LAB 3: SDN

Network Architecture

Universitat Pompeu Fabra

Yihan Jin - 253297

Zihao Zhou - 285879

Martí Esquius - 267444

2024-2025

Table of Contents

1. Introduction

This practical session focused on implementing Software-Defined Networking (SDN) concepts using Mininet and the Ryu controller. The lab will provide valuable hands-on experience in network emulation and SDN configuration, complementing the theoretical foundations covered in lectures.

The primary objectives are to:

- Construct a functional network topology with three Open vSwitches and six hosts
- Configure network addressing and routing programmatically
- Implement full connectivity between all nodes using SDN principles

Through this exercise, we will gain practical insights into SDN operation while developing skills in network troubleshooting and configuration management.

2. Laboratory work

2.1. Initial Setup

First of all, we defined the network topology and installed curl with the command *sudo apt install curl*. After running this command, we could check if curl was installed correctly using the command *which curl*.

2.2. Linear topology with 3 OVS switches and 6 hosts

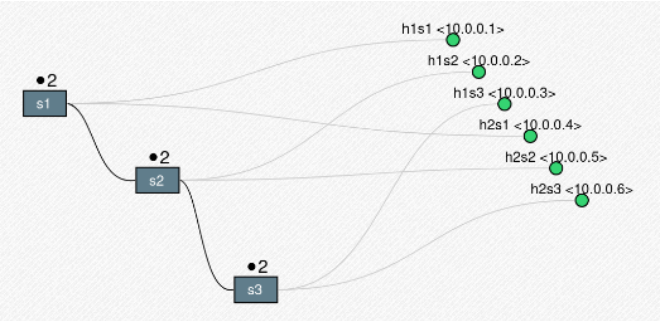
Using mininet to execute a network with 3 OVS, connected in line by 2 hosts per OVS we obtained the following network configuration:

```
mininet@arqxxs:~$ sudo mn --topo linear,3,2 --mac --switch ovsk --controller remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Adding hosts:
h1s1 h1s2 h1s3 h2s1 h2s2 h2s3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1s1, s1) (h1s2, s2) (h1s3, s3) (h2s1, s1) (h2s2, s2) (h2s3, s3) (s2, s1) (s3, s2)
*** Configuring hosts
h1s1 h1s2 h1s3 h2s1 h2s2 h2s3
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
```

Then we used the commands *dump* and *links*

```
mininet> dump
<Host h1s1: h1s1-eth0:10.0.0.1 pid=13743>
<Host h1s2: h1s2-eth0:10.0.0.2 pid=13745>
<Host h1s3: h1s3-eth0:10.0.0.3 pid=13747>
<Host h2s1: h2s1-eth0:10.0.0.4 pid=13749>
<Host h2s2: h2s2-eth0:10.0.0.5 pid=13751>
<Host h2s3: h2s3-eth0:10.0.0.6 pid=13753>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None pid=13758>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None,s2-eth4:None pid=13761>
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None,s3-eth3:None pid=13764>
<RemoteController c0: 127.0.0.1:6653 pid=13735>
mininet> links
h1s1-eth0<->s1-eth1 (OK OK)
h1s2-eth0<->s2-eth1 (OK OK)
h1s3-eth0<->s3-eth1 (OK OK)
h2s1-eth0<->s1-eth2 (OK OK)
h2s2-eth0<->s2-eth2 (OK OK)
h2s3-eth0<->s3-eth2 (OK OK)
s2-eth3<->s1-eth3 (OK OK)
s3-eth3<->s2-eth4 (OK OK)
```

The final graphic result is:



2.3. Configure host IP addresses on 192.168.x.0/24 subnets

Then, we manually modify the addresses of every node in the network, by sequentially removing the default addresses and associating them with the right addresses and mask, as we can see in the following figure:

```
mininet> h1s1 ip addr del 10.0.0.1/8 dev h1s1-eth0
mininet> h1s1 ip addr add 192.168.1.2/24 dev h1s1-eth0
mininet> h2s1 ip addr del 10.0.0.4/8 dev h2s1-eth0
mininet> h2s1 ip addr add 192.168.1.3/24 dev h2s1-eth0
mininet> h1s2 ip addr del 10.0.0.2/8 dev h1s2-eth0
mininet> h1s2 ip addr add 192.168.2.2/24 dev h1s2-eth0
mininet> h2s2 ip addr del 10.0.0.5/8 dev h2s2-eth0
mininet> h2s2 ip addr add 192.168.2.3/24 dev h2s2-eth0
mininet> h1s3 ip addr del 10.0.0.3/8 dev h1s3-eth0
mininet> h1s3 ip addr add 192.168.3.2/24 dev h1s3-eth0
mininet> h2s3 ip addr del 10.0.0.6/8 dev h2s3-eth0
mininet> h2s3 ip addr add 192.168.3.3/24 dev h2s3-eth0
```

Then we tried to use the ping command to test the connection between hosts under the same switch and between two different switches.

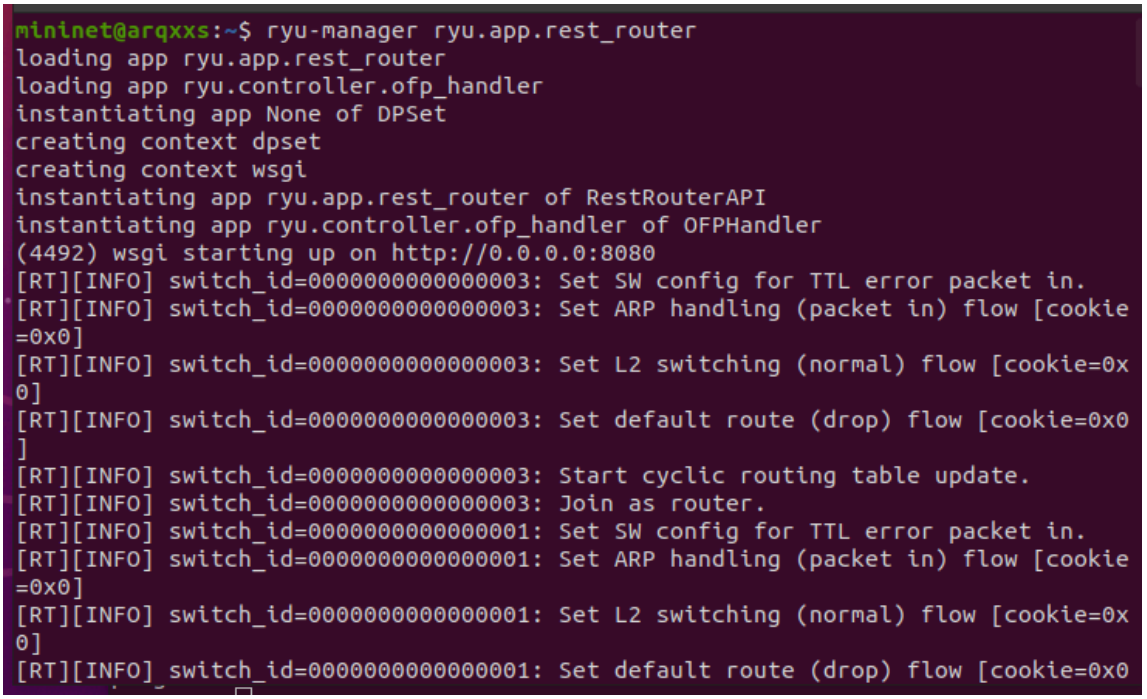
```
mininet> h1s1 ping h2s1
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
From 192.168.1.2 icmp_seq=1 Destination Host Unreachable
From 192.168.1.2 icmp_seq=2 Destination Host Unreachable
From 192.168.1.2 icmp_seq=3 Destination Host Unreachable
^C
--- 192.168.1.3 ping statistics ---
4 packets transmitted, 0 received, +3 errors, 100% packet
pipe 4
mininet> h1s1 ping h1s2
ping: connect: Network is unreachable
```

As we have seen, the first ping command (inside the same LAN) fails due to the *Destination Host Unreachable* error.

This is due to the fact that we have not yet defined a router in the ip 192.168.1.1, we will later on see how adding a router in this address allows this communication inside the same LAN.
When testing connectivity between different LANs, we have seen that it throws a Network is unreachable error. This is due to the fact that they have not in its internal scope a valid way of accessing the other network.

2.4. Launch Ryu controller with rest_router application

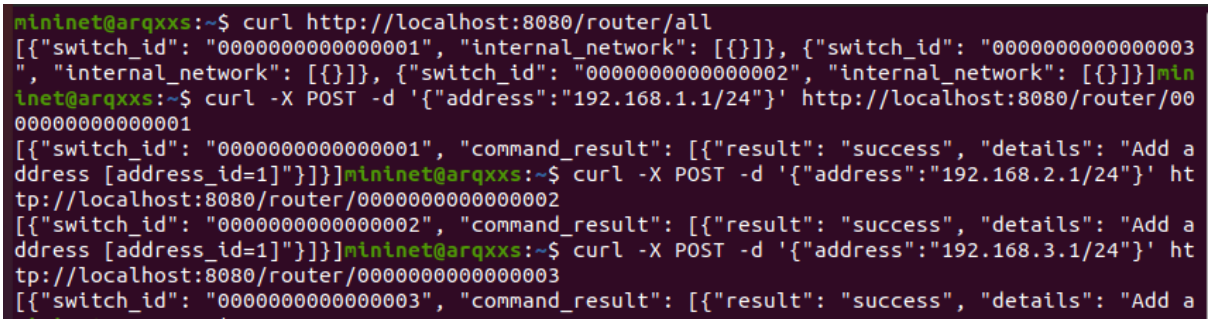
In a separate terminal, we launched the Ryu controller with the rest_router application using ryu-manager ryu.app.rest_router. This initiated an HTTP server on port 8080, ready to process REST API calls for network configuration. The controller's console output displayed real-time notifications of switch connections and flow modifications, providing visibility into the control plane's operation.



2.5. Set OVS LAN interfaces via REST API calls

Using curl commands, we could access the switch interfaces via REST API.

Each switch received its respective LAN interface IP (192.168.x.1/24). We verified these configurations by querying the controller's current state with curl *http://localhost:8080/router/all*, which returned JSON-formatted data showing all configured switches and their interfaces.



After we had the SDN process running we used the following commands:

· OVS1 (s1) in LAN 1 (192.168.1.0/24)

> curl -X POST -d '{"address":"192.168.1.1/24"}'

http://localhost:8080/router/0000000000000001

· OVS2 (s2) in LAN 2 (192.168.2.0/24)

> curl -X POST -d '{"address":"192.168.2.1/24"}'

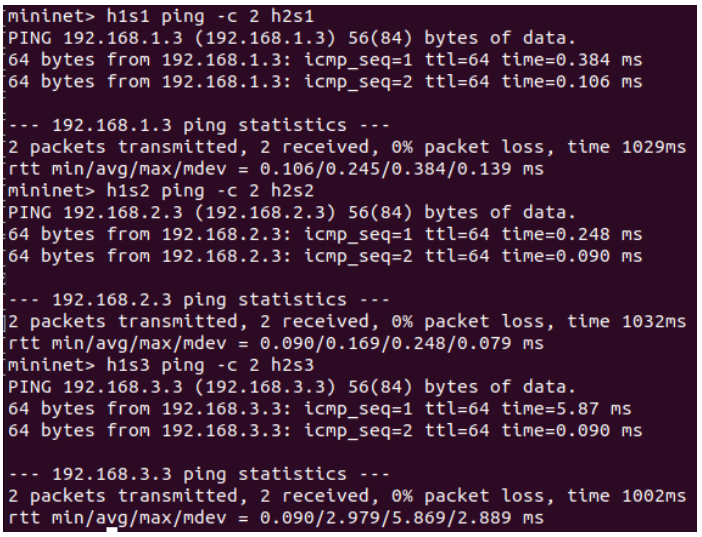
http://localhost:8080/router/0000000000000002

· OVS3 (s3) in LAN 3 (192.168.3.0/24)

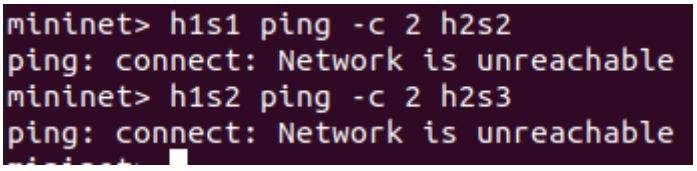
> curl -X POST -d '{"address":"192.168.3.1/24"}'

http://localhost:8080/router/0000000000000003

Then we did a ping test inside the same LANS. As we can see in the following figure, the result was successful:



But the ping test inside different LANS: fails.



2.6. Configure default gateways for all hosts

We established default routes on all hosts pointing to their local switch's IP address:

```
mininet> h1s1 ip route add default via 192.168.1.1
mininet> h2s1 ip route add default via 192.168.1.1
mininet> h1s2 ip route add default via 192.168.2.1
mininet> h2s2 ip route add default via 192.168.2.1
mininet> h1s3 ip route add default via 192.168.3.1
mininet> h2s3 ip route add default via 192.168.3.1
```

```
mininet> h1s1 ping -c 1 h2s1
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=0.052 ms

--- 192.168.1.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.052/0.052/0.052/0.000 ms
mininet> h1s2 ping -c 1 h2s2
PING 192.168.2.3 (192.168.2.3) 56(84) bytes of data.
64 bytes from 192.168.2.3: icmp_seq=1 ttl=64 time=0.236 ms

--- 192.168.2.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.236/0.236/0.236/0.000 ms
mininet> h1s3 ping -c 1 h2s3
PING 192.168.3.3 (192.168.3.3) 56(84) bytes of data.
64 bytes from 192.168.3.3: icmp_seq=1 ttl=64 time=0.195 ms

--- 192.168.3.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.195/0.195/0.195/0.000 ms
```

Same-LAN pings succeed as hosts communicate directly via layer-2 switching, but cross-LAN pings fail because hosts lack gateways and switches haven't been configured with inter-subnet routing rules yet.

```
mininet> h1s1 ping h1s2
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
^C
--- 192.168.2.2 ping statistics ---
43 packets transmitted, 0 received, 100% packet loss, time 43007ms
```

2.7. Establish point-to-point links between OVS switches

The point-to-point links between switches were configured with addresses from the 10.0.1.0/30 and 10.0.2.0/30 subnets:

```
mininet@arqxxs:~$ curl -X POST -d '{"address":"10.0.1.1/30"}' http://localhost:8080/router/00
00000000000001
[{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "Add a
ddress [address_id=2]"}]}]mininet@arqxxs:~$ curl -X POST -d '{"address":"10.0.1.2/30"}' http:
//localhost:8080/router/0000000000000002
[{"switch_id": "0000000000000002", "command_result": [{"result": "success", "details": "Add a
ddress [address_id=2]"}]}]mininet@arqxxs:~$ curl -X POST -d '{"address":"10.0.2.1/30"}' http:
//localhost:8080/router/0000000000000002
[{"switch_id": "0000000000000002", "command_result": [{"result": "success", "details": "Add a
ddress [address_id=3]"}]}]mininet@arqxxs:~$ curl -X POST -d '{"address":"10.0.2.2/30"}' http:
//localhost:8080/router/0000000000000003
```

```
mininet> h1s1 ping -c 1 h2s1
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=0.236 ms

--- 192.168.1.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.236/0.236/0.236/0.000 ms
mininet> h1s2 ping -c 1 h2s2
PING 192.168.2.3 (192.168.2.3) 56(84) bytes of data.
64 bytes from 192.168.2.3: icmp_seq=1 ttl=64 time=0.202 ms

--- 192.168.2.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.202/0.202/0.202/0.000 ms
mininet> h1s3 ping -c 1 h2s3
PING 192.168.3.3 (192.168.3.3) 56(84) bytes of data.
64 bytes from 192.168.3.3: icmp_seq=1 ttl=64 time=0.180 ms

--- 192.168.3.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.180/0.180/0.180/0.000 ms
```

The successful switch-to-switch pings (s1 ↔ s2) confirm the inter-switch links are properly established at layer 3, but the absence of end-to-end routing prevents host-to-host communication across different subnets.

```
mininet> h1s1 ping h1s2
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
^C
--- 192.168.2.2 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2037ms
```

Link 1: OVS1 (s1) ↔ OVS2 (s2)

OVS1 (s1) Interface (s1-eth3):

curl -X POST -d '{"address":"10.0.1.1/30"}' http://localhost:8080/router/0000000000000001

OVS2 (s2) Interface (s2-eth3):

curl -X POST -d '{"address":"10.0.1.2/30"}' http://localhost:8080/router/0000000000000002

Link 2: OVS2 (s2) ↔ OVS3 (s3)

OVS2 (s2) Interface (s2-eth4):

curl -X POST -d '{"address":"10.0.2.1/30"}' http://localhost:8080/router/0000000000000002

OVS3 (s3) Interface (s3-eth3):

curl -X POST -d '{"address":"10.0.2.2/30"}' http://localhost:8080/router/0000000000000003

2.8. Implement static routing for full network connectivity

By programming static routes via Ryu's REST API (e.g., destination: 192.168.2.0/24, gateway: 10.0.1.2), we achieved full connectivity. The pingall command confirmed 100% reachability, validating our SDN routing implementation.

```
mininet@arqxxs:~$ curl -X POST -d '{"destination":"192.168.2.0/24","gateway":"10.0.1.2"}' htt
p://localhost:8080/router/0000000000000001
[{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "Add r
oute [route_id=1]"}]}]mininet@arqxxs:~$ curl -X POST -d '{"destination":"192.168.3.0/24","gat
eway":"10.0.1.2"}' http://localhost:8080/router/0000000000000001
[{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "Add r
oute [route_id=2]"}]}]mininet@arqxxs:~$ curl -X POST -d '{"destination":"192.168.1.0/24","gat
eway":"10.0.2.1"}' http://localhost:8080/router/0000000000000003
[{"switch_id": "0000000000000003", "command_result": [{"result": "success", "details": "Add r
oute [route_id=1]"}]}]mininet@arqxxs:~$ curl -X POST -d '{"destination":"192.168.2.0/24","gat
eway":"10.0.2.1"}' http://localhost:8080/router/0000000000000003
[{"switch_id": "0000000000000003", "command_result": [{"result": "success", "details": "Add r
oute [route_id=2]"}]}]mininet@arqxxs:~$
```

OVS1 (s1): Route LAN2 and LAN3 traffic via OVS2 (10.0.1.2)

curl -X POST -d '{"destination":"192.168.2.0/24","gateway":"10.0.1.2"}' http://localhost:8080/router/0000000000000001
curl -X POST -d '{"destination":"192.168.3.0/24","gateway":"10.0.1.2"}' http://localhost:8080/router/0000000000000001

OVS3 (s3): Route LAN1 and LAN2 traffic via OVS2 (10.0.2.1)

curl -X POST -d '{"destination":"192.168.1.0/24","gateway":"10.0.2.1"}' http://localhost:8080/router/0000000000000003
curl -X POST -d '{"destination":"192.168.2.0/24","gateway":"10.0.2.1"}' http://localhost:8080/router/0000000000000003

OVS2 needs explicit routes to forward traffic between LAN1 and LAN3

```
mininet@arqxxs:~$ curl -X POST -d '{"destination":"192.168.1.0/24","gateway":"10.0.1.1"}' htt
p://localhost:8080/router/0000000000000002
[{"switch_id": "0000000000000002", "command_result": [{"result": "success", "details": "Add r
oute [route_id=1]"}]}]mininet@arqxxs:~$ curl -X POST -d '{"destination":"192.168.3.0/24","gat
eway":"10.0.2.2"}' http://localhost:8080/router/0000000000000002
[{"switch_id": "0000000000000002", "command_result": [{"result": "success", "details": "Add r
oute [route_id=2]"}]}]mininet@arqxxs:~$
```

After configuring the routes in the OVSs, we make use of 2 more *curl* REST calls from the API of the router Ryu Script to define the route for a given destination network in the following way:

Route LAN1 (192.168.1.0/24) via OVS1 (10.0.1.1)

> *curl -X POST -d '{"destination":"192.168.1.0/24","gateway":"10.0.1.1"}' http://localhost:8080/router/0000000000000002*

Route LAN3 (192.168.3.0/24) via OVS3 (10.0.2.2)

> *curl -X POST -d '{"destination":"192.168.3.0/24","gateway":"10.0.2.2"}' http://localhost:8080/router/0000000000000002*

Then, before testing full connectivity between all the six nodes, we tested if from a host in switch 1 we could reach a host in switch 2, using *ping* command as we can see:

```
mininet> h1s1 ping -c1 h1s2
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=0.555 ms

--- 192.168.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.555/0.555/0.555/0.000 ms
mininet> h1s1 ping -c1 h1s3
PING 192.168.3.2 (192.168.3.2) 56(84) bytes of data.
64 bytes from 192.168.3.2: icmp_seq=1 ttl=61 time=4.90 ms

--- 192.168.3.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 4.903/4.903/4.903/0.000 ms
mininet> h1s2 ping -c1 h1s3
PING 192.168.3.2 (192.168.3.2) 56(84) bytes of data.
64 bytes from 192.168.3.2: icmp_seq=1 ttl=62 time=0.245 ms

--- 192.168.3.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.245/0.245/0.245/0.000 ms
```

Since the connection was successful, we proceeded with the final test.

Pingall test shows that we successfully built the connections, and we can establish communication between hosts of both the same and different LANs, as we can see in the following image:

```
mininet> pingall
*** Ping: testing ping reachability
h1s1 -> h1s2 h1s3 h2s1 h2s2 h2s3
h1s2 -> h1s1 h1s3 h2s1 h2s2 h2s3
h1s3 -> h1s1 h1s2 h2s1 h2s2 h2s3
h2s1 -> h1s1 h1s2 h1s3 h2s2 h2s3
h2s2 -> h1s1 h1s2 h1s3 h2s1 h2s3
h2s3 -> h1s1 h1s2 h1s3 h2s1 h2s2
*** Results: 0% dropped (30/30 received)
mininet>
```

3. Conclusion

This lab session deepened our work with SDN principles through hands-on experience with Mininet and the Ryu controller. We built a multi-switch topology, configured host addressing, and implemented routing through the controller's REST API, testing connectivity at each stage. The process gave us practical insight into how SDN's centralized control simplifies network management compared to traditional approaches.

By systematically configuring the network components and troubleshooting connectivity issues, we reinforced key concepts of subnetting, routing, and programmable network control. The successful implementation of full network connectivity demonstrated SDN's operational advantages in real-world scenarios.

The session provided valuable experience with SDN tools and methodologies, establishing a foundation for more advanced network programmability concepts in future work. The hands-on approach effectively bridged theoretical knowledge with practical implementation.