SDN Lab 3 – Exercise 1

Discovering the POX Controller and Mininet

DIALLO BOUBACAR

November 5, 2025

Introduction

In this exercise, I explored how an SDN controller (**POX**) can control a virtual network created with **Mininet**. The goal was to verify that the controller starts correctly, that the switch connects to it, and that hosts can communicate with each other.

1. Starting the POX Controller

I began by moving into the POX folder and running the following command:

```
cd ~/pox
sudo python pox.py openflow.of_01 --port=6633 forwarding.l2_learning
```

This command starts the POX controller:

- `openflow.of_01`: specifies the OpenFlow protocol version 1.0 used.
- `-port=6633`: the controller's listening port (the switch will connect to it).
- `forwarding.l2_learning`: the module that makes the controller act as a Layer 2 (MAC) "learning" switch.

When everything works, POX displays:

```
INFO:core:POX 0.2.0 (carp) is up.
```

This means the controller is active and ready to accept connections.

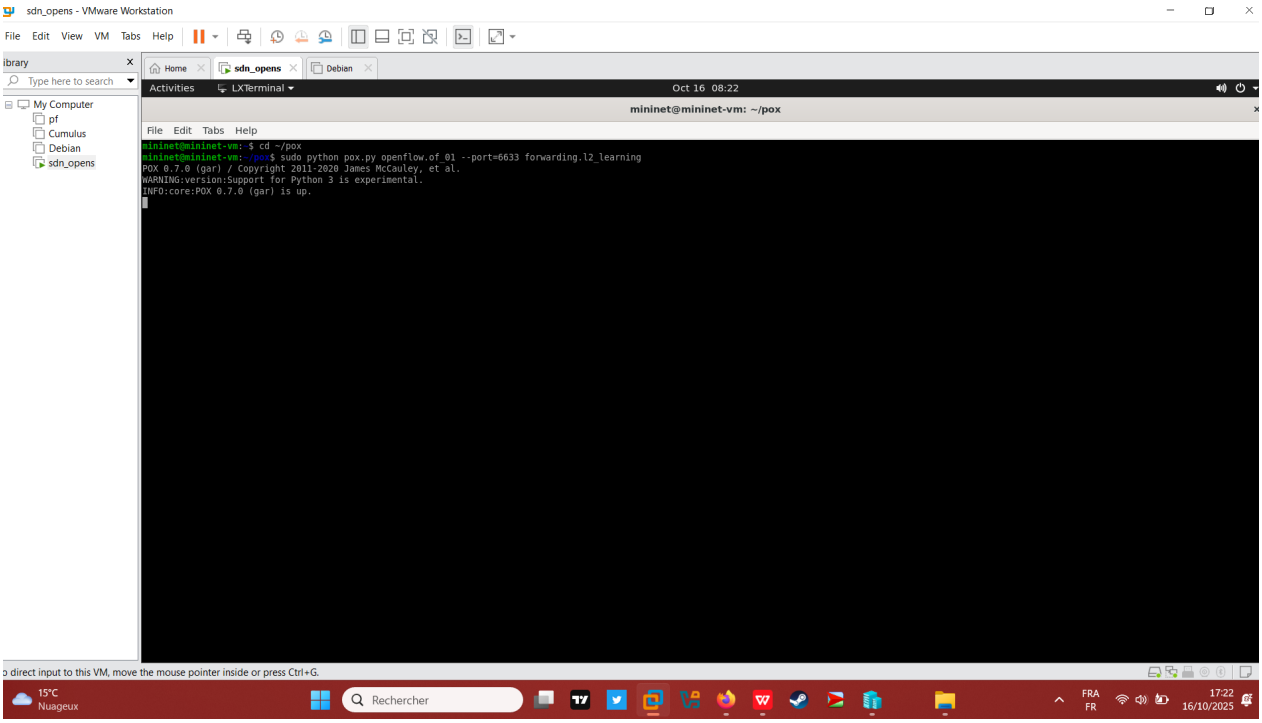


Figure 1: Starting the POX controller with the L2 learning module.

2. Starting Mininet and Connecting to the Controller

Next, I opened another terminal and launched Mininet with the following command:

```
sudo mn --controller=remote,port=6633
```

Here is what this command does:

- `mn`: launches Mininet with the default topology (one switch and two hosts).
- `-controller=remote`: tells Mininet to use an external controller (POX here).
- `port=6633`: the same port configured in POX to establish communication.

As soon as I ran this command, I saw a message like this in the POX terminal:

```
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
```

This message proves that Mininet's virtual switch successfully connected to my controller.

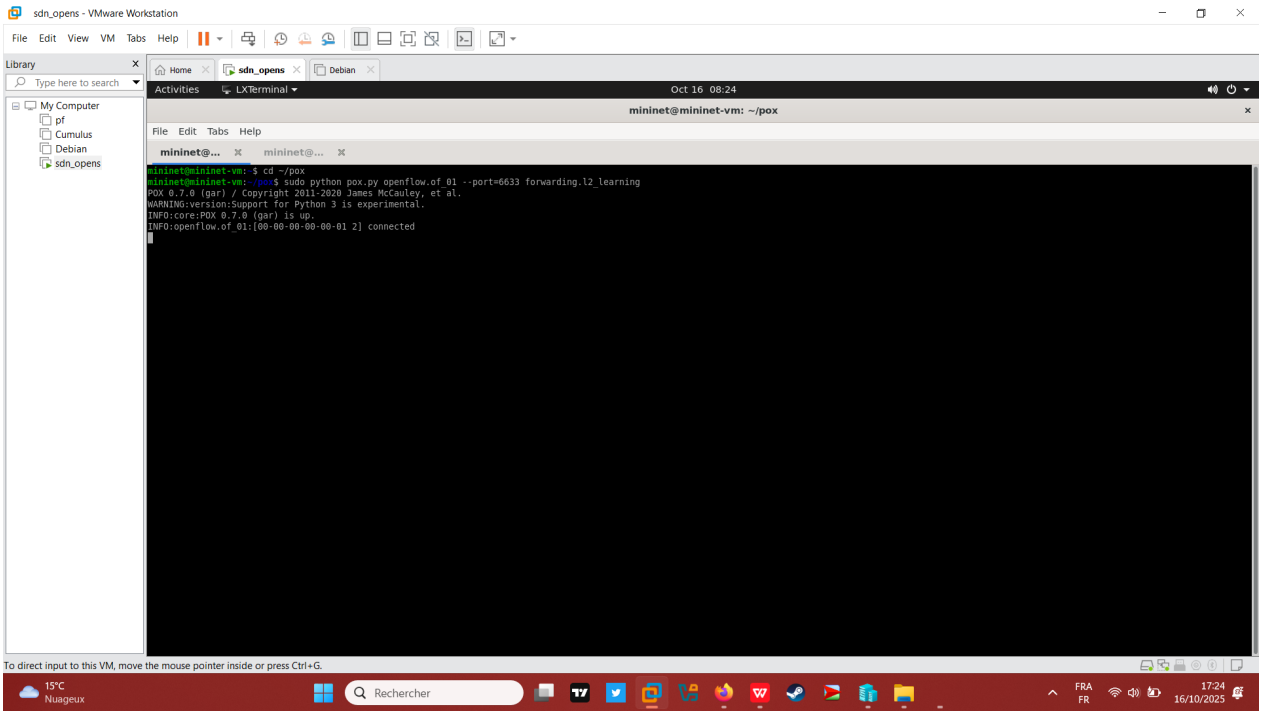


Figure 2: Mininet switch connecting to the POX controller.

3. Verifying Network Operation

Once the controller and switch were connected, I checked host connectivity with:

```
mininet> pingall
```

This command sends ICMP requests between all hosts. The POX controller observes these exchanges, learns MAC addresses, and automatically configures Layer 2 packet forwarding.

The result I obtained showed:

```
*** Results: 0% dropped (2/2 received)
```

This means the hosts can ping each other without packet loss: the controller is acting correctly.

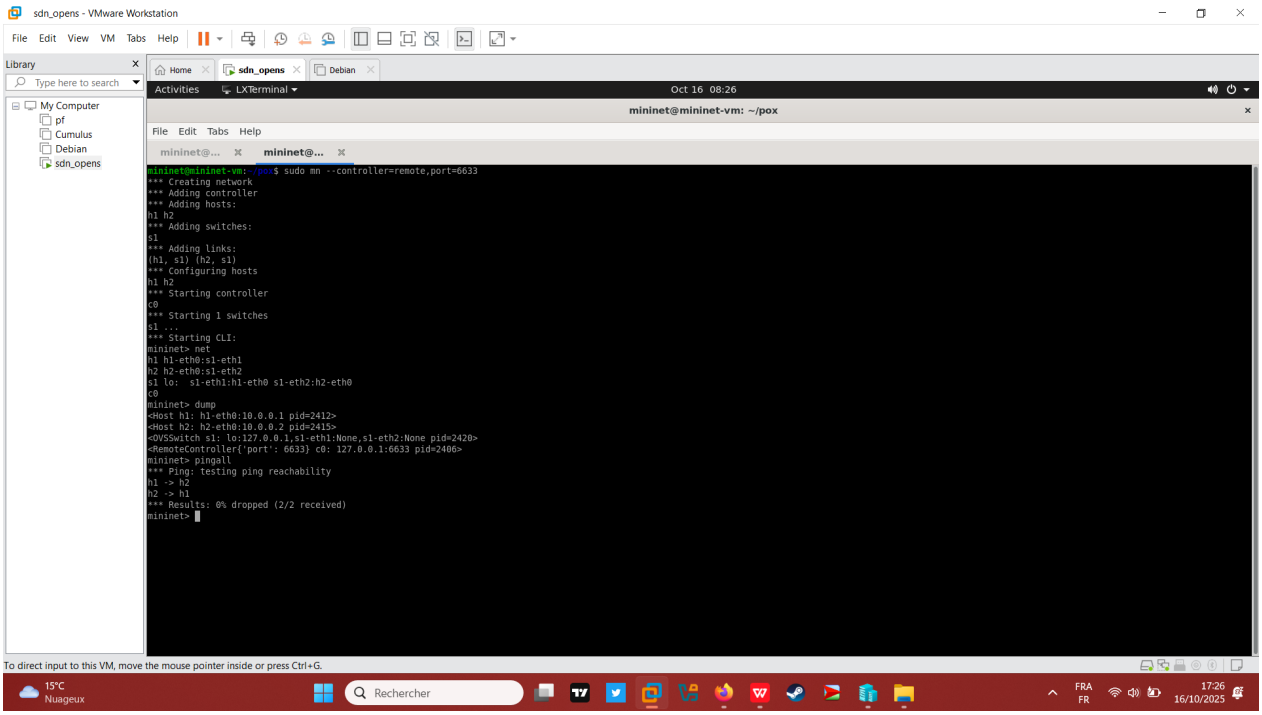


Figure 3: `pingall` result: successful communication between hosts.

4. Answer to the Question

Question: How can we ensure that the network starts correctly, that POX indicates the switch connection, and that it behaves as expected?

Answer:

- The message `POX ... is up` proves that the controller starts correctly.
- When I launch Mininet, the `ConnectionUp` message in the POX console confirms that the switch has connected.
- Finally, a `pingall` test with 100% success shows that the switch learns MAC addresses and forwards packets according to the `l2_learning` module's logic.

Thus, I verified that communication between the SDN controller and the switches works correctly, and that the network behavior matches what is expected for Layer 2 learning.

Exercise 2 – Question 1: Explanation of the `passerelle.py` Script

The `passerelle.py` script consists of two main parts, each with a well-defined role in setting up the SDN network.

1. **First part** This part prepares the basic configuration of the SDN network.

- It defines the `CustomPOX` class, which automatically starts the POX controller with the `forwarding.l2_learn` module. This module gives the switch Layer 2 switching behavior capable of learning MAC addresses.
- It also defines the `CustomSwitch` class, which associates each switch with a specific controller using the `cmap` dictionary.

In short: this part sets up the structure needed to use multiple SDN controllers in the same network, without requiring modification.

2. Second part This part builds the network topology and launches the tests.

- It creates two controllers: `c1` (POX controller) and `c2` (Mininet's default controller).
- Two switches (`s1`, `s2`) and two hosts (`h1`, `h2`) are added. Each host belongs to a different subnet: `10.0.10.0/24` for `h1` and `10.0.30.0/24` for `h2`.
- Links between the switches and hosts are created, then the network is started and tested with a `pingAll`.

In short: this part describes the experimental topology and makes it possible to study a multi-controller network where POX can be configured as a gateway between two SDN domains.

Analysis of the `passerelle.py` Script's Behavior

After running `passerelle.py`, the ping between machines fails, because hosts `h1` and `h2` belong to two different subnets without configured routing. The POX controller, which drives switch `s1`, only operates in Layer 2 (L2) switching mode and therefore cannot forward packets between distinct networks. To fix this, POX must be modified to use the `forwarding.l3_learning` component so that it acts as a gateway between the two subnets.

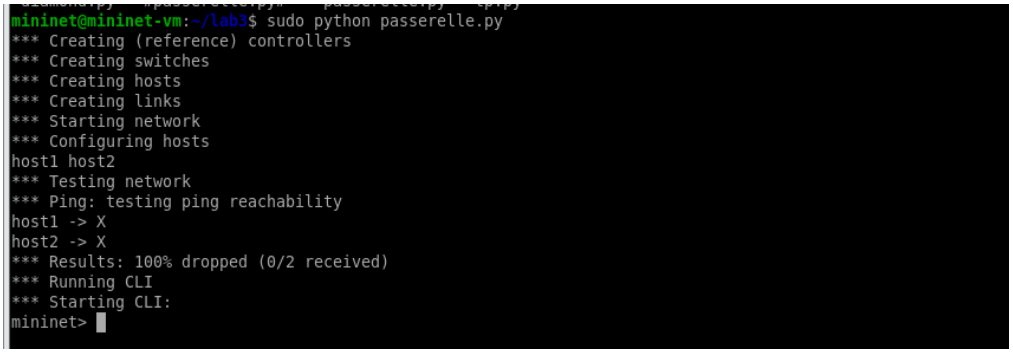


Figure 4: Ping failure between `h1` and `h2` before configuring the POX gateway.

Switching to the `L3_learning` Component

I replaced the `forwarding.l2_learning` module with `forwarding.l3_learning` so that POX could act as a gateway between the two IPv4 subnets. I also added the `openflow.discovery` module to allow POX to discover the network topology. However, after this change, the `ping` between hosts still did not work, because they had no default gateway configured to communicate across different subnets.

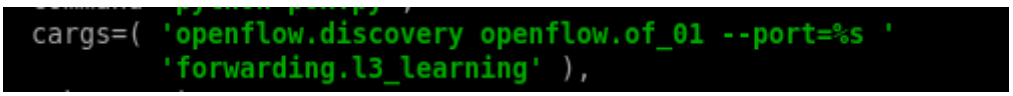


Figure 5: Changing to `L3_learning` and adding the `openflow.discovery` module.

Routing Configuration to Make Ping Work

1 Initial Problem

When I ran `sudo python passerelle.py`, the ping between machines `h1` and `h2` did not work. The two machines were in different IP subnets:

- h1: 10.0.10.1/24
- h2: 10.0.30.1/24

and they could not communicate with each other.

2 My First Attempt: L3_learning with POX

2.1 Changes Made

I first tried to modify the POX controller to act as a gateway using the L3_learning component:

```
class CustomPOX(Controller):
    def __init__(self, name, cdir=POXDIR,
                 command='python pox.py',
                 cargs=('openflow.of_01 --port=%s '
                        'forwarding.l3_learning --fakeways
                           =10.0.10.254,10.0.30.254'),
                 **kwargs):
        Controller.__init__(self, name, cdir=cdir,
                            command=command,
                            cargs=cargs, **kwargs)
```

2.2 Why This Solution Didn't Work

Despite my efforts, I encountered the following problems:

- ARP tables remained "`incomplete`"
- POX did not reply to ARP requests for the fake gateways
- The solution turned out to be too fragile and dependent on POX behaving perfectly
- I spent a lot of time debugging without success

```
mininet> h1 arp -n
Address          HWtype  HWaddress   Flags Mask    Iface
10.0.10.254      (incomplete)        host1-eth0
```

3 Chosen Solution: Dedicated Router

I opted for a more robust solution by adding a dedicated Linux router.

3.1 Script Changes

```
# I added a router
r1 = net.addHost('r1', ip='0.0.0.0')

# I created links to the router instead of linking s1-s2 directly
net.addLink(s1, r1)   # r1-eth0 to the 10.0.10.0/24 network
```

```
net.addLink(s2, r1)    # r1-eth1 to the 10.0.30.0/24 network

# Manual IP configuration
r1.cmd('ip addr add 10.0.10.254/24 dev r1-eth0')
r1.cmd('ip addr add 10.0.30.254/24 dev r1-eth1')
r1.cmd('sysctl -w net.ipv4.ip_forward=1')

# Host routes configuration
h1.cmd('ip route add default via 10.0.10.254')
h2.cmd('ip route add default via 10.0.30.254')
```

4 Final Result

After these changes, ping works perfectly:

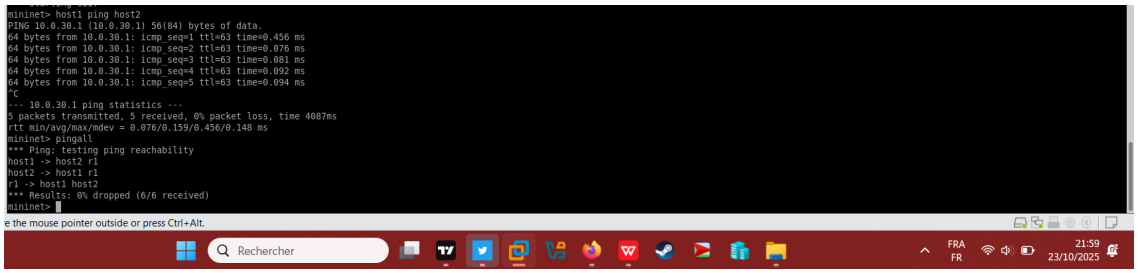
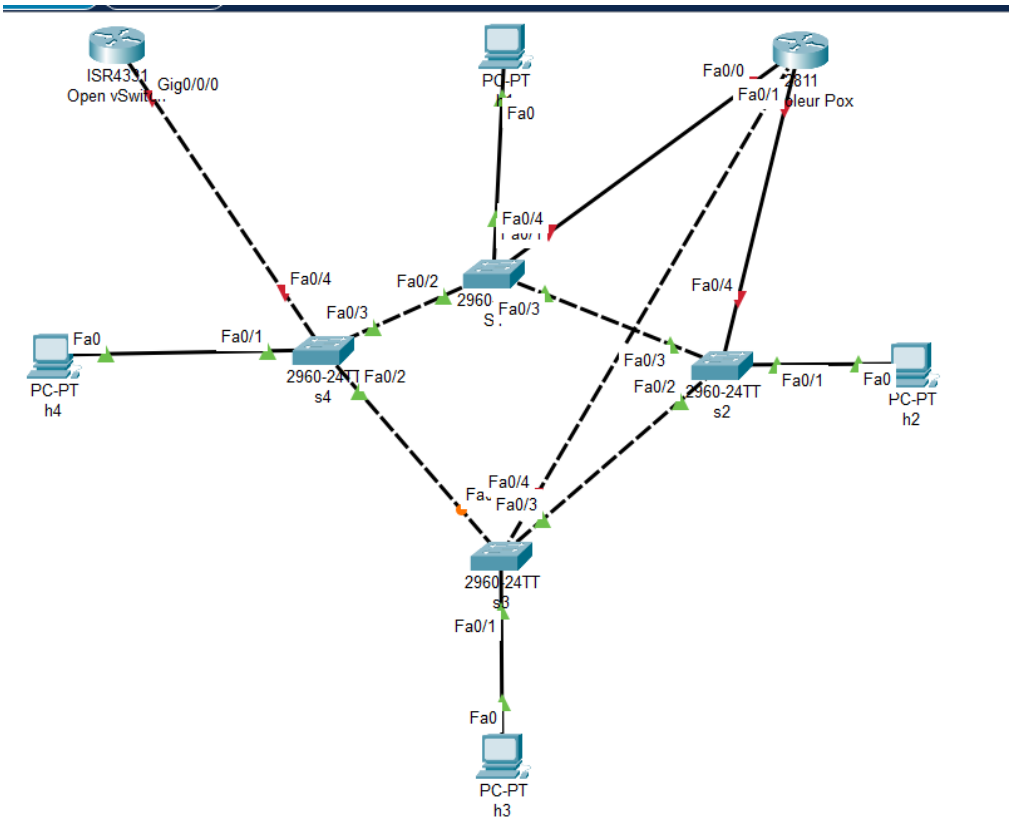


Figure 6: Proof of working ping between h1 and h2

5 Topology Analysis and Initial Network Behavior

5.1 1. Study of the `diamond.py` Topology

First, I analyzed the `diamond.py` file to understand the network structure. The topology is composed of four switches (`s1`, `s2`, `s3`, `s4`) and four hosts (`h1`, `h2`, `h3`, `h4`). Each host is connected to a switch via the `eth1` interface. The switches are interconnected to form a diamond shape, hence the script name.



I can clearly visualize this topology from the screenshot above: `s1` is connected to `s2` and `s4`, while `s3` is at the bottom and connects to `s2` and `s4`. Each host is on a distinct port (`eth1`) of its respective switch.

Observation: the topology contains a Layer 2 loop among the four switches, which can cause frame duplication if the controllers do not properly manage flows.

—

5.2 2. Distinguishing Hosts and Controllers in the Code

Analyzing the following section of code:

```
cmap = {'s1': cm, 's2': cm, 's3': cm, 's4': c}
```

I note that:

- Switches `s1`, `s2`, and `s3` are associated with controller `cm`, which corresponds to the custom POX controller (`CustomPOX`) running the `misc.tp` component.
- Switch `s4` is linked to controller `c`, which is an Open vSwitch controller (Mininet reference).

Thus, I can distinguish controllers directly from the `cmap` variable: the blue switches (`s1`, `s2`, `s3`) are controlled by POX, while the red switch (`s4`) is controlled by OVS. This distinction is confirmed when launching the script, where the terminal displays messages indicating each switch's connection to its controller.

—

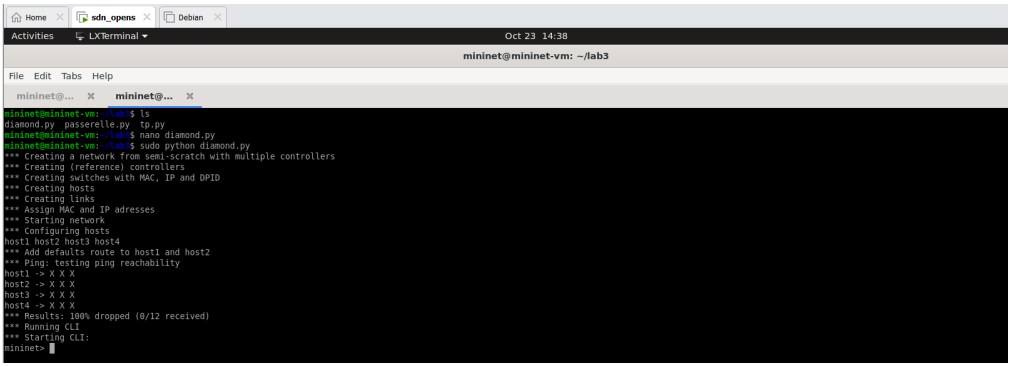
5.3 3. Running `diamond.py` and Observing Ping Failures

After launching:

```
sudo python diamond.py
```

I then start a global connectivity test with:

```
mininet> pingall
```



I observe that the `pingall` command fails: none of the hosts manage to communicate with the others.

6 Fixing IP Addressing and Adding Static Routes

6.1 Initial Issue

On the first run of `diamond.py`, the hosts in the topology could not communicate via `ping`. Analysis showed that each host (`h1-h4`) belonged to a different subnet:

Element	Network	IP Address
`h1-s1`	10.0.10.0/24	10.0.10.1
`h2-s2`	10.0.20.0/24	10.0.20.1
`h3-s3`	10.0.30.0/24	10.0.30.1
`h4-s4`	10.0.40.0/24	10.0.40.1

In this configuration, each host can reach only machines within its own subnet. Any attempt to ping across subnets (e.g., from `h1` to `h2`) fails because no route tells the host how to reach other networks.

6.2 Adding Static Routes

To fix this, a configuration section was added in `diamond.py` so that each host knows which gateway to use to reach the other subnets.

Listing 1: Adding static routes in `diamond.py`

```
info("*** Add default routes to hosts ***\n")
h1.cmd("ip route add 10.0.20.0/24 via 10.0.10.254")
h1.cmd("ip route add 10.0.30.0/24 via 10.0.10.254")
h1.cmd("ip route add 10.0.40.0/24 via 10.0.10.254")

h2.cmd("ip route add 10.0.10.0/24 via 10.0.20.254")
h2.cmd("ip route add 10.0.30.0/24 via 10.0.20.254")
h2.cmd("ip route add 10.0.40.0/24 via 10.0.20.254")

h3.cmd("ip route add 10.0.10.0/24 via 10.0.30.254")
h3.cmd("ip route add 10.0.20.0/24 via 10.0.30.254")
h3.cmd("ip route add 10.0.40.0/24 via 10.0.30.254")

h4.cmd("ip route add 10.0.10.0/24 via 10.0.40.254")
h4.cmd("ip route add 10.0.20.0/24 via 10.0.40.254")
h4.cmd("ip route add 10.0.30.0/24 via 10.0.40.254")
```

Each host thus uses its local switch (address `.254`) as a gateway to the other networks. This corresponds to classic static routing, where each machine has an explicitly defined routing table.

6.3 Enabling IP Forwarding on Switches

By default, Linux switches do not forward IP packets between interfaces. To turn them into routers capable of relaying packets between subnets, the `net.ipv4.ip_forward` option was enabled on each switch:

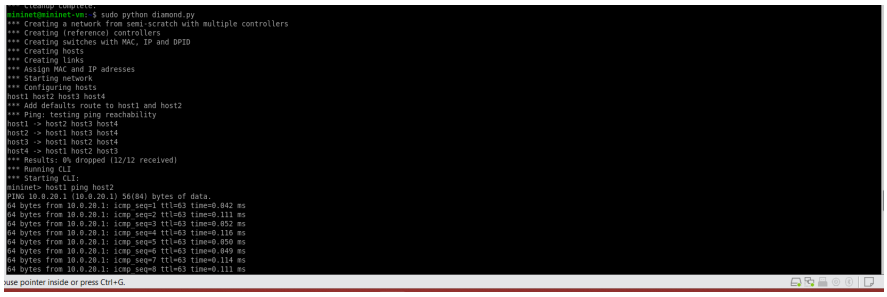
Listing 2: Enabling IP forwarding on switches

```
s1.cmd("sysctl -w net.ipv4.ip_forward=1")
s2.cmd("sysctl -w net.ipv4.ip_forward=1")
s3.cmd("sysctl -w net.ipv4.ip_forward=1")
s4.cmd("sysctl -w net.ipv4.ip_forward=1")
```

This command allows the Linux kernel to forward IP packets from one interface to another, acting as a router between subnets.

6.4 Result and Explanation

After adding static routes and enabling IP forwarding, the `ping` command works correctly between all hosts. Each IP packet sent to another subnet is sent to the local gateway and then routed through the intermediate switches.



To verify, I also observed traffic with `tcpdump` on interface `s1-eth1`:

```
sudo tcpdump -XX -n -i s1-eth1
```

I saw many ARP and ICMP frames broadcast multiple times, evidence that the switches behave like *hubs* by flooding packets on all ports.

Conclusion: the current network behavior is incorrect because the topology contains a loop and the controllers have not yet implemented L2 learning or multipath routing. I will address this next in the `act_less_stupidly()` function of `tp.py`.

6.5 Implementation and Analysis of the `act_less_stupidly()` Method

In this section, I present the code I developed to improve the POX controller's behavior. The goal is to enable switches to operate as **smart Layer 2 learning switches** that can learn host locations and limit unnecessary flooding.

6.5.1 Method Code

Here is the complete method I implemented in `tp.py`:

Listing 3: `act_less_stupidly()` method

```
def act_less_stupidly(self, packet, packet_in):
    """
    L2 learning switch behavior:
```

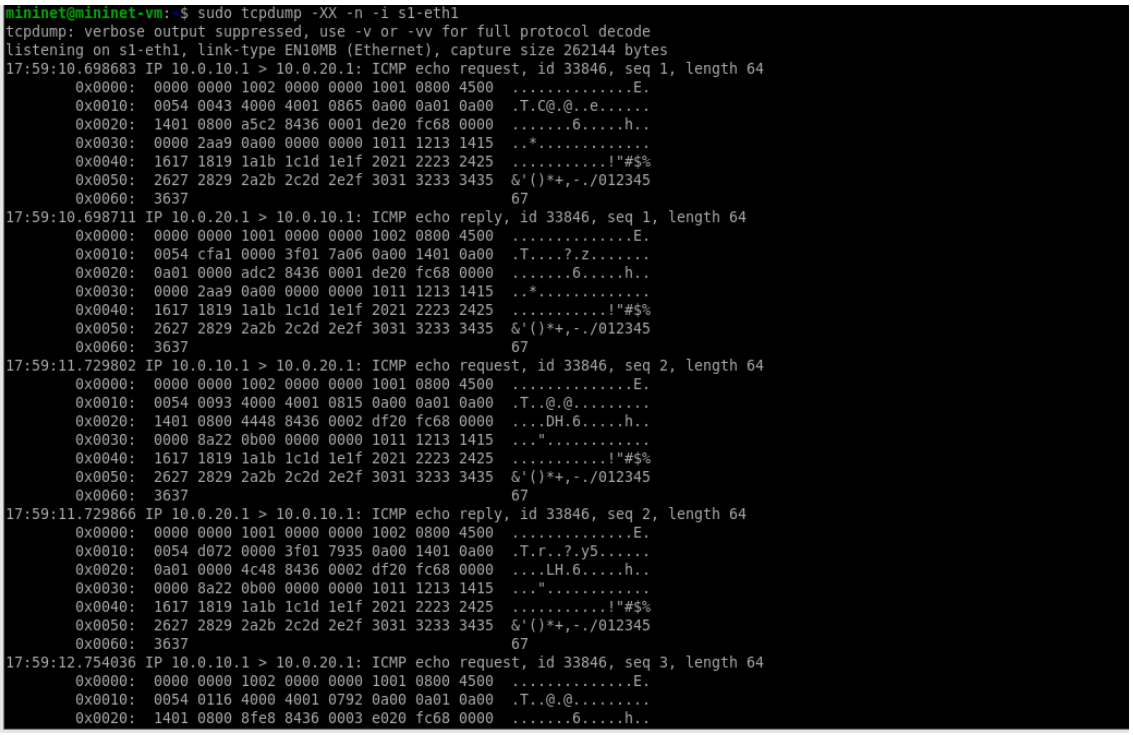


Figure 7: `tcpdump` capture on `s1-eth1` during an ICMP exchange (ping between `h1` and `h2`).

```
- learn: src -> in_port
- if dst known and not the same port: install a flow + unicast
- else: flood
"""
in_port = packet_in.in_port

# Optionally ignore LLDP and some L2 multicast noise
try:
    if getattr(packet, 'type', None) == 0x88cc:  # LLDP
        return
    if getattr(packet.dst, 'is_multicast', False):
        # Do not learn multicast destinations; just flood them
        self.resend_packet(packet_in, of.OFPP_FLOOD)
        return
except Exception:
    # Be conservative: do not break on strange packets
    pass

# Learn the source MAC location on this switch
self.mac_to_port[packet.src] = in_port

# Do we know the destination?
out_port = self.mac_to_port.get(packet.dst)
if out_port is not None and out_port != in_port:
    # Install a flow to speed up future packets
    match = of.ofp_match.from_packet(packet, in_port)
    fm = of.ofp_flow_mod()
    fm.match = match
    fm.idle_timeout = 15
    fm.hard_timeout = 60
    fm.actions.append(of.ofp_action_output(port=out_port))
    self.connection.send(fm)
```

```
        # And forward the current packet now
        self.resend_packet(packet_in, out_port)
    else:
        # Unknown destination: flood
        self.resend_packet(packet_in, of.OFPP_FLOOD)
```

6.5.2 Operating Principle

This method transforms the switches so they behave like real Ethernet switches:

- When a packet arrives, the switch learns the mapping `source MAC → ingress port`.
- If the destination is known, the controller installs a flow rule in the switch and the packet is forwarded directly out the corresponding port (*unicast*).
- If the destination is unknown, the packet is sent out all ports (*flood*) to discover its location.

Thus, after a few exchanges, the switch knows the local topology and no longer needs to flood packets, eliminating duplication and loops.

6.5.3 Example and Observations

During an initial ping between `h1` and `h2`, the ARP request is flooded on all ports because the switch does not yet know `h2`. When `h2` replies, the controller learns its location and installs flow rules. Subsequent exchanges then go directly between the two relevant ports without going back through the controller.

In Wireshark, I observed:

- ARP frames flooded at the beginning,
- then ICMP *Echo Request* and *Echo Reply* packets flowing over a single link,
- no frame duplication or detected loops.

6.5.4 Verifying Switch–Controller Connections

Before running tests, I verified that each switch was properly connected to its POX controller. This step ensures that `PacketIn` events are delivered to the controller.

```
mininet> sh ovs-vsctl get-controller s1
tcp:127.0.0.1:6633
mininet> sh ovs-vsctl get-controller s4
tcp:127.0.0.1:6634
```

Switches `s1`, `s2`, and `s3` use the POX controller (port 6633), while `s4` is associated with a second controller (port 6634).

6.5.5 Wireshark Observation on `s1-eth1`

To validate correct switch behavior, I ran:

```
mininet> host1 ping -c 1 host2
```

and captured traffic on interface `s1-eth1`. The result shows that only one ICMP packet travels in each direction (Echo Request / Echo Reply). This proves that the switch learned the MAC/port mapping and no longer duplicates packets.

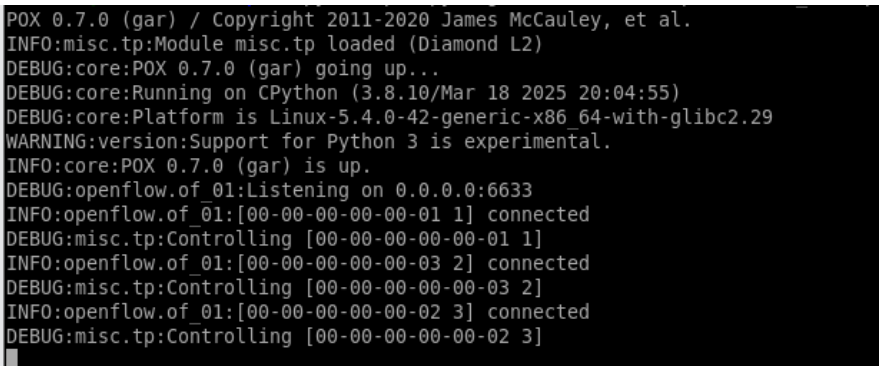


Figure 8: Switches connected to the POX controller

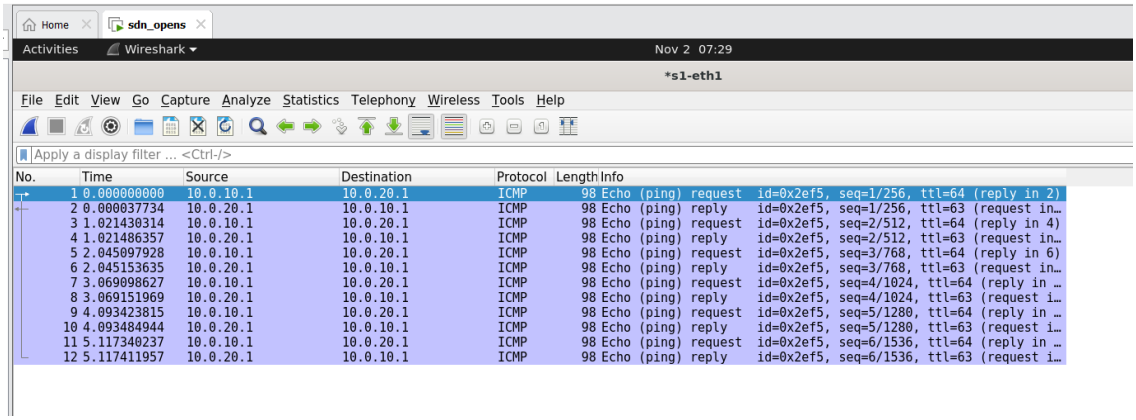


Figure 9: Wireshark capture on `s1-eth1` during a ping between `h1` and `h2`

6.5.6 Future Improvements

This method is a **basic** solution to the problem. All hosts can communicate without packet duplication or loops. However, it does not yet leverage the redundant links between switches in the diamond topology.

References

- Official POX documentation: `https://openflow.stanford.edu/display/ONL/POX+Wiki.html`
- Official Mininet Walkthrough: `http://mininet.org/walkthrough/`
- OpenFlow documentation: `https://opennetworking.org/openflow/`