Student Name: Marco Ermini  
Email: mermini@asu.edu  
Submission Date: 14th Jun, 2021  
Class Name and Term: CSE548 Summer 2021

Project Report 2 – SDN-Based Stateless Firewall

# Project Overview

In this lab we are exploring how to set up a software defined environment based on mininet and containernet. We also get to practice how to set up an OpenFlow based flow-level firewall on SDN. Finally, we need to set up and practice flow-based firewall filtering policies such as enabling the ability to accept, drop, or reject the incoming flows thus ensuring the safety of the system from malicious attacking network traffic.

All the files and configurations used for this lab have been uploaded on GitHub; references are provided throughout the text and in the Appendix A at the bottom of the File.

# Network Setup

Since I have experienced some issues with connecting the VM to my lab through my host PC running Windows 10, I have chosen to set up the VM in VirtualBox in a bridged network configuration.

In this way, I could avoid configuring a static IP address to the VM and simply opted to fetch an IP for the VM via DHCP; this IP is then assigned directly to my router, which can be useful to troubleshoot eventual connectivity issues. On the Ubuntu/Linux side this has brought no issues whatsoever, once the configuration commands are adjusted (e.g. use “dhclient br0” rather than assigning an IP address with “ifconfig br0”).

Because of the use of DHCP, depending on the lab run and when I restarted the VM, it may have assumed a different address in the 192.168.1/24 network. This does not affect the outcome of the lab exercises, but it may look inconsistent in the screenshots. Apologies about that.

Please find below the initial set-up of the virtual infrastructure as I have configured it in VirtualBox.

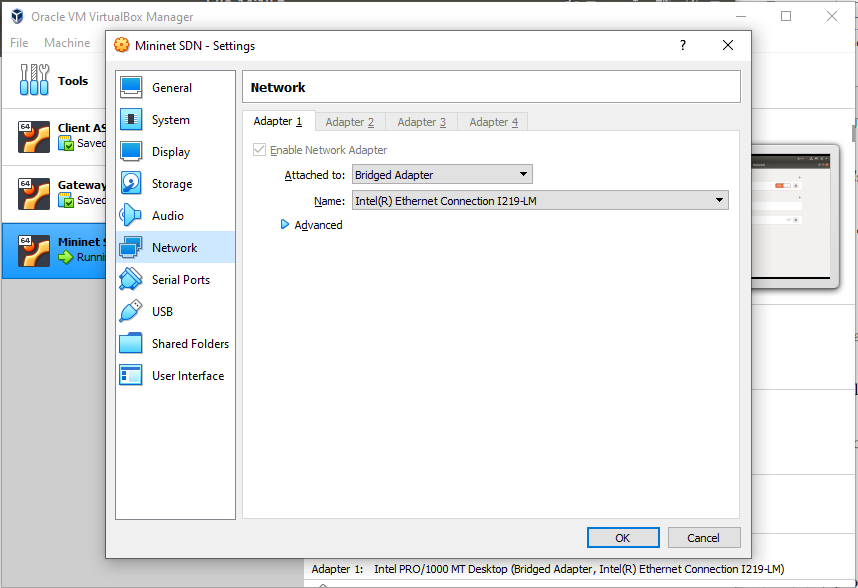


Figure 1 - Bridged network setup in VirtualBox

# Software

For this first lab, the following software has been used:

* Various network tools (specifically, tcpdump, ping, traceroute, hping3, and nc – netcat)
* POX (GitHub link: <https://noxrepo.github.io/pox-doc/html/>
* Open vSwitch: <http://www.openvswitch.org/>
* Open vSwitch Cheat Sheet: <https://therandomsecurityguy.com/openvswitch-cheatsheet/>
* Containernet: <https://containernet.github.io/>
* Containernet tutorial: <https://github.com/containernet/containernet/wiki/Tutorial:-Getting-Started>

# Project Description

In this assignment, I have executed the various labs steps and obtained the proofs that the assignments have been completed.

## Lab “lab-cs-net-00006 OpenVirtual Switch”

As mentioned in the “Network Setup” section, the interesting part of this lab happens because a bridge is required, and I am using DHCP. The following screenshots illustrate how DHCP is used on the bridge and the MAC address of the VM external interface is assigned to the bridge.

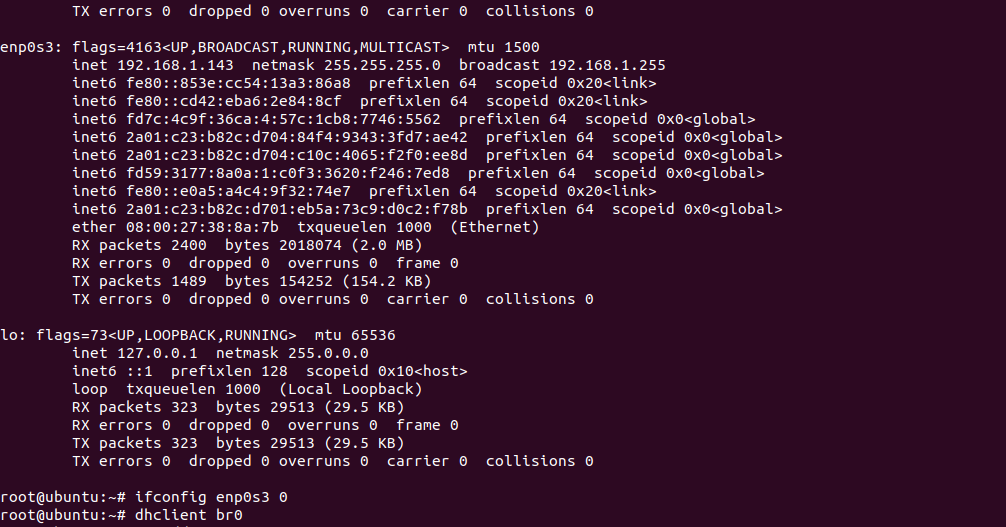


Figure 2 - On the created bridge br0, dhclient is run

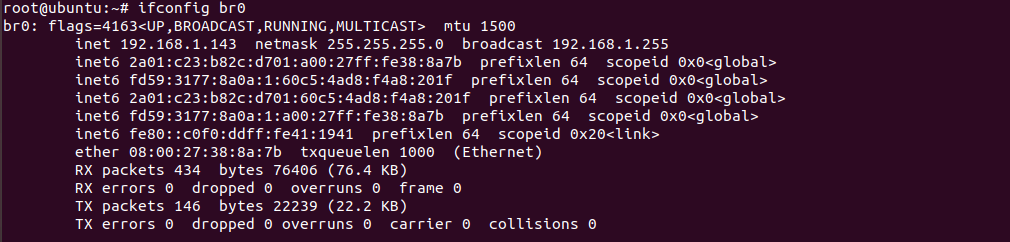


Figure 3 - The bridge inherits the same IP of the enp0s3 interface

The bridge obtains the same IP from the router because it uses the same MAC address of the interface, and therefore the route is just reusing a DHCP lease already assigned.

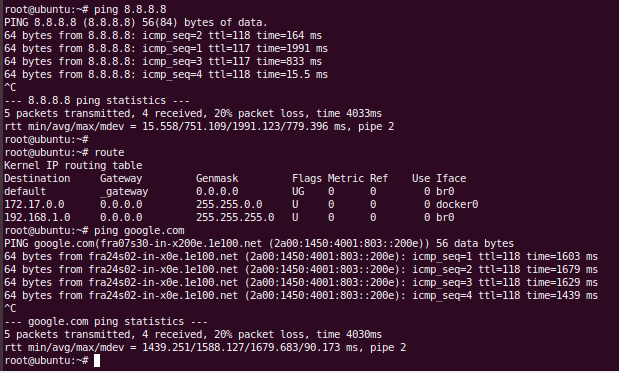


Figure 4 - Pinging the outside world

As we can see, pinging on some Internet hosts works – even in IPv6! However, there is a performance penalty to be paid by the double software bridge (on the Linux VM and on the VirtualBox host), which causes packet losses. Generally speaking, it caused no issues – I could run *apt* and update and install packages as needed.

### Lab Assessment 1: The OVS…

* …can correctly show the created bridges via the open vswitch command:

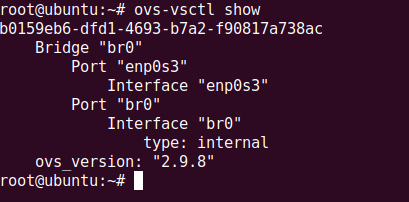


Figure 5 - lab-cs-net-00006 - Lab 1-1

* … can show the correct bound between the br0 and ens33 (in my case, enp0s3):

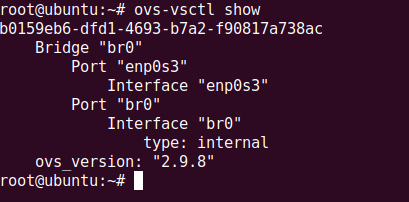


Figure 6 - lab-cs-net-00006 - Lab 1-2

* …can show the correct IP address assigned to the br0 (in my case, it is obtained via DHCP):

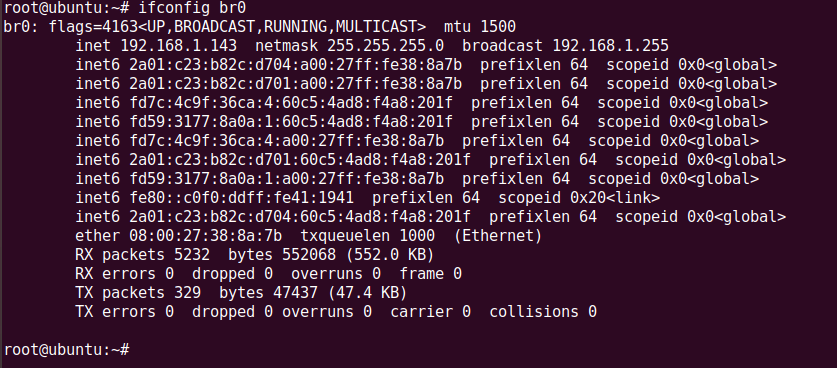


Figure 7- lab-cs-net-00006 - Lab 1-3/1

As a counterproof, I am showing that the ep0s3 interface has no (IPv4) address. To note that IPv6 is unaffected by “ifconfig 0” (I wonder if this has to do with the packet loss; note that both interfaces’ statistics do not report any transmission error):

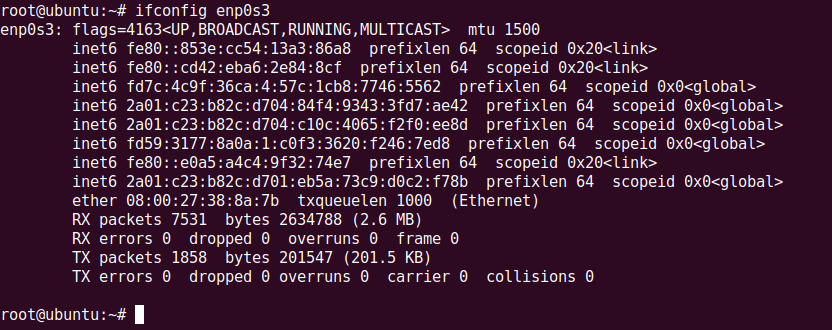


Figure 8 - lab-cs-net-00006 - Lab 1-3/2

### Lab Assessment 2: After finishing the configuration of br0 in OVS, the machine…

* …can show the correct routing table for especially for the br0 and ens3 (in my case, enp0s3):

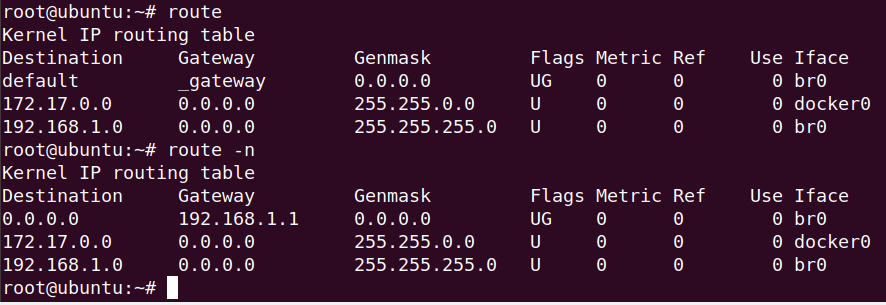


Figure 9 - lab-cs-net-00006 - Lab 2-1

* …can ping outside such as ping 8.8.8.8 correctly:

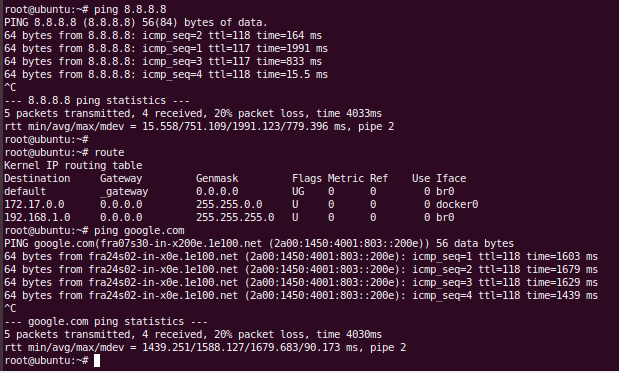


Figure 10 - Ping outside

## Lab “lab-cs-net-00007 - Mininet”

### Setting up mininet and Running mininet topology…

* *…can correctly create the topology with a single switch and four hosts*

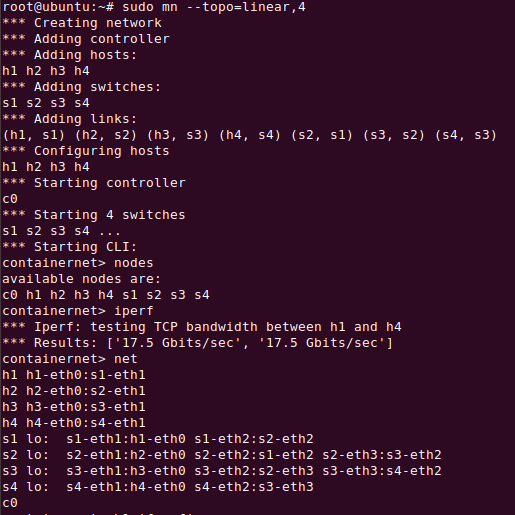


Figure 11 - lab-cs-net-0007 - Lab 1-1

* …can correctly create the linear topology with five switch and one host for every switch:

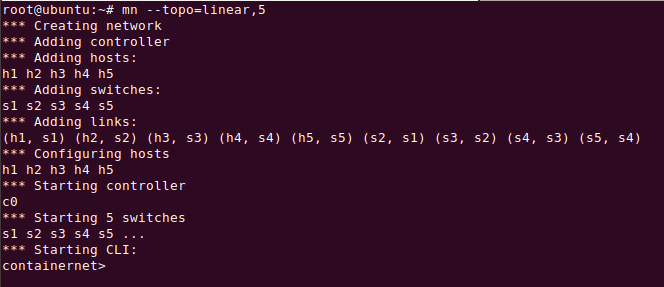


Figure 12- lab-cs-net-0007 - Lab 1-2

### For each topology above, they should be able to:

* show the correct number of nodes within the current topology

Single switch and four hosts:

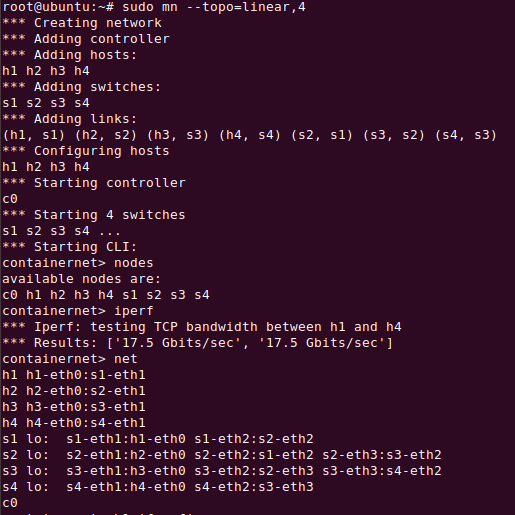


Figure 13 - lab-cs-net-0007 - Lab 2-1 – topology 1

Linear topology of five switch and one host for each switch:

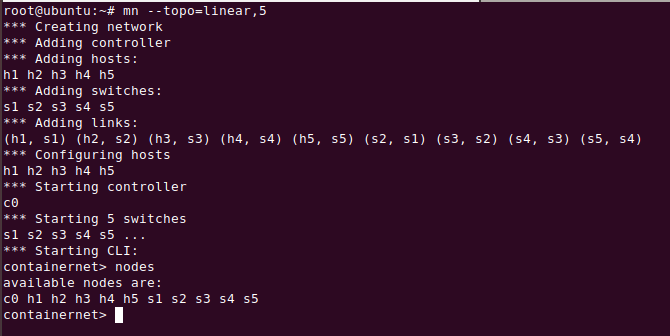


Figure 14 - lab-cs-net-0007 - Lab 2-1 – topology 2

* …perform bandwidth measurement.

Single switch and four hosts:

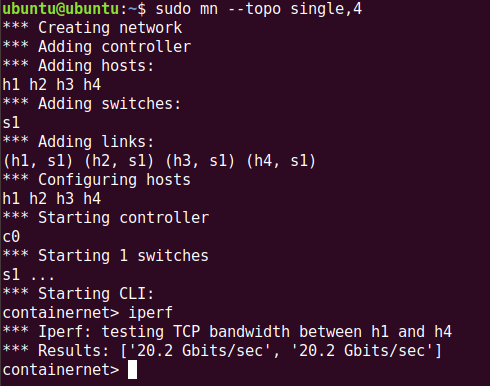


Figure 15- lab-cs-net-0007 - Lab 2-2 - Topology 1

Linear topology of five switch and one host for each switch:

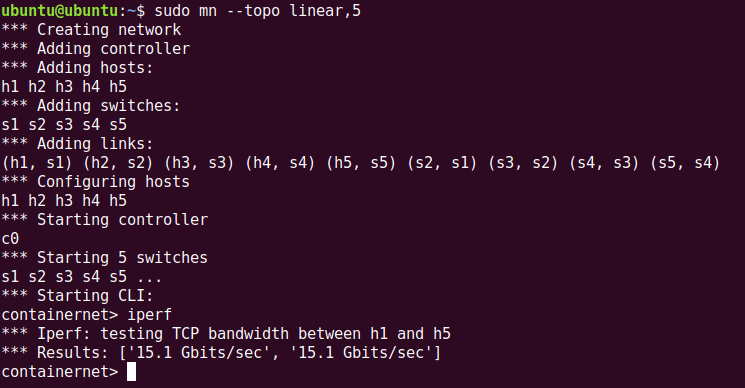


Figure 16- lab-cs-net-0007 - Lab 2-2 - Topology 2

* …display the correct link information among hosts and switches:

Single switch and four hosts:

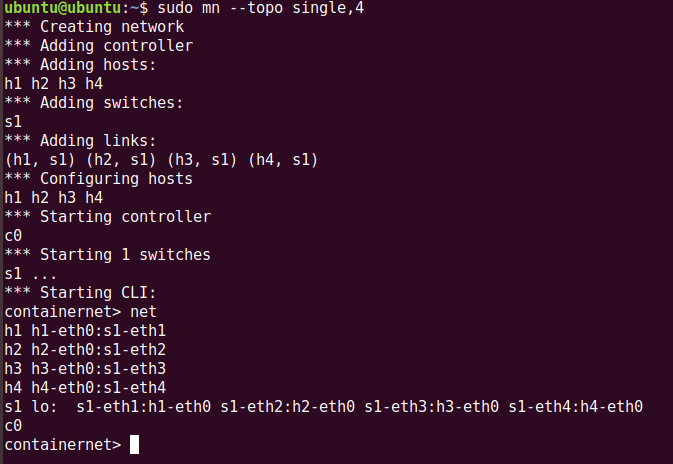


Figure 17- lab-cs-net-0007 - Lab 2-3 - Topology 1

Linear topology of five switch and one host for each switch:

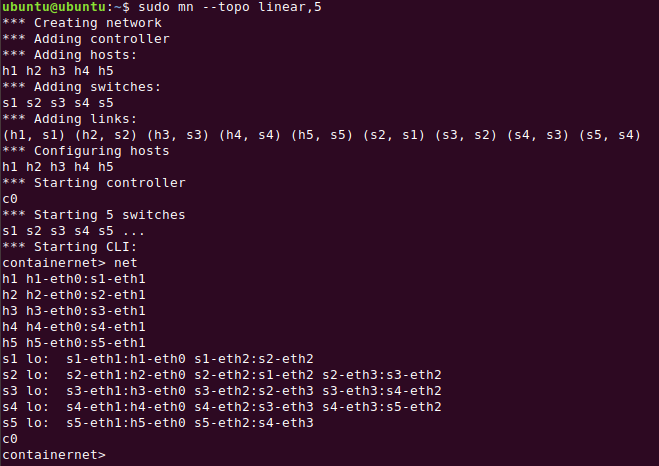


Figure 18- lab-cs-net-0007 - Lab 2-3 - Topology 2

### Create another tree topology of depth 2 and fanout 8.

* Startup with “*sudo mn --topo tree,depth=2 fanout=8*”:

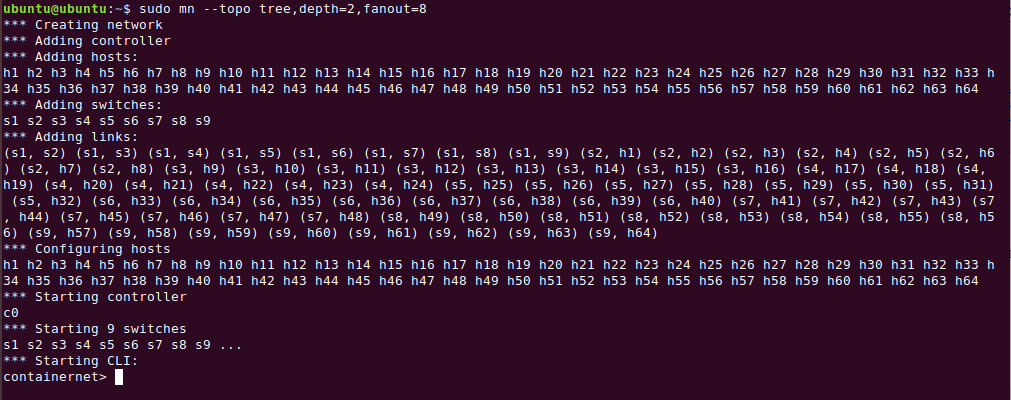


Figure 19- lab-cs-net-0007 - Lab 3-1

* the host1 can correctly ping the host64:

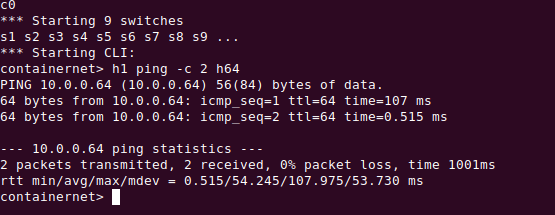


Figure 20- lab-cs-net-0007 - Lab 3-2

(End of “lab-cs-net-00007 - Mininet”)

## Lab “CS-NET-00008 – SDN Controller (POX)”

### The POX controller can:

* connect to a mininet topology:

The behavior of pox and mininet can be observed by switching between the two tabs where the programs are running, showing that mininet connected to openflow:

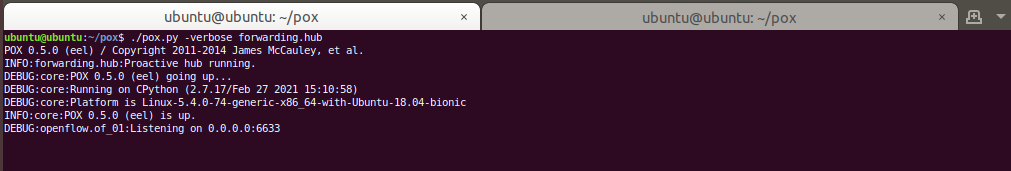


Figure 21 - lab-cs-net-0087 - Lab 1-1/1

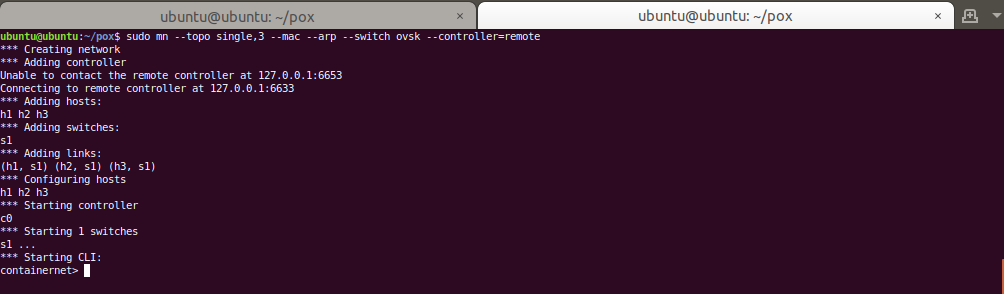


Figure 22 - lab-cs-net-0087 - Lab 1-1/2

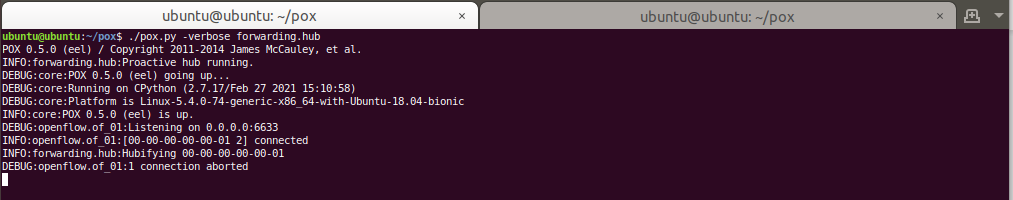


Figure 23 - lab-cs-net-0087 - Lab 1-1/3

* start the forwarding.l3 learning component correctly

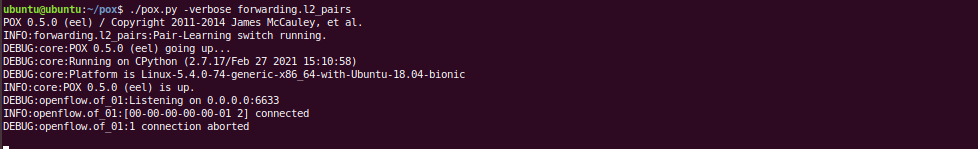


Figure 24 - lab-cs-net-0087 - Lab 1-2/1

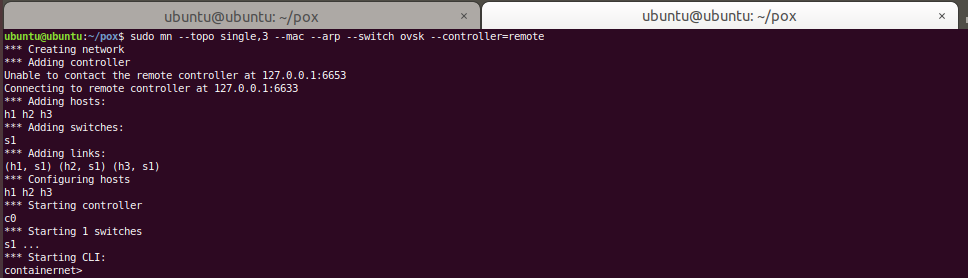


Figure 25 - lab-cs-net-0087 - Lab 1-2/2

### While connect to the controller, the mininet topology can (Please take snapshots for each task)

* open the xterm terminal correctly
* enable the ping command between two different host in two different xterm terminal (two different hosts)

On this first image, I have opened the three xterm with the command “xterm h1 h2 h3” highlighted on the bottom. I have also run a couple of ping commands.

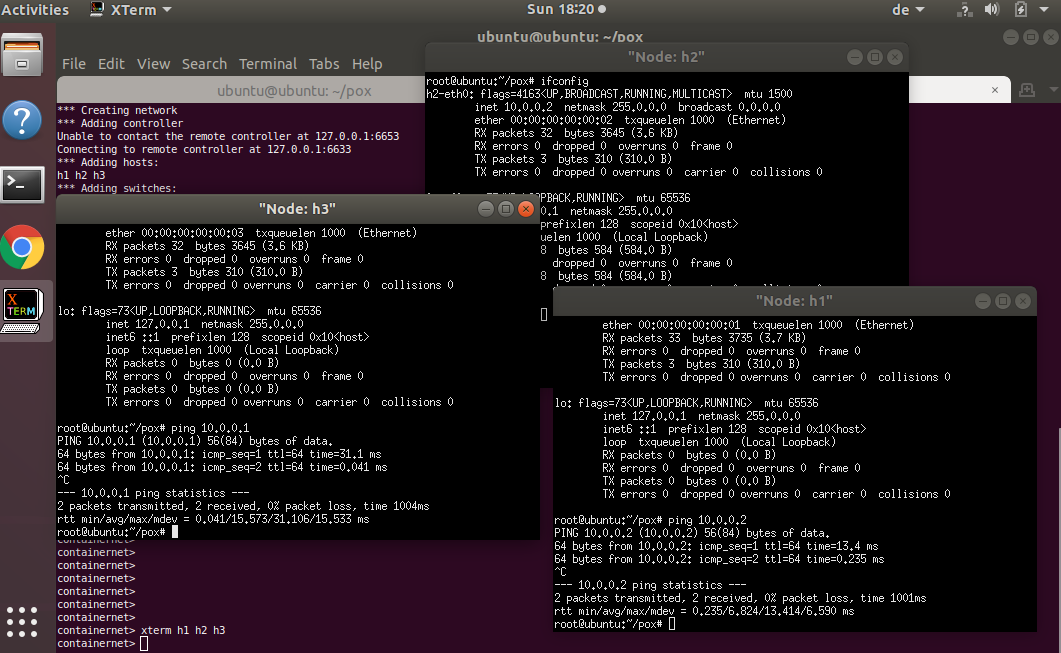


Figure 26 - lab-cs-net-0087 - Lab 2-1

In this screenshot, I am running tcpdump on one xterm while executing a ping from another. In this way it is possible to notice the incoming packets.

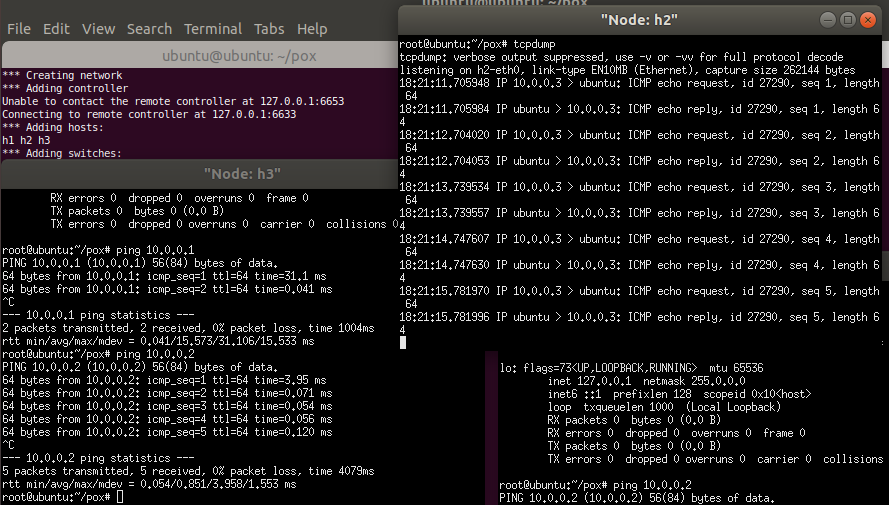


Figure 27 - lab-cs-net-0087 - Lab 2-2

### Extra Activity

I have started the pox webserver connected to mininet to test its functionality.

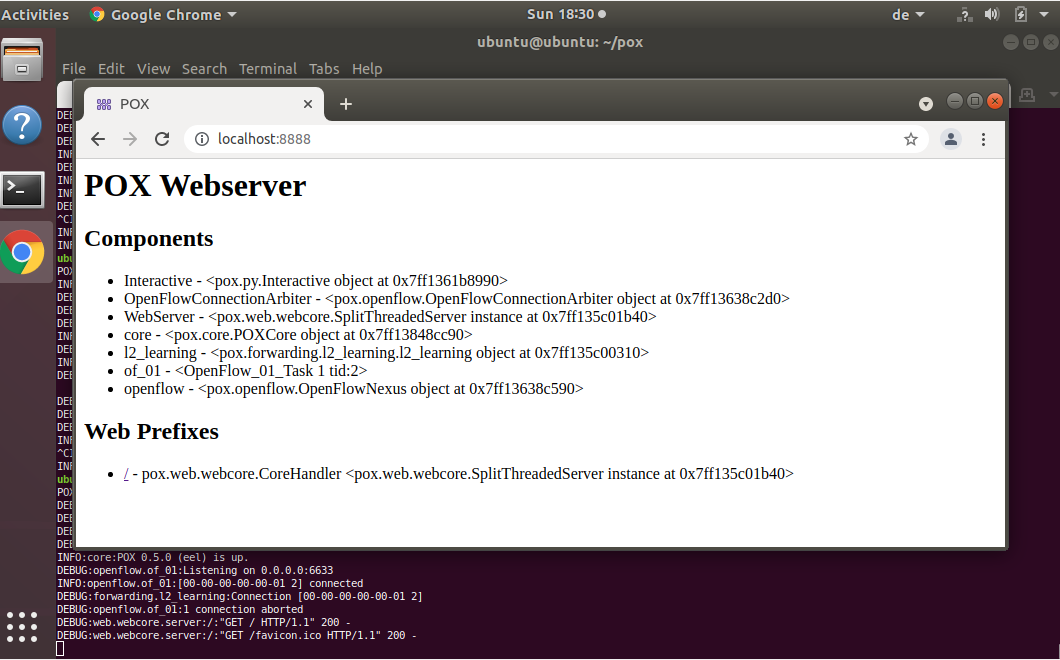
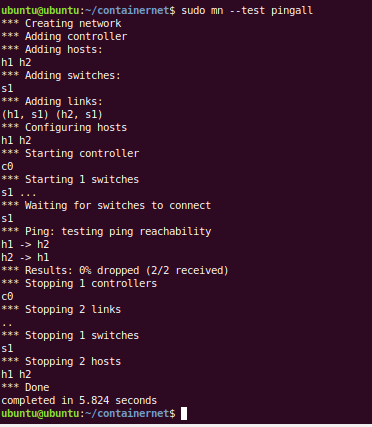


Figure 28 - - lab-cs-net-0087 - Extra Activity

## Lab “CS-NET-00009 – Containernet”

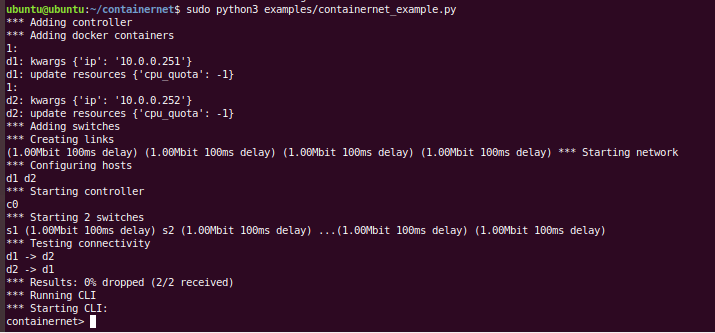
### The containernet topology can:

* successfully implement the test ping command in a temporary topology

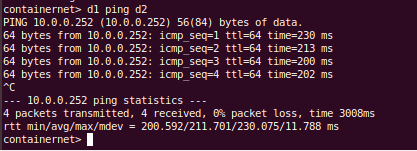


After started the scripts the container example.py can

* create the indicated topology



* enable the communication between the d1 and d2 (link up the two switches):



(End of “lab-cs-net-00009 - Containernet”)

## Lab “CS-CNS-00101” – OpenFlow Based Stateless firewall

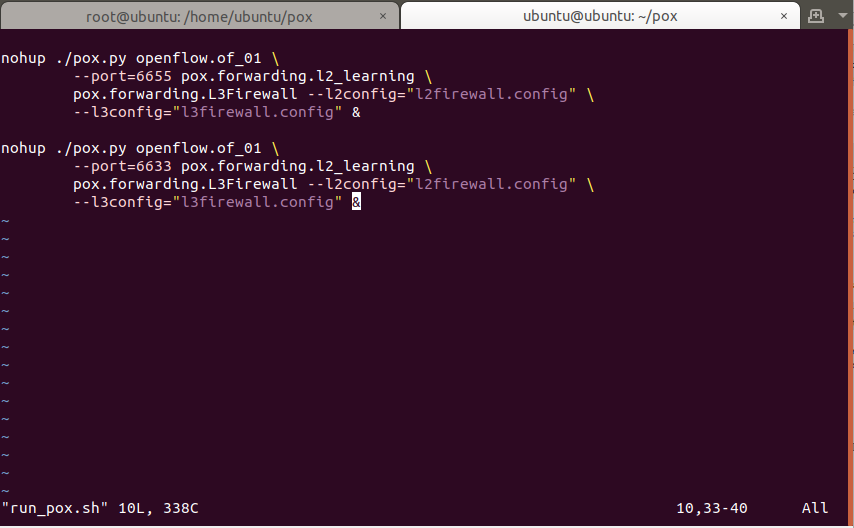
In this lab, the students are required to verify working of stateless firewall and try adding different rules using

config files.

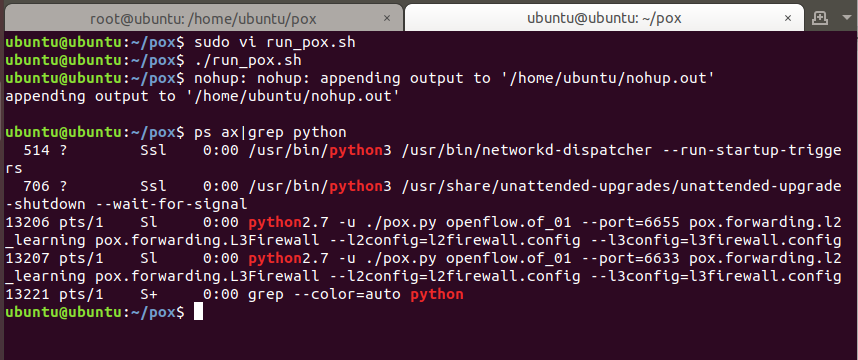
### Create a mininet based topology with 4 container hosts and one controller switches and run it.

* + Add link from controller1 to switch 1.
  + Add link from controller2 to switch 1.
  + Add link from switch 1 to container 1.
  + Add link from switch 1 to container 2.
  + Add link from switch 1 to container 3.
  + Add link from switch 1 to container 4.

This is accomplished by two steps: first, creating two controller with pox (only one of the two will be really used). I have created a script, called “run\_pox.sh” (present on GitHub and the Appendices), which spawns two controllers, as show below.



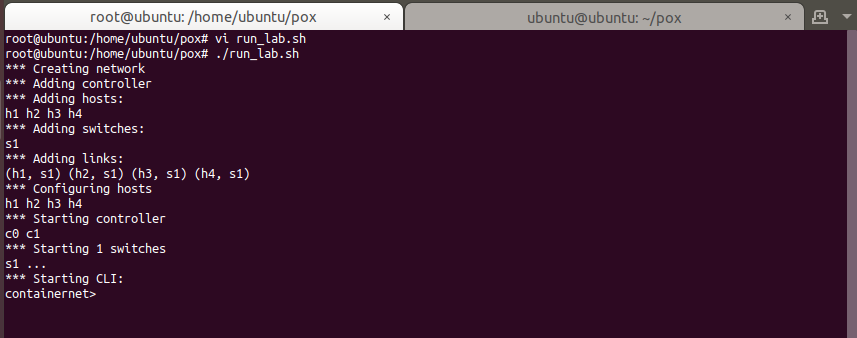
The controllers are identifiable as python programs from the Ubuntu process status:



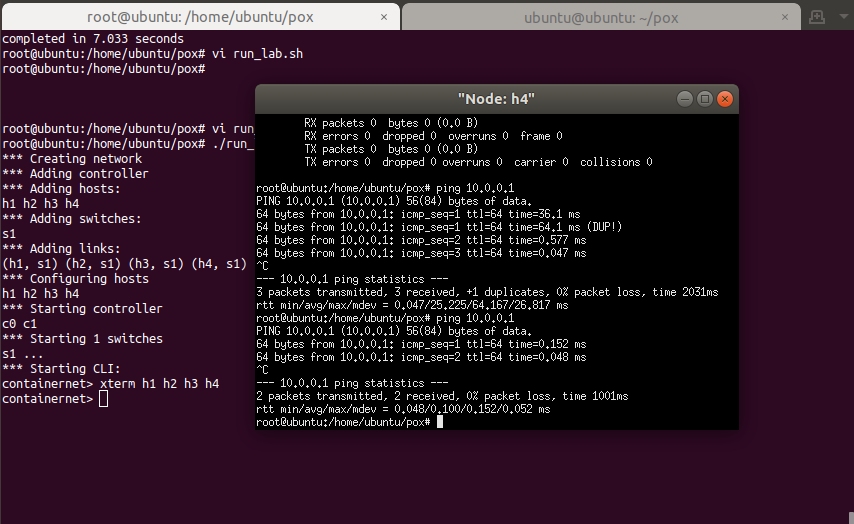
The second step is to run another script from another Terminal tab or windows, launching mininet. This file, called “run\_lab.sh”, is also presented in the Appendices and GitHub.



After mininet starts, it will bind to the two pox controllers without errors (if everything goes smoothly):



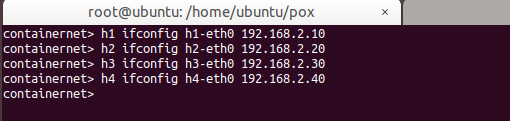
At this point we can run the xterm and observe that the containers work.



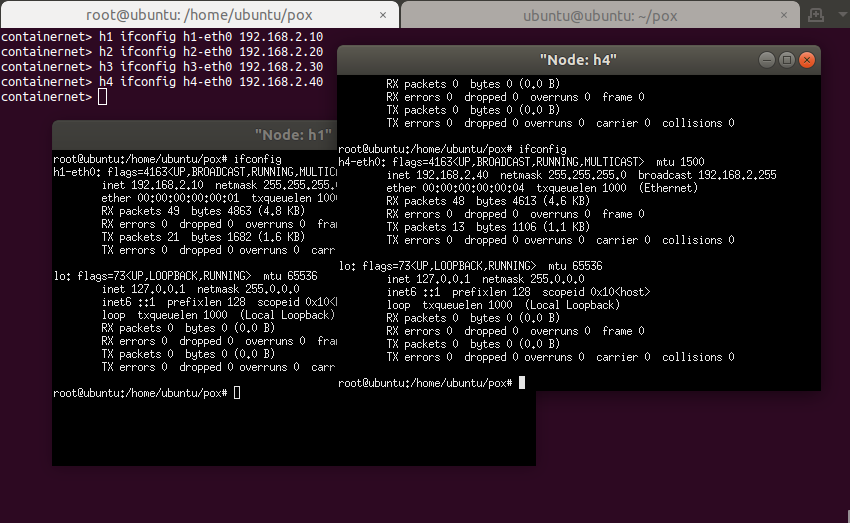
### Make the interfaces up and assign IP addresses to interfaces of container hosts.

* *Assign IP address 192.168.2.10 to container host #1.*
* *Assign IP address 192.168.2.20 to container host #2.*
* *Assign IP address 192.168.2.30 to container host #3.*
* *Assign IP address 192.168.2.40 to container host #4*

The following commands are ran from the mininet CLI to assign them the addresses requested by the lab:



The next screenshot to document that the containers have all assumed the correct IP addresses.



### Add new rule to l3config file for blocking ICMP traffic from source IP 192.168.2.10 and destination IP 192.168.2.30.

### Add new rule to l3config file for blocking ICMP traffic from source IP 192.168.2.20 and destination IP 192.168.2.40.

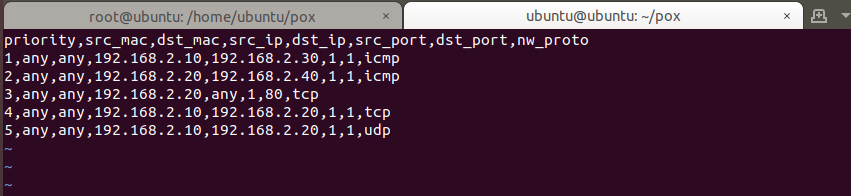
### Add new rule to l3config file for blocking HTTP traffic from source IP 192.168.2.20.

### Add new rule to l2config file for blocking traffic from MAC address 00:00:00:00:00:02 to destination MAC address 00:00:00:00:00:04.

### Add new rule to l3config file for blocking tcp traffic from 192.168.2.10 to 192.168.2.20.

### Add new rule to l3config file for blocking udp traffic from 192.168.2.10 to 192.168.2.20.

The following screenshot documents the l3config.file:



priority,src\_mac,dst\_mac,src\_ip,dst\_ip,src\_port,dst\_port,nw\_proto

1,any,any,192.168.2.10,192.168.2.30,1,1,icmp

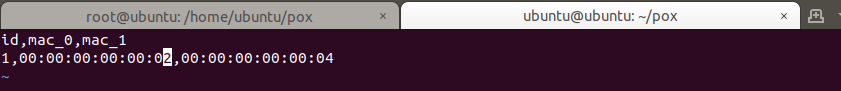
2,any,any,192.168.2.20,192.168.2.40,1,1,icmp

3,any,any,192.168.2.20,any,1,80,tcp

4,any,any,192.168.2.10,192.168.2.20,1,1,tcp

5,any,any,192.168.2.10,192.168.2.20,1,1,udp

The following screenshot documents the l2config.file:

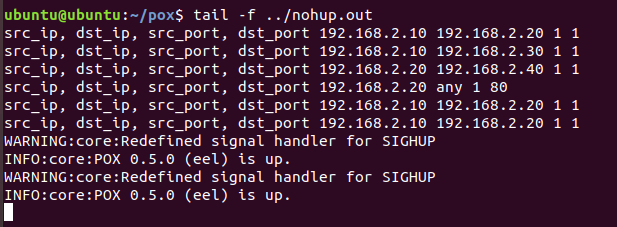


id,mac\_0,mac\_1

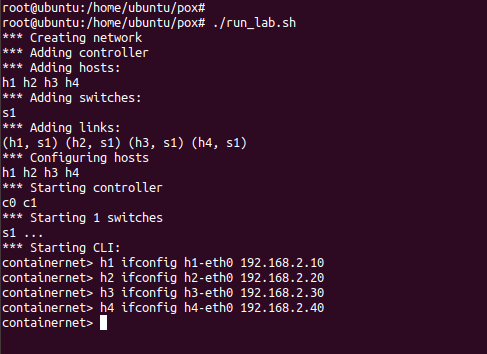
1,00:00:00:00:00:02,00:00:00:00:00:04

At this point, we restart both pox and mininet.

From looking at the “nohup.out” generated by the run\_pox.sh, it is possible to see the Layer3 rules enabled (but not the Layer2):

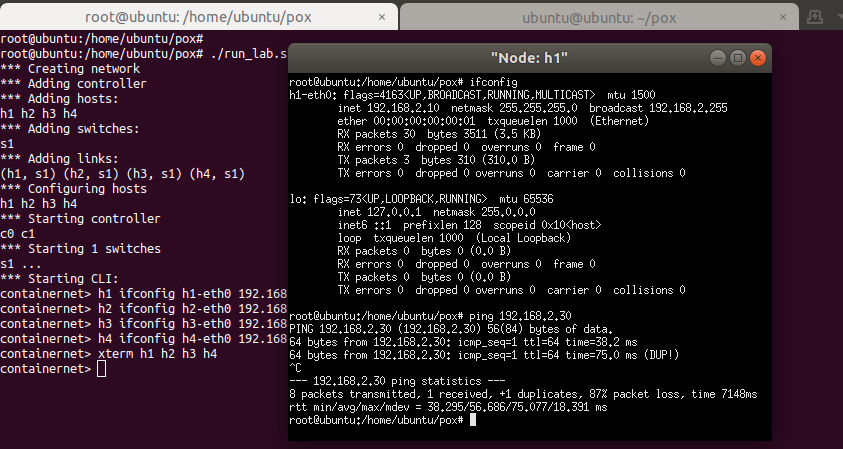


After we restart mininet, we must re-assign the correct IP addresses (fortunately, the CLI history has stored the command, so we must just retrieve them with the arrows and feed them to the Open vSwitch pressing Enter):

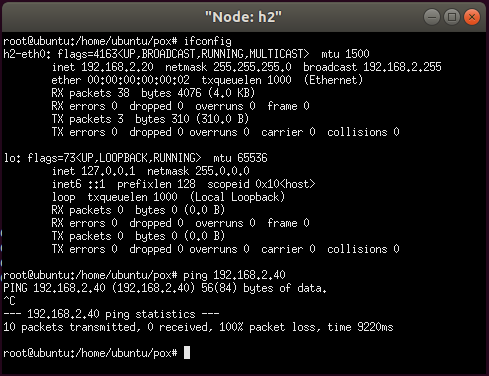


Let’s now test out rules to see if they work.

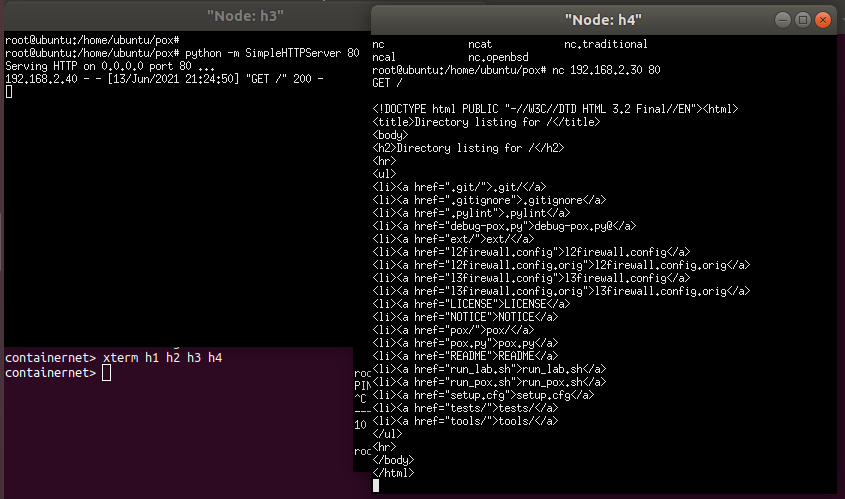
Let’s ping h3 from h1. The ping should fail, but in fact, the first packet gets true, but all the subsequent are blocked. Seems like one rule must be triggered first before the vSwitch “loads” them in, but at this point they are all enabled.



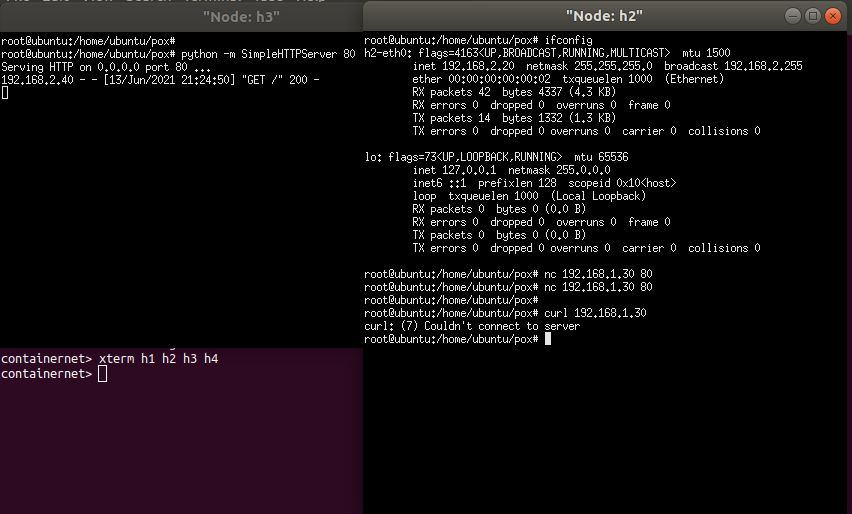
In fact, pinging h4 from h2 fails (not even the first packet gets through):



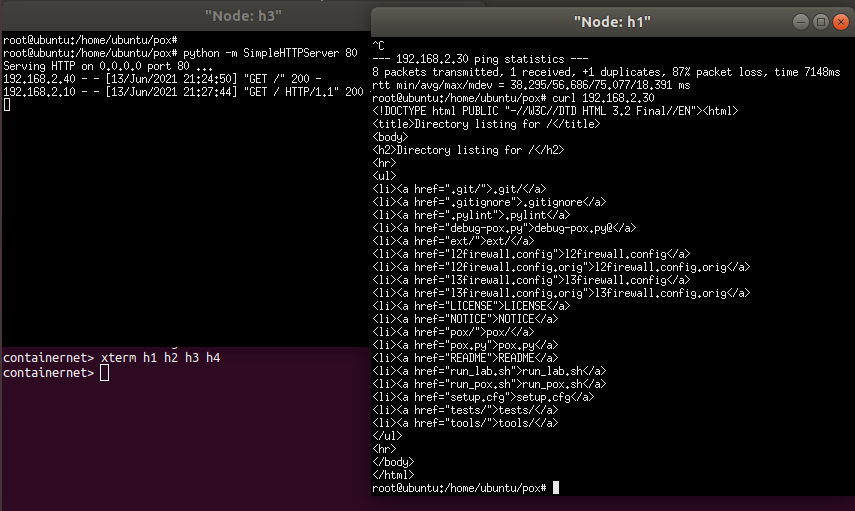
I start the Python SimpleHTTPServer on port 80 of node h3 and test fetching a page from the node h4, which works (as expected):



However, trying to “browse” the web page at port 80 from node h2 – no matter if with netcar or curl – fails as the connection is dropped and, as we can see, it never reaches the node h3 – confirming the Layer3 rule works as intended:

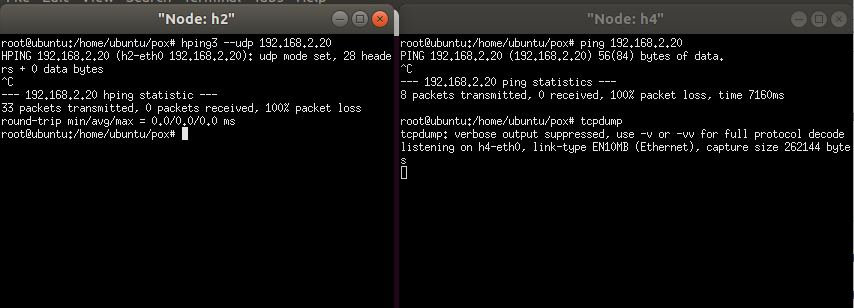


Another test against the web server on h3, this time from h1, shows that effectively only h2 is blocked from reaching web pages.

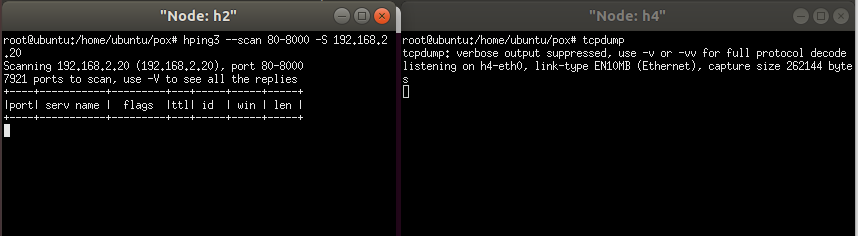


Finally, between h2 and h4, the block should be at the Layer2 level. To test this, I have tried first using ping, then running tcpdump on h4, while doing scans from node h2.

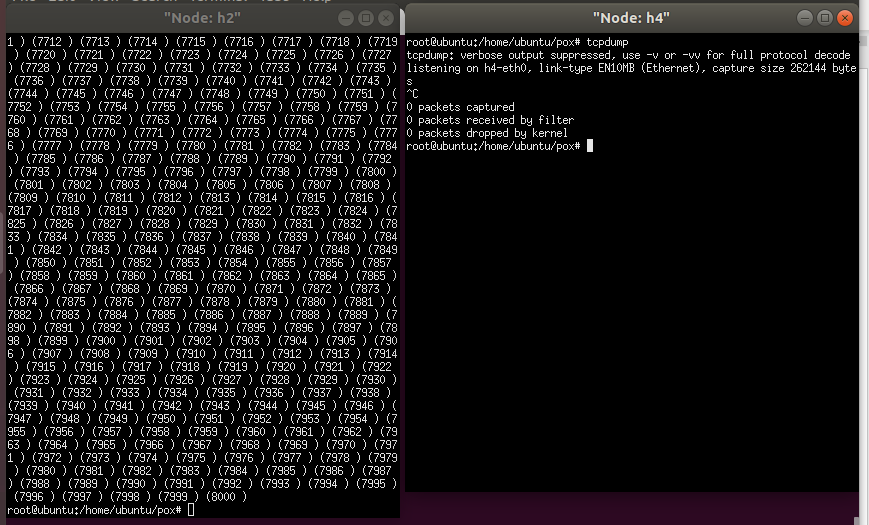
The first is a UDP scan:



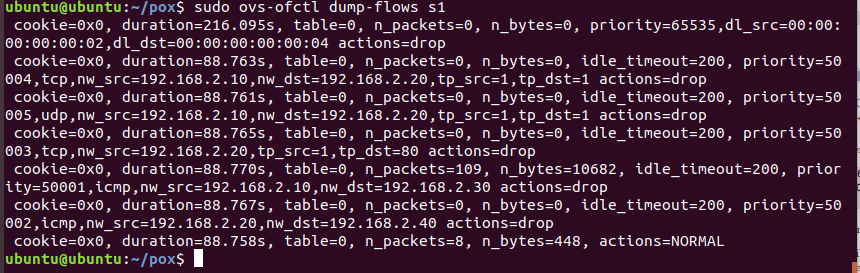
The second is a TCP scan on some 8000 circa ports. As we can see, no packet ever reaches node h4 – be it ICPM, TCP or UDP.



This is how hping3 spits out its failure against h4; the result from h4 shows that no packet has ever reached the interface.



Finally, we can dump the flows directly from the vSwitch, to verify that rules are in place (packet counts only show a few rules because the screenshot has been taken throughout restarts):



# Appendix A: Files for the Lab

Please find the list of files created for this lab and mentioned throughout this document, plus their GitHub link for download.

|  |  |
| --- | --- |
| rc.sh | <https://github.com/markoer73/CSE-548/blob/main/Project%201%20-%20Packet%20Filter%20Firewall/rc.sh> |
| index.html | <https://github.com/markoer73/CSE-548/blob/main/Project%201%20-%20Packet%20Filter%20Firewall/index.html> |
| test-and-demo.conf | <https://github.com/markoer73/CSE-548/blob/main/Project%201%20-%20Packet%20Filter%20Firewall/test-and-demo.conf> |
| ports.conf | <https://github.com/markoer73/CSE-548/blob/main/Project%201%20-%20Packet%20Filter%20Firewall/ports.conf> |
| 02-gateway-networks.yaml | <https://github.com/markoer73/CSE-548/blob/main/Project%201%20-%20Packet%20Filter%20Firewall/02-gateway-networks.yaml> |
| 02-internal-network.yaml | <https://github.com/markoer73/CSE-548/blob/main/Project%201%20-%20Packet%20Filter%20Firewall/02-internal-network.yaml> |

# References

* Linux NAT Tutorial: <https://www.karlrupp.net/en/computer/nat_tutorial>
* Ubuntu “Basic Iptables HOWTO”: <https://help.ubuntu.com/community/IptablesHowTo>
* “Iptables Tutorial: Ultimate Guide to Linux Firewall”: <https://phoenixnap.com/kb/iptables-tutorial-linux-firewall>

VIII. Table of Figures

[Figure 1 - Network Map of the Lab 1](#_Toc73261405)

[Figure 2 - VirtualBox Preferences 3](file:///C:\Users\DJSM0911\Documents\Personal\Training\ASU%20MasterTrack\CSE%20548%20-%20Advanced%20Computer%20Network%20Security\GitHub\CSE-548\Project%201%20-%20Packet%20Filter%20Firewall\Project-Report-1%20Packet%20Filter%20Firewall.docx#_Toc73261406)

[Figure 3 - VirtualBox Networks Configuration 3](file:///C:\Users\DJSM0911\Documents\Personal\Training\ASU%20MasterTrack\CSE%20548%20-%20Advanced%20Computer%20Network%20Security\GitHub\CSE-548\Project%201%20-%20Packet%20Filter%20Firewall\Project-Report-1%20Packet%20Filter%20Firewall.docx#_Toc73261407)

[Figure 4 - InternetNetwork configuration 3](file:///C:\Users\DJSM0911\Documents\Personal\Training\ASU%20MasterTrack\CSE%20548%20-%20Advanced%20Computer%20Network%20Security\GitHub\CSE-548\Project%201%20-%20Packet%20Filter%20Firewall\Project-Report-1%20Packet%20Filter%20Firewall.docx#_Toc73261408)

[Figure 5 - NatNetwork Configuration 3](file:///C:\Users\DJSM0911\Documents\Personal\Training\ASU%20MasterTrack\CSE%20548%20-%20Advanced%20Computer%20Network%20Security\GitHub\CSE-548\Project%201%20-%20Packet%20Filter%20Firewall\Project-Report-1%20Packet%20Filter%20Firewall.docx#_Toc73261409)

[Figure 6 - Gateway’s Adapter 1 set up on VirtualBox 3](file:///C:\Users\DJSM0911\Documents\Personal\Training\ASU%20MasterTrack\CSE%20548%20-%20Advanced%20Computer%20Network%20Security\GitHub\CSE-548\Project%201%20-%20Packet%20Filter%20Firewall\Project-Report-1%20Packet%20Filter%20Firewall.docx#_Toc73261410)

[Figure 7 - Gateway’s Adapter 2 set up on VirtualBox 3](file:///C:\Users\DJSM0911\Documents\Personal\Training\ASU%20MasterTrack\CSE%20548%20-%20Advanced%20Computer%20Network%20Security\GitHub\CSE-548\Project%201%20-%20Packet%20Filter%20Firewall\Project-Report-1%20Packet%20Filter%20Firewall.docx#_Toc73261411)

[Figure 8 - Client's Adapter Configuration 4](#_Toc73261412)

[Figure 9 - Gateway VM's interfaces as they appear at the Linux console 4](file:///C:\Users\DJSM0911\Documents\Personal\Training\ASU%20MasterTrack\CSE%20548%20-%20Advanced%20Computer%20Network%20Security\GitHub\CSE-548\Project%201%20-%20Packet%20Filter%20Firewall\Project-Report-1%20Packet%20Filter%20Firewall.docx#_Toc73261413)

[Figure 10 - Client VM's interface as it appears at the Linux console 4](file:///C:\Users\DJSM0911\Documents\Personal\Training\ASU%20MasterTrack\CSE%20548%20-%20Advanced%20Computer%20Network%20Security\GitHub\CSE-548\Project%201%20-%20Packet%20Filter%20Firewall\Project-Report-1%20Packet%20Filter%20Firewall.docx#_Toc73261414)

[Figure 11 - Gateway VM's initial forward routing and iptable rules 4](#_Toc73261415)

[Figure 12 - Client's VM initial network situation 5](#_Toc73261416)

[Figure 13 - Client's routing table 5](#_Toc73261417)

[Figure 14 - Initial Gateway internet connection 5](#_Toc73261418)

[Figure 15 - Gateway's routing table 5](#_Toc73261419)

[Figure 16 - Evidence of Gateway connectivity 7](#_Toc73261420)

[Figure 17 - Kernel routing on the Gateway 8](#_Toc73261421)

[Figure 18 - Routing on the Client 9](#_Toc73261422)

[Figure 19 - Lazy NAT configuration on the Gateway 10](#_Toc73261423)

[Figure 20 - Successfully pinging the Internet from the client 10](#_Toc73261424)

[Figure 21 - apt update command 10](#_Toc73261425)

[Figure 22 - apt dist-upgrade 11](#_Toc73261426)

[Figure 23 - Installation of tcpdump on the Gateway 11](#_Toc73261427)

[Figure 24 - Installation of Apache2 on the Gateway 11](#_Toc73261428)

[Figure 25 - Disabling of ufw 12](#_Toc73261429)

[Figure 26 - Adding the host name to /etc/hosts 13](#_Toc73261430)

[Figure 27 - the index.html of our test web site 13](#_Toc73261431)

[Figure 28 - Firefox browser testing out web site on the Gateway 14](#_Toc73261432)

[Figure 29 - /etc/hosts file on the Client 15](#_Toc73261433)

[Figure 30 - Firefox browser testing out web site from the Client 15](#_Toc73261434)

[Figure 31 - One run of the rc.sh script 16](#_Toc73261435)

[Figure 32 - Testing connectivity from the Client 17](#_Toc73261436)

[Figure 33 - Pinging Google DNS from the Client 17](#_Toc73261437)

[Figure 34 - tcpdump run from the Gateway 18](#_Toc73261438)

[Figure 35 - Running Wireshark on the Gateway 18](#_Toc73261439)

[Figure 36 - Internet web sites timing out on the Client 19](#_Toc73261440)

[Figure 37 - Example of monitoring for rule drops via SYSLOG. 20](#_Toc73261441)