Universidade de Aveiro

Comunicações Móveis

SDN - Project 1 Report

André Clérigo (98485), João Amaral (98373), Pedro Rocha (98256)

Departamento de Eletrónica, Telecomunicações e Informática

January 6, 2023

Contents

List of Figures

Chapter 1

General Information

The contributions between each member of the group were equal. The tutorials followed in the next chapter can be found here `https://docs.openvswitch.org/en/latest/tutorials/index.html`
The project's repository can be viewed here: `https://github.com/PedroRocha9/CM`
Since we didn't have time to demonstrate our work, there are some demonstrations that may be checked here: `https://drive.google.com/drive/folders/1d9-uB5swGpNYoYP5r9WEmvAvvjQl6ojL?usp=share_link`

Chapter 2

Tutorials

2.1 OVS Faucet Tutorial

2.1.1 Overview

Firstly we set up Faucet appropriately following the tutorial, using a docker container image with the correct configuration, not requiring any modification to system-level files or directories.
We downloaded a copy of the Open vSwitch repository using Git:

```
git clone https://github.com/openvswitch/ovs.git
```

And configured and built the Open vSwitch using:

```
./boot.sh
./configure
make -j4
```

Then we downloaded a copy of the Faucet source repository using Git:

```
git clone https://github.com/faucetsdn/faucet.git
cd faucet
```

There were several commands we learned that came in handy throughout the whole project, for creating, building the docker image (inside faucet folder), as well as others, namely:
Docker build:

```
sudo docker build -t faucet/faucet -f Dockerfile.faucet .
```

Docker Run:

```
sudo docker run -d --name faucet --restart=always -v $(pwd)/inst
    /:/etc/faucet/ -v $(pwd)/inst/:/var/log/faucet/ -p 6653:6653
    -p 9302:9302 faucet/faucet
```

Refreshing the configuration (hot reload):

```
sudo docker exec faucet pkill -HUP -f faucet.faucet
```

After the hot reload, we can do the following command to verify if the configuration was done correctly:

```
cat faucet/inst/faucet.log
```

In this tutorial, we learned how to implement switching in a L2 network with Faucet, routing between multiple L3 networks and ACLs, by adding and modifying control rules.

2.1.2 Switching

We started by configurating a switch with 5 ports, each having a VLAN associated (100 to ports 1, 2 and 3, 200 to ports 4 and 5), as well as a "bridge" br0, with simulated ports to it named p1 through p5, to connecto to the Faucet controller.

```
dps:
    switch-1:
        dp_id: 0x1
        timeout: 7201
        arp_neighbor_timeout: 3600
        interfaces:
            1:
                native_vlan: 100
            2:
                native_vlan: 100
            3:
                native_vlan: 100
            4:
                native_vlan: 200
            5:
                native_vlan: 200
vlans:
    100:
    200:
```

One problem we found was that the timeout value needed to be greater than twice the *arp_neighbor_timeout* value, hence the 7201 value used instead of 3600 recommended in the tutorial.

Looking at the OpenFlow tables, we learned that those tables handle packets differently having in mind the type of packet, source and destination, as well as VLAN (e.g. some tables drop inappropriate packets, other pushes those packets to the VLANs configured for the ports).

OpenFlow also lets us track the specific path a particular packet would take through Open vSwitch, coming to the conclusion that information follows the numbered tables. We also saw how packets are sent to the controller to trigger MAC learning.

2.1.3 Routing

To implement routing, all we needed to do was to change the configuration (faucet.yaml), in the vlans section, to specify a router OP address for each VLAN, defining a router between them:

```
dps:
    switch-1:
        dp_id: 0x1
        timeout: 7201
        arp_neighbor_timeout: 3600
        interfaces:
            1:
                native_vlan: 100
            2:
                native_vlan: 100
            3:
                native_vlan: 100
            4:
                native_vlan: 200
            5:
                native_vlan: 200
vlans:
    100:
        faucet_vips: ["10.100.0.254/24"]
    200:
        faucet_vips: ["10.200.0.254/24"]
routers:
    router-1:
        vlans: [100, 200]
```

Looking into the OpenFlow Layer, we acquired the understanding that we got some additional OpenFlow tables, **ipv4_fib** and **vip**, where the first one creates flows for verifying that the packets are indeed addressed to a network IP that Faucet knows how to route. The latter one indicates a few different things, namely sending ARP requests for the router IPs and switching other ARP packets, broadcasting or unicasting them.

By tracing the packets, we can learn the path/steps the packets take when two hosts have not been in communication recently:

1. The host sends an ARP request to the router
2. The router sends ARP reply to the host
3. The host then sends an IP packet via the router's ethernet address
4. The router broadcasts an ARP request to the ports that carry the vlan network
5. The host sends an ARP reply to the router
6. Finally, the router either sends the IP packet to the host or times out

2.1.4 ACLs

To end this tutorial, we set up an ACL, where Faucet converts it to fit into the OpenFlow pipeline:

```
(...)
routers:
    router-1:
        vlans: [100, 200]
acls:
    1:
        - rule:
            dl_type: 0x800
            nw_proto: 6
            tcp_dst: 8080
            actions:
                allow: 0
        - rule:
            actions:
                allow: 1
```

The first rule states that if the packet is an IPv4 packet, with the TCP protocol and destination port as 8080, it gets blocked. The second states that for all the other packets, the action to be taken is to allow the packet. This results in Faucet creating a new table to the start of the pipeline to process this ACL.

2.2 IPSec

2.2.1 Overview

In this tutorial we acquired the understanding of the fundamentals to run an IPSec tunnel in Open vSwitch, allowing two networks or devices to securely communicate with each other by encapsulating the data in a secure, encrypted tunnel.

2.2.2 IPsec tunnel configuration

In this case, we built an IPSec tunnel between two hosts by first setting up an OVS bridge in both of them, as well as the tunnel itself, which can be configured with 3 different authentication methods: using a pre-shared key, a self-signed certificate or CA-signed certificate.

To create the secure connection with pre-shared key we simply need to set this command on each host:

host_1:

```
ovs-vsctl add-port br-ipsec tun -- \
            set interface tun type=gre \
                          options:remote_ip=$ip_host_2 \
                          options:psk=swordfish
```

host_2:

```
ovs-vsctl add-port br-ipsec tun -- \
            set interface tun type=gre \
                          options:remote_ip=$ip_host_1 \
                          options:psk=swordfish
```

This the simplest way to do this as we only need to check that both pre-shared keys are equal.

To create the connection with a self-signed certificate we first need to create the certificate, and then exchange the certificates

```
ovs-pki req -u host_x
ovs-pki self-sign host_x
scp host_x-cert.pem $ip_y:/etc/keys/host_x-cert.pem
```

After that we can setup the configuration.

```
ovs-vsctl set Open_vSwitch . \
           other_config:certificate=/etc/keys/host_x-cert.pem \
           other_config:private_key=/etc/keys/host_x-privkey.pem
```

To finalize the tunnel setup we need to run the following command on each host

```
ovs-vsctl set Interface tun options:local_ip=$ip_x
```

Now that the secure connection was setted up we now had to figure out if the secure connection was working, for that, we generated a ping with a pattern from one host to another

```
ping -p A $ip_2
```

When capturing traffic on host_2 using tcpdump we see indeed that the data inside de ICMP Packet does not contain the pattern, to double check we followed the same procedure without using the secure connection and saw that the pattern is present.

SERA QUE REFAÇO OS PASSOS E METO AQUI PRINTS?

We encountered two problems while following the troubleshooting part of this tutorial. One of them was running the following command:

```
ovs-appctl -t ovs-monitor-ipsec tunnels/show
```

Which supposedly showed the internal representation of the tunnel configuration, but for us it only returned an error, after some debugging with no solution we continued following the tutorials that we had in mind.

2.3 Conntrack

We tried doing the OVS Conntrack tutorial, however we ran into problems. While setting up the network, right from the beginning, it wasn't specified anywhere that we needed to change the state of the interfaces to 'up'.

We figured out that we need to run

```
sudo ip netns exec left sudo ip link set veth_l1 up
sudo ip netns exec right sudo ip link set veth_r1 up
```

If we didn't the do this, an error would appear when running the required commands on scapy, stating that the network was down.

Even after fixing that, we still couldn't get any connection between the hosts, so we assumed it was a gateway error. We couldn't fix the problem, and even signed up to the emailing list and sent an email to the OVS organization discuss mailing list, but got no response.

2.4 Advanced Features

The last tutorial we completed to make sure we had a good understanding of the possibilities of the OVS was the *Advanced features*, which cover aspects of admission control, VLAN input and output processing, MAC+VLAN learning for ingress port and, finally, look up destination ports.

The goal of this tutorial was to demonstrate the power of Open vSwitc flow tables, which works through the implementation of a MAC-learning switch with VLAN trunk and access ports.

The first table implemented was the *Admission Control table*, where the packets enter the switch and are discarded for specific reasons (e.g. packets with a multicast source address are not valid). In this table, we have the freedom to add flows to drop other protocols.

The second table implemented was the *VLAN Input Processing table*. The purpose of this table (table 1) is to validate the packet's VLAN, based on the VLAN configuration of the switch port through which the packet entered the switch. We started by adding a low-priority flow that drops all the packets, like a 'default drop' flow. On trunk ports, we accept any packets and resubmits them to the next table. Finally, on access ports, we just want to accept packets that don't have any VLAN header, then tag them with the access port's VLAN number and pass it along to the next stage.

To learn that the packet's source MAC is located on the packet's ingress port, we implemented a third table, the *MAC+VLAN Learning for Ingress Port table*. This table is a good example why the table 1 added a VLAN tag to packets that entered the switch through an access port, because we want to associate a MAC+VLAN with a port. To do this, we use the "learn" action, an Open vSwitch extension to OpenFlow), that modifies a table (in this case, table 10) based on the content of the flow currently being processed.

The next table (table 3) implemented was the *Look Up Destination Port table*, which figures out what port we should send the packet to based on the destination MAC and VLAN: That is, if we've learned the location of the destination (from table 2), then we want to send the packet there. To do this, we first send the packet to the table 10, the table that the "learn" modifies, as stated before. If the destination for our packet hasn't been learned, there will be no matching flow, and an internal register 0 will have the value 0 instead of the correct port, and the packet will be flooded.

The final table we established, the *Output Processing table*, just needs to output the packets. We know that the internal register 0 contains either the desired output port or is zero if the packet should be flooded. For the output to the access ports, we just have to strip the VLAN header before outputting the packet. If the packet needs to be flooded, due to unlearned destinations, we make sure the packets contain the 802.1Q header when going to the trunk port.

This tutorial gave us some new insights regarding how tables affect the flows of the packets, as well as how an implementation of a switch with MAC-learning and VLAN trunk and access ports.

Chapter 3

Faucet

To deepen our knowledge about the use of the SDN Faucet controller, together with the Open vSwitch, we created a set of scenarios to explore its usefulness in VLANs, as well as the functionality and advantages of using ACLs to control traffic within a network or apply security policies, as well as other features. In this chapter we will also demonstrate our entire process in developing an SDN App, including all the setup details and problems faced.

Some important commands that are usefull to run these setups: Docker build:

```
sudo docker build -t faucet/faucet -f Dockerfile.faucet .
```

Docker Run:

```
sudo docker run -d --name faucet --restart=always -v $(pwd)/inst
    /:/etc/faucet/ -v $(pwd)/inst/:/var/log/faucet/ -p 6653:6653
    -p 9302:9302 faucet/faucet
```

Refreshing the configuration (hot reload):

```
sudo docker exec faucet pkill -HUP -f faucet.faucet
```

Checking if the configuration was correctly updated:

```
cat faucet/inst/faucet.log
```

3.1 VLANs

To explore the topic of VLANs, we created a network with the use of Mininet, which is an emulator of a network and used to visualize the switches and application of the software-defined networks in a virtualized environment. With that, we created the followind network topology, which will be used to explain everything we did in this section:

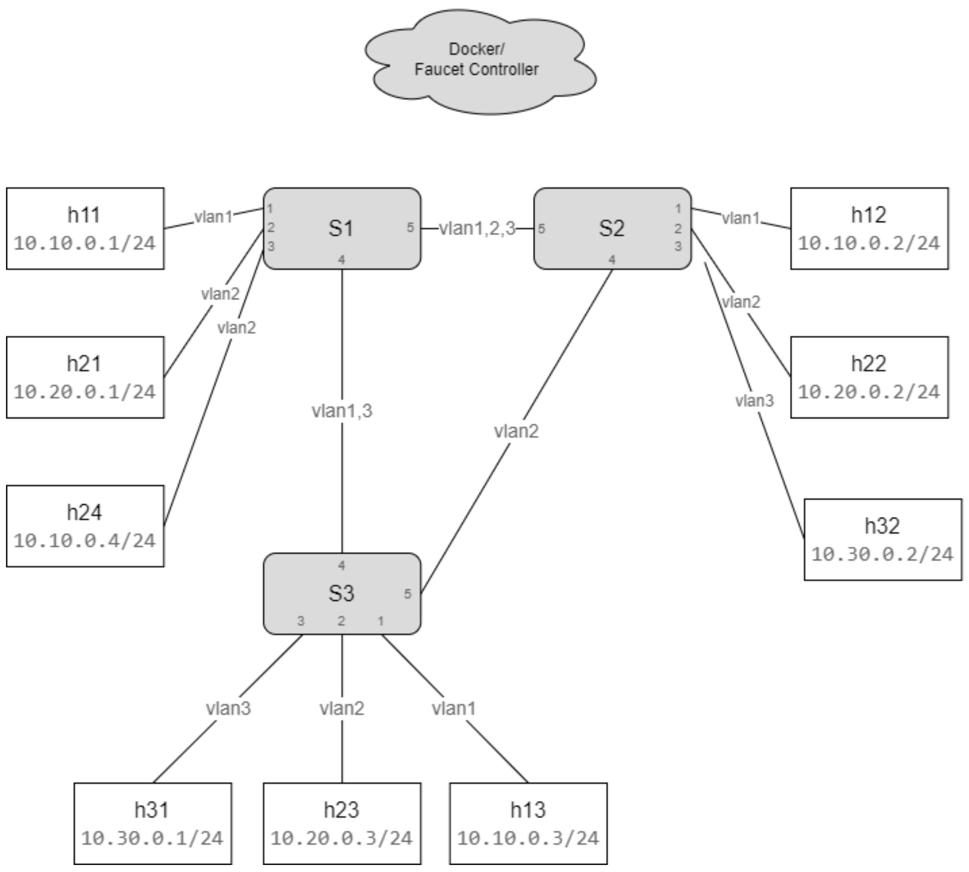


Figure 3.1: Network topoly for the VLAN topic exploration

Taking into account the topology illustrated above, our objective was to explore the practicality of using the controller to define VLANs on switches, understand how packets traveled the indicated paths, what happened when networks were misconfigured and criticize the ease of developing such a network.

We intentionally made decisions in the design of this topology to test these different scenarios. The IP defined in host h24 is incorrect because it does not go according to the VLAN in which it is present. We also define trunk ports on the switches so that, for example, a ping from host h13 to h12 has to take the longest path through all 3 switches, to test the integrity and correctness of the setup.

The configuration file *faucet.yaml* for the network is the following, which includes the detailed configuration of the switches, and their relative trunk ports and connections, as well as the VLANs and their IPs:

```
dps:
  s1:
   description: s1
   dp_id: 0x0000000000000001
   interfaces:
      1:
          description: Connection s1 to h11
          native_vlan: 10
      2:
          description: Connection s1 to h21
          native_vlan: 20
      3:
          description: Connection s1 to h24
          native_vlan: 20
      4:
          description: Connection s1 to s3
          tagged_vlans: [10, 30]
      5:
          description: Connection s1 to s2
          tagged_vlans: [10, 20, 30]

  s2:
    description: s2
    dp_id: 0x0000000000000002
    interfaces:
       1:
         description: Connection s2 to h12
         native_vlan: 10
       2:
         description: Connection s2 to h22
         native_vlan: 20
       3:
         description: Connection s2 to h32
         native_vlan: 30
       4:
         description: Connection s2 to s3
         native_vlan: 20
       5:
         description: Connection s2 to s1
         tagged_vlans: [10, 20, 30]

  s3:
    description: s3
    dp_id: 0x0000000000000003
    interfaces:
      1:
        description: Connection s3 to h13
        native_vlan: 10
      2:
        description: Connection s3 to h23
        native_vlan: 20
      3:
        description: Connection s3 to h31
        native_vlan: 30
      4:
        description: Connection s3 to s1
        tagged_vlans: [10, 30]
      5:
        description: Connection s3 to s2
        native_vlan: 20

vlans:
  10:
    name: VLAN10
    faucet_vips: ["10.10.0.254/24"]
  20:
    name: VLAN20
    faucet_vips: ["10.20.0.254/24"]
  30:
    name: VLAN30
    faucet_vips: ["10.30.0.254/24"]

routers:
  router-1:
    vlans: [10, 20, 30]
```

From the configuration above, the main information displayed is the configuration of the switches with their relative connections and VLANs associated with those connections. We also create each VLAN, with their number and associated network.

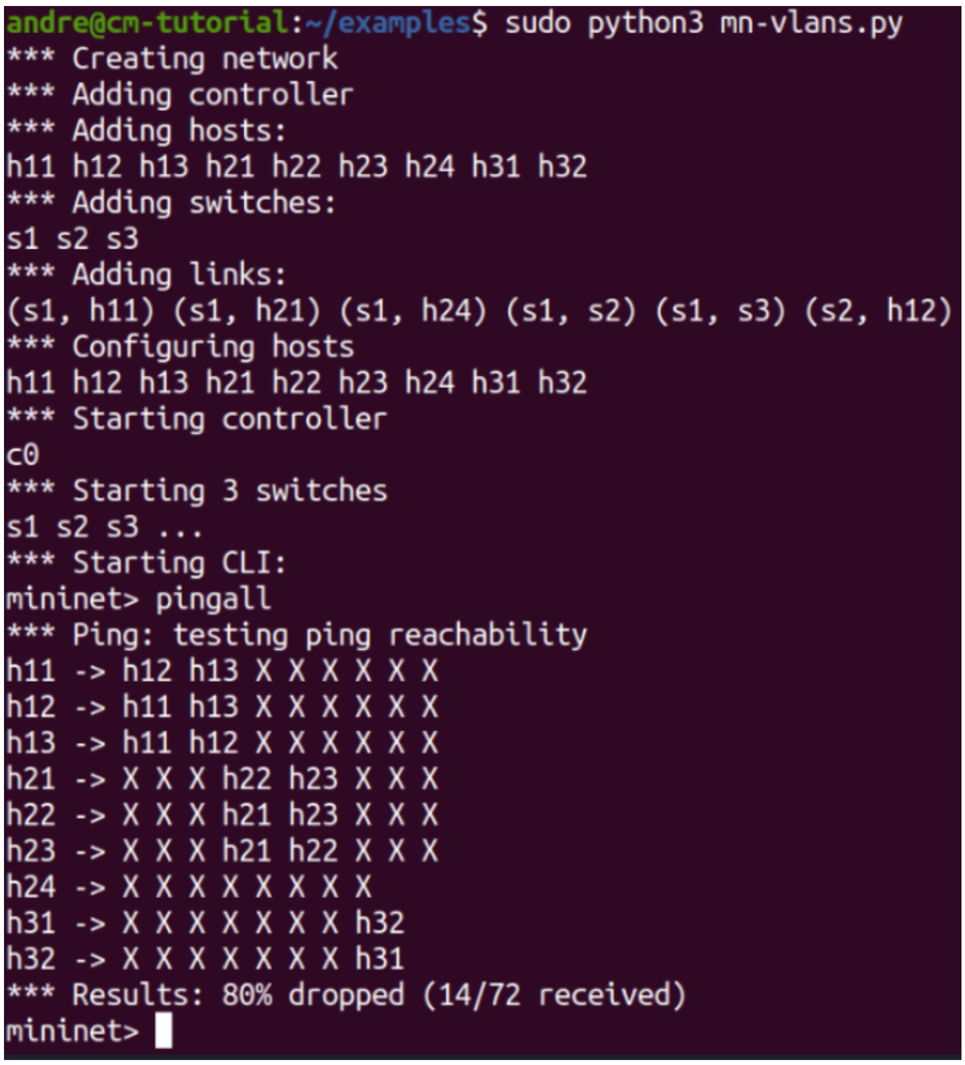
Now, to test this setup we created a mininet script (mn-vlans.py) where we create the hosts and switches defined in the topology with the respective IPs and connections.

```python
from mininet.net import Mininet
from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.node import Host, RemoteController, OVSSwitch
from mininet.topo import Topo

class TopoExample(Topo):
    def __init__(self, *args, **kwargs):
        Topo.__init__(self, *args, **kwargs)

        # creating switches
        s1 = self.addSwitch('s1', dpid='1')
        s2 = self.addSwitch('s2', dpid='2')
        s3 = self.addSwitch('s3', dpid='3')

        # creagting hosts
        h11 = self.addHost('h11', ip='10.10.0.1/24')
        h12 = self.addHost('h12', ip='10.10.0.2/24')
        h13 = self.addHost('h13', ip='10.10.0.3/24')
        h21 = self.addHost('h21', ip='10.20.0.1/24')
        h22 = self.addHost('h22', ip='10.20.0.2/24')
        h23 = self.addHost('h23', ip='10.20.0.3/24')
        # wrong ip for h24 using vlan10 address range
        h24 = self.addHost('h24', ip='10.10.0.4/24')
        h31 = self.addHost('h31', ip='10.30.0.1/24')
        h32 = self.addHost('h32', ip='10.30.0.2/24')

        # Connections for s1
        self.addLink(s1, h11, port1=1, port2=1)
        self.addLink(s1, h21, port1=2, port2=1)
        self.addLink(s1, h24, port1=3, port2=1)

        # Connections for s2
        self.addLink(s2, h12, port1=1, port2=1)
        self.addLink(s2, h22, port1=2, port2=1)
        self.addLink(s2, h32, port1=3, port2=1)

        # Connections for s3
        self.addLink(s3, h13, port1=1, port2=1)
        self.addLink(s3, h23, port1=2, port2=1)
        self.addLink(s3, h31, port1=3, port2=1)

        # Connections between switches
        self.addLink(s1, s2, port1=5, port2=5)
        self.addLink(s1, s3, port1=4, port2=4)
        self.addLink(s2, s3, port1=4, port2=5)


def main():
    net = Mininet(topo=TopoExample(), switch=OVSSwitch, cleanup=
    True,
    controller=RemoteController('c0', ip='localhost', port=6653,
    protocols="OpenFlow13"))

    net.start()
    CLI(net)
    net.stop()


if __name__ == "__main__":
    setLogLevel('info')
    main()
```

To test that our VLANs are configured correctly we can run the following command on the mininet CLI:

```
mininet> pingall
```

From that we get the following result:



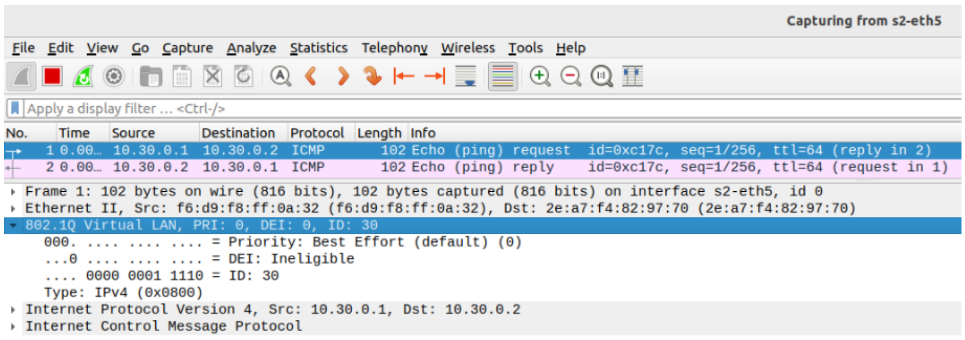
```
andre@cm-tutorial:~/examples$ sudo python3 mn-vlans.py
*** Creating network
*** Adding controller
*** Adding hosts:
h11 h12 h13 h21 h22 h23 h24 h31 h32
*** Adding switches:
s1 s2 s3
*** Adding links:
(s1, h11) (s1, h21) (s1, h24) (s1, s2) (s1, s3) (s2, h12)
*** Configuring hosts
h11 h12 h13 h21 h22 h23 h24 h31 h32
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h11 -> h12 h13 X X X X X X
h12 -> h11 h13 X X X X X X
h13 -> h11 h12 X X X X X X
h21 -> X X X h22 h23 X X X
h22 -> X X X h21 h23 X X X
h23 -> X X X h21 h22 X X X
h24 -> X X X X X X X X
h31 -> X X X X X X X h32
h32 -> X X X X X X X h31
*** Results: 80% dropped (14/72 received)
mininet> █
```

Figure 3.2: Usage of the *pingall* command and respective results

We can see that the results got are to be expected, the hosts from VLAN10 (h11, h12, h13), can only ping among themselves, and the same happens for the hosts in VLAN20 and VLAN30. The only thing noted is that h24 does not have connectivity with any host, this is to be expected because h24 has the wrong IP, the port connected to h24 has a native vlan 20, and it's IP is from the VLAN10 network.

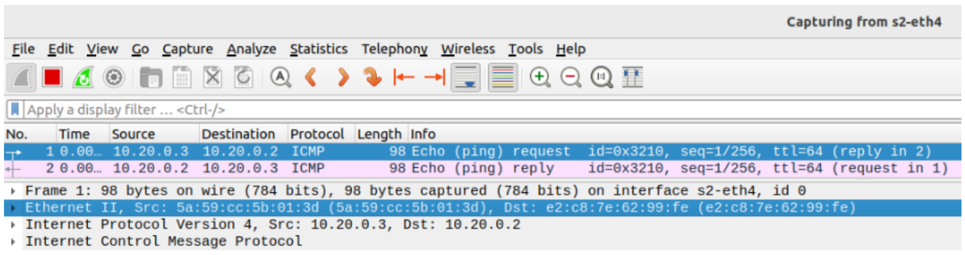
To gauge the effectiveness of the implementation, we captured packets in two different connections: one on the port 5 of the switch 2 and the other on the port 4 of that same switch and performed the following actions, firstly we pinged h32 from h31, and secondly we pinged h23 from h22.

These were the packets captured in the interfaces


Capturing from s2-eth5
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time   Source      Destination Protocol Length Info
   1 0.00…  10.30.0.1   10.30.0.2   ICMP     102 Echo (ping) request  id=0xc17c, seq=1/256, ttl=64 (reply in 2)
   2 0.00…  10.30.0.2   10.30.0.1   ICMP     102 Echo (ping) reply    id=0xc17c, seq=1/256, ttl=64 (request in 1)

Frame 1: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface s2-eth5, id 0
Ethernet II, Src: f6:d9:f8:ff:0a:32 (f6:d9:f8:ff:0a:32), Dst: 2e:a7:f4:82:97:70 (2e:a7:f4:82:97:70)
802.1Q Virtual LAN, PRI: 0, DEI: 0, ID: 30
   000. .... .... .... = Priority: Best Effort (default) (0)
   ...0 .... .... .... = DEI: Ineligible
   .... 0000 0001 1110 = ID: 30
   Type: IPv4 (0x0800)
Internet Protocol Version 4, Src: 10.30.0.1, Dst: 10.30.0.2
Internet Control Message Protocol
```

Figure 3.3: Capturing packets in port 5 of switch 2


Capturing from s2-eth4
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time   Source      Destination Protocol Length Info
   1 0.00…  10.20.0.3   10.20.0.2   ICMP      98 Echo (ping) request  id=0x3210, seq=1/256, ttl=64 (reply in 2)
   2 0.00…  10.20.0.2   10.20.0.3   ICMP      98 Echo (ping) reply    id=0x3210, seq=1/256, ttl=64 (request in 1)

Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface s2-eth4, id 0
Ethernet II, Src: 5a:59:cc:5b:01:3d (5a:59:cc:5b:01:3d), Dst: e2:c8:7e:62:99:fe (e2:c8:7e:62:99:fe)
Internet Protocol Version 4, Src: 10.20.0.3, Dst: 10.20.0.2
Internet Control Message Protocol
```

Figure 3.4: Capturing packets in port 4 of switch 2

We can see two different things here, firstly, we can see that the ping packet from h31 to h32 goes through switch 1 before arriving to h32, which can be observed because the ping is captured on s2-eth5 that is connected to switch 1. We can also see that ping from h22 to h23 goes directly to switch 2 because the packet is captured on s2-eth4 and that port is connected directly to switch 3.

Secondly, we see that the links that are used as trunk ports (like the link between switch 1 and switch 2) add a Virtual LAN tag on the ping packet, this is probably because the faucet.yaml defines that as using ”tagged_vlans”, on the link with only one vlan (like the link between switch 3 and switch 2), no tag is added, on the configuration file we define this as ”native_vlan”.

3.2 ACLs

ACLs, or access control lists, are used to specify what traffic is allowed or denied on a network. In the context of this project, ACLs are used to define rules for how traffic should be handled within a network. These rules can be based on various criteria, such as the source or destination IP address, the protocol being used, or the port number. Moreover, they can be used to secure a network by blocking traffic in various ways.

We started by updating the topology used in the previous section with the introduction of a new switch (switch 4) as well as 4 new hosts to try out the developed ACLs:

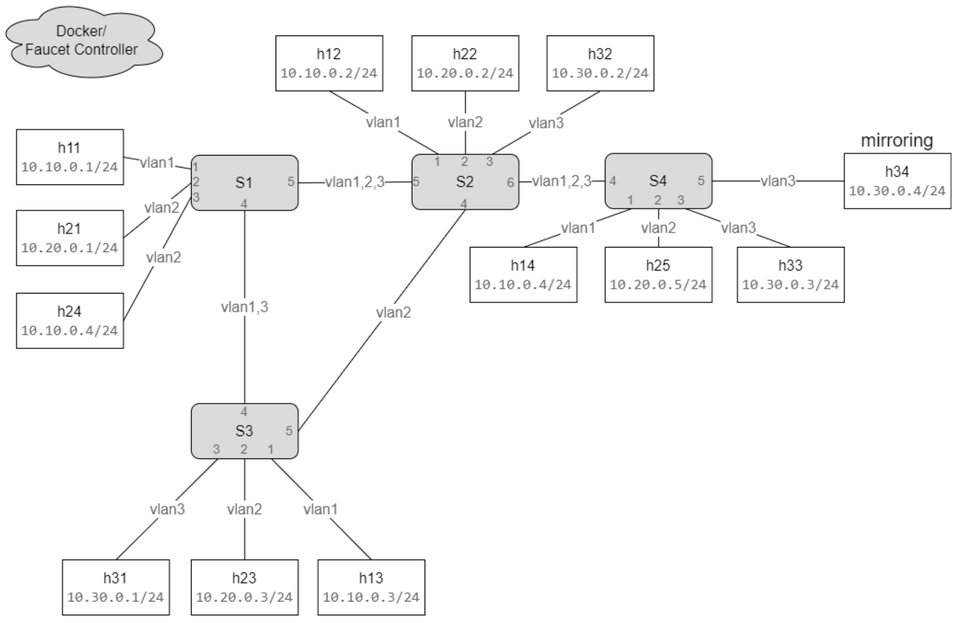


Figure 3.5: Network topoly for the ACLs topic exploration

The switch 4 will be the switch that contains the ACLs we felt important to implement and fully test. First, we implemented an access control list that blocks pings coming to and from the host h14 (connected to the port where this ACL will be deployed):

```
block-ping:
    - rule:
        dl-type: 0x800
        ip_proto: 1
        actions:
            allow: False
            mirror: 5
    - rule:
        actions:
            allow: true
```

To better understand what was written above, we named this ACL ”block_ping” and defined two rules. The first one starts by filtering packets whose *dl-type* is 0x800 (IPv4) and *ip_proto* is 1 (ICMP) and taking a certain action, in this case it blocks that packet from going to the specified port and mirrors it to the port number 5. The second rule can be seen as a default rule, where all the other packets can go through without any other action.

The second ACL we deployed on that switch blocks all connections to and from the network 10.0.20.0/24, which we considered being a malicious network:

```
block-ip:
    - rule:
        dl-type: 0x800
        ipv4_src: 10.20.0.0/24
        actions:
            allow: False
            mirror: 5
    - rule:
        actions:
            allow: true
```

Here, we block all IPv4 packets to and from the network 10.20.0.0/24, mirroring them to the port 5 as well and also setting a default rule to allow all the other packets go through the port to that network.

We also created two other ACLs, however these ones weren’t fully tested in our network:

```
priority-traffic:
    - rule:
        nw_src: 1.2.3.4
        actions:
            priority: 100
```

```
block-p2p:
    - rule:
        dl_type: 0x800
        id_proto: 6
        tp_dst: [6881, 6889]
        actions:
            allow: False
    -rule:
        dl_type: 0x800
        id_proto: 17
        tp_dst: [4662]
        actions:
            allow: False
     -rule:
        actions:
            allow: True
```

The first rule, as the name suggests, prioritizes the traffic of a specific IP (QoS) when going through that switch. The second rule blocks peer-to-peer communications to go through that switch by analysing the ports of the packets, as well as the protocol, and blocks them.

To deploy these ACLs into the switch 4 in the network, we edited the *faucet.yaml* by adding the following lines:

```
include:
  - acls.yaml

vlans:
  10:
    name: VLAN10
    faucet_vips: ["10.10.0.254/24"]
  20:
    name: VLAN20
    faucet_vips: ["10.20.0.254/24"]
  30:
    name: VLAN30
    faucet_vips: ["10.30.0.254/24"]

routers:
  router-1:
    vlans: [10, 20, 30]

dps:
  s1:
   description: s1
   dp_id: 0x0000000000000001
   interfaces:
      1:
          description: Connection s1 to h11
          native_vlan: 10
      2:
          description: Connection s1 to h21
          native_vlan: 20
      3:
          description: Connection s1 to h24
          native_vlan: 20
      4:
          description: Connection s1 to s3
          tagged_vlans: [10, 30]
      5:
          description: Connection s1 to s2
          tagged_vlans: [10, 20, 30]

  s2:
    description: s2
    dp_id: 0x0000000000000002
    interfaces:
       1:
         description: Connection s2 to h12
         native_vlan: 10
       2:
         description: Connection s2 to h22
         native_vlan: 20
       3:
         description: Connection s2 to h32
         native_vlan: 30
       4:
         description: Connection s2 to s3
         native_vlan: 20
       5:
         description: Connection s2 to s1
         tagged_vlans: [10, 20, 30]
       6:
         description: Connection s2 to s4
         tagged_vlans: [10, 20, 30]

  s3:
    description: s3
    dp_id: 0x0000000000000003
    interfaces:
      1:
        description: Connection s3 to h13
        native_vlan: 10
      2:
        description: Connection s3 to h23
        native_vlan: 20
      3:
        description: Connection s3 to h31
        native_vlan: 30
      4:
        description: Connection s3 to s1
        tagged_vlans: [10, 30]
      5:
        description: Connection s3 to s2
        native_vlan: 20

  # Switch used to test ACLs
  s4:
    description: s4
    dp_id: 0x0000000000000004
    interfaces:
      1:
        description: Connection s4 to h14
        native_vlan: 10
        acls_in: [block-ping]
      2:
        description: Connection s4 to h25
        native_vlan: 20
        acls_in: [block-ip]
      3:
        description: Connection s4 to h33
        native_vlan: 30
      4:
        description: Connection s4 to s2
        tagged_vlans: [10, 20, 30]
      5:
        description: Connection s4 to h34
        native_vlan: 30
```

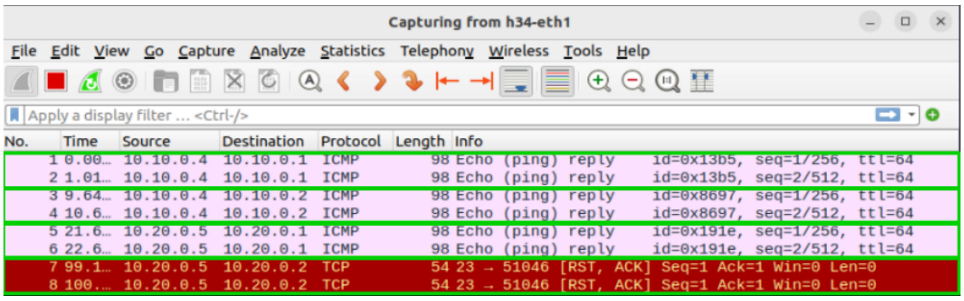
Note that in the beggining of the faucet.yaml file we include the acls.yaml file, this eases the process of debug and is easier for the programmer for the acls definition, this was the content of acls.yaml.

```
acls:
  block-ping:
    - rule:
      dl_type: 0x800      # IPv4
      ip_proto: 1         # ICMP
      actions:
        allow: False
        mirror: 5
    - rule:
      actions:
        allow: True

  block-ip:
    - rule:
      dl_type: 0x800
      ipv4_src: 10.20.0.0/24
      actions:
        allow: False
        mirror: 5
    - rule:
      actions:
        allow: True

# priority-traffic:
#   - rule:
#     nw_src: 1.2.3.4
#     actions:
#       priority: 100
# block-p2p:
#   - rule:
#     dl_type: 0x800
#     ip_proto: 6
#     tp_dst: [6881, 6889]
#     actions:
#       allow: False
#    - rule:
#      dl_type: 0x800
#      ip_proto: 17
#      tp_dst: [4662]
#      actions:
#        allow: False
#     - rule:
#       actions:
#         allow: True
```

Now, to test this setup we created a mininet script (mn-acls.py) where we create the hosts and switches defined in the topology with the respective IPs and connections.

```python
from mininet.net import Mininet
from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.node import Host, RemoteController, OVSSwitch
from mininet.topo import Topo

class TopoExample(Topo):
    def __init__(self, *args, **kwargs):
        Topo.__init__(self, *args, **kwargs)

        # creating switches
        s1 = self.addSwitch('s1', dpid='1')
        s2 = self.addSwitch('s2', dpid='2')
        s3 = self.addSwitch('s3', dpid='3')
        s4 = self.addSwitch('s4', dpid='4')

        # creagting hosts
        h11 = self.addHost('h11', ip='10.10.0.1/24')
        h12 = self.addHost('h12', ip='10.10.0.2/24')
        h13 = self.addHost('h13', ip='10.10.0.3/24')
        h14 = self.addHost('h14', ip='10.10.0.4/24')
        h21 = self.addHost('h21', ip='10.20.0.1/24')
        h22 = self.addHost('h22', ip='10.20.0.2/24')
        h23 = self.addHost('h23', ip='10.20.0.3/24')
        # wrong ip for h24 using vlan10 address range
        h24 = self.addHost('h24', ip='10.10.0.4/24')
        h25 = self.addHost('h25', ip='10.20.0.5/24')
        h31 = self.addHost('h31', ip='10.30.0.1/24')
        h32 = self.addHost('h32', ip='10.30.0.2/24')
        h33 = self.addHost('h33', ip='10.30.0.3/24')
        h34 = self.addHost('h34', ip='10.30.0.4/24')

        # Connections for s1
        self.addLink(s1, h11, port1=1, port2=1)
        self.addLink(s1, h21, port1=2, port2=1)
        self.addLink(s1, h24, port1=3, port2=1)

        # Connections for s2
        self.addLink(s2, h12, port1=1, port2=1)
        self.addLink(s2, h22, port1=2, port2=1)
        self.addLink(s2, h32, port1=3, port2=1)

        # Connections for s3
        self.addLink(s3, h13, port1=1, port2=1)
        self.addLink(s3, h23, port1=2, port2=1)
        self.addLink(s3, h31, port1=3, port2=1)

        # Connections for s4
        self.addLink(s4, h14, port1=1, port2=1)
        self.addLink(s4, h25, port1=2, port2=1)
        self.addLink(s4, h33, port1=3, port2=1)
        self.addLink(s4, h34, port1=5, port2=1)

        # Connections between switches
        self.addLink(s1, s2, port1=5, port2=5)
        self.addLink(s1, s3, port1=4, port2=4)
        self.addLink(s2, s3, port1=4, port2=5)
        self.addLink(s2, s4, port1=6, port2=4)


def main():
    net = Mininet(topo=TopoExample(), switch=OVSSwitch, cleanup=
   True, controller=RemoteController('c0', ip='localhost', port
   =6653, protocols="OpenFlow13"))

    net.start()
    CLI(net)
    net.stop()


if __name__ == "__main__":
    setLogLevel('info')
    main()
```

In order to test this implementation to ensure it is working properly, we sent some pings to the host h14, sent pings and tcp packets to h25 and captured them in the connection between the switch 4 (port 5) and the host 34, the host that receives all the mirrored packets:



Capturing from h34-eth1
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
No.  Time    Source      Destination  Protocol  Length Info
   1 0.00…   10.10.0.4   10.10.0.1    ICMP      98 Echo (ping) reply    id=0x13b5, seq=1/256, ttl=64
   2 1.01…   10.10.0.4   10.10.0.1    ICMP      98 Echo (ping) reply    id=0x13b5, seq=2/512, ttl=64
   3 9.64…   10.10.0.4   10.10.0.2    ICMP      98 Echo (ping) reply    id=0x8697, seq=1/256, ttl=64
   4 10.6…   10.10.0.4   10.10.0.2    ICMP      98 Echo (ping) reply    id=0x8697, seq=2/512, ttl=64
   5 21.6…   10.20.0.5   10.20.0.1    ICMP      98 Echo (ping) reply    id=0x191e, seq=1/256, ttl=64
   6 22.6…   10.20.0.5   10.20.0.1    ICMP      98 Echo (ping) reply    id=0x191e, seq=2/512, ttl=64
   7 99.1…   10.20.0.5   10.20.0.2    TCP       54 23 → 51046 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
   8 100.…   10.20.0.5   10.20.0.2    TCP       54 23 → 51046 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
```

Figure 3.6: Captured packets to show the implemented ACLs

The first 2 captured packets show that the pings that go through the port that connects to the h14, in this case the port 1, are mirrored to the interface 5. We also tested the second ACL, which blocks all IPv4 traffic that goes to the network 10.20.0.0/24, and, as seen in the image, we tested pings (3rd pair of packets) and telnets (4th pair) to test the blockage of the different types of packets.

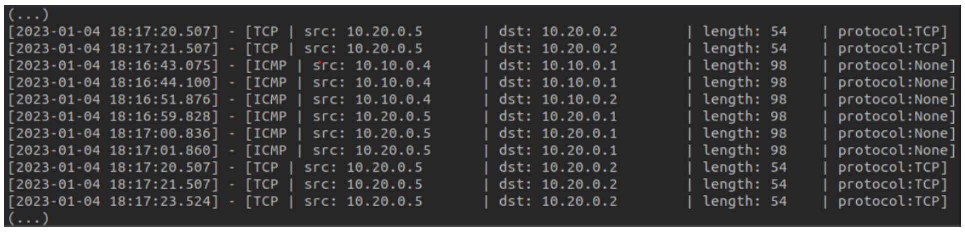
3.3 Monitorization

By taking advantage of the mirroring feature, which allows us to make copys of the packets that we blocked, we developed a Python program that sniffs all the packets that come to it's interface and log them into a file:

```python
import pyshark
#import to show timestamps
from datetime import datetime

iface_name = 'h34-eth1'
capture = pyshark.LiveCapture(
    interface=iface_name
)
#create "log.txt" file
file = open("log.txt", "w")
while True:
    capture.sniff(timeout=5, packet_count=1)
    if len(capture) > 0:
        for packet in capture:
            try:
                output = "["
                timestamp = datetime.fromtimestamp(float(packet.
    sniff_timestamp))
                output += timestamp.strftime('%Y-%m-%d %H:%M:%S
    .%f')[:-3]
                output += "] - ["
                #type of packet
                output += str(packet.highest_layer)
                output += " | src: "
                #source ip
                output += str(packet.ip.src
                output += " \t| dst: "
                #destination ip
                output += str(packet.ip.dst)
                output += " \t| length: "
                #length of packet
                output += str(packet.length)
                output += " \t| protocol:"
                #protocol
                output += str(packet.transport_layer)
                output += "]"
                #write to file
                print(output)
                file.write(output + "\n")
            except Exception as e:
                print(e)
                pass
```

The result of this logger can be seen in the file *log.txt* and a portion of it contains, for example, the following:

This logger can have several advantages by logging the traffic that is being redirected by the ACLs. From an administration perspective, this



```
(...)
[2023-01-04 18:17:20.507] - [TCP | src: 10.20.0.5      | dst: 10.20.0.2       | length: 54    | protocol:TCP]
[2023-01-04 18:17:21.507] - [TCP | src: 10.20.0.5      | dst: 10.20.0.2       | length: 54    | protocol:TCP]
[2023-01-04 18:16:43.075] - [ICMP | src: 10.10.0.4     | dst: 10.10.0.1       | length: 98    | protocol:None]
[2023-01-04 18:16:44.100] - [ICMP | src: 10.10.0.4     | dst: 10.10.0.1       | length: 98    | protocol:None]
[2023-01-04 18:16:51.876] - [ICMP | src: 10.10.0.4     | dst: 10.10.0.2       | length: 98    | protocol:None]
[2023-01-04 18:16:59.828] - [ICMP | src: 10.20.0.5     | dst: 10.20.0.1       | length: 98    | protocol:None]
[2023-01-04 18:17:00.836] - [ICMP | src: 10.20.0.5     | dst: 10.20.0.1       | length: 98    | protocol:None]
[2023-01-04 18:17:01.860] - [ICMP | src: 10.20.0.5     | dst: 10.20.0.1       | length: 98    | protocol:None]
[2023-01-04 18:17:20.507] - [TCP | src: 10.20.0.5      | dst: 10.20.0.2       | length: 54    | protocol:TCP]
[2023-01-04 18:17:21.507] - [TCP | src: 10.20.0.5      | dst: 10.20.0.2       | length: 54    | protocol:TCP]
[2023-01-04 18:17:23.524] - [TCP | src: 10.20.0.5      | dst: 10.20.0.2       | length: 54    | protocol:TCP]
(...)
```

Figure 3.7: Portion of contents of the *log.txt* file

logger can help you to keep track of network activity and identify trends or patterns over time, which can be useful for capacity planning or identifying potential bottlenecks in the blocked networks or host. It can also be useful from a security point of view, by monitoring suspicious or malicious activities (the network 10.20.0.0/24 was considered malicious and we can see the packets that have it as source/destination). It's also very helpful for troubleshooting issues that can arise on the network, not necessarily in our case, but with different ACLs can result in different goals for the logger.

3.4 SDN Application

A software-defined networking (SDN) application is a program that is designed to use and control a software-defined networking system. SDN technology allows a network administrator to manage network services through an abstracted high-level functionality, such as traffic management, rather than at the individual hardware level. This can make it easier to deploy and manage network services and can improve network flexibility and efficiency.

In order to implement a SDN app, we followed two approaches that are going to be explained on the next subsections.

3.4.1 SDN app using mininet

In this approach, we decided to implement, using the mininet script that defined the network, a trigger to drop packets that would be activated when the bandwith arriving a port of a host exceeded 50KB/s, as it can be seen in the code snippet's function *traffic_monitor* below.

```python
# Start by importing the necessary modules
from mininet.net import Mininet
from mininet.node import RemoteController, OVSSwitch
from mininet.cli import CLI
from mininet.log import lg, info, setLogLevel
import threading

# Create a Mininet object and add a Faucet controller
net = Mininet(switch=OVSSwitch)
```

```
c1 = net.addController(name='c1', controller=RemoteController,
    ip='localhost', port=6653)

# Add an OVS switch and connect it to the controller
s1 = net.addSwitch('s1')

# Add two hosts and connect them to the switch
h1 = net.addHost('h1')
h2 = net.addHost('h2')
net.addLink(h1, s1, port1=1, port2=1)
net.addLink(h2, s1, port1=1, port2=2)
setLogLevel('info')

# Start the Mininet network
net.start()

def traffic_monitor(h2):
    while True:
        traffic = h2.cmd('ifstat -i h2-eth1 1 1')
        try:
            traffic = float(traffic.split()[-2])
        except:
            traffic = 0
        # For debug purposes -> check that traffic as started
        if traffic > 1:
            info(traffic)
            info('\n')
        if traffic > 50:
            # For debug purposes -> check if it actually enters
    the if condition when the traffic exceeded 50KB/s
            info('happened\n')
            s1.cmd('ovs-ofctl add-flow s1 priority=15,in_port=1,
    actions=drop')

# Start the traffic monitor in a separate thread
thread = threading.Thread(target=traffic_monitor, args=(h2,))
thread.start()

# Enter the Mininet CLI to interact with the network
CLI(net)

# Stop the Mininet network
net.stop()
```

In order to keep the network being monitored and use mininet's CLI - that allows us to interact with the network, which is crucial to test if the SDN app is doing its purpose -, we needed to start the function traffic_monitor in a separated thread, which is probably the reason why it isn't working since, as a result of our research, mininet uses threading in its implementation and it may cause problems. However, there were a couple times where, even though we could see the print "happened" on the mininet CLI, the command "ovs-ofctl add-flow" wasn't executed. Even though we couldn't actually implement this application, we conclude that it would be a crucial part of SDN, since if the network was flexible in real time it would facilitate the network administration and control, e.g., when there's a client needing more resources/priority, more bandwith could be allocated to it.

In furtherance of implementing this script, we should use this yaml file as faucet.yaml, which was prepared to implement a flow that when the bandwith surpassed the 50KB/s the host VLAN's would change.

```
vlans:
vlan100:
    vid: 100
    description: "VLAN - Trigger"

dps:
    s1:
        dp_id: 0x1
        interfaces:
            1:
                name: "h1-eth0"
                native_vlan: vlan100
            2:
                name: "h2-eth0"
                native_vlan: vlan100
```

To test if this is working we can use the following commands on mininet CLI for a server: Start the *iperf* server and print on the terminal the bandwith:

```
mininet> xterm h1 h2
h2 terminal> iperf -s &
h2 terminal> ifstat -i h2-eth0
```

Start the *iperf* client with a bandwith of 1MB/s and during 15 seconds in order to create traffic:

```
h1 terminal> iperf -c 10.0.0.2 -b 1M -t 15
```

3.4.2 SDN app using Faucet's API

In this approach, even tough there isn't an actual API documentation, we found (a few hours - approximately 6 - before the project's presentation), via Faucet's Twitter page, that the code that could be useful to take advantage using Faucet's API, could be found somewhere else than on Faucet's Github `https://github.com/faucetsdn/faucet/tree/master/faucet`), where it is quite scattered between several files. Which is here `https://docs.faucet.nz/en/latest/source/apidoc/faucet.html#`, where it is more organized, even though it isn't a real API documentation.

However, we tried to implement a SDN application that would do an HTTP POST of an ACL change on the network that would change the VLAN 10 tag of sw1-eth1 to VLAN 1, even though it sent back a "200 OK" status code, it didn't really change the VLAN, since we could ping the host of VLAN 10. Furthermore, in order to get the network information of a specific port of a switch, given an ID and the port defined in the parameters with an HTTP POST and doing an HTTP GET to print that network information. It sent a "200 OK" status code, but it was only possible to print all of the information on API url `http://localhost:9302/api/v1/config`.

```python
from mininet.topo import Topo
from mininet.node import RemoteController, OVSSwitch
from mininet.net import Mininet
from mininet.cli import CLI
import requests

# Create a custom topology
class MyTopo(Topo):
    def __init__(self, *args, **kwargs):
        Topo.__init__(self, *args, **kwargs)

        # creating switches
        s1 = self.addSwitch('s1', dpid='1')
        s2 = self.addSwitch('s2', dpid='2')
        s3 = self.addSwitch('s3', dpid='3')
        s4 = self.addSwitch('s4', dpid='4')

        # creagting hosts
        h11 = self.addHost('h11', ip='10.10.0.1/24')
        h12 = self.addHost('h12', ip='10.10.0.2/24')
        h13 = self.addHost('h13', ip='10.10.0.3/24')
        h14 = self.addHost('h14', ip='10.10.0.4/24')
        h21 = self.addHost('h21', ip='10.20.0.1/24')
        h22 = self.addHost('h22', ip='10.20.0.2/24')
        h23 = self.addHost('h23', ip='10.20.0.3/24')
        # wrong ip for h24 using vlan10 address range
        h24 = self.addHost('h24', ip='10.20.0.4/24')
        h25 = self.addHost('h25', ip='10.20.0.5/24')
        h31 = self.addHost('h31', ip='10.30.0.1/24')
        h32 = self.addHost('h32', ip='10.30.0.2/24')
        h33 = self.addHost('h33', ip='10.30.0.3/24')
        h34 = self.addHost('h34', ip='10.30.0.4/24')

        # Connections for s1
        self.addLink(s1, h11, port1=1, port2=1)
        self.addLink(s1, h21, port1=2, port2=1)
        self.addLink(s1, h24, port1=3, port2=1)

        # Connections for s2
        self.addLink(s2, h12, port1=1, port2=1)
        self.addLink(s2, h22, port1=2, port2=1)
        self.addLink(s2, h32, port1=3, port2=1)

        # Connections for s3
        self.addLink(s3, h13, port1=1, port2=1)
        self.addLink(s3, h23, port1=2, port2=1)
        self.addLink(s3, h31, port1=3, port2=1)

        # Connections for s4
        self.addLink(s4, h14, port1=1, port2=1)
        self.addLink(s4, h25, port1=2, port2=1)
        self.addLink(s4, h33, port1=3, port2=1)
        self.addLink(s4, h34, port1=5, port2=1)

        # Connections between switches
        self.addLink(s1, s2, port1=5, port2=5)
        self.addLink(s1, s3, port1=4, port2=4)
        self.addLink(s2, s3, port1=4, port2=5)
        self.addLink(s2, s4, port1=6, port2=4)

def main():
    net = Mininet(topo=MyTopo(), switch=OVSSwitch, cleanup=True,
     controller=RemoteController('c0', ip='localhost', port=6653,
     protocols="OpenFlow13"))

    # Set the parameters for the traffic management
    configuration
    data = {
      "acls": [
        {
          "rule": {
            "actions": {
              "allow": 1
            },
            "dl_type": 2048,
            "nw_proto": 6
          },
          "name": "tcp_allow"
        }
      ],
      "dp_id": "0x1",
      "interfaces": [
        {
          "name": "eth1",
          "native_vlan": 1
        }
      ]
    }

    # Set the parameters for the traffic management
    configuration
    data2 = {
        "dp_id": "0x1",
        "interfaces": [
            {
            "name": "eth1",
            "native_vlan": 1
            }
        ]
    }

    api_url = "http://localhost:9302/api/v1/config"

    response = requests.post(api_url, json=data)
    print(response.status_code)

    response = requests.post(api_url, json=data2)
    print(response.status_code)

    api_url = "http://localhost:9302/api/v1/switches"
    params = {
        "dp_id": "0x1"
    }

    response = requests.get(api_url, params=params)
    print(response.text)
    print(response.status_code)

    net.start()

    CLI(net)

    net.stop()

if __name__ == "__main__":
    main()
```

In order to implement this script, we should use this yaml file as faucet.yaml

```
include:
  - acls.yaml

vlans:
  10:
    name: VLAN10
    faucet_vips: ["10.10.0.254/24"]
  20:
    name: VLAN20
    faucet_vips: ["10.20.0.254/24"]
  30:
    name: VLAN30
    faucet_vips: ["10.30.0.254/24"]

routers:
  router-1:
    vlans: [10, 20, 30]

dps:
  s1:
   description: s1
   dp_id: 0x0000000000000001
   interfaces:
      1:
          description: Connection s1 to h11
          native_vlan: 10
      2:
          description: Connection s1 to h21
          native_vlan: 20
      3:
          description: Connection s1 to h24
          native_vlan: 20
      4:
          description: Connection s1 to s3
          tagged_vlans: [10, 30]
      5:
          description: Connection s1 to s2
          tagged_vlans: [10, 20, 30]

  s2:
    description: s2
    dp_id: 0x0000000000000002
    interfaces:
       1:
         description: Connection s2 to h12
         native_vlan: 10
       2:
         description: Connection s2 to h22
         native_vlan: 20
       3:
         description: Connection s2 to h32
         native_vlan: 30
       4:
         description: Connection s2 to s3
         native_vlan: 20
       5:
         description: Connection s2 to s1
         tagged_vlans: [10, 20, 30]
       6:
         description: Connection s2 to s4
         tagged_vlans: [10, 20, 30]

  s3:
    description: s3
    dp_id: 0x0000000000000003
    interfaces:
      1:
        description: Connection s3 to h13
        native_vlan: 10
      2:
        description: Connection s3 to h23
        native_vlan: 20
      3:
        description: Connection s3 to h31
        native_vlan: 30
      4:
        description: Connection s3 to s1
        tagged_vlans: [10, 30]
      5:
        description: Connection s3 to s2
        native_vlan: 20

  # Switch used to test ACLs
  s4:
    description: s4
    dp_id: 0x0000000000000004
    interfaces:
      1:
        description: Connection s4 to h14
        native_vlan: 10
        acls_in: [block-ping]
      2:
        description: Connection s4 to h25
        native_vlan: 20
        acls_in: [block-ip]
      3:
        description: Connection s4 to h33
        native_vlan: 30
      4:
        description: Connection s4 to s2
        tagged_vlans: [10, 20, 30]
      5:
        description: Connection s4 to h34
        native_vlan: 30
```

Besides that, like it was explained on the ACL section, we should use this additional yaml file.

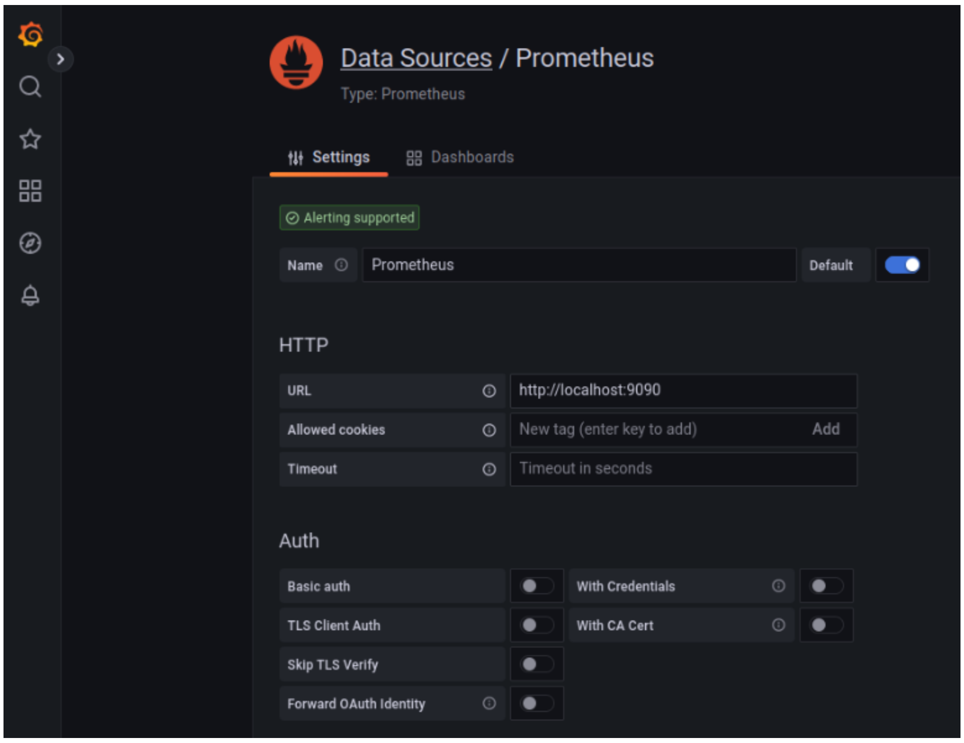
```
acls:
  block-ping:
    - rule:
      dl_type: 0x800      # IPv4
      ip_proto: 1         # ICMP
      actions:
        allow: False
        mirror: 5
    - rule:
      actions:
        allow: True

# Blocking ip from malicious IP
  block-ip:
    - rule:
      dl_type: 0x800
      ipv4_src: 10.20.0.0/24
      actions:
        allow: False
        mirror: 5
    - rule:
      actions:
        allow: True
```

3.5 Faucet Dashboard

At this point we had everything done and were trying to get a Dashboard of Faucet running and probably execute some commands remotely. With that in mind we thought it was best to do a new VM and follow the Faucet tutorial "Installing Faucet for the first time" `https://docs.faucet.nz/en/latest/tutorials/first_time.html`. We start by installing faucet-all-in-one, configuring prometheus and starting grafana.

```
sudo apt-get install faucet-all-in-one
(we can configure prometheus but we left it default)
sudo systemctl restart prometheus
sudo systemctl daemon-reload
sudo systemctl enable grafana-server
sudo systemctl start grafana-server
```

After this we accessed the grafana dashboard by going to `http://localhost:3000`, we logged in with the default username and password (both admin). To test that our grafana dashboard was working we added the prometheus as a data source.



Name: Prometheus Default: on

HTTP
URL: http://localhost:9090
Allowed cookies: New tag (enter key to add) Add
Timeout: Timeout in seconds

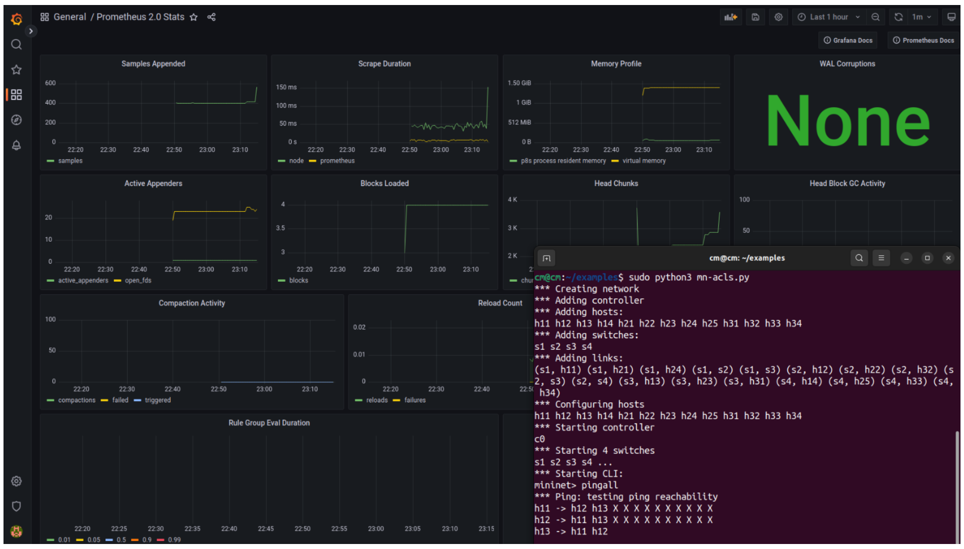
Auth
Basic auth: off
With Credentials: off
TLS Client Auth: off
With CA Cert: off
Skip TLS Verify: off
Forward OAuth Identity: off

Figure 3.8: Prometheus data source on Grafana

Prometheus is used in Faucet to provide monitoring and alerting capabilities for a network. In this context, Prometheus can be used to collect and store metrics data from the controller and the network devices, and to provide alerts when certain conditions are met.

Prometheus can be integrated with Faucet using an exporter, which is a piece of software that exposes the metrics data from Faucet in a format that Prometheus can scrape and store. Once the metrics data are collected, it can be visualized using Grafana .

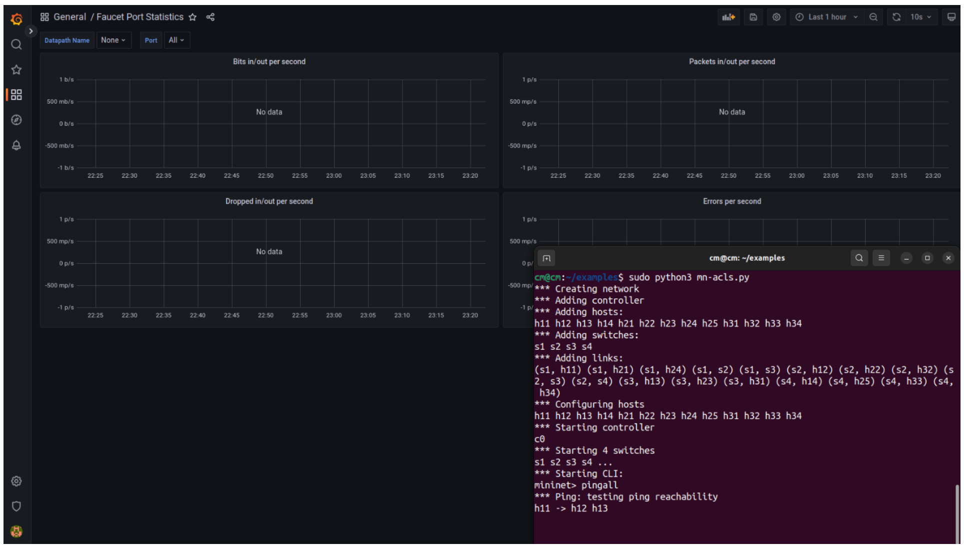
After that, we added two default dashboards that we can import from the grafana "library" for Prometheus, as we can see below prometheus is working while we run a test with Faucet (using the configuration files and scripts used for testing the ACLs)



```
cm@cm:~/examples$ sudo python3 mn-acls.py
*** Creating network
*** Adding controller
*** Adding hosts:
h11 h12 h13 h14 h21 h22 h23 h24 h25 h31 h32 h33 h34
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(s1, h11) (s1, h21) (s1, h24) (s1, s2) (s1, s3) (s2, h12) (s2, h22) (s2, h32) (s
2, s3) (s2, s4) (s3, h13) (s3, h23) (s3, h31) (s4, h14) (s4, h25) (s4, h33) (s4,
 h34)
*** Configuring hosts
h11 h12 h13 h14 h21 h22 h23 h24 h25 h31 h32 h33 h34
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h11 -> h12 h13 X X X X X X X X X X
h12 -> h11 h13 X X X X X X X X X X
h13 -> h11 h12
```

Figure 3.9: Prometheus 2.0 stats dashboard

We tried to use the dashboards that are provided in the Faucet tutorial and adding them to grafana dashboard but none of them worked, as we can see in the figure below, the added dashboard is always stating no data, and this behaviour is the same with the other dashboards provided.



```
cm@cm:~/examples$ sudo python3 mn-acls.py
*** Creating network
*** Adding controller
*** Adding hosts:
h11 h12 h13 h14 h21 h22 h23 h24 h25 h31 h32 h33 h34
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(s1, h11) (s1, h21) (s1, h24) (s1, s2) (s1, s3) (s2, h12) (s2, h22) (s2, h32) (s
2, s3) (s2, s4) (s3, h13) (s3, h23) (s3, h31) (s4, h14) (s4, h25) (s4, h33) (s4,
 h34)
*** Configuring hosts
h11 h12 h13 h14 h21 h22 h23 h24 h25 h31 h32 h33 h34
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h11 -> h12 h13
```

Figure 3.10: Faucet port statistics dashboard

These were the grafana dashboards tested:

1. `https://docs.faucet.nz/en/latest/_static/grafana-dashboards/faucet_instrumentation.json`
2. `https://docs.faucet.nz/en/latest/_static/grafana-dashboards/faucet_inventory.json`
3. `https://docs.faucet.nz/en/latest/_static/grafana-dashboards/faucet_port_statistics.json`

At this point we hitted a roadblock and since this was done just in the last moments of finishing the project we couldn't find a configuration for the dashboard that worked for us.

3.6 IPSec

We also tried to implement IPSec with the topology and configurations used on Faucet VLANs.

What we tried was, inside mininet, open xterm terminals for h11 and h12 and following the tutorial.

On host11 we executed the following commands:

```
ovs-vsctl set interface s1-eth1 type=gre \
                options:remote_ip=10.10.0.1 \
                options:psk=swordfish
ovs-vsctl set Interface s1-eth1 options:local_ip=10.10.0.1
```

And on host12 we executed the following commands:

```
ovs-vsctl set interface s2-eth1 type=gre \
                options:remote_ip=10.10.0.1 \
                options:psk=swordfish
ovs-vsctl set Interface s2-eth1 options:local_ip=10.10.0.2
```

From this, we either had no connection or, when we changed some of the settings and/or connections we had a connection that was not secure (using the ping with pattern method). After some debugging we found out that on the case where we got no connection the problem was that h11 was doing ARP requests to the network and no replies were received, this was a problem we couldn't fix due to the time constraint of this project.

3.7 GNS3

In addition to the use of mininet, we also tried to figure out (in the beggining of the project), on how to test our scenarios in GNS3.

Firstly we tied a list of Linux distributions that supported OpenV Switch and were light in requirements so we could instantiate multiple VMs.

We tried these distrbutions:

- Debian 11 - Lubuntu - Kubuntu - Xubuntu - Ubuntu Server - CentOS

Finally, we ended up choosing Ubuntu Server because it was easy and fast to instantiate, it only required 5GB of storage, 1GB of RAM and 1 CPU. Having familiarity with the distribution was also a factor in this process.

Secondly we created two templates of VMs, one for the controller and another for the switch. On the switch's VM we first install every package needed to have OpenV Switch on the machine, than we set up a bridge.

```
sudo ovs-vsctl add-br ovsbr0
```

Lastly, we need to create a script setup.sh with content below, this script will be used to execute commands on startup for configurations that are not saved when the machine is turned off.

```
#!/bin/bash

ifconfig ens3 up
ifconfig ens4 up
ifconfig ens5 up
ifconfig ens6 up
ifconfig ens7 up
ifconfig ens8 up

ovs-vsctl set-controller ovsbr0 tcp:192.168.122.43:6653
ovs-vsctl add-port ovsbr0 ens4
ovs-vsctl add-port ovsbr0 ens5
ovs-vsctl add-port ovsbr0 ens6
ovs-vsctl add-port ovsbr0 ens7
ovs-vsctl add-port ovsbr0 ens8
```

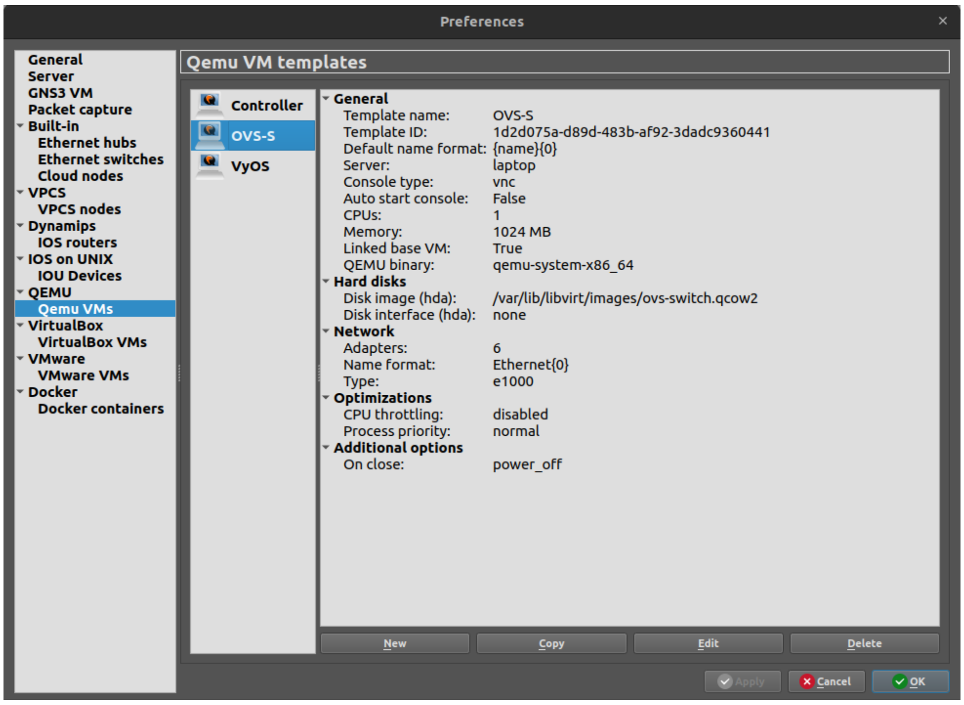
This configuration activates six network interfaces, this is done like this because our template on GNS3 will have six network adapters, than we define a controller on bridge "ovsbr0" and five ports that are on that bridge, this is because eth0, or in this case ens3 will be connected to controller. To execute this script on startup we create a file openvswitch-settings.service, inside the /etc/systemc/system directory, the content of openvswitch-settings.service follow below.

```
[Unit]
Description=OpenVSwitch Settings

[Service]
Type=oneshot
ExecStart=/bin/bash /home/controller/configure.sh

[Install]
WantedBy=multi-user.target
```

To achieve the desired behaviour we need to run the following commands.

```
sudo systemctl start openvswitch-settings.service
sudo systemctl enable openvswitch-settings.service
sudo systemctl daemon-reload
```

For the controller VM we only download faucetsdn/faucet repository and a docker-compose.yml for ONOS. After that we create the VMs template on GNS3 and mounte the topology.



General
Template name: OVS-S
Template ID: 1d2d075a-d89d-483b-af92-3dadc9360441
Default name format: {name}{0}
Server: laptop
Console type: vnc
Auto start console: False
CPUs: 1
Memory: 1024 MB
Linked base VM: True
QEMU binary: qemu-system-x86_64
Hard disks
Disk image (hda): /var/lib/libvirt/images/ovs-switch.qcow2
Disk interface (hda): none
Network
Adapters: 6
Name format: Ethernet{0}
Type: e1000
Optimizations
CPU throttling: disabled
Process priority: normal
Additional options
On close: power_off

Figure 3.11: GNS3 VM template

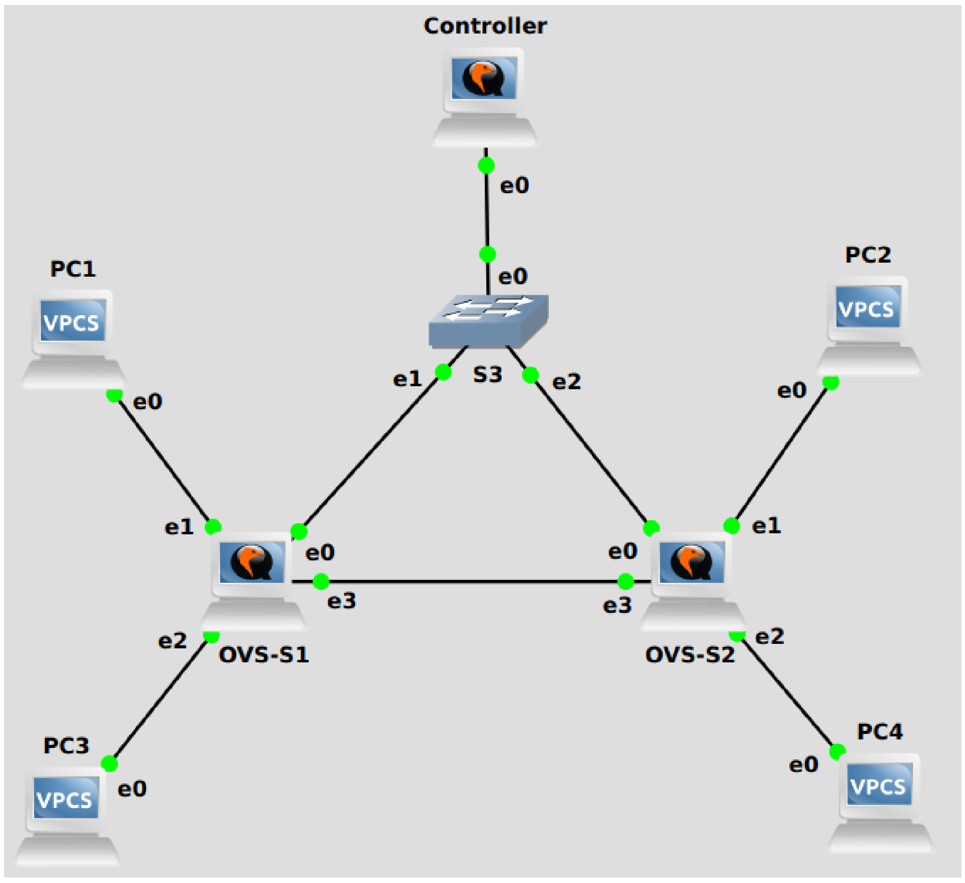


Figure 3.12: GNS3 test topology

After the network setup we hit a roadblock, we couldn't have connectivity with Faucet nor ONOS, after some debugging we saw that we couldn't even have a simple pign connectivity between the switches and the controller VM, probably because we were defining network interfaces on GNS3 template and the VM was recognizing those as ens3 instead of eth, nonetheless, by this time we had our first setup with mininet and found out we could run programs inside a mininet host by opening a terminal with xterm. Given the time constrain with decide not to pursuit this test topology, but realizing that this would be a better environment to test performance of the SDN controller (i.e. Faucet vs. ONOS) and also a better environment to do something like monitorization programs inside a host.

Chapter 4

ONOS Controller

We also though it was good to explore other controllers such as ONOS. ONOS (Open Network Operating System) is an open-source SDN controller, just like Faucet that is designed to support high-scale data center networks and telecom networks. It is designed to be modular, highly available, and scalable, and it supports a wide range of networking protocols and devices. Faucet in the other end is designed specifically for use in enterprise networks. It is designed to be easy to use and to provide a simple, yet powerful, way to manage and automate the configuration of network devices.

Both ONOS and Faucet can be used to manage and automate the configuration of network devices in a variety of environments. However, ONOS is geared more towards large-scale data center and telecom networks.

To test ONOS we used the following docker-compose.yml to create a ONOS container.

```
version: "3"
services:
  onos:
    image: onosproject/onos:2.7.0
    hostname: onos
    container_name: onos
    ports:
      - "8181:8181" # HTTP API/ webapp
      - "8101:8101" # SSH ONOS CLI
      - "6653:6653" # OVS  controller port
      - "5000:5000" # Java remote debug
    environment:
      - ONOS_APPS=gui2,drivers,optical-model,openflow-base,
          openflow,proxyarp,lldpprovider,hostprovider,fwd
```

After creating the docker-compose.yml file we create the ONOS container by running the following command:

```
docker compose up -d
```

The topology for our test case will be the one displayed in the following image.

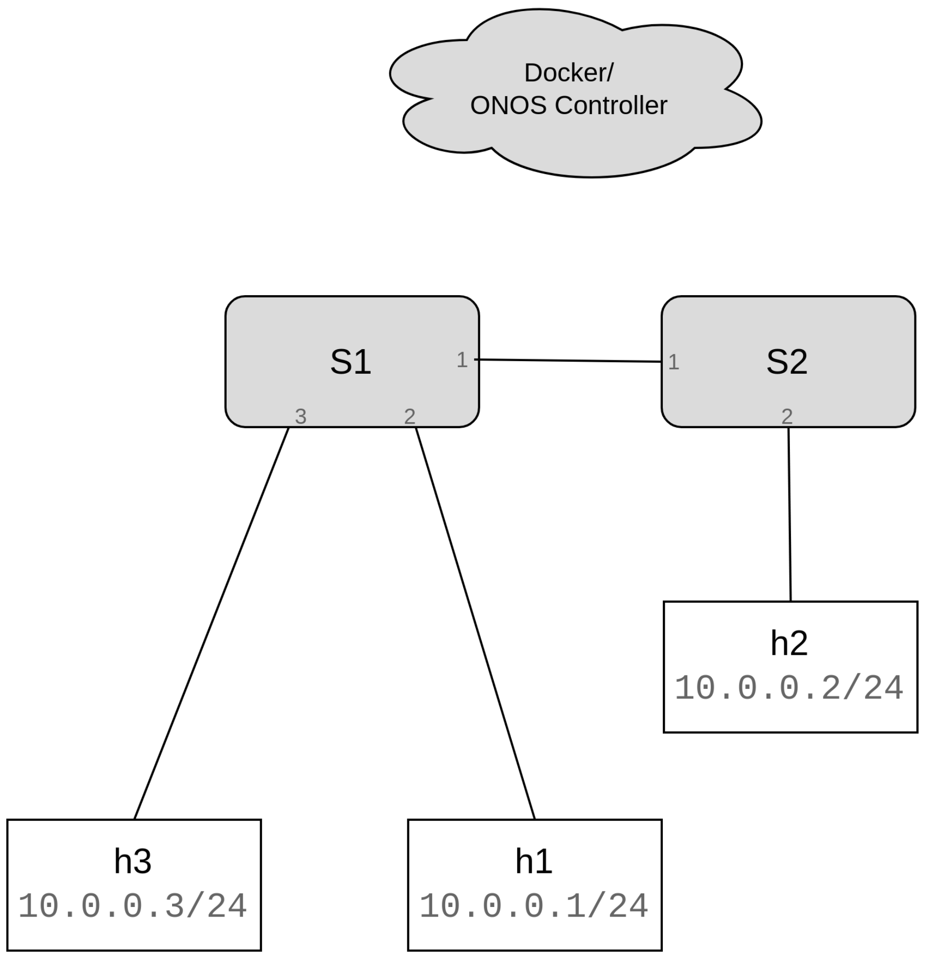


Figure 4.1: ONOS test topology

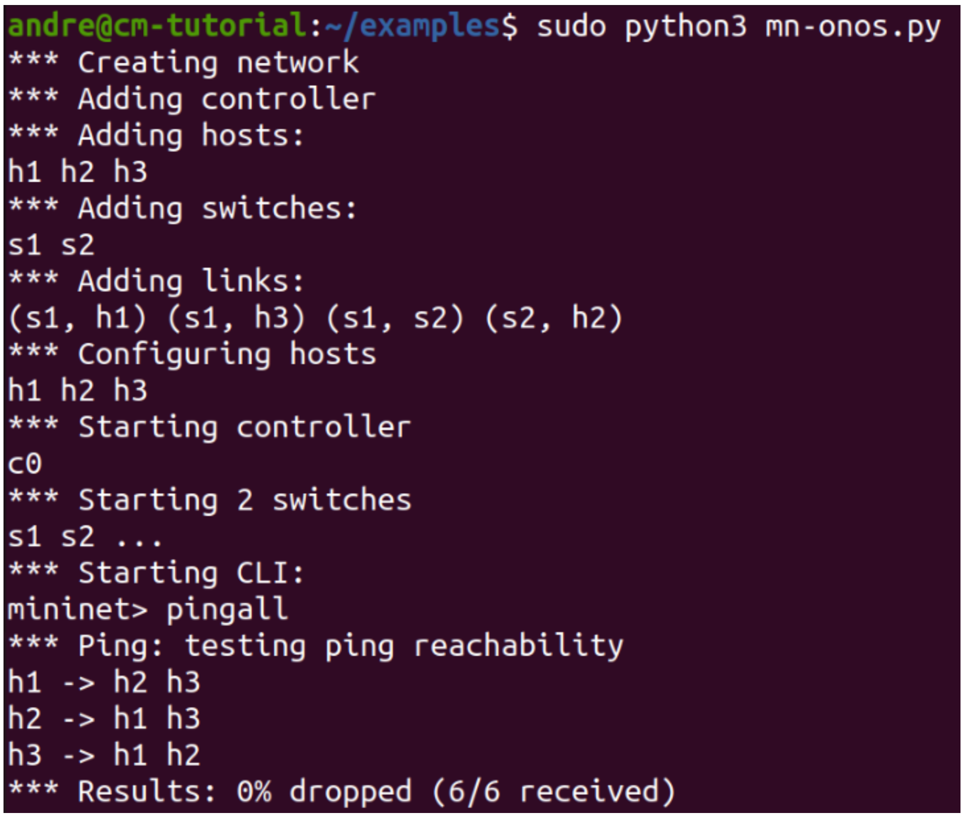
Now, to test this setup we created a mininet script (mn-onos.py) where create the hosts and switches defined in the topology with the respective IPs and connections.

```
from mininet.net import Mininet
from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.node import Host, RemoteController, OVSSwitch
from mininet.topo import Topo

class TopoExample(Topo):
    def __init__(self, *args, **kwargs):
        Topo.__init__(self, *args, **kwargs)

        s1 = self.addSwitch('s1', mac="00:00:00:00:00:01",
    protocols="OpenFlow13")
        s2 = self.addSwitch('s2', mac="00:00:00:00:00:02",
    protocols="OpenFlow13")

        h1 = self.addHost('h1', ip='10.0.0.1/24')
        h2 = self.addHost('h2', ip='10.0.0.2/24')
        h3 = self.addHost('h3', ip='10.0.0.3/24')

        self.addLink(s1, s2, port1=1, port2=1)

        self.ad\clearpagedLink(s1, h1, port1=2, port2=1)
        self.addLink(s1, h3, port1=3, port2=1)
        self.addLink(s2, h2, port1=2, port2=1)

def main():
    net = Mininet(topo=TopoExample(), switch=OVSSwitch, cleanup=
    True, controller=RemoteController('c0', ip='localhost', port
    =6653, protocols="OpenFlow13"))

    net.start()
    CLI(net)
    net.stop()


if __name__ == "__main__":
    setLogLevel('info')
    main()
```

To test if everything configured correctly we can run the following command on the mininet CLI:

```
mininet > pingall
```

From that we get the following result



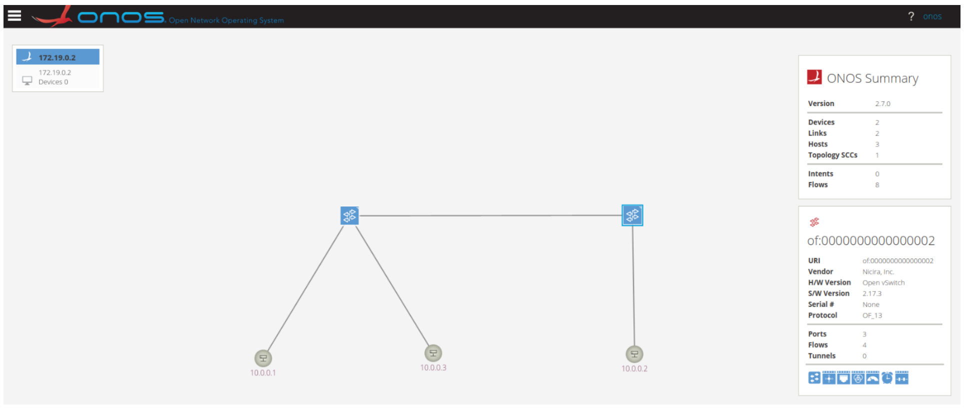
```
andre@cm-tutorial:~/examples$ sudo python3 mn-onos.py
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2
*** Adding links:
(s1, h1) (s1, h3) (s1, s2) (s2, h2)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 2 switches
s1 s2 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
```

Figure 4.2: Usage of the *pingall* command on mn-onos.py

We can see that full connectivity is achieved, this is to be expected because our ONOS configuration is default and in this mode the network is discovered and all the packets are forwarded correctly. Something to note is that some time connectivity isn't reached at the first try, this is because first pings help the ARPs reach the controller, after which the controller populates the switch tables.

Another advantage to ONOS is that it comes with a already incorporated admin page, to access it we need to go to `http://localhost:8181/onos/ui`, the credentials to login by default are username: onos, password: rocks.

We can see that with the network above we can see the topology that ONOS discovers.



ONOS Summary

Version	2.7.0
Devices	2
Links	2
Hosts	3
Topology SCCs	1
Intents	0
Flows	8

of:0000000000000002

URI	of:0000000000000002
Vendor	Nicira, Inc.
H/W Version	Open vSwitch
S/W Version	2.17.3
Serial #	None
Protocol	OF_13
Ports	3
Flows	4
Tunnels	0

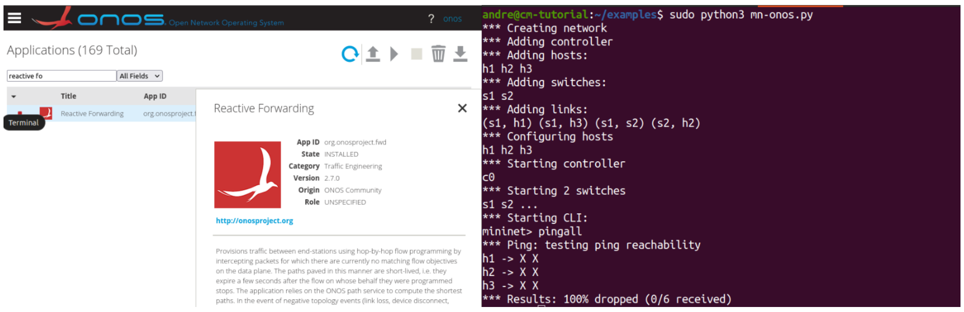
Figure 4.3: ONOS UI

In this page we can see the switches, hosts, connections used, detailed information like mac addresses, switch ids, IPs, VLANs, Flows, etc.

To test one of the advantages of ONOS we go to `http://localhost:8181/onos/ui/#/app` and find the Reactive Forwarding App and deactivate it. The Reactive Forwarding app is a software module that implements reactive packet forwarding. Reactive forwarding is based on ”store and forward”, this means that a network device receives a packet, stores it in memory, and then forwards it to its destination.

The Reactive Forwarding is responsible for receiving packet-in from devices and determining the appropriate action. It can then send packet-out messages to devices to instruct them to forward the packet to its destination.

Now, if we try to do the previous command to test connectivity we see that no connectivity will be achieved, it is also seamless to upload applications to ONOS the UI, and we can also debug code with the help of code editor like Jetbrain’s IntelliJ IDEA and connecting to ONOS Java Debug Port.



Reactive Forwarding

App ID	org.onosproject.fwd
State	INSTALLED
Category	Traffic Engineering
Version	2.7.0
Origin	ONOS Community
Role	UNSPECIFIED

http://onosproject.org

Provisions traffic between end-stations using hop-by-hop flow programming by intercepting packets for which there are currently no matching flow objectives on the data plane. The paths paved in this manner are short-lived, i.e. they expire a few seconds after the flow on whose behalf they were programmed stops. The application relies on the ONOS path service to compute the shortest paths. In the event of negative topology events (link loss, device disconnect,

```
andre@cm-tutorial:~/examples$ sudo python3 mn-onos.py
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2
*** Adding links:
(s1, h1) (s1, h3) (s1, s2) (s2, h2)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 2 switches
s1 s2 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X
h2 -> X X
h3 -> X X
*** Results: 100% dropped (0/6 received)
```

Figure 4.4: Pingall with Reactive Forwarding disable

Another test that we've done with ONOS was having 2 controllers in our network. The topology tested is demonstrated by the image below.

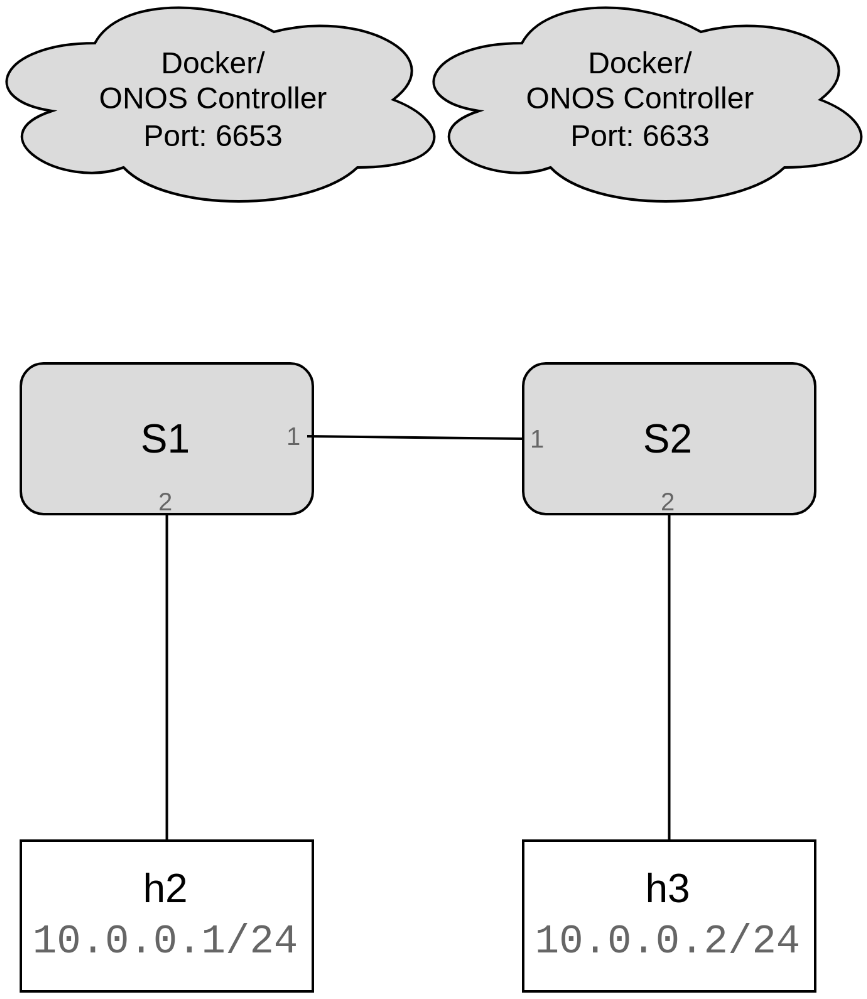


Figure 4.5: ONOS test topology with 2 controllers

To create another ONOS controller we used the following onos-6633.yml to create another ONOS container.

```
version: "3"
services:
  onos6633:
    image: onosproject/onos:2.7.0
    hostname: onos6633
    container_name: onos6633
    ports:
      - "8182:8182" # HTTP API/ webapp
      - "8102:8102" # SSH ONOS CLI
      - "6633:6653" # OVS  controller port
      - "5001:5001" # Java remote debug
    environment:
      - ONOS_APPS=gui2,drivers,optical-model,openflow-base,
        openflow,proxyarp,lldpprovider,hostprovider,fwd
```

After creating the onos-6633.yml file we create the ONOS container by running the following command:

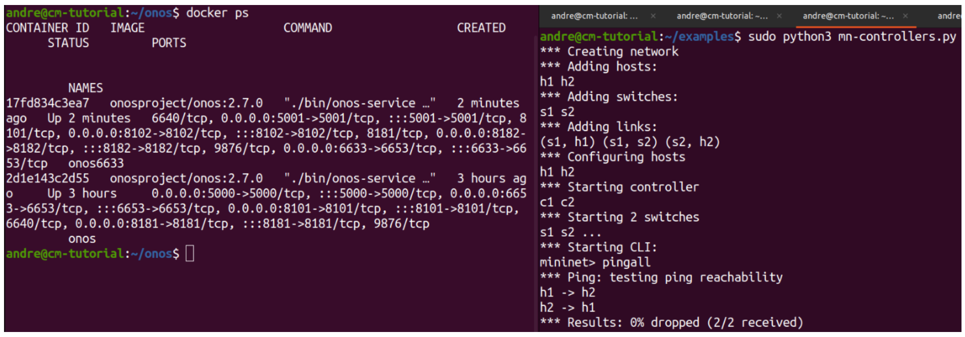
```
docker compose -f onos-6633.yml up -d
```

Now, to test this setup we created a mininet script (mn-controllers.py) that creates the hosts and switches defined in the topology with the respective IPs and connections.

```
from mininet.net import Mininet
from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.node import Host, RemoteController, OVSSwitch
from mininet.topo import Topo

class TopoExample(Topo):
    def __init__(self, *args, **kwargs):
        Topo.__init__(self, *args, **kwargs)

        s1 = self.addSwitch('s1', mac="00:00:00:00:00:01",
    protocols="OpenFlow13")
        s2 = self.addSwitch('s2', mac="00:00:00:00:00:02",
    protocols="OpenFlow13")

        h1 = self.addHost('h1', ip='10.0.0.1/24')
        h2 = self.addHost('h2', ip='10.0.0.2/24')

        self.addLink(s1, s2, port1=1, port2=1) # connect s1/1
    <-> s2/1

        self.addLink(s1, h1, port1=2, port2=1) # connect s1/2
    <-> h1
        self.addLink(s2, h2, port1=2, port2=1) # connect s2/2
    <-> h2


def main():
```

```python
    net = Mininet(topo=TopoExample(), switch=OVSSwitch, cleanup=True, controller=None)
    net.addController('c1', controller=RemoteController, ip='localhost', port=6653, protocols="OpenFlow13")
    net.addController('c2', controller=RemoteController, ip='localhost', port=6633, protocols="OpenFlow13")

    net.start()
    CLI(net)
    net.stop()


if __name__ == "__main__":
    setLogLevel('info')
    main()
```

By viewing the image below we see that full connectivity is achieved when both containers are up and running.

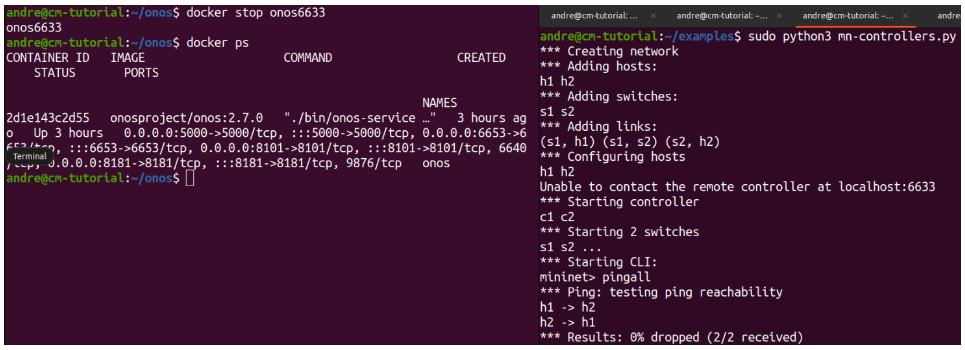


```
andre@cm-tutorial:~/onos$ docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED
      STATUS          PORTS


         NAMES
17fd834c3ea7   onosproject/onos:2.7.0   "./bin/onos-service …"   2 minutes
ago   Up 2 minutes   6640/tcp, 0.0.0.0:5001->5001/tcp, :::5001->5001/tcp, 8
101/tcp, 0.0.0.0:8102->8102/tcp, :::8102->8102/tcp, 8181/tcp, 0.0.0.0:8182-
>8182/tcp, :::8182->8182/tcp, 9876/tcp, 0.0.0.0:6633->6653/tcp, :::6633->66
53/tcp   onos6633
2d1e143c2d55   onosproject/onos:2.7.0   "./bin/onos-service …"   3 hours ag
o     Up 3 hours     0.0.0.0:5000->5000/tcp, :::5000->5000/tcp, 0.0.0.0:665
3->6653/tcp, :::6653->6653/tcp, 0.0.0.0:8101->8101/tcp, :::8101->8101/tcp,
6640/tcp, 0.0.0.0:8181->8181/tcp, :::8181->8181/tcp, 9876/tcp
         onos
andre@cm-tutorial:~/onos$
```

```
andre@cm-tutorial:~/examples$ sudo python3 mn-controllers.py
*** Creating network
*** Adding hosts:
h1 h2
*** Adding switches:
s1 s2
*** Adding links:
(s1, h1) (s1, s2) (s2, h2)
*** Configuring hosts
h1 h2
*** Starting controller
c1 c2
*** Starting 2 switches
s1 s2 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```

Figure 4.6: Pingall with both controllers active

And when we run a command to stop one of the controllers, we can also see that full connectivity is maintained, this is because the controller left takes control and keeps the network running.



```
andre@cm-tutorial:~/onos$ docker stop onos6633
onos6633
andre@cm-tutorial:~/onos$ docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED
    STATUS       PORTS

                                                            NAMES
2d1e143c2d55   onosproject/onos:2.7.0   "./bin/onos-service …"   3 hours ag
o   Up 3 hours   0.0.0.0:5000->5000/tcp, :::5000->5000/tcp, 0.0.0.0:6653->6
653/tcp, :::6653->6653/tcp, 0.0.0.0:8101->8101/tcp, :::8101->8101/tcp, 6640
/tcp, 0.0.0.0:8181->8181/tcp, :::8181->8181/tcp, 9876/tcp   onos
andre@cm-tutorial:~/onos$
```

```
andre@cm-tutorial:~/examples$ sudo python3 mn-controllers.py
*** Creating network
*** Adding hosts:
h1 h2
*** Adding switches:
s1 s2
*** Adding links:
(s1, h1) (s1, s2) (s2, h2)
*** Configuring hosts
h1 h2
Unable to contact the remote controller at localhost:6633
*** Starting controller
c1 c2
*** Starting 2 switches
s1 s2 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```

Figure 4.7: Pingall with one controller deactivated

Chapter 5

Hybrid Networks

A hybrid network is one that combines elements of traditional networking (legacy devices) and software-defined networking (SDN). In a hybrid network, some parts of the network may be controlled using traditional networking methods, while others are controlled using SDN.

One example of a hybrid network is one that uses SDN to control some parts of the network, such as the data center or the cloud, while using traditional networking methods to control other parts of the network, such as the access layer or the edge of the network.

Even thought this use currently in the scientific research domain we can though on some ways we could use this test case to have some degree of control (by the controller) on the legacy network behaviour.

In a scenario where you have a pipeline of SDN switches that are connected to a controller, and somewhere in the pipeline there is a legacy switch. During the link discovery process, this legacy switch is interrupting the propagation of LLDP packets, which is causing the controller to be unable to discover the rest of the network.

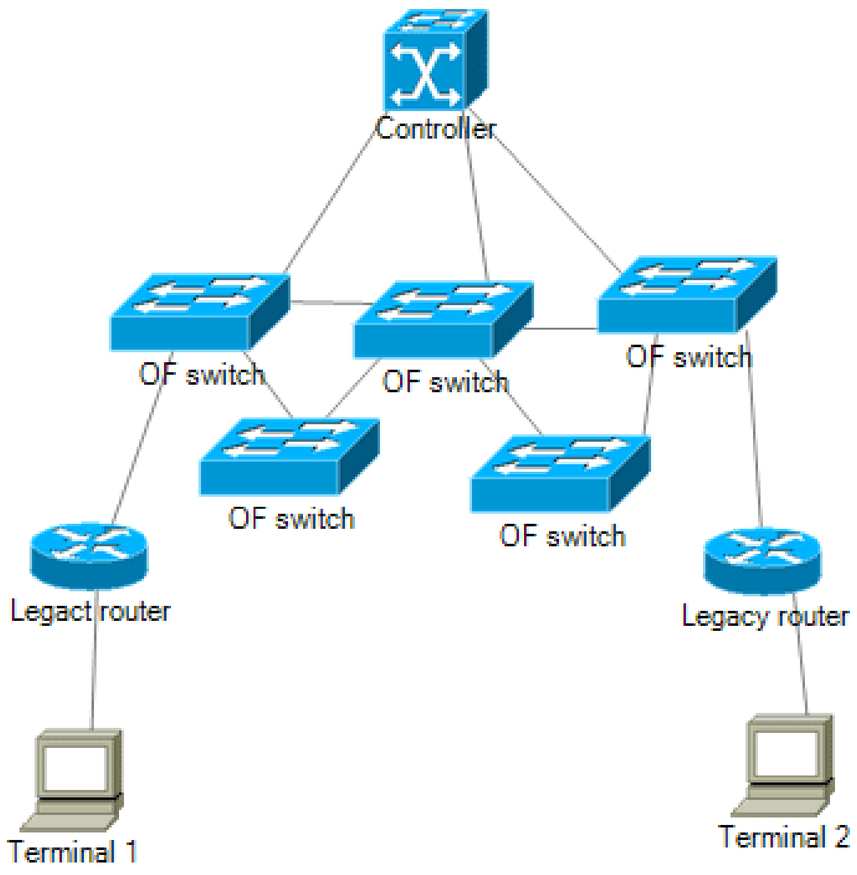


Figure 5.1: Example of a hybrid network

There are a few potential solutions you could consider in this situation:

1. If the legacy switch supports LLDP packets, we could try to determine whether the switch is simply passing the packets through without interacting with them, or if there is some way that the switch can be ”seen” by the controller.

2. We could try enabling link discovery with BDDP packets, which legacy switches may not drop. However, you would need to determine whether this is sufficient for the controller to discover these switches.

3. We could modify the controller by modifying the existing link discovery app or developing a new app that is able to support the necessary modifications for one of the above scenarios.