INSTITUTO SUPERIOR TÉCNICO

TRAFFIC ENGINEERING

METI

Lab Report IV and V

Software Defined Networking and OpenFlow

2018/2019

Group 6
André Mendes - 78079
Filipe Fernandes - 78083

1 Introduction

TODO: Briefly describe both lab 4 and 5.

This report is divided in two major sections. Laboratory 4 and 5.

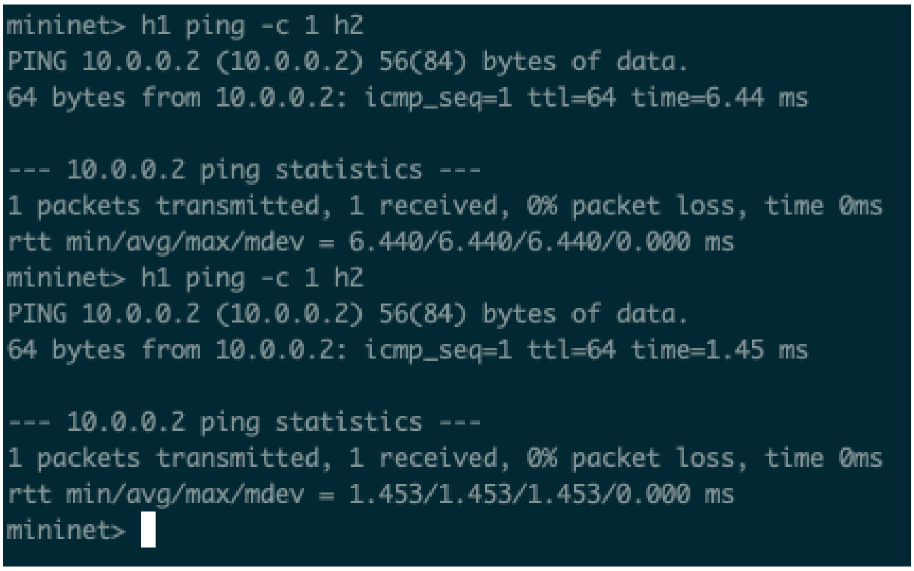
2 Laboratory 4

2.1 Mininet minimal topology

Mininet minimal topology is composed by 2 hosts, 1 switch and 1 basic controller. This topology can be launched with a single command from mininet like `sudo mn` or `sudo mn --topo=minimal`.

2.1.1 Testing connectivity between hosts

On the next figure we can see the output of two consecutive pings from host 1 to host 2.



```
mininet> h1 ping -c 1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=6.44 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 6.440/6.440/6.440/0.000 ms
mininet> h1 ping -c 1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1.45 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.453/1.453/1.453/0.000 ms
mininet>
```

Figure 1: Ping from h1 to h2

As we can see the first one has a much higher delay than the second one. This happens because on the second ping a flow entry covering *ICMP* ping traffic was already installed in the switch, so no control traffic was generated, and the packets immediately pass through the switch. We can confirm this by looking at the packet trace captured during the first ping using *wireshark*, displayed on figure 2.

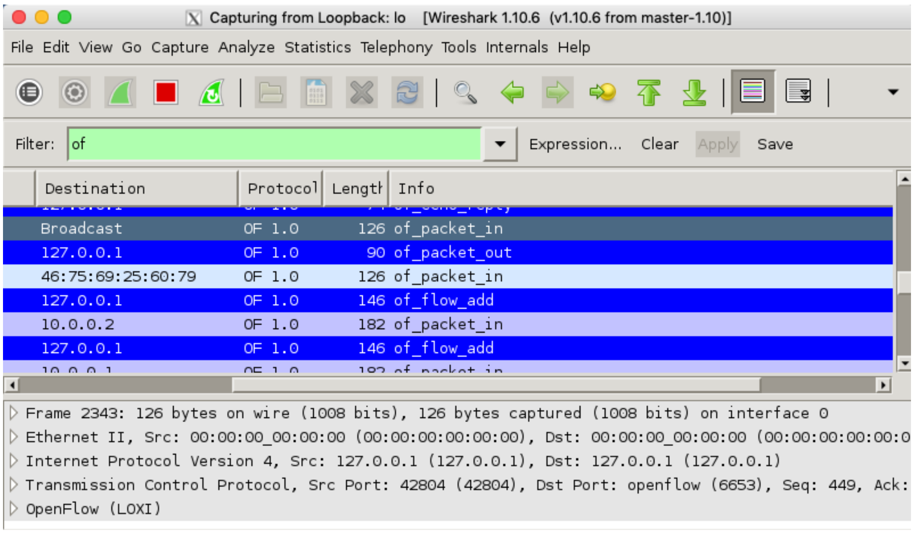


Figure 2: Wireshark capture on lo0 of mininet host during first ping

2.2 Mininet single topology

This topology includes 3 hosts, 1 OpenFlow Switch and 1 controller as displayed in figure 2.

2.2.1 Without controller

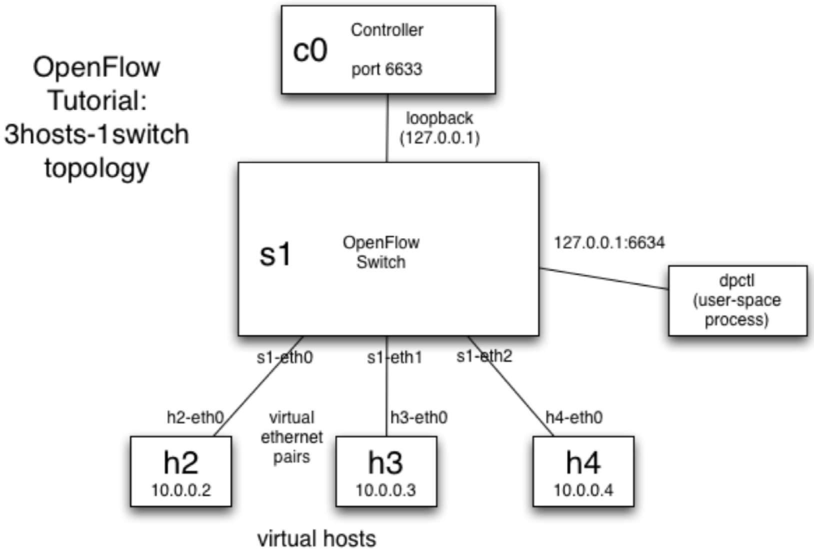


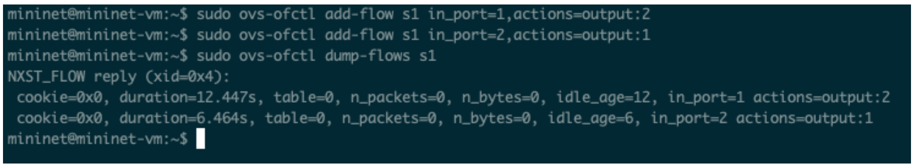
Figure 3: Mininet simple topology with 3 hosts, hosts actually are h1(10.0.0.1), h2(10.0.0.2) and h3(10.0.0.3) instead of h2, h3 and h4

Right after deploying this topology the switch flow table is empty. So if we try to ping it will fail because the switch hasn't any configurations and it doesn't even act has an hub.

If we add the following flows on the switch flow-table it will start doing what we expected firstly.

```
sudo ovs-ofctl add-flow s1 in_port=1,actions=output:2
sudo ovs-ofctl add-flow s1 in_port=2,actions=output:1
```

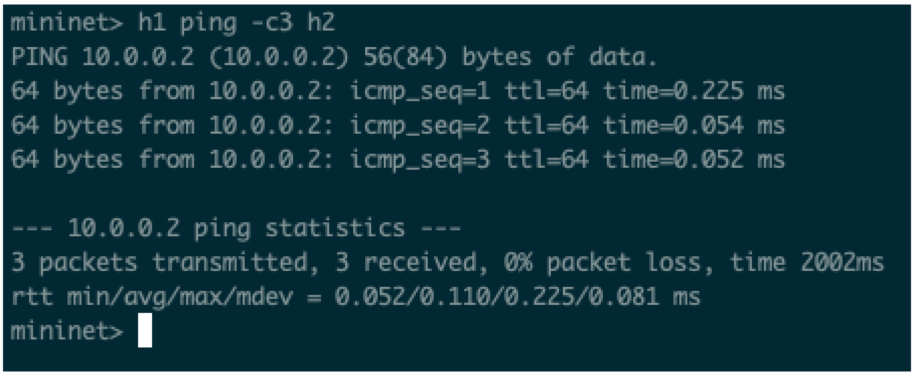
We are configuring the switch to redirect traffic from input port 1 to output port 2 and vice-versa. The content of the flow-table after these commands are displayed on the next figure.



```
mininet@mininet-vm:~$ sudo ovs-ofctl add-flow s1 in_port=1,actions=output:2
mininet@mininet-vm:~$ sudo ovs-ofctl add-flow s1 in_port=2,actions=output:1
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=12.447s, table=0, n_packets=0, n_bytes=0, idle_age=12, in_port=1 actions=output:2
 cookie=0x0, duration=6.464s, table=0, n_packets=0, n_bytes=0, idle_age=6, in_port=2 actions=output:1
mininet@mininet-vm:~$
```

Figure 4: Install flows on switch s1

And now we can successfully ping between h1 and h2 as seen on the next figure.

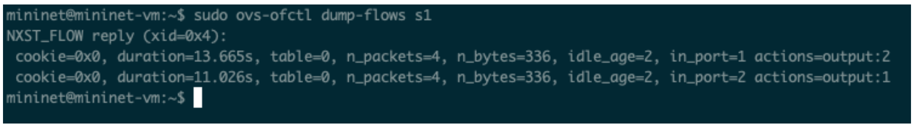


```
mininet> h1 ping -c3 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.225 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.054 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.052 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 0.052/0.110/0.225/0.081 ms
mininet>
```

Figure 5: Ping between h1 and h2 with flows installed

After each traffic flow that belongs to the flows installed on s1, the flow-table of s1 is updated.



```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=13.665s, table=0, n_packets=4, n_bytes=336, idle_age=2, in_port=1 actions=output:2
 cookie=0x0, duration=11.026s, table=0, n_packets=4, n_bytes=336, idle_age=2, in_port=2 actions=output:1
mininet@mininet-vm:~$
```

Figure 6: Flow table of s1 after ping

All pings executed between other hosts than h1 to h2 and h2 to h1 will fail. Except if we add/edit the flows for other routes. If we delete the current flows the connection between h1 and h2 will also start to fail.

2.2.2 With controller

If we repeat the steps above but this time with a controller, the behaviour of the network is different.
Let's start by launching a controller on mininet: `$ controller ptcp:6653`.
When the controller is starts a lot of messages are exchanged.
This messages are:

- Hello - (Controller<->Switch)
- Features - (RequestController->Switch)
- Set Config - (RequestController->Switch)
- Features Reply - (Switch->Controller)
- Port Status - (Switch->Controller)

Some of them are captured on figure 7.

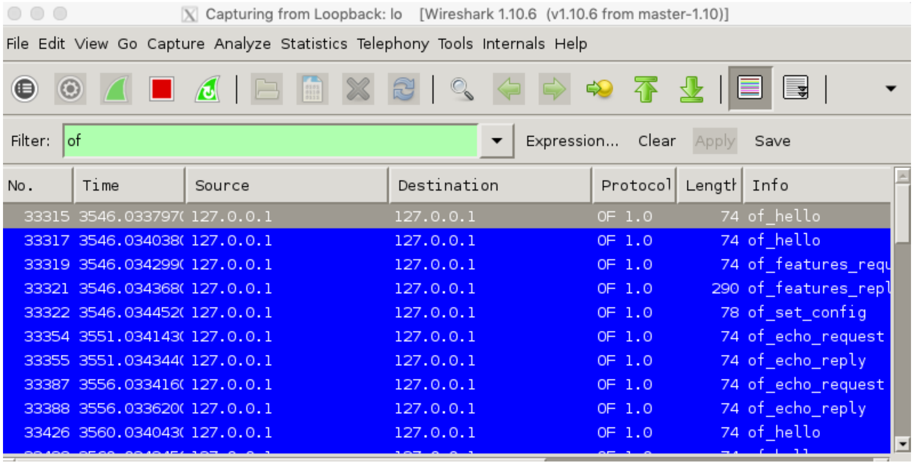


Figure 7: Wireshark capture of messages being exchanged between controller and switch.

When we generate packets (`h1 ping -c1 h2`) new messages are exchanged.
This messages can be:

- Packet-In (Switch->Controller)
- Packet-Out (Controller->Switch)
- Flow-Mod (Controller->Switch)
- Flow-Expired (Switch->Controller)

On figure 8 some of these are captured. This is an example of OpenFlow in reactive mode (flows are pushed down in response to individual packets). The other possible mode is proactive mode where flows can be pushed down before packets to avoid the round-trip times and flow insertion delays.

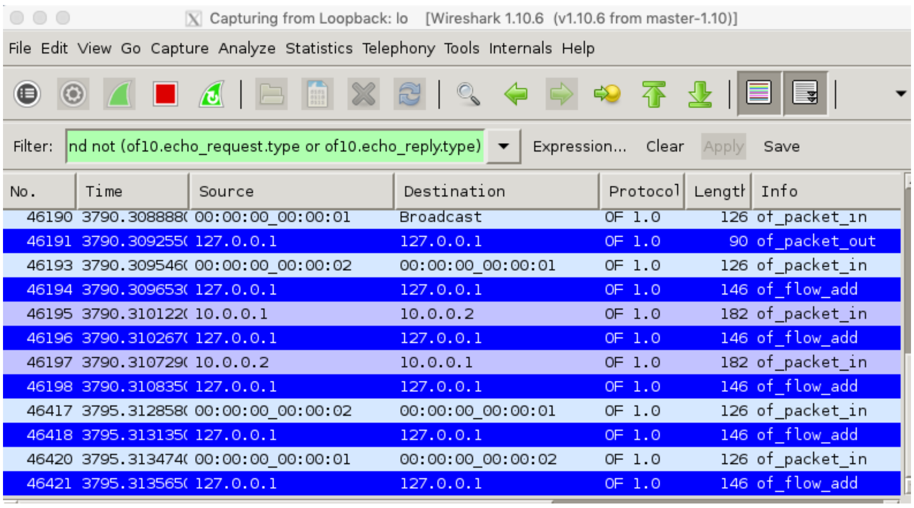


Figure 8: Wireshark capture of flows being pushed down in response to individual packets.

TODO: Explain this new messages

2.2.3 Benchmark controller with `iperf`

To benchmark the reference controller we will use iperf, a tool used to check speeds between two computers.

On the next figure we can see the results of iperf between all hosts. The higher speed obtained is 24.7 Gbits/s on link h1<->h2. But all hosts have about the same speed between them, arround 24Gbits/s.

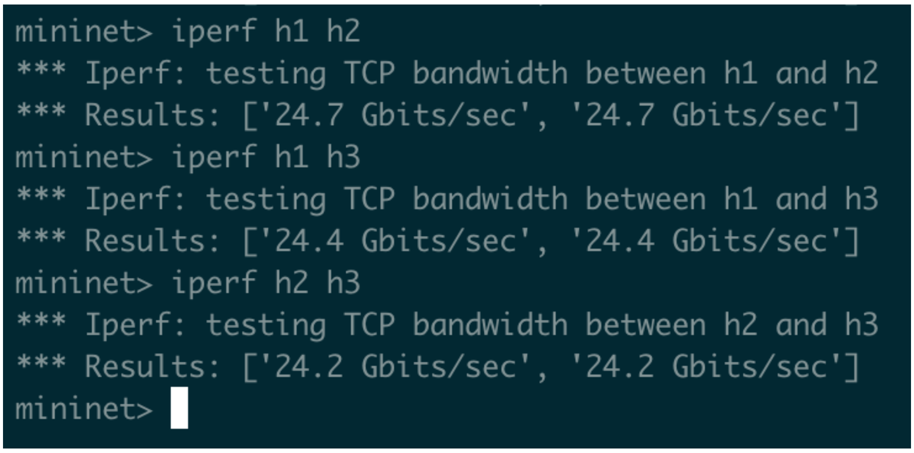


Figure 9: Output of `iperf h1 h2`, `iperf h1 h3` and `iperf h2 h3`.

3 Laboratory 5

In this section we will compare the behaviour of the system in three different behaviours. Like an Hub, using a learning switch without installing table flow entries and finnally a learning switch with table flow entries.

3.1 Hub behaviour

In Hub behaviour, if three or more hosts are connected to the same switch, all packets sent from one host to another are flooded to all ports except the incoming one. This is show on the next figure. Where it is performed two pings from h1 to h2 and one ping from h1 to h3. In both cases we are running `tcpdump` on h2 and h3.

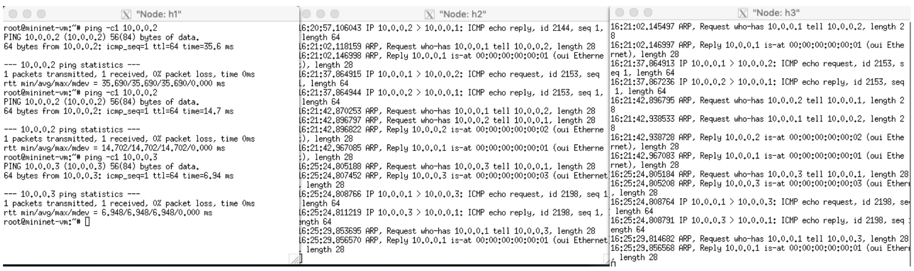


Figure 10: Output of ping in h1 and `tcpdump` on h2 and h3

On figure 10 we can confirm that even on the second ping from h1 to h2, h3 is also flooded. This is a typical Hub behaviour.

3.2 Implementing a learning switch without installing table flow entries

In this scenario we expect that all packets are going to be sent to the controller, and the controller is the responsible to re-route to the respective host.

On the following figure we can confirm that when performing two consecutive pings from h1 to h2, only the first packet is flooded, the second one is only sent to h2.

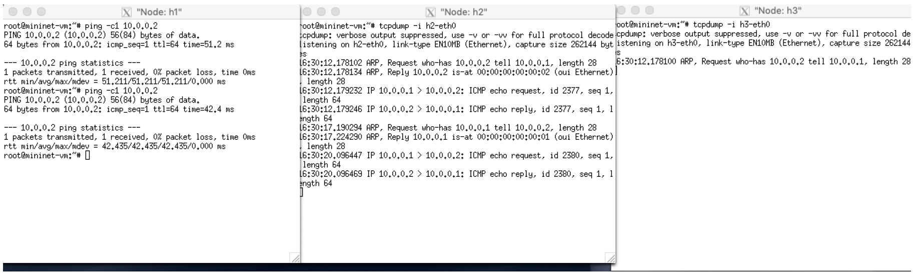


Figure 11: Output of ping in h1 to h2 and `tcpdump` on h2 and h3

A similar behaviour is verified when we perform another two pings from h1 to h3, only the first is flooded to h2. As seen on figure 12.

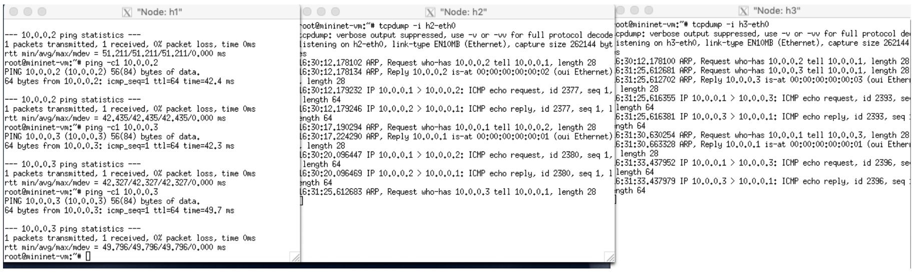
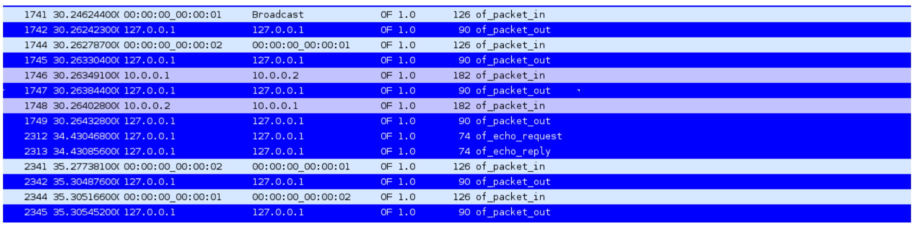


Figure 12: Output of ping in h1 to h2, h3 and `tcpdump` on h2 and h3

On the following image we present a wireshark print, captured on mininet VM, interface lo0 that confirms all packets are sent to controller.



1741	30.24624400(	00:00:00_00:00:01	Broadcast	OF 1.0	126	of_packet_in
1742	30.26242300(	127.0.0.1	127.0.0.1	OF 1.0	90	of_packet_out
1744	30.26278700(	00:00:00_00:00:02	00:00:00_00:00:01	OF 1.0	126	of_packet_in
1745	30.26330400(	127.0.0.1	127.0.0.1	OF 1.0	90	of_packet_out
1746	30.26349100(	10.0.0.1	10.0.0.2	OF 1.0	182	of_packet_in
1747	30.26384400(	127.0.0.1	127.0.0.1	OF 1.0	90	of_packet_out
1748	30.26402800(	10.0.0.2	10.0.0.1	OF 1.0	182	of_packet_in
1749	30.26432800(	127.0.0.1	127.0.0.1	OF 1.0	90	of_packet_out
2312	34.43046800(	127.0.0.1	127.0.0.1	OF 1.0	74	of_echo_request
2313	34.43085600(	127.0.0.1	127.0.0.1	OF 1.0	74	of_echo_reply
2341	35.27738100(	00:00:00_00:00:02	00:00:00_00:00:01	OF 1.0	126	of_packet_in
2342	35.30487600(	127.0.0.1	127.0.0.1	OF 1.0	90	of_packet_out
2344	35.30516600(	00:00:00_00:00:01	00:00:00_00:00:02	OF 1.0	126	of_packet_in
2345	35.30545200(	127.0.0.1	127.0.0.1	OF 1.0	90	of_packet_out

Figure 13: Wireshark capture on interface Loopback0 with "of" filter on mininet VM

And also as expected no routes are installed at s1, but mac to port table is properly learned. See figures 14 and 15

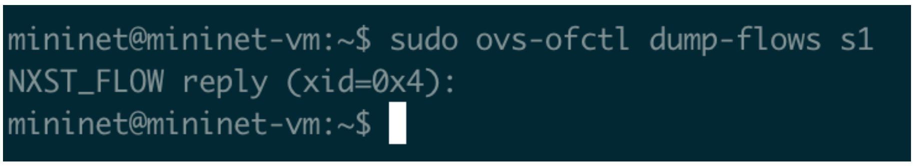


Figure 14: Switch 1 flow table entries

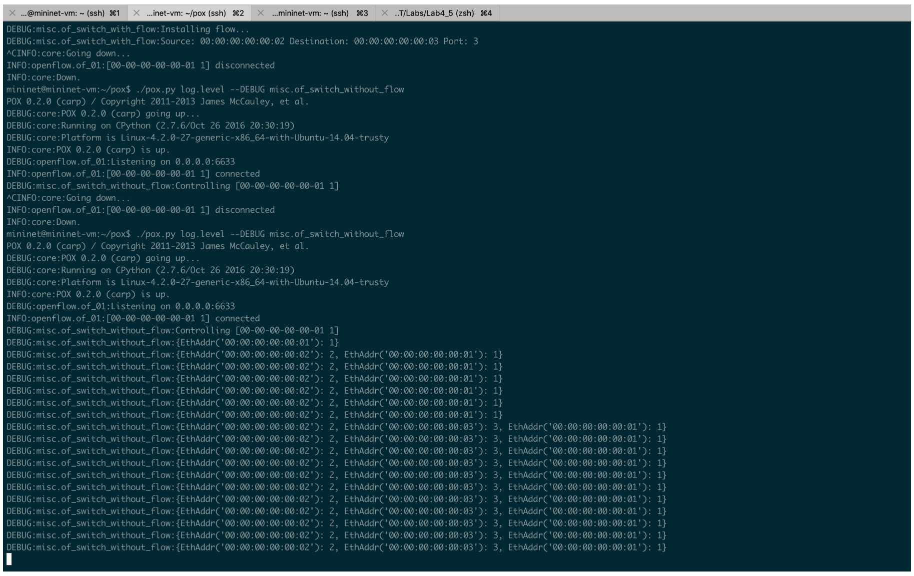


Figure 15: Log of POX with all mac to port table entries being displayed

3.3 Implementing a learning switch with table flow entries

On this subsection, the behaviour is similar to the previous but now table flow entries are installed at s1. This means that most of the packets won't go through the controller. As we will confirm.

We started by performing the same pings as we did before, two consecutive pings from h1 to h2. As expected only the first is flooded to h3 and sent to the controller. See figures 16 and 17.

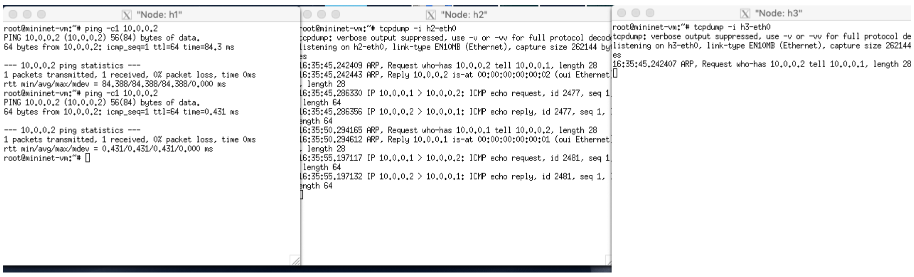
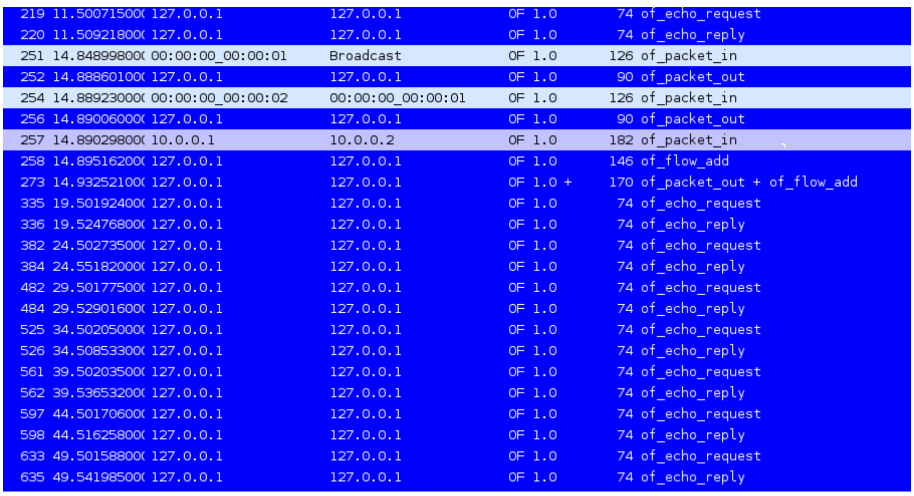


Figure 16: Output of ping in h1 to h2 and `tcpdump` on h2 and h3



219	11.50071500(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_request
220	11.50921800(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_reply
251	14.84899800(	00:00:00_00:00:01	Broadcast	OF 1.0	126 of_packet_in
252	14.88860100(	127.0.0.1	127.0.0.1	OF 1.0	90 of_packet_out
254	14.88923000(	00:00:00_00:00:02	00:00:00_00:00:01	OF 1.0	126 of_packet_in
256	14.89006000(	127.0.0.1	127.0.0.1	OF 1.0	90 of_packet_out
257	14.89029800(	10.0.0.1	10.0.0.2	OF 1.0	182 of_packet_in
258	14.89516200(	127.0.0.1	127.0.0.1	OF 1.0	146 of_flow_add
273	14.93252100(	127.0.0.1	127.0.0.1	OF 1.0 +	170 of_packet_out + of_flow_add
335	19.50192400(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_request
336	19.52476800(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_reply
382	24.50273500(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_request
384	24.55182000(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_reply
482	29.50177500(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_request
484	29.52901600(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_reply
525	34.50205000(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_request
526	34.50853300(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_reply
561	39.50203500(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_request
562	39.53653200(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_reply
597	44.50170600(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_request
598	44.51625800(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_reply
633	49.50158800(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_request
635	49.54198500(	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_reply

Figure 17: Wireshark capture on interface Loopback0 with "of" filter on mininet VM, displaying the addition of a flow and packet re-route

This happens because the first time packets are sent to the controller, flow table entries are installed at s1. See figures 18 and 19.

```
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
DEBUG:misc.of_switch_with_flow:Controlling [00-00-00-00-00-01 1]
DEBUG:misc.of_switch_with_flow:Installing flow...
DEBUG:misc.of_switch_with_flow:Source: 00:00:00:00:00:02 Destination: 00:00:00:00:00:01 Port: 1
DEBUG:misc.of_switch_with_flow:Installing flow...
DEBUG:misc.of_switch_with_flow:Source: 00:00:00:00:00:01 Destination: 00:00:00:00:00:02 Port: 2
```

Figure 18: Log of running POX displaying messages when installing flows on s1

```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=36.698s, table=0, n_packets=3, n_bytes=238, idle_age=26, in_port=2,dl_dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=36.66s, table=0, n_packets=2, n_bytes=140, idle_age=26, in_port=1,dl_dst=00:00:00:00:00:02 actions=output:2
mininet@mininet-vm:~$
```

Figure 19: Flows installed at s1 after ping from h1 to h2

On the next figure we can also confirm that mac to port table is properly learned.

```
mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG misc.of_switch_with_flow
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
DEBUG:core:POX 0.2.0 (carp) going up...
DEBUG:core:Running on CPython (2.7.6/Oct 26 2016 20:30:19)
DEBUG:core:Platform is Linux-4.2.0-27-generic-x86_64-with-Ubuntu-14.04-trusty
INFO:core:POX 0.2.0 (carp) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
DEBUG:misc.of_switch_with_flow:Controlling [00-00-00-00-00-01 1]
DEBUG:misc.of_switch_with_flow:{EthAddr('00:00:00:00:00:01'): 1}
DEBUG:misc.of_switch_with_flow:{EthAddr('00:00:00:00:00:02'): 2, EthAddr('00:00:00:00:00:01'): 1}
DEBUG:misc.of_switch_with_flow:Installing flow...
DEBUG:misc.of_switch_with_flow:Source: 00:00:00:00:00:02 Destination: 00:00:00:00:00:01 Port: 1
DEBUG:misc.of_switch_with_flow:{EthAddr('00:00:00:00:00:02'): 2, EthAddr('00:00:00:00:00:01'): 1}
DEBUG:misc.of_switch_with_flow:Installing flow...
DEBUG:misc.of_switch_with_flow:Source: 00:00:00:00:00:01 Destination: 00:00:00:00:00:02 Port: 2
DEBUG:misc.of_switch_with_flow:{EthAddr('00:00:00:00:00:02'): 2, EthAddr('00:00:00:00:00:01'): 1}
DEBUG:misc.of_switch_with_flow:{EthAddr('00:00:00:00:00:02'): 2, EthAddr('00:00:00:00:00:03'): 3, EthAddr('00:00:00:00:00:01'): 1}
DEBUG:misc.of_switch_with_flow:Installing flow...
DEBUG:misc.of_switch_with_flow:Source: 00:00:00:00:00:03 Destination: 00:00:00:00:00:01 Port: 1
DEBUG:misc.of_switch_with_flow:{EthAddr('00:00:00:00:00:02'): 2, EthAddr('00:00:00:00:00:03'): 3, EthAddr('00:00:00:00:00:01'): 1}
DEBUG:misc.of_switch_with_flow:Installing flow...
DEBUG:misc.of_switch_with_flow:Source: 00:00:00:00:00:01 Destination: 00:00:00:00:00:03 Port: 3
```

Figure 20: Log of POX with all mac to port table entries being displayed

On figure 21 we present the final table flow entry of s1 after performing pings from all to all. Our flows have two matching conditions and one output action.

- `msg.match.in_port = packet_in.in_port`
- `msg.match.dl_dst = packet.dst`
- `msg.actions.append(of.ofp_action_output(port = self.mac_to_port.get(packet.dst)))`

Summarizing, packets from a certain port to a certain destination are re-routed to the respective destination port.

```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=31.314s, table=0, n_packets=3, n_bytes=238, idle_age=21, in_port=3,dl_dst=00:00:00:00:00:02 actions=output:2
 cookie=0x0, duration=1749.538s, table=0, n_packets=7, n_bytes=518, idle_age=32, in_port=2,dl_dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=1747.159s, table=0, n_packets=6, n_bytes=420, idle_age=18, in_port=3,dl_dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=31.277s, table=0, n_packets=2, n_bytes=140, idle_age=21, in_port=2,dl_dst=00:00:00:00:00:03 actions=output:3
 cookie=0x0, duration=1747.121s, table=0, n_packets=5, n_bytes=322, idle_age=18, in_port=1,dl_dst=00:00:00:00:00:03 actions=output:3
 cookie=0x0, duration=1749.501s, table=0, n_packets=6, n_bytes=420, idle_age=32, in_port=1,dl_dst=00:00:00:00:00:02 actions=output:2
mininet@mininet-vm:~$
```

Figure 21: Flows installed at s1 after ping from all to all

4 Conclusions

Regarding laboratory 4 our goal was mainly to get hands on Mininet and OpenFlow protocol. We found mininet very interesting because it allows us to use kernel linux/unix functions with just a few commands.

We now understand main benefits of SDN (Software Defined Networks) and how it can help us create an abstraction layer between the equipment's configuration and the topology of the network.

On laboratory 5 we had the opportunity to test POX, a python framework for communicating with SDN switches using OpenFlow. We started by analysing the behaviour of an Hub, followed by a learning switch with and without table flow entries. In all cases we performed pings to test connectivity and RTT(round trip times). We conclude that the lower RTT happens when using learning switch with table flow entries (after the flows being inserted on switch, values around 0.431ms). And the higher is when using a learning switch with a controller but without installing table flow entries (about 49 ms). The Hub behaviour stays in the middle when comparing RTTs (+/- 7 ms).

Annex:

A

Annex A - of_switch_without_flow.py

```
# Copyright 2012 James McCauley
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
  implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
This component is for use with the OpenFlow tutorial.

It acts as a simple hub, but can be modified to act like an L2
learning switch.

It's roughly similar to the one Brandon Heller did for NOX.
"""

from pox.core import core
import pox.openflow.libopenflow_01 as of

log = core.getLogger()



class Tutorial (object):
  """
  A Tutorial object is created for each switch that connects.
  A Connection object for that switch is passed to the __init__
    function.
  """
  def __init__ (self, connection):
    # Keep track of the connection to the switch so that we can
    # send it messages!
    self.connection = connection

    # This binds our PacketIn event listener
    connection.addListeners(self)

    # Use this table to keep track of which ethernet address is on
```

```
    # which switch port (keys are MACs, values are ports).
    self.mac_to_port = {}


  def resend_packet (self, packet_in, out_port):
    """
    Instructs the switch to resend a packet that it had sent to us.
    "packet_in" is the ofp_packet_in object the switch had sent to
        the
    controller due to a table-miss.
    """
    msg = of.ofp_packet_out()
    msg.data = packet_in

    # Add an action to send to the specified port
    action = of.ofp_action_output(port = out_port)
    msg.actions.append(action)

    # Send message to switch
    self.connection.send(msg)


  def act_like_hub (self, packet, packet_in):
    """
    Implement hub-like behavior -- send all packets to all ports
        besides
    the input port.
    """

    # We want to output to all ports -- we do that using the special
    # OFPP_ALL port as the output port.  (We could have also used
    # OFPP_FLOOD.)
    self.resend_packet(packet_in, of.OFPP_ALL)

    # Note that if we didn't get a valid buffer_id, a slightly better
    # implementation would check that we got the full data before
    # sending it (len(packet_in.data) should be == packet_in.
        total_len)).


  def act_like_switch (self, packet, packet_in):
    """
    Implement switch-like behavior.
    """

    # Learn the port for the source MAC
    self.mac_to_port.update({packet.src: packet_in.in_port})

    log.debug(self.mac_to_port)

    if  self.mac_to_port.get(packet.dst) is not None:
      # Send packet out the associated port
```

```
            self.resend_packet(packet_in, self.mac_to_port.get(packet.dst
              ))
        else:
          self.resend_packet(packet_in, of.OFPP_ALL)


    def _handle_PacketIn (self, event):
      """
      Handles packet in messages from the switch.
      """

      packet = event.parsed # This is the parsed packet data.
      if not packet.parsed:
        log.warning("Ignoring incomplete packet")
        return

      packet_in = event.ofp # The actual ofp_packet_in message.

      self.act_like_switch(packet, packet_in)



def launch ():
    """
    Starts the component
    """
    def start_switch (event):
      log.debug("Controlling %s" % (event.connection,))
      Tutorial(event.connection)
    core.openflow.addListenerByName("ConnectionUp", start_switch)
```

B

Annex B - of_switch_with_flow.py

```
# Copyright 2012 James McCauley
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
This component is for use with the OpenFlow tutorial.
```

```python
It acts as a simple hub, but can be modified to act like an L2
learning switch.

It's roughly similar to the one Brandon Heller did for NOX.
"""

from pox.core import core
import pox.openflow.libopenflow_01 as of

log = core.getLogger()



class Tutorial (object):
  """
  A Tutorial object is created for each switch that connects.
  A Connection object for that switch is passed to the __init__
     function.
  """
  def __init__ (self, connection):
    # Keep track of the connection to the switch so that we can
    # send it messages!
    self.connection = connection

    # This binds our PacketIn event listener
    connection.addListeners(self)

    # Use this table to keep track of which ethernet address is on
    # which switch port (keys are MACs, values are ports).
    self.mac_to_port = {}


  def resend_packet (self, packet_in, out_port):
    """
    Instructs the switch to resend a packet that it had sent to us.
    "packet_in" is the ofp_packet_in object the switch had sent to
       the
    controller due to a table-miss.
    """
    msg = of.ofp_packet_out()
    msg.data = packet_in

    # Add an action to send to the specified port
    action = of.ofp_action_output(port = out_port)
    msg.actions.append(action)

    # Send message to switch
    self.connection.send(msg)


  def act_like_hub (self, packet, packet_in):
    """
```

```
        Implement hub-like behavior -- send all packets to all ports
            besides
        the input port.
        """

        # We want to output to all ports -- we do that using the special
        # OFPP_ALL port as the output port.  (We could have also used
        # OFPP_FLOOD.)
        self.resend_packet(packet_in, of.OFPP_ALL)

        # Note that if we didn't get a valid buffer_id, a slightly better
        # implementation would check that we got the full data before
        # sending it (len(packet_in.data) should be == packet_in.
            total_len)).


    def act_like_switch (self, packet, packet_in):
        """
        Implement switch-like behavior.
        """

        # Here's some psuedocode to start you off implementing a learning
        # switch.  You'll need to rewrite it as real Python code.

        # Learn the port for the source MAC
        #self.mac_to_port ... <add or update entry>
        self.mac_to_port.update({packet.src: packet_in.in_port})

        log.debug(self.mac_to_port)

        if  self.mac_to_port.get(packet.dst) is not None:

            # Send packet out the associated port
            self.resend_packet(packet_in, self.mac_to_port.get(packet.dst
                ))

            # Once you have the above working, try pushing a flow entry
            # instead of resending the packet (comment out the above and
            # uncomment and complete the below.)

            log.debug("Installing␣flow...")
            # Maybe the log statement should have source/destination/port
                ?
            log.debug("Source:␣" + str(packet.src) + "␣Destination:␣" +
                str(packet.dst) + "␣Port:␣" + str(self.mac_to_port.get(
                packet.dst)))

            msg = of.ofp_flow_mod()
            #msg.match = of.ofp_match.from_packet(packet)
            msg.match.in_port = packet_in.in_port
            msg.match.dl_dst = packet.dst
            msg.actions.append(of.ofp_action_output(port = self.
```

```python
            mac_to_port.get(packet.dst)))

        self.connection.send(msg)
        #
        ## Set fields to match received packet
        #msg.match = of.ofp_match.from_packet(packet)
        #
        #< Set other fields of flow_mod (timeouts? buffer_id?) >
        #
        #< Add an output action, and send -- similar to resend_packet
            () >

    else:
      # Flood the packet out everything but the input port
      # This part looks familiar, right?
      self.resend_packet(packet_in, of.OFPP_ALL)


  def _handle_PacketIn (self, event):
    """
    Handles packet in messages from the switch.
    """

    packet = event.parsed # This is the parsed packet data.
    if not packet.parsed:
      log.warning("Ignoring incomplete packet")
      return

    packet_in = event.ofp # The actual ofp_packet_in message.

    # Comment out the following line and uncomment the one after
    # when starting the exercise.
    #self.act_like_hub(packet, packet_in)
    self.act_like_switch(packet, packet_in)



def launch ():
  """
  Starts the component
  """
  def start_switch (event):
    log.debug("Controlling %s" % (event.connection,))
    Tutorial(event.connection)
  core.openflow.addListenerByName("ConnectionUp", start_switch)
```