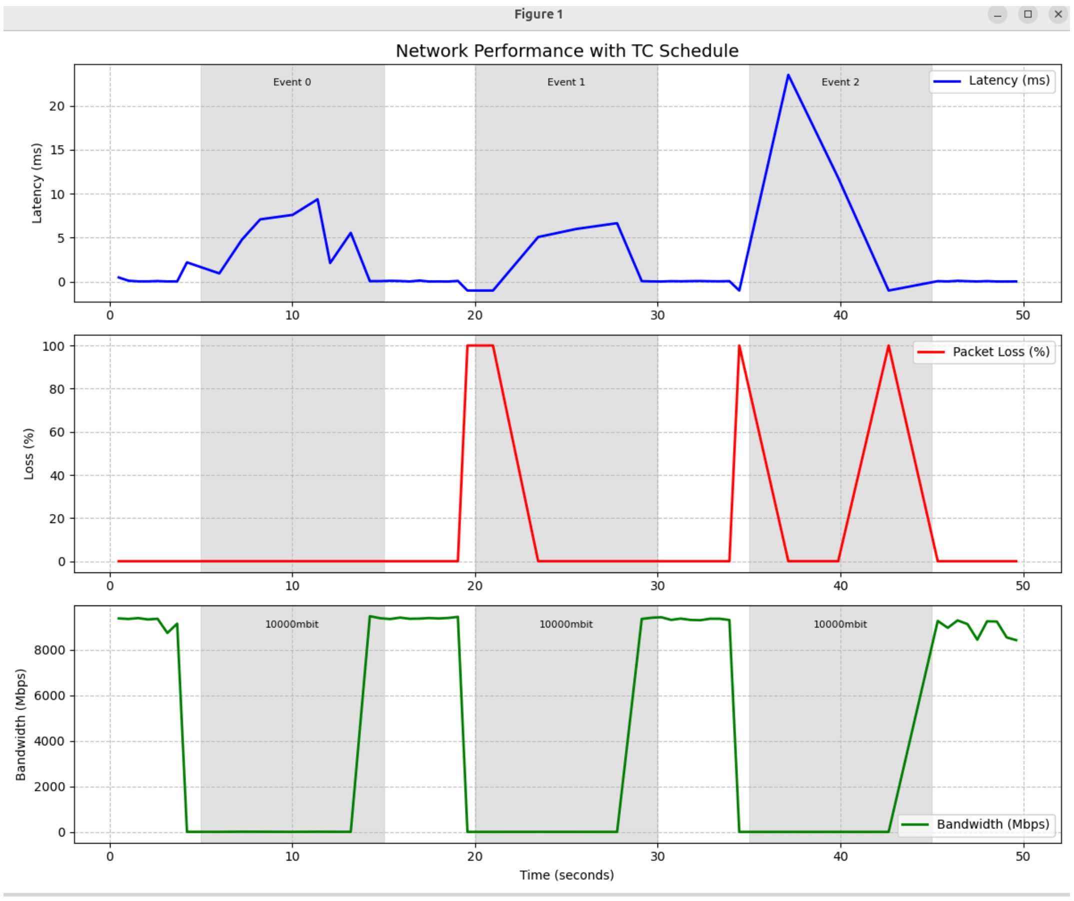
For the Latency, as we expected lower bandwidth can produce higher peaks as we have seen in the first plot. Loss packets represent a 0% of the total in all the configurations. For the Bandwidth measurements, it closely matched the configured limits, confirming accuracy.

2.6.2. Jitter effect in Bandwidth

To test the effectiveness of Jitter over the final bandwidth, we set up a small delay of 3, 6 and 10ms. With a respective jitter (or standard deviation of the delay) of the same value respectively. We did the following modifications on the `metrics.py` script:

```
# Define baseline and schedule
baseline = {'bw': "10000mbit", 'loss': 0, 'delay': "0ms", 'jitter': "0ms"}

schedule = [
    {'start': 5,  'end': 15, 'params': {'bw': "10000mbit", 'loss': 5, 'delay': "3ms", 'jitter': "3ms"}},
    {'start': 20, 'end': 30, 'params': {'bw': "10000mbit",  'loss': 10, 'delay': "6ms", 'jitter': "6ms"}},
    {'start': 35, 'end': 45, 'params': {'bw': "10000mbit", 'loss': 20, 'delay': "10ms", 'jitter': "10ms"}}
]
```

And obtained the following results:



A higher value for Jitter increased packet delay variation, causing some packets to arrive out of order, which may trigger retransmissions and produce a higher Loss %. In our case, we observed the maximum and minimum delay during Event 2, due to the fact that the higher value of delay was followed by a higher value of the Jitter (or standard deviation of delay).

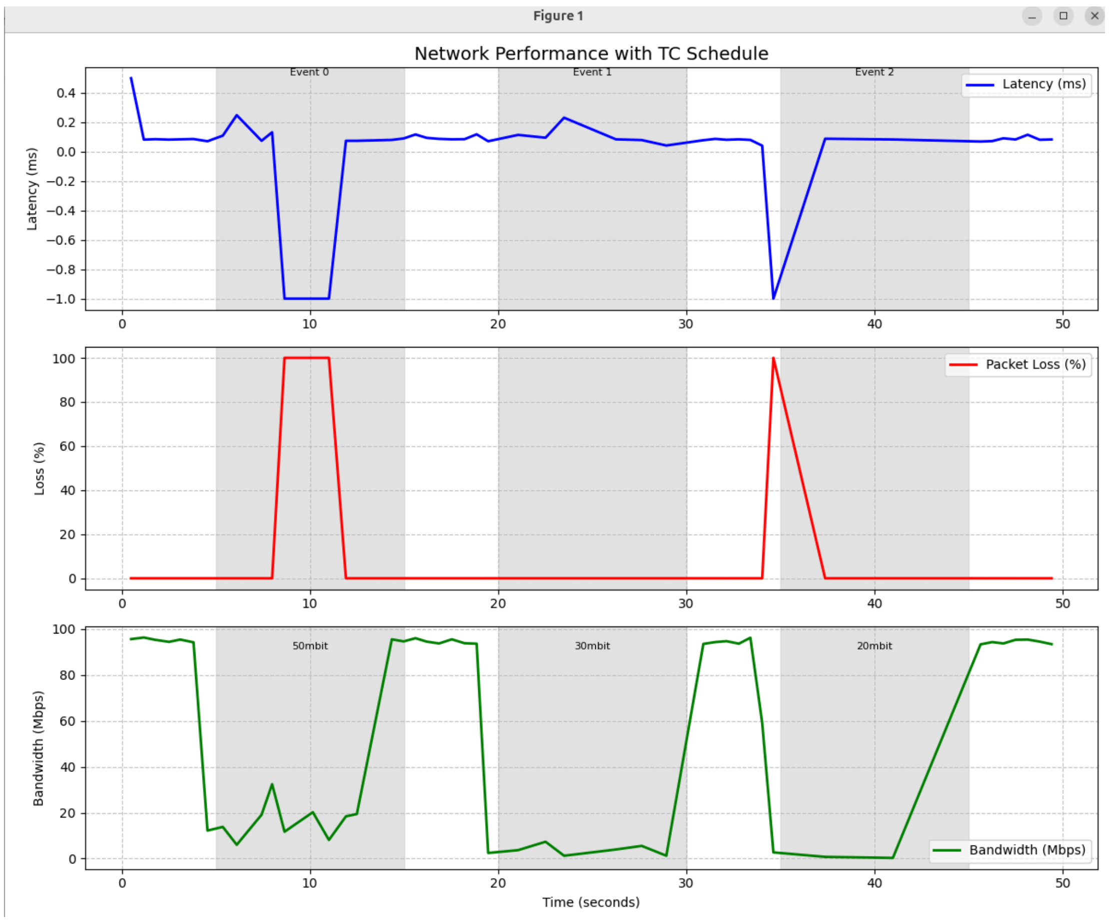
Bandwidth did not have a significant variation, but we noticed that Jitter reduces a little bit of effective bandwidth by causing packet delays and variability.

2.6.3. Packet Loss effect in Bandwidth

To test the effectiveness of packet loss over the final bandwidth, we tested a packet loss probability of 5, 10 and 20%. The following modifications were made on the `metrics.py` script:

```
# Define baseline and schedule
baseline = {'bw': "100mbit", 'loss': 0, 'delay': "0ms", 'jitter': "0ms"}

schedule = [
    {'start': 5,  'end': 15, 'params': {'bw': "50mbit", 'loss': 5, 'delay': "0ms", 'jitter': "0ms"}},
    {'start': 20, 'end': 30, 'params': {'bw': "30mbit",  'loss': 10, 'delay': "0ms", 'jitter': "0ms"}},
    {'start': 35, 'end': 45, 'params': {'bw': "20mbit", 'loss': 20, 'delay': "0ms", 'jitter': "0ms"}}
]
```

We obtained the following results:



We have seen that the Loss % measured does not exactly match the one we input in the script. Regarding the performance in terms of Latency, higher loss rates did produce slightly high waiting times.

Again, here we also got -1ms values that correspond to 100% Loss discrete points in time, we will discuss this behaviour in the following section. Regarding Bandwidth, lower error rates produced an effective higher bandwidth.

2.6.4. Loss effect discussion

Observations made in the past exercise evidenced an error during the measurements. Sending only one packet per measurement round (with -c 1) causes the loss measurement to be binary

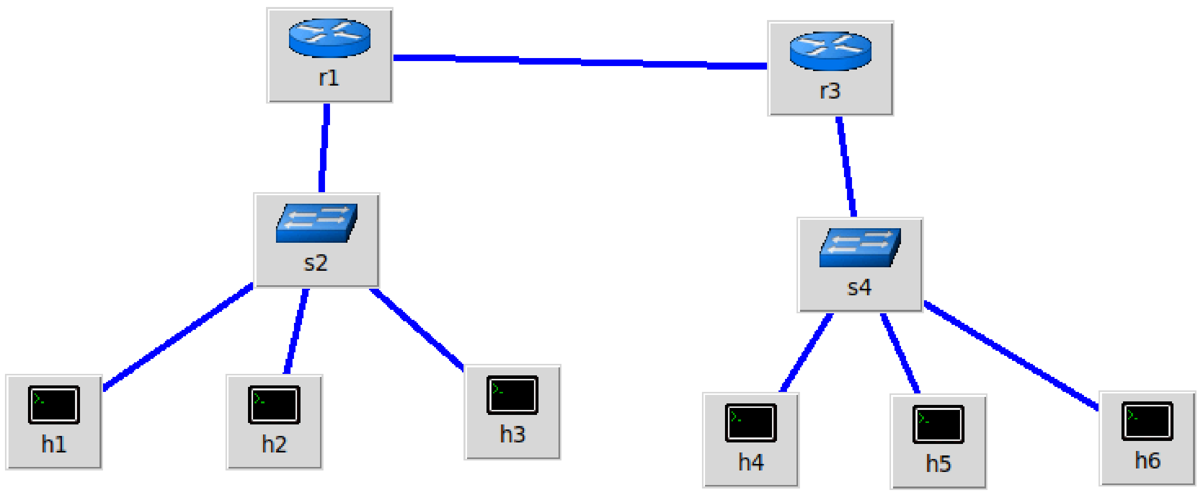
```python
# Run ping test more frequently
if current_timestamp - last_ping_time >= ping_interval:
    with cmd_lock:
        # Use faster ping command (-c 1 = single packet, -W 0.1 = 0.1s timeout)
        ping_output = host1.cmd(f"ping -c 1 -W 0.1 {host2_ip}")
        latency, loss = parse_ping_output(ping_output)
```

The observation that loss values are either 0% or very high in each interval stems from using a single packet ping test with a short timeout, which does not capture partial loss rates. In contrast, iperf's averaging over many packets shows a smoother transition in bandwidth values. The discrepancy is not because the loss effect isn't applied correctly—instead, it's a measurement artifact due to the ping command's configuration.

3. Part 2: Miniedit Network Setup and Static Routing

3.1. Creation of a Network Topology

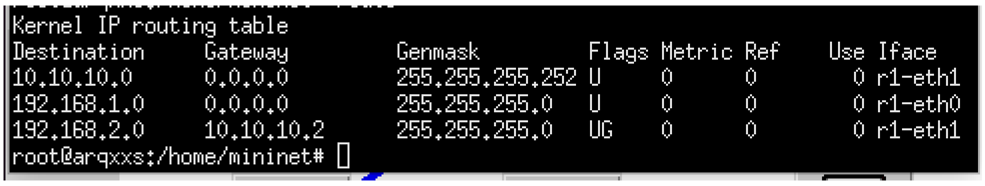
The final network topology is as we can see in the following figure:



3.2. Save and Run the Designed Network (Q3&Q4)

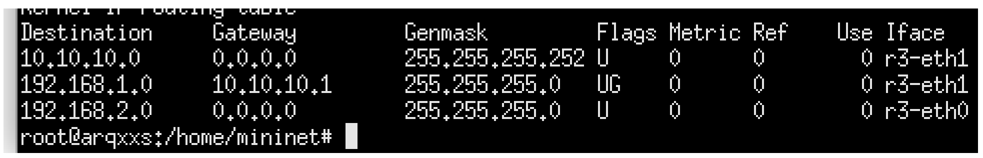
We can use the mouse to right-click the host and choose the properties to define its IP and gateway. After that we run the program and open the terminal window of r1(router 1) and r3(router 3) We use ifconfig r1-eth0 192.168.1.1/24 to define the IP of router 1, and ifconfig r3-eth0 192.168.2.1/24. To build p2p, we need to assign IP of p2p for each router, ifconfig r1-eth1 10.10.10.1/30 and ifconfig r3-eth1 10.10.10.2/30. We use the command ip route add 192.168.2.0/24 via 10.10.10.2 in r1 and ip route add 192.168.1.0/24 via 10.10.10.1 to make sure of the connection.

R1 route table:



```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.10.10.0      0.0.0.0         255.255.255.252 U     0      0        0 r1-eth1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 r1-eth0
192.168.2.0     10.10.10.2      255.255.255.0   UG    0      0        0 r1-eth1
root@arqxxs:/home/mininet# 
```

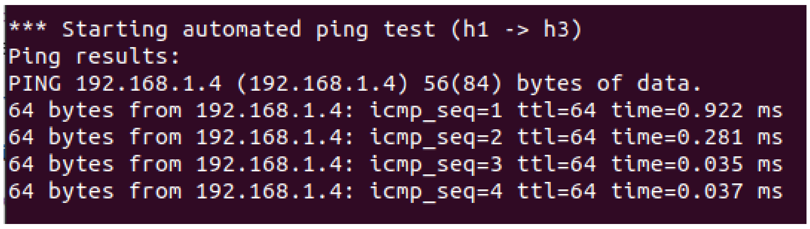
R3 route table:



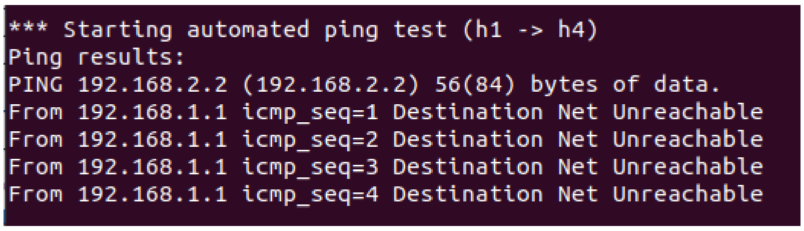
```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.10.10.0      0.0.0.0         255.255.255.252 U     0      0        0 r3-eth1
192.168.1.0     10.10.10.1      255.255.255.0   UG    0      0        0 r3-eth1
192.168.2.0     0.0.0.0         255.255.255.0   U     0      0        0 r3-eth0
root@arqxxs:/home/mininet# 
```

3.3. Router CLI network tests (Q5)

After modifying the script to add a ping test before calling the CLI, we observed that the test did succeed inside the same switch, but it was unreachable if the nodes were disposed under different routers.

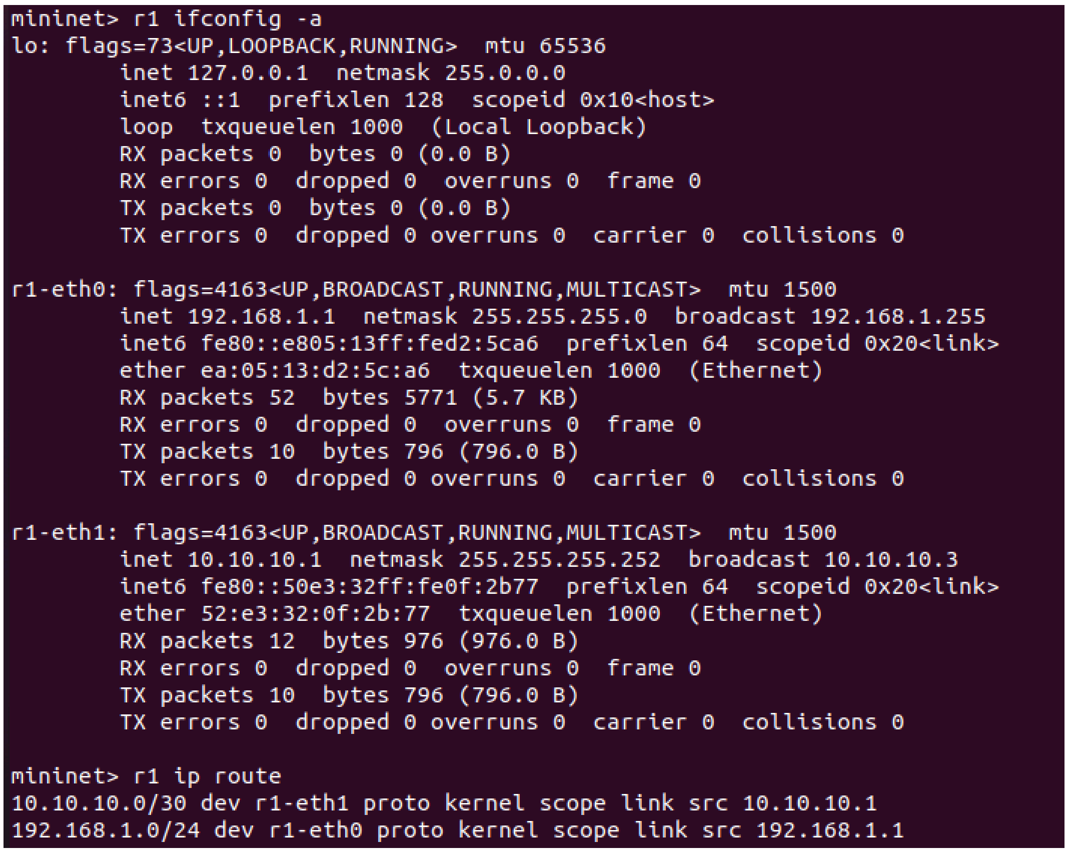


```
*** Starting automated ping test (h1 -> h3)
Ping results:
PING 192.168.1.4 (192.168.1.4) 56(84) bytes of data.
64 bytes from 192.168.1.4: icmp_seq=1 ttl=64 time=0.922 ms
64 bytes from 192.168.1.4: icmp_seq=2 ttl=64 time=0.281 ms
64 bytes from 192.168.1.4: icmp_seq=3 ttl=64 time=0.035 ms
64 bytes from 192.168.1.4: icmp_seq=4 ttl=64 time=0.037 ms
```



```
*** Starting automated ping test (h1 -> h4)
Ping results:
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
From 192.168.1.1 icmp_seq=1 Destination Net Unreachable
From 192.168.1.1 icmp_seq=2 Destination Net Unreachable
From 192.168.1.1 icmp_seq=3 Destination Net Unreachable
From 192.168.1.1 icmp_seq=4 Destination Net Unreachable
```

Now from the CLI we run the following commands in the router node:

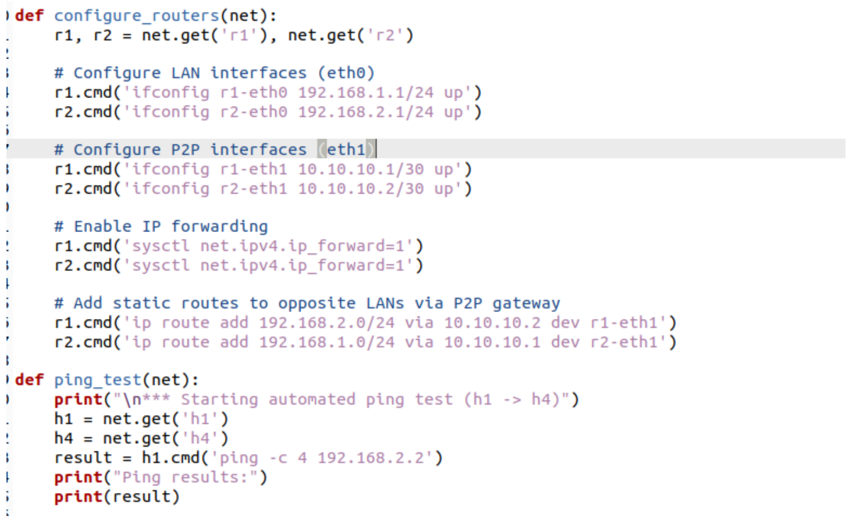


```
mininet> r1 ifconfig -a
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

r1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.1  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::e805:13ff:fed2:5ca6  prefixlen 64  scopeid 0x20<link>
        ether ea:05:13:d2:5c:a6  txqueuelen 1000  (Ethernet)
        RX packets 52  bytes 5771 (5.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 10  bytes 796 (796.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

r1-eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.10.10.1  netmask 255.255.255.252  broadcast 10.10.10.3
        inet6 fe80::50e3:32ff:fe0f:2b77  prefixlen 64  scopeid 0x20<link>
        ether 52:e3:32:0f:2b:77  txqueuelen 1000  (Ethernet)
        RX packets 12  bytes 976 (976.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 10  bytes 796 (796.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

mininet> r1 ip route
10.10.10.0/30 dev r1-eth1 proto kernel scope link src 10.10.10.1
192.168.1.0/24 dev r1-eth0 proto kernel scope link src 192.168.1.1
```

From the results obtained using *ip route*, we know that we have configured two routes for our router.

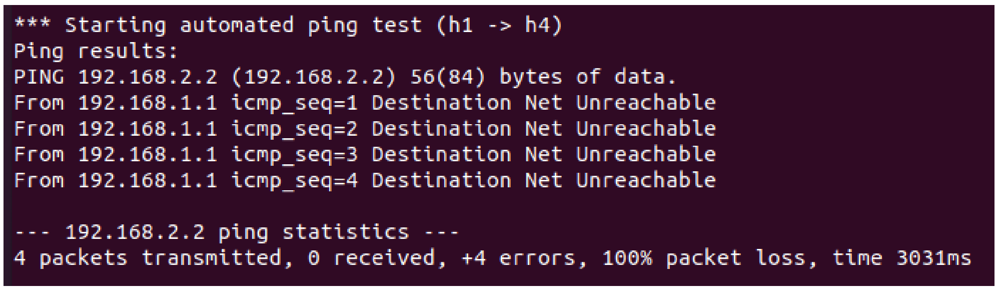
3.4. Network Interface Configuration and Routing Verification (Q6)

After running *ifconfig* the final commands present in the python script in order to configure the router nodes are:



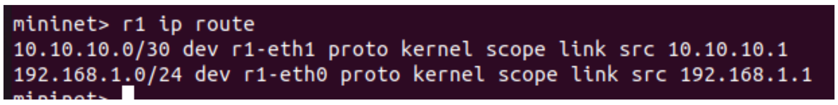
```python
def configure_routers(net):
    r1, r2 = net.get('r1'), net.get('r2')

    # Configure LAN interfaces (eth0)
    r1.cmd('ifconfig r1-eth0 192.168.1.1/24 up')
    r2.cmd('ifconfig r2-eth0 192.168.2.1/24 up')

    # Configure P2P interfaces (eth1)
    r1.cmd('ifconfig r1-eth1 10.10.10.1/30 up')
    r2.cmd('ifconfig r2-eth1 10.10.10.2/30 up')

    # Enable IP forwarding
    r1.cmd('sysctl net.ipv4.ip_forward=1')
    r2.cmd('sysctl net.ipv4.ip_forward=1')

    # Add static routes to opposite LANs via P2P gateway
    r1.cmd('ip route add 192.168.2.0/24 via 10.10.10.2 dev r1-eth1')
    r2.cmd('ip route add 192.168.1.0/24 via 10.10.10.1 dev r2-eth1')

def ping_test(net):
    print("\n*** Starting automated ping test (h1 -> h4)")
    h1 = net.get('h1')
    h4 = net.get('h4')
    result = h1.cmd('ping -c 4 192.168.2.2')
    print("Ping results:")
    print(result)
```

We first tried running the automated ping test to check the connectivity between server h1 and a client h4:

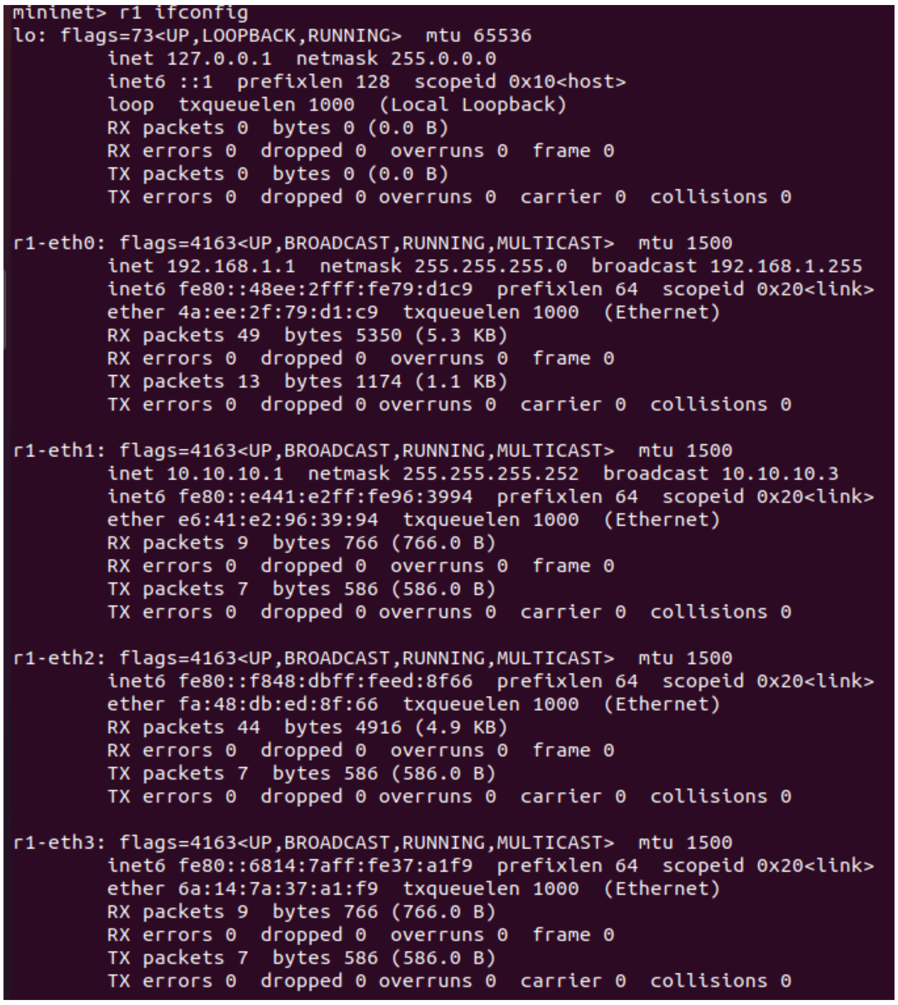


```
*** Starting automated ping test (h1 -> h4)
Ping results:
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
From 192.168.1.1 icmp_seq=1 Destination Net Unreachable
From 192.168.1.1 icmp_seq=2 Destination Net Unreachable
From 192.168.1.1 icmp_seq=3 Destination Net Unreachable
From 192.168.1.1 icmp_seq=4 Destination Net Unreachable

--- 192.168.2.2 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3031ms
```

As we can see, all packets were lost. Now we ran the commands *ifconfig* and *ip route* and obtained, ifconfig shows more detailed information about routers, we can simply use “r1 ip route” to check the route path it has.



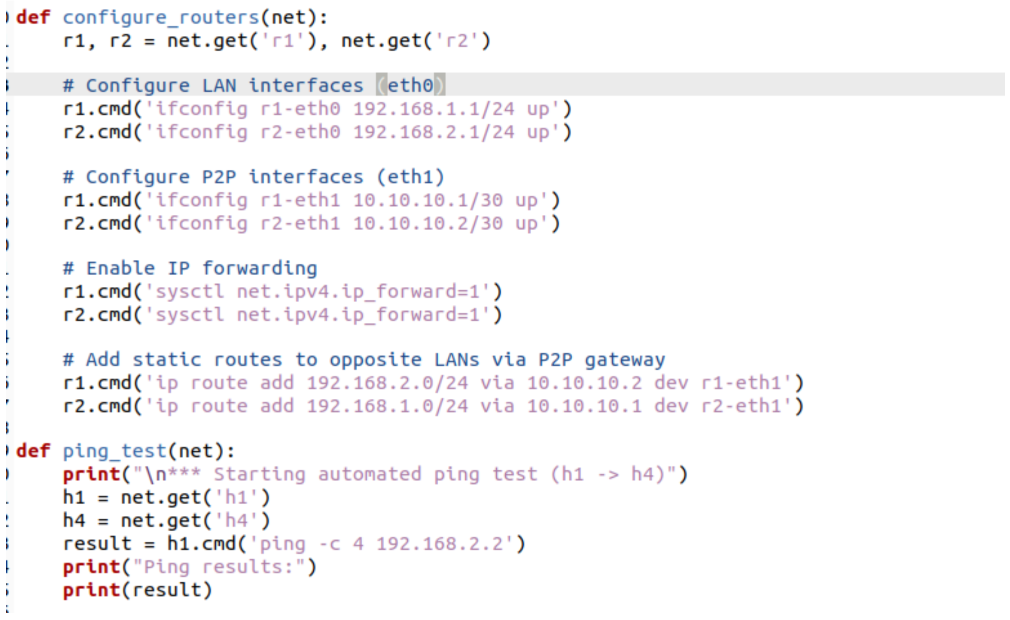
```
mininet> r1 ip route
10.10.10.0/30 dev r1-eth1 proto kernel scope link src 10.10.10.1
192.168.1.0/24 dev r1-eth0 proto kernel scope link src 192.168.1.1
```



```
mininet> r1 ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

r1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.1  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::48ee:2fff:fe79:d1c9  prefixlen 64  scopeid 0x20<link>
        ether 4a:ee:2f:79:d1:c9  txqueuelen 1000  (Ethernet)
        RX packets 49  bytes 5350 (5.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 13  bytes 1174 (1.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

r1-eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.10.10.1  netmask 255.255.255.252  broadcast 10.10.10.3
        inet6 fe80::e441:e2ff:fe96:3994  prefixlen 64  scopeid 0x20<link>
        ether e6:41:e2:96:39:94  txqueuelen 1000  (Ethernet)
        RX packets 9  bytes 766 (766.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 7  bytes 586 (586.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

r1-eth2: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::f848:dbff:feed:8f66  prefixlen 64  scopeid 0x20<link>
        ether fa:48:db:ed:8f:66  txqueuelen 1000  (Ethernet)
        RX packets 44  bytes 4916 (4.9 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 7  bytes 586 (586.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

r1-eth3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::6814:7aff:fe37:a1f9  prefixlen 64  scopeid 0x20<link>
        ether 6a:14:7a:37:a1:f9  txqueuelen 1000  (Ethernet)
        RX packets 9  bytes 766 (766.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 7  bytes 586 (586.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

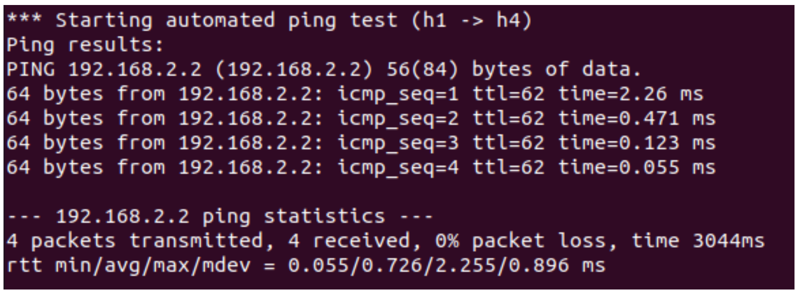
3.5. Static Route Configuration and End-to-End Connectivity Test (Q7)

Building the connectivity and ping is successful, we need to configure the P2P gateway which allows ping between 2 hosts under different routers.

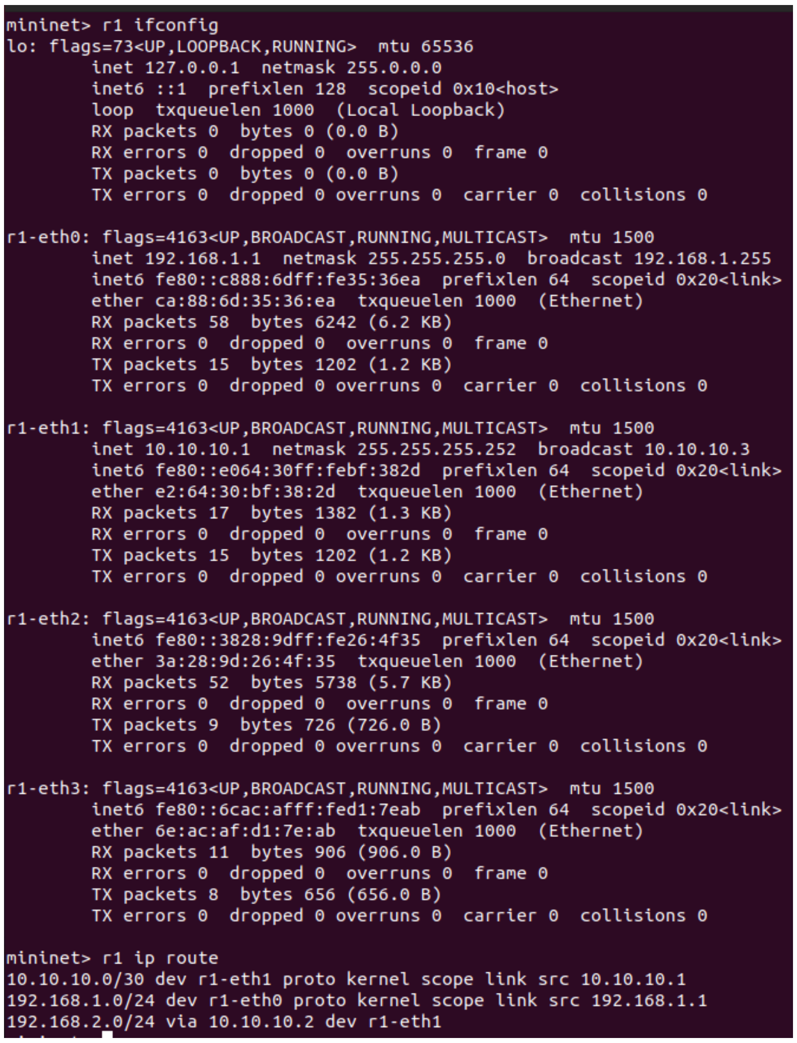


```python
def configure_routers(net):
    r1, r2 = net.get('r1'), net.get('r2')

    # Configure LAN interfaces (eth0)
    r1.cmd('ifconfig r1-eth0 192.168.1.1/24 up')
    r2.cmd('ifconfig r2-eth0 192.168.2.1/24 up')

    # Configure P2P interfaces (eth1)
    r1.cmd('ifconfig r1-eth1 10.10.10.1/30 up')
    r2.cmd('ifconfig r2-eth1 10.10.10.2/30 up')

    # Enable IP forwarding
    r1.cmd('sysctl net.ipv4.ip_forward=1')
    r2.cmd('sysctl net.ipv4.ip_forward=1')

    # Add static routes to opposite LANs via P2P gateway
    r1.cmd('ip route add 192.168.2.0/24 via 10.10.10.2 dev r1-eth1')
    r2.cmd('ip route add 192.168.1.0/24 via 10.10.10.1 dev r2-eth1')

def ping_test(net):
    print("\n*** Starting automated ping test (h1 -> h4)")
    h1 = net.get('h1')
    h4 = net.get('h4')
    result = h1.cmd('ping -c 4 192.168.2.2')
    print("Ping results:")
    print(result)
```

The final result we have seen is:



```
*** Starting automated ping test (h1 -> h4)
Ping results:
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
64 bytes from 192.168.2.2: icmp_seq=1 ttl=62 time=2.26 ms
64 bytes from 192.168.2.2: icmp_seq=2 ttl=62 time=0.471 ms
64 bytes from 192.168.2.2: icmp_seq=3 ttl=62 time=0.123 ms
64 bytes from 192.168.2.2: icmp_seq=4 ttl=62 time=0.055 ms

--- 192.168.2.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3044ms
rtt min/avg/max/mdev = 0.055/0.726/2.255/0.896 ms
```

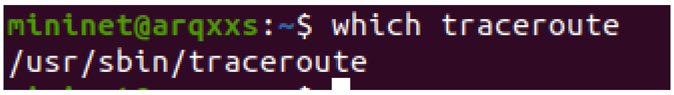


```
mininet> r1 ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

r1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.1  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::c888:6dff:fe35:36ea  prefixlen 64  scopeid 0x20<link>
        ether ca:88:6d:35:36:ea  txqueuelen 1000  (Ethernet)
        RX packets 58  bytes 6242 (6.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 15  bytes 1202 (1.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

r1-eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.10.10.1  netmask 255.255.255.252  broadcast 10.10.10.3
        inet6 fe80::e064:30ff:febf:382d  prefixlen 64  scopeid 0x20<link>
        ether e2:64:30:bf:38:2d  txqueuelen 1000  (Ethernet)
        RX packets 17  bytes 1382 (1.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 15  bytes 1202 (1.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

r1-eth2: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::3828:9dff:fe26:4f35  prefixlen 64  scopeid 0x20<link>
        ether 3a:28:9d:26:4f:35  txqueuelen 1000  (Ethernet)
        RX packets 52  bytes 5738 (5.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 9  bytes 726 (726.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

r1-eth3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::6cac:afff:fed1:7eab  prefixlen 64  scopeid 0x20<link>
        ether 6e:ac:af:d1:7e:ab  txqueuelen 1000  (Ethernet)
        RX packets 11  bytes 906 (906.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8  bytes 656 (656.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

mininet> r1 ip route
10.10.10.0/30 dev r1-eth1 proto kernel scope link src 10.10.10.1
192.168.1.0/24 dev r1-eth0 proto kernel scope link src 192.168.1.1
192.168.2.0/24 via 10.10.10.2 dev r1-eth1
```

Compared to the previous question we can find added line information in the terminal in router 1 by using the command "r1 ip route", which is the static route, which allows the next hop to jump to another router's domain. (r2 in this case).

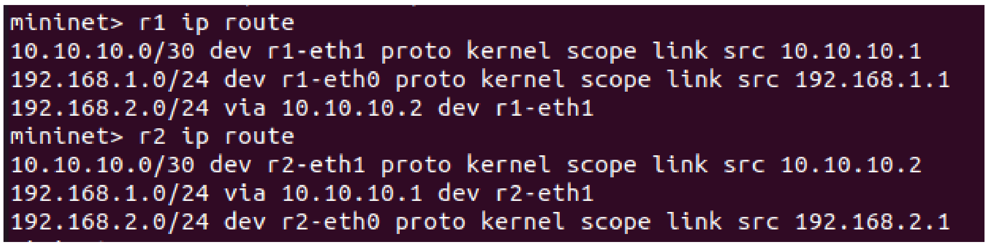
3.6. Trace Routes

First of all, we wanted to check whether the traceroute utility was already installed. To do it we used *which traceroute*



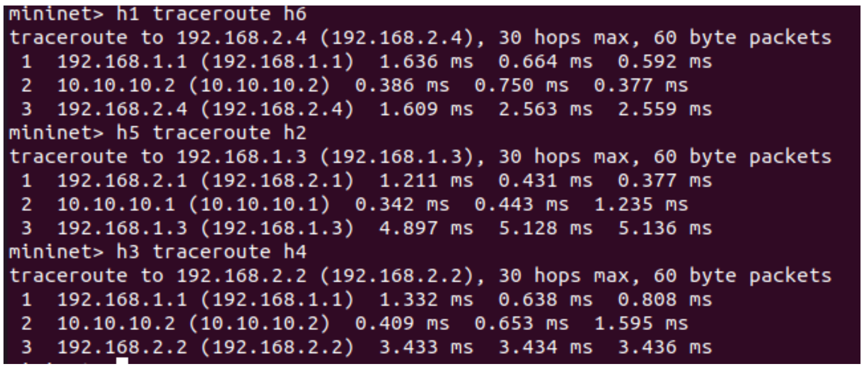
```
mininet@arqxxs:~$ which traceroute
/usr/sbin/traceroute
```

Then we used the *ip route* to check the route table of each router.



```
mininet> r1 ip route
10.10.10.0/30 dev r1-eth1 proto kernel scope link src 10.10.10.1
192.168.1.0/24 dev r1-eth0 proto kernel scope link src 192.168.1.1
192.168.2.0/24 via 10.10.10.2 dev r1-eth1
mininet> r2 ip route
10.10.10.0/30 dev r2-eth1 proto kernel scope link src 10.10.10.2
192.168.1.0/24 via 10.10.10.1 dev r2-eth1
192.168.2.0/24 dev r2-eth0 proto kernel scope link src 192.168.2.1
```

Now, in order to see the path the packets follow in a network, we started from node h1. We will use the node h6 as destination and h3 to h4 (cross-subnet) and From h5 to h2 (reverse path):



```
mininet> h1 traceroute h6
traceroute to 192.168.2.4 (192.168.2.4), 30 hops max, 60 byte packets
 1  192.168.1.1 (192.168.1.1)  1.636 ms  0.664 ms  0.592 ms
 2  10.10.10.2 (10.10.10.2)  0.386 ms  0.750 ms  0.377 ms
 3  192.168.2.4 (192.168.2.4)  1.609 ms  2.563 ms  2.559 ms
mininet> h5 traceroute h2
traceroute to 192.168.1.3 (192.168.1.3), 30 hops max, 60 byte packets
 1  192.168.2.1 (192.168.2.1)  1.211 ms  0.431 ms  0.377 ms
 2  10.10.10.1 (10.10.10.1)  0.342 ms  0.443 ms  1.235 ms
 3  192.168.1.3 (192.168.1.3)  4.897 ms  5.128 ms  5.136 ms
mininet> h3 traceroute h4
traceroute to 192.168.2.2 (192.168.2.2), 30 hops max, 60 byte packets
 1  192.168.1.1 (192.168.1.1)  1.332 ms  0.638 ms  0.808 ms
 2  10.10.10.2 (10.10.10.2)  0.409 ms  0.653 ms  1.595 ms
 3  192.168.2.2 (192.168.2.2)  3.433 ms  3.434 ms  3.436 ms
```

We do traceroute commands from h1 to h6 and h3 to h4 to make sure the static route path works correctly, which allows hop to jump to r2 from r1. Conversely, we do traceroute commands from h5 to h2 to ensure the reverse route path works well.

With this we have successfully configured the routers with static routes manually and individually for each router and are looking forward to doing this dynamically and from a central, software managed controller in the next Lab.

4. Conclusion

This lab session allowed us to continue working with Mininet, extending both the CLI-based functionalities and the graphical tools provided by MiniEdit. We tested basic connectivity, measured network performance under different conditions, and set up simple client-server applications within the emulated network. Additionally, we explored how parameters like bandwidth, delay, jitter, and packet loss affect network behavior.

In the second part, we designed a dual-LAN topology with routers, manually configured interfaces and static routes, and verified connectivity across the entire setup. This process helped consolidate the concepts of IP addressing, routing, and basic network troubleshooting.

Overall, the lab provided useful practice with both the technical tools and the underlying networking concepts, forming a solid base for the next sessions where more advanced and dynamic configurations will be introduced.