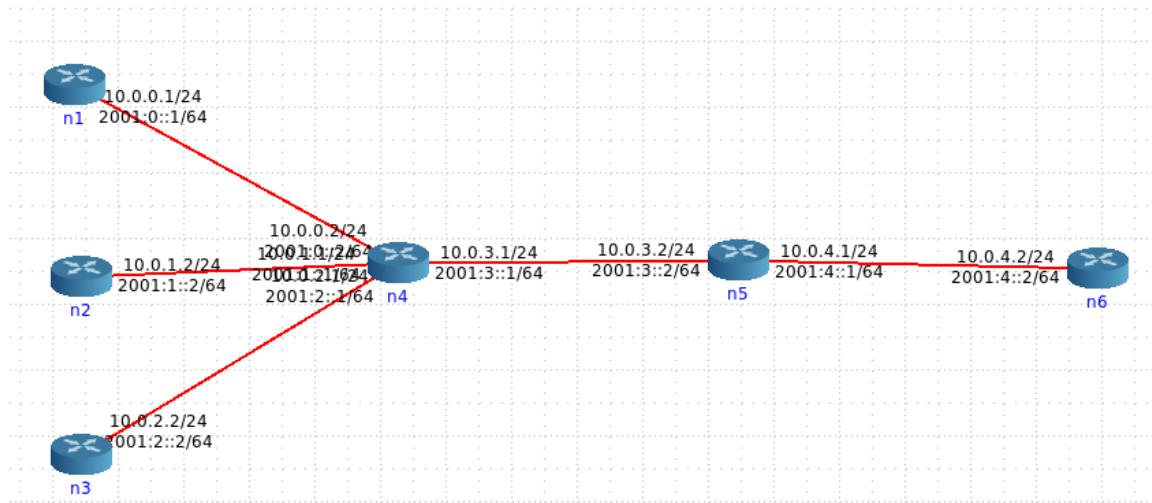


Network Security – CSCI_6541_80

Namana Y Tarikere – G21372717

Homework Assignment – 7

Firewall Section (10pts)



Useful commands:

- Flush the filter table: `iptables -F`
- Flush the NAT table: `iptables -F -t NAT`
- List the rules in the filter table: `iptables -L`
- List the rules in the filter table with #packet, #bytes matching each rule: `iptables -L -n -v`
- Change the default policy of the INPUT chain of the filter table to DROP: `iptables -P INPUT DROP`

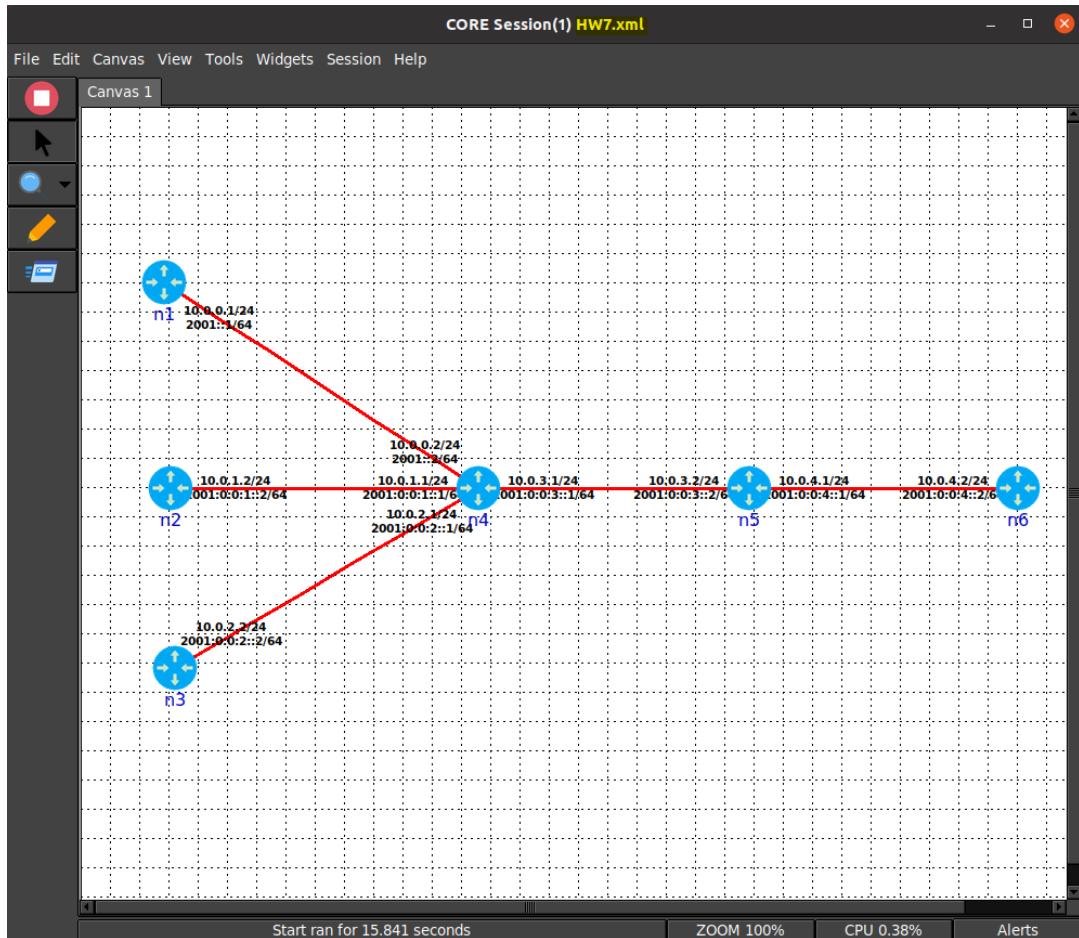
Use IPTTables (<http://www.netfilter.org/projects/iptables/index.html>) to apply the following rules **at n4** to the network below.

For each question, **provide a list of firewall rules to apply at n4**. Associate rules with each of the requirements above. Please provide screenshots and make each expand to the width of the page.

Example:

1. a. `iptables -A etc.`

Core GUI:



1) (2.5pts) Drop 40% of all ICMP packets from n2

- Show the iptables command you used?

Command for dropping input and forward packets coming to n4 from n2:

- iptables -A INPUT -s 10.0.1.2 -p icmp -m statistic --mode random --probability 0.4 -j DROP**
- iptables -A FORWARD -s 10.0.1.2 -p icmp -m statistic --mode random - probability 0.4 -j DROP**
- iptables -L**

```
root@n4:/tmp/pycore.1/n4.conf# iptables -A INPUT -s 10.0.1.2 -p icmp -m statistic --mode random --probability 0.4 -j DROP
root@n4:/tmp/pycore.1/n4.conf# iptables -A FORWARD -s 10.0.1.2 -p icmp -m statistic --mode random --probability 0.4 -j DROP
root@n4:/tmp/pycore.1/n4.conf# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source          destination
DROP      icmp --  10.0.1.2        anywhere           statistic mode random probability 0.39999999991

Chain FORWARD (policy ACCEPT)
target     prot opt source          destination
DROP      icmp --  10.0.1.2        anywhere           statistic mode random probability 0.39999999991

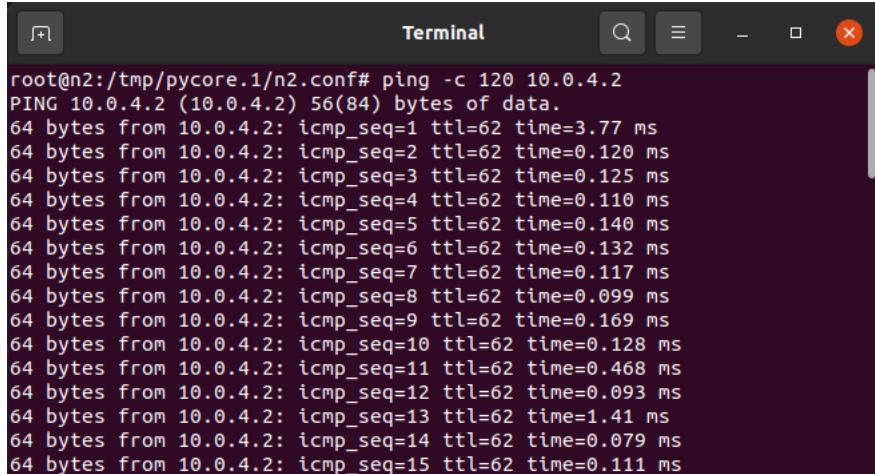
Chain OUTPUT (policy ACCEPT)
target     prot opt source          destination
root@n4:/tmp/pycore.1/n4.conf# root@n4:/tmp/pycore.1/n4.conf# iptables -L -n -v
Chain INPUT (policy ACCEPT 40 packets, 2824 bytes)
  pkts bytes target     prot opt in     out     source          destination
    0     0   DROP      icmp -- *       *      10.0.1.2        0.0.0.0/0           statistic mode random probability 0.39999999991

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
  pkts bytes target     prot opt in     out     source          destination
    0     0   DROP      icmp -- *       *      10.0.1.2        0.0.0.0/0           statistic mode random probability 0.39999999991

Chain OUTPUT (policy ACCEPT 40 packets, 2824 bytes)
  pkts bytes target     prot opt in     out     source          destination
    0     0   DROP      icmp -- *       *      10.0.1.2        0.0.0.0/0           statistic mode random probability 0.39999999991
root@n4:/tmp/pycore.1/n4.conf#
```

b. Use ping command to ping from n2 to n6 for 120 seconds. Show the following:

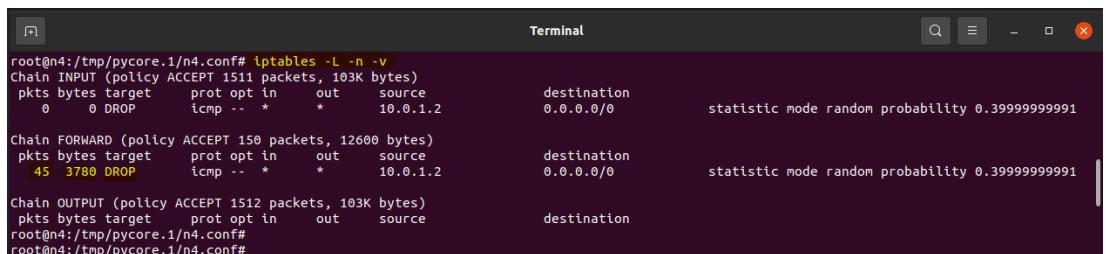
Command to send pings for 120 seconds from n2 to n6(10.0.4.2): **ping -c 120 10.0.4.2**



```
root@n2:/tmp/pycore.1/n2.conf# ping -c 120 10.0.4.2
PING 10.0.4.2 (10.0.4.2) 56(84) bytes of data.
64 bytes from 10.0.4.2: icmp_seq=1 ttl=62 time=3.77 ms
64 bytes from 10.0.4.2: icmp_seq=2 ttl=62 time=0.120 ms
64 bytes from 10.0.4.2: icmp_seq=3 ttl=62 time=0.125 ms
64 bytes from 10.0.4.2: icmp_seq=4 ttl=62 time=0.110 ms
64 bytes from 10.0.4.2: icmp_seq=5 ttl=62 time=0.140 ms
64 bytes from 10.0.4.2: icmp_seq=6 ttl=62 time=0.132 ms
64 bytes from 10.0.4.2: icmp_seq=7 ttl=62 time=0.117 ms
64 bytes from 10.0.4.2: icmp_seq=8 ttl=62 time=0.099 ms
64 bytes from 10.0.4.2: icmp_seq=9 ttl=62 time=0.169 ms
64 bytes from 10.0.4.2: icmp_seq=10 ttl=62 time=0.128 ms
64 bytes from 10.0.4.2: icmp_seq=11 ttl=62 time=0.468 ms
64 bytes from 10.0.4.2: icmp_seq=12 ttl=62 time=0.093 ms
64 bytes from 10.0.4.2: icmp_seq=13 ttl=62 time=1.41 ms
64 bytes from 10.0.4.2: icmp_seq=14 ttl=62 time=0.079 ms
64 bytes from 10.0.4.2: icmp_seq=15 ttl=62 time=0.111 ms
```

i. Show iptable counts to see # of packet drops corresponding to this rule.

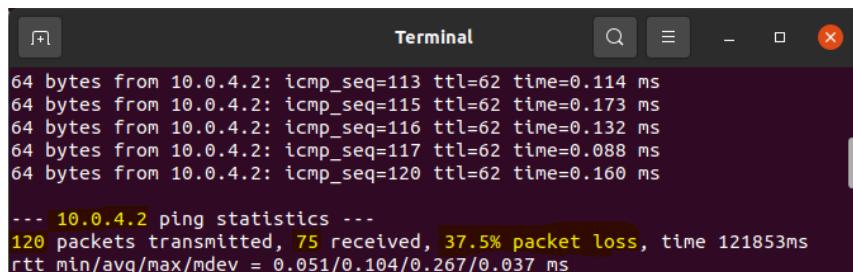
1. You can use the command “**iptables -L -n -v**”



```
root@n4:/tmp/pycore.1/n4.conf# iptables -L -n -V
Chain INPUT (policy ACCEPT 1511 packets, 103K bytes)
  pkts bytes target     prot opt in     out    source               destination
    0     0  DROP        icmp -- *      *      10.0.1.2          0.0.0.0/0      statistic mode random probability 0.399999999991
Chain FORWARD (policy ACCEPT 150 packets, 12600 bytes)
  pkts bytes target     prot opt in     out    source               destination
   45   3780  DROP        icmp -- *      *      10.0.1.2          0.0.0.0/0      statistic mode random probability 0.399999999991
Chain OUTPUT (policy ACCEPT 1512 packets, 103K bytes)
  pkts bytes target     prot opt in     out    source               destination
root@n4:/tmp/pycore.1/n4.conf#
root@n4:/tmp/pycore.1/n4.conf#
```

In the above screenshot, the output of **iptables -L -n -V** command showed **45** packets loss which matches with the forward drop rule as shown in the below ping statistics section.

ii. Show ping statistics reflecting the loss (lost packets should roughly match matched packets count in i.1)



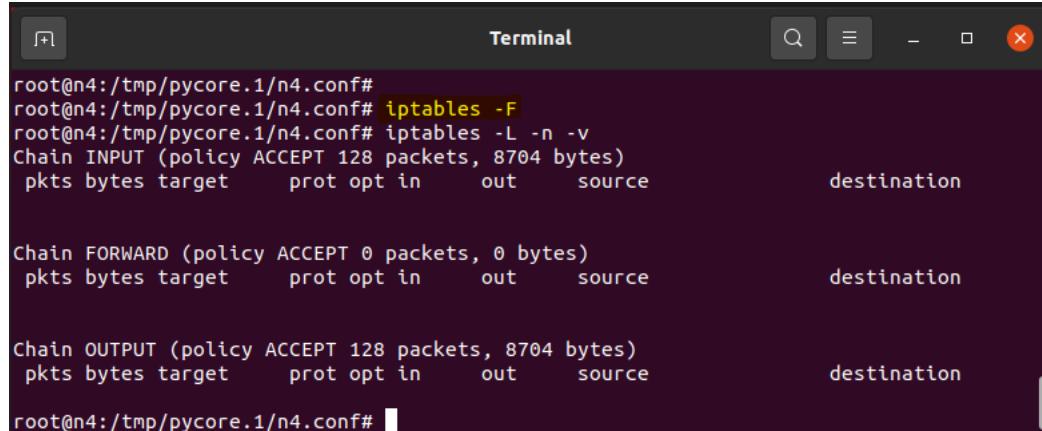
```
64 bytes from 10.0.4.2: icmp_seq=113 ttl=62 time=0.114 ms
64 bytes from 10.0.4.2: icmp_seq=115 ttl=62 time=0.173 ms
64 bytes from 10.0.4.2: icmp_seq=116 ttl=62 time=0.132 ms
64 bytes from 10.0.4.2: icmp_seq=117 ttl=62 time=0.088 ms
64 bytes from 10.0.4.2: icmp_seq=120 ttl=62 time=0.160 ms

--- 10.0.4.2 ping statistics ---
120 packets transmitted, 75 received, 37.5% packet loss, time 121853ms
rtt min/avg/max/mdev = 0.051/0.104/0.267/0.037 ms
```

The ping statistics show that only 75 packets were successfully received out of 120 packets transmitted. This results in a loss of **45** packets ($120 - 75 = 45$), leading to a **37.5%** packet loss. This closely aligns with the ~40% packet drop rate implemented by the iptables rule, as verified by the output of the command **iptables -L -n -v** as mentioned above.

c. Reset iptables by flushing the rules as indicated in the “useful commands” section

Command to flush the iptables: **iptables -F**

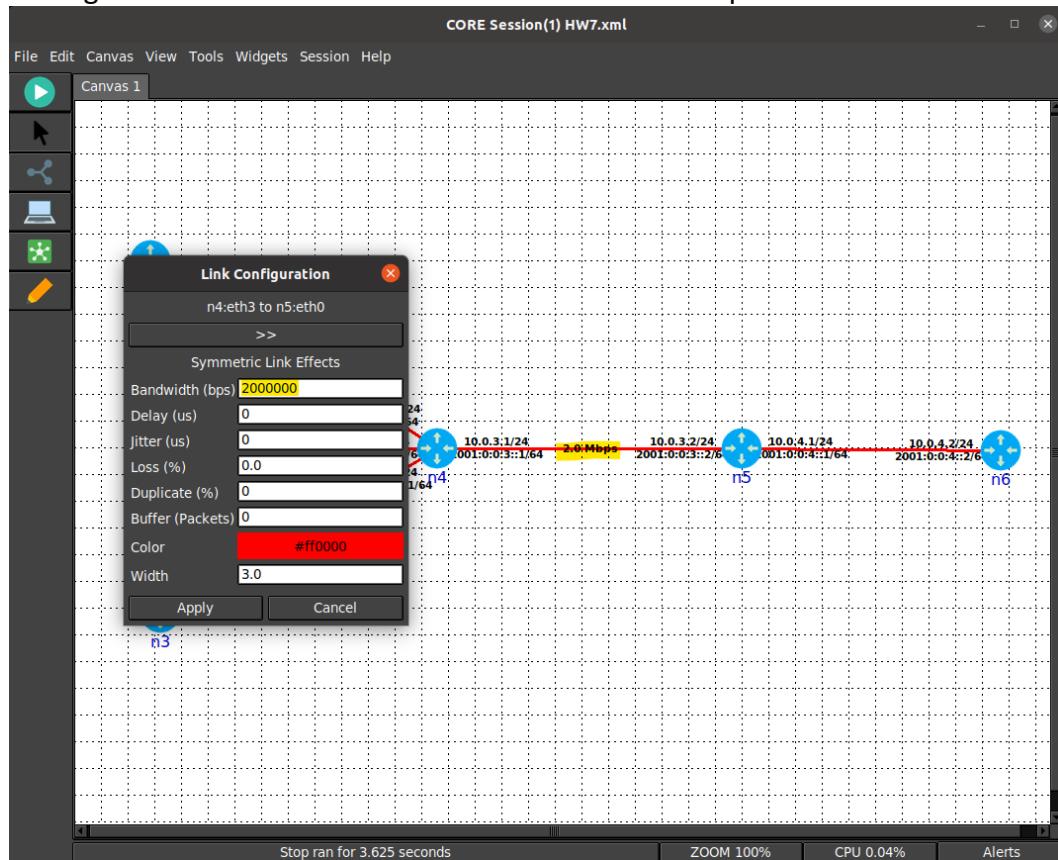


```
root@n4:/tmp/pycore.1/n4.conf# 
root@n4:/tmp/pycore.1/n4.conf# iptables -F
root@n4:/tmp/pycore.1/n4.conf# iptables -L -n -v
Chain INPUT (policy ACCEPT 128 packets, 8704 bytes)
pkts bytes target     prot opt in     out     source               destination
Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out     source               destination
Chain OUTPUT (policy ACCEPT 128 packets, 8704 bytes)
pkts bytes target     prot opt in     out     source               destination
root@n4:/tmp/pycore.1/n4.conf# 
```

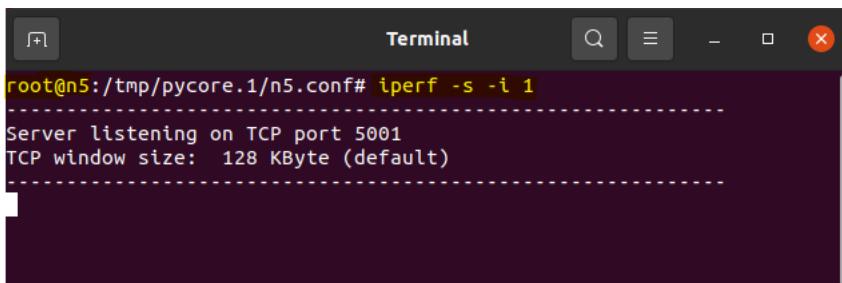
2. (2.5pts) Imagine n5 is a server that you are running and that n1, n2, and n3 are clients. Node n1 is an attacker who is using hping3 to run a UDP flooding attack against you. Nodes n2 and n3 are good nodes

- Set the link between n4 and n5 to be 2Mbps.

Setting the link between node n4 to node n5 to be 2Mbps as shown below:



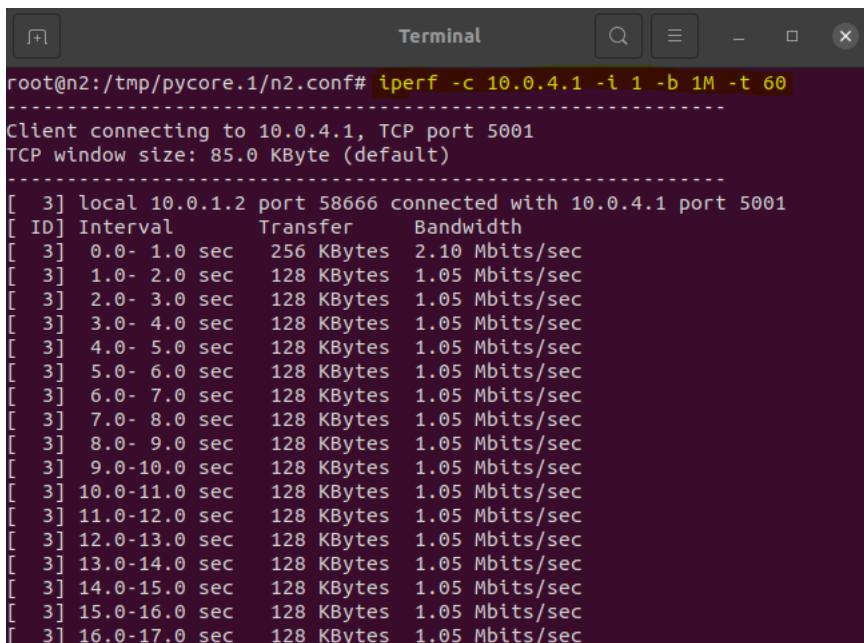
b. Run a TCP iperf server on node n5: `iperf -s -i 1`



```
root@n5:/tmp/pycore.1/n5.conf# iperf -s -i 1
-----
Server listening on TCP port 5001
TCP window size: 128 KByte (default)
-----
```

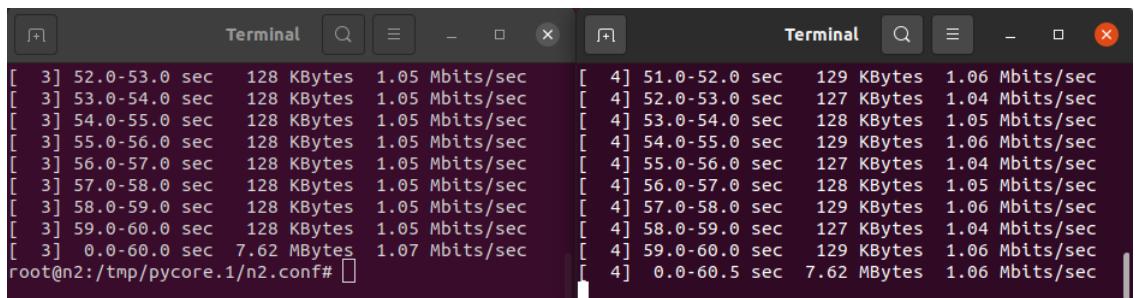
c. Try to run an iperf TCP client from node n2 to node n5. Set client to send 1mbps of data (-b option) for 1 minute (-t option).

i) `iperf -c 10.0.4.1 -i 1 -b 1M -t 60`



```
root@n2:/tmp/pycore.1/n2.conf# iperf -c 10.0.4.1 -i 1 -b 1M -t 60
-----
Client connecting to 10.0.4.1, TCP port 5001
TCP window size: 85.0 KByte (default)
-----
[ 3] local 10.0.1.2 port 58666 connected with 10.0.4.1 port 5001
[ ID] Interval Transfer Bandwidth
[ 3] 0.0- 1.0 sec 256 KBytes 2.10 Mbits/sec
[ 3] 1.0- 2.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 2.0- 3.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 3.0- 4.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 4.0- 5.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 5.0- 6.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 6.0- 7.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 7.0- 8.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 8.0- 9.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 9.0-10.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 10.0-11.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 11.0-12.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 12.0-13.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 13.0-14.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 14.0-15.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 15.0-16.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 16.0-17.0 sec 128 KBytes 1.05 Mbits/sec
```

d. What performance does iperf report?

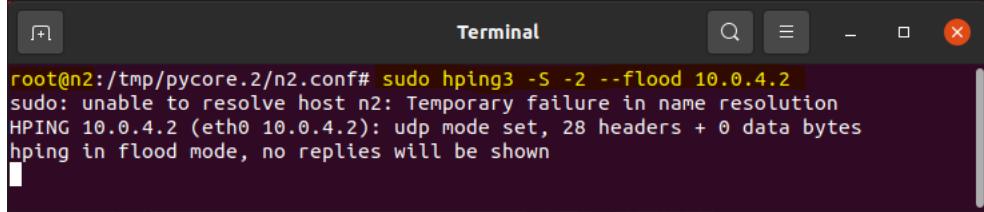


```
[ 3] 52.0-53.0 sec 128 KBytes 1.05 Mbits/sec [ 4] 51.0-52.0 sec 129 KBytes 1.06 Mbits/sec
[ 3] 53.0-54.0 sec 128 KBytes 1.05 Mbits/sec [ 4] 52.0-53.0 sec 127 KBytes 1.04 Mbits/sec
[ 3] 54.0-55.0 sec 128 KBytes 1.05 Mbits/sec [ 4] 53.0-54.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 55.0-56.0 sec 128 KBytes 1.05 Mbits/sec [ 4] 54.0-55.0 sec 129 KBytes 1.06 Mbits/sec
[ 3] 56.0-57.0 sec 128 KBytes 1.05 Mbits/sec [ 4] 55.0-56.0 sec 127 KBytes 1.04 Mbits/sec
[ 3] 57.0-58.0 sec 128 KBytes 1.05 Mbits/sec [ 4] 56.0-57.0 sec 128 KBytes 1.05 Mbits/sec
[ 3] 58.0-59.0 sec 128 KBytes 1.05 Mbits/sec [ 4] 57.0-58.0 sec 129 KBytes 1.06 Mbits/sec
[ 3] 59.0-60.0 sec 128 KBytes 1.05 Mbits/sec [ 4] 58.0-59.0 sec 127 KBytes 1.04 Mbits/sec
[ 3] 0.0-60.0 sec 7.62 MBytes 1.07 Mbits/sec [ 4] 59.0-60.0 sec 129 KBytes 1.06 Mbits/sec
root@n2:/tmp/pycore.1/n2.conf# [ 4] 0.0-60.5 sec 7.62 MBytes 1.06 Mbits/sec
```

The iperf client reported a performance of 128 Kbytes of data transfers on a 1.05 Mbits/sec bandwidth.

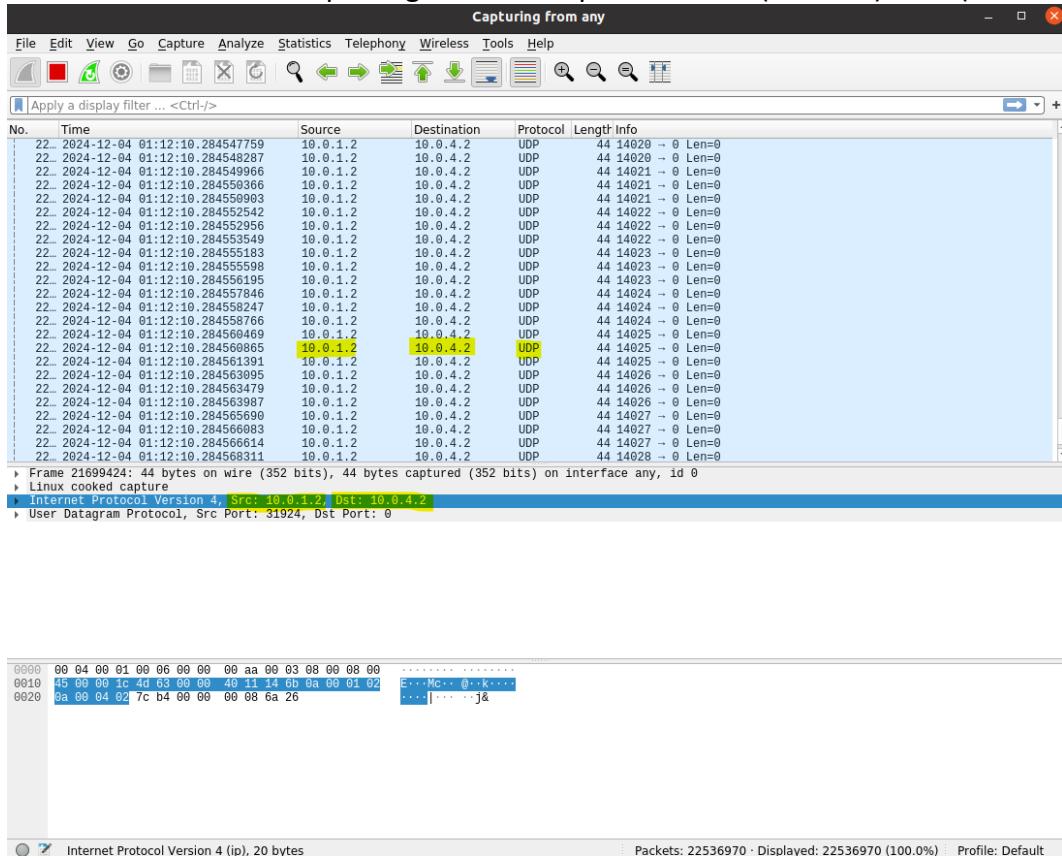
e. Now, use hping3 to run a SYN flood attack from node n2 to node n6

Command to run a SYN flood attack from n2 to n6: `sudo hping3 -S -2 --flood 10.0.4.2`



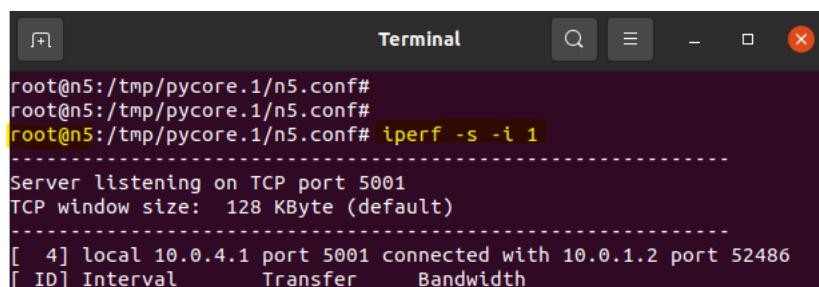
```
root@n2:/tmp/pycore.2/n2.conf# sudo hping3 -S -2 --flood 10.0.4.2
sudo: unable to resolve host n2: Temporary failure in name resolution
HPING 10.0.4.2 (eth0 10.0.4.2): udp mode set, 28 headers + 0 data bytes
hpPing in flood mode, no replies will be shown
```

Wireshark screenshot capturing the UDP request from n2 (10.0.1.2) to n6(10.0.4.2):



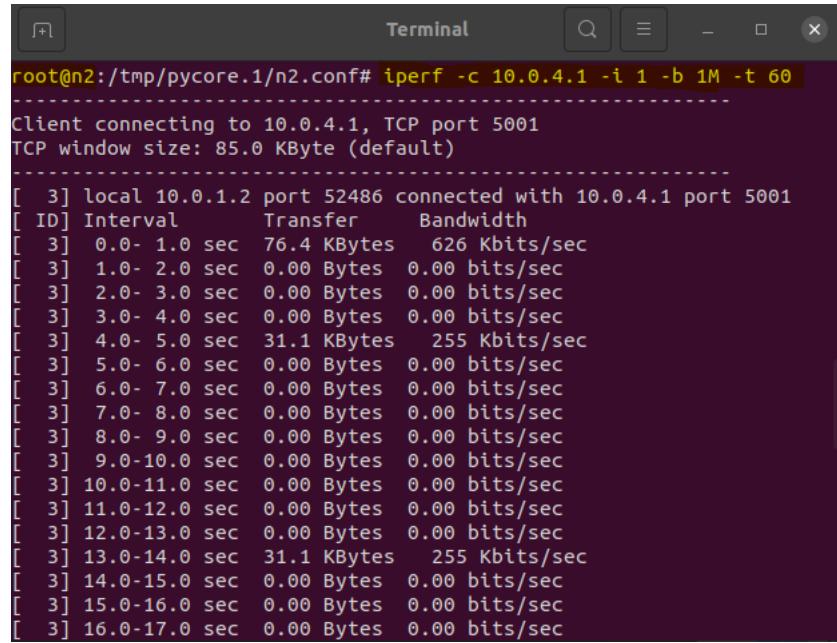
f. Repeat c, d

- Starting the iperf server on n5 using the command: `iperf -s -i 1`



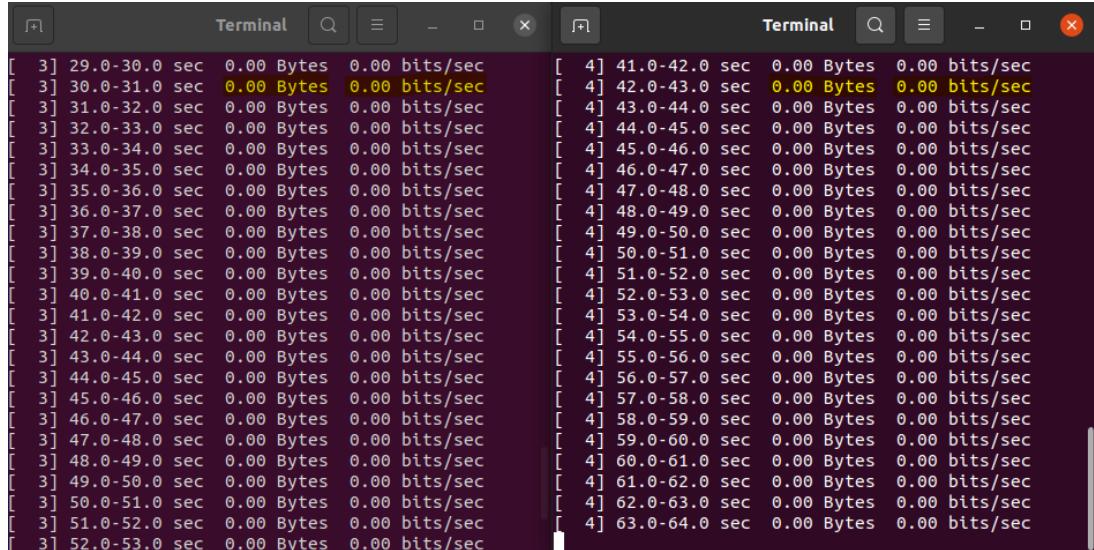
```
root@n5:/tmp/pycore.1/n5.conf#
root@n5:/tmp/pycore.1/n5.conf#
root@n5:/tmp/pycore.1/n5.conf# iperf -s -i 1
-----
[ 4] local 10.0.4.1 port 5001 connected with 10.0.1.2 port 52486
[ ID] Interval      Transfer     Bandwidth
```

- Starting iperf client on n2 while running a SYN flood attack using the command:
iperf -c 10.0.4.1 -i 1 -b 1M -t 60



```
root@n2:/tmp/pycore.1/n2.conf# iperf -c 10.0.4.1 -i 1 -b 1M -t 60
-----
Client connecting to 10.0.4.1, TCP port 5001
TCP window size: 85.0 KByte (default)
[ 3] local 10.0.1.2 port 52486 connected with 10.0.4.1 port 5001
[ ID] Interval Transfer Bandwidth
[ 3] 0.0- 1.0 sec 76.4 KBytes 626 Kbits/sec
[ 3] 1.0- 2.0 sec 0.00 Bytes 0.00 bits/sec
[ 3] 2.0- 3.0 sec 0.00 Bytes 0.00 bits/sec
[ 3] 3.0- 4.0 sec 0.00 Bytes 0.00 bits/sec
[ 3] 4.0- 5.0 sec 31.1 KBytes 255 Kbits/sec
[ 3] 5.0- 6.0 sec 0.00 Bytes 0.00 bits/sec
[ 3] 6.0- 7.0 sec 0.00 Bytes 0.00 bits/sec
[ 3] 7.0- 8.0 sec 0.00 Bytes 0.00 bits/sec
[ 3] 8.0- 9.0 sec 0.00 Bytes 0.00 bits/sec
[ 3] 9.0-10.0 sec 0.00 Bytes 0.00 bits/sec
[ 3] 10.0-11.0 sec 0.00 Bytes 0.00 bits/sec
[ 3] 11.0-12.0 sec 0.00 Bytes 0.00 bits/sec
[ 3] 12.0-13.0 sec 0.00 Bytes 0.00 bits/sec
[ 3] 13.0-14.0 sec 31.1 KBytes 255 Kbits/sec
[ 3] 14.0-15.0 sec 0.00 Bytes 0.00 bits/sec
[ 3] 15.0-16.0 sec 0.00 Bytes 0.00 bits/sec
[ 3] 16.0-17.0 sec 0.00 Bytes 0.00 bits/sec
```

- In the below screenshot the [3] is representing iperf client on n2 and [4] is representing iperf server on n5. We can observe that as the SYN flood attack is in progress, the iperf client could not transfer information to the server.



Client (Left)	Server (Right)
[3] 29.0-30.0 sec 0.00 Bytes 0.00 bits/sec	[4] 41.0-42.0 sec 0.00 Bytes 0.00 bits/sec
[3] 30.0-31.0 sec 0.00 Bytes 0.00 bits/sec	[4] 42.0-43.0 sec 0.00 Bytes 0.00 bits/sec
[3] 31.0-32.0 sec 0.00 Bytes 0.00 bits/sec	[4] 43.0-44.0 sec 0.00 Bytes 0.00 bits/sec
[3] 32.0-33.0 sec 0.00 Bytes 0.00 bits/sec	[4] 44.0-45.0 sec 0.00 Bytes 0.00 bits/sec
[3] 33.0-34.0 sec 0.00 Bytes 0.00 bits/sec	[4] 45.0-46.0 sec 0.00 Bytes 0.00 bits/sec
[3] 34.0-35.0 sec 0.00 Bytes 0.00 bits/sec	[4] 46.0-47.0 sec 0.00 Bytes 0.00 bits/sec
[3] 35.0-36.0 sec 0.00 Bytes 0.00 bits/sec	[4] 47.0-48.0 sec 0.00 Bytes 0.00 bits/sec
[3] 36.0-37.0 sec 0.00 Bytes 0.00 bits/sec	[4] 48.0-49.0 sec 0.00 Bytes 0.00 bits/sec
[3] 37.0-38.0 sec 0.00 Bytes 0.00 bits/sec	[4] 49.0-50.0 sec 0.00 Bytes 0.00 bits/sec
[3] 38.0-39.0 sec 0.00 Bytes 0.00 bits/sec	[4] 50.0-51.0 sec 0.00 Bytes 0.00 bits/sec
[3] 39.0-40.0 sec 0.00 Bytes 0.00 bits/sec	[4] 51.0-52.0 sec 0.00 Bytes 0.00 bits/sec
[3] 40.0-41.0 sec 0.00 Bytes 0.00 bits/sec	[4] 52.0-53.0 sec 0.00 Bytes 0.00 bits/sec
[3] 41.0-42.0 sec 0.00 Bytes 0.00 bits/sec	[4] 53.0-54.0 sec 0.00 Bytes 0.00 bits/sec
[3] 42.0-43.0 sec 0.00 Bytes 0.00 bits/sec	[4] 54.0-55.0 sec 0.00 Bytes 0.00 bits/sec
[3] 43.0-44.0 sec 0.00 Bytes 0.00 bits/sec	[4] 55.0-56.0 sec 0.00 Bytes 0.00 bits/sec
[3] 44.0-45.0 sec 0.00 Bytes 0.00 bits/sec	[4] 56.0-57.0 sec 0.00 Bytes 0.00 bits/sec
[3] 45.0-46.0 sec 0.00 Bytes 0.00 bits/sec	[4] 57.0-58.0 sec 0.00 Bytes 0.00 bits/sec
[3] 46.0-47.0 sec 0.00 Bytes 0.00 bits/sec	[4] 58.0-59.0 sec 0.00 Bytes 0.00 bits/sec
[3] 47.0-48.0 sec 0.00 Bytes 0.00 bits/sec	[4] 59.0-60.0 sec 0.00 Bytes 0.00 bits/sec
[3] 48.0-49.0 sec 0.00 Bytes 0.00 bits/sec	[4] 60.0-61.0 sec 0.00 Bytes 0.00 bits/sec
[3] 49.0-50.0 sec 0.00 Bytes 0.00 bits/sec	[4] 61.0-62.0 sec 0.00 Bytes 0.00 bits/sec
[3] 50.0-51.0 sec 0.00 Bytes 0.00 bits/sec	[4] 62.0-63.0 sec 0.00 Bytes 0.00 bits/sec
[3] 51.0-52.0 sec 0.00 Bytes 0.00 bits/sec	[4] 63.0-64.0 sec 0.00 Bytes 0.00 bits/sec
[3] 52.0-53.0 sec 0.00 Bytes 0.00 bits/sec	

The iperf client reported a performance of 0 Bytes of data transfers on a 0.00 bits/sec bandwidth as shown above.

- g. Now add a firewall rule at node n4 to drop all packets coming in from n1. What rule did you add?

Command to add a firewall at node n4 to drop all input and forward packets coming in from n2 and view the output of iptables:

- **iptables -A INPUT -s 10.0.0.1 -j DROP**
- **iptables -A FORWARD -s 10.0.0.1 -j DROP**
- **iptables -L -n -v**

```

root@n4:/tmp/pycore.1/n4.conf# iptables -A INPUT -s 10.0.0.1 -j DROP
root@n4:/tmp/pycore.1/n4.conf# iptables -A FORWARD -s 10.0.0.1 -j DROP
root@n4:/tmp/pycore.1/n4.conf# iptables -L -n -v
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target      prot opt in     out      source          destination
      0     0  DROP        all    --  *       *      10.0.0.1        0.0.0.0/0

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target      prot opt in     out      source          destination
      0     0  DROP        all    --  *       *      10.0.0.1        0.0.0.0/0

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target      prot opt in     out      source          destination
root@n4:/tmp/pycore.1/n4.conf#

```

The ping statistics from n1 to n4 reveals a 100% loss of packets since all packets from n1 were dropped at the firewall at n4 as shown below:

```

root@n1:/tmp/pycore.1/n1.conf# ping 10.0.4.1
PING 10.0.4.1 (10.0.4.1) 56(84) bytes of data.

^C
--- 10.0.4.1 ping statistics ---
184 packets transmitted, 0 received, 100% packet loss, time 187429ms
root@n1:/tmp/pycore.1/n1.conf#
root@n1:/tmp/pycore.1/n1.conf#

```

3. (2.5pts) Perform a NAT for all traffic forwarded by n4 to n5

- a. Show the iptables command you used?

Command used: **iptables -t nat -A POSTROUTING -o eth3 -j MASQUERADE**

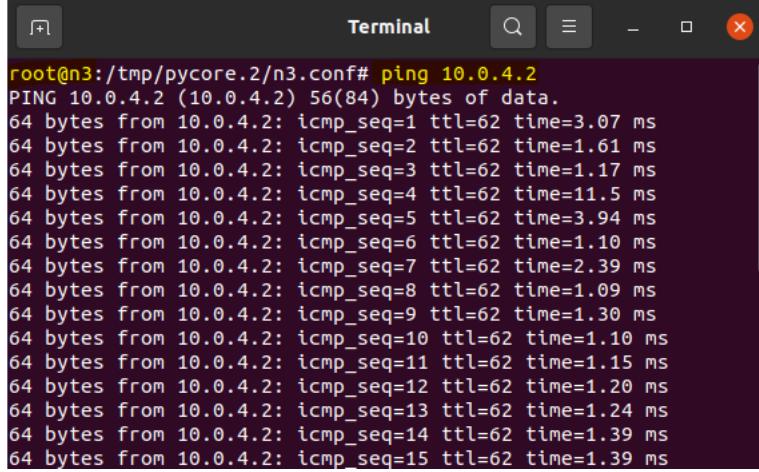
```

root@n4:/tmp/pycore.1/n4.conf# iptables -t nat -A POSTROUTING -o eth3 -j MASQUERADE
root@n4:/tmp/pycore.1/n4.conf# 

```

b. Use ping command to ping from n3 to n6. Show the following:

Pinging from n3 to n6 using the command: **ping 10.0.4.2**

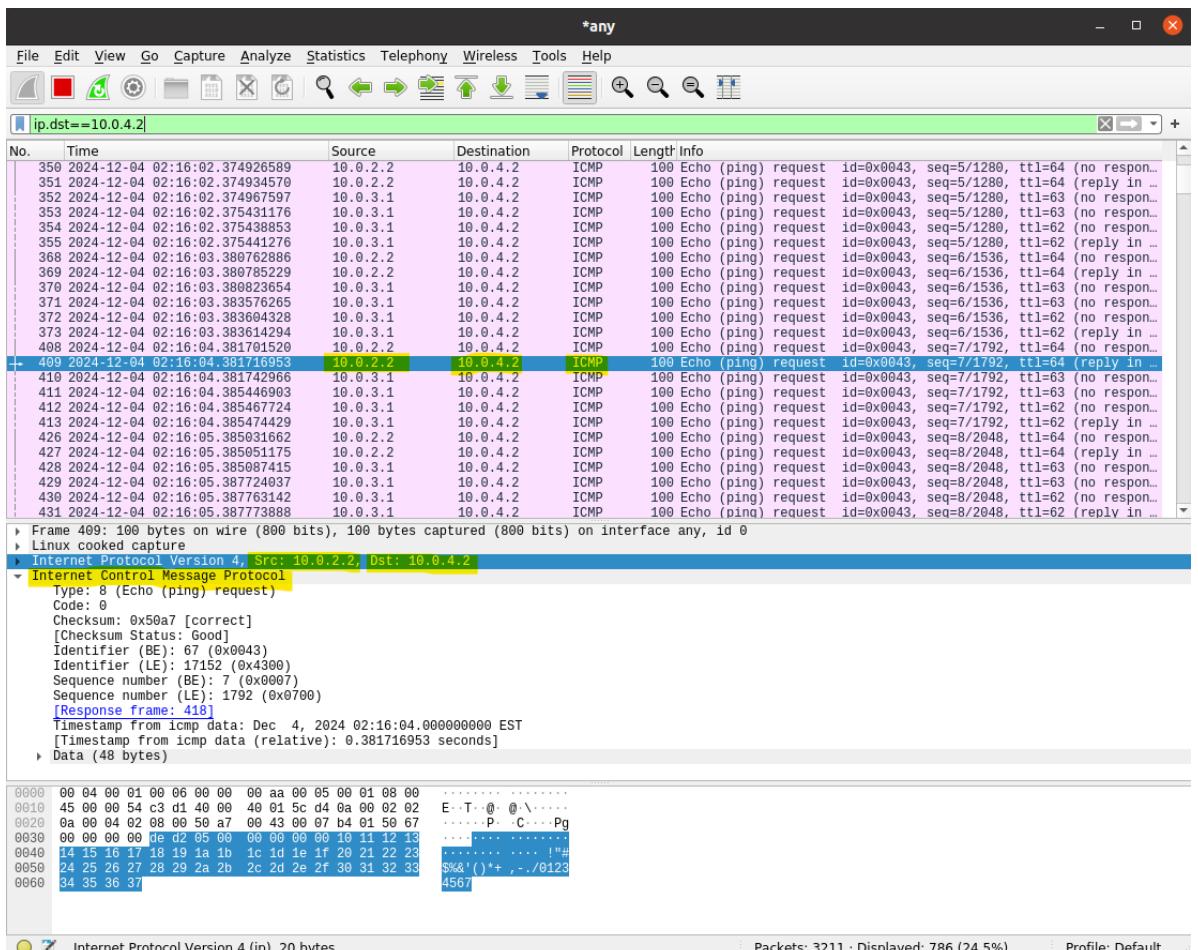


```
root@n3:/tmp/pycore.2/n3.conf# ping 10.0.4.2
PING 10.0.4.2 (10.0.4.2) 56(84) bytes of data.
64 bytes from 10.0.4.2: icmp_seq=1 ttl=62 time=3.07 ms
64 bytes from 10.0.4.2: icmp_seq=2 ttl=62 time=1.61 ms
64 bytes from 10.0.4.2: icmp_seq=3 ttl=62 time=1.17 ms
64 bytes from 10.0.4.2: icmp_seq=4 ttl=62 time=11.5 ms
64 bytes from 10.0.4.2: icmp_seq=5 ttl=62 time=3.94 ms
64 bytes from 10.0.4.2: icmp_seq=6 ttl=62 time=1.10 ms
64 bytes from 10.0.4.2: icmp_seq=7 ttl=62 time=2.39 ms
64 bytes from 10.0.4.2: icmp_seq=8 ttl=62 time=1.09 ms
64 bytes from 10.0.4.2: icmp_seq=9 ttl=62 time=1.30 ms
64 bytes from 10.0.4.2: icmp_seq=10 ttl=62 time=1.10 ms
64 bytes from 10.0.4.2: icmp_seq=11 ttl=62 time=1.15 ms
64 bytes from 10.0.4.2: icmp_seq=12 ttl=62 time=1.20 ms
64 bytes from 10.0.4.2: icmp_seq=13 ttl=62 time=1.24 ms
64 bytes from 10.0.4.2: icmp_seq=14 ttl=62 time=1.39 ms
64 bytes from 10.0.4.2: icmp_seq=15 ttl=62 time=1.39 ms
```

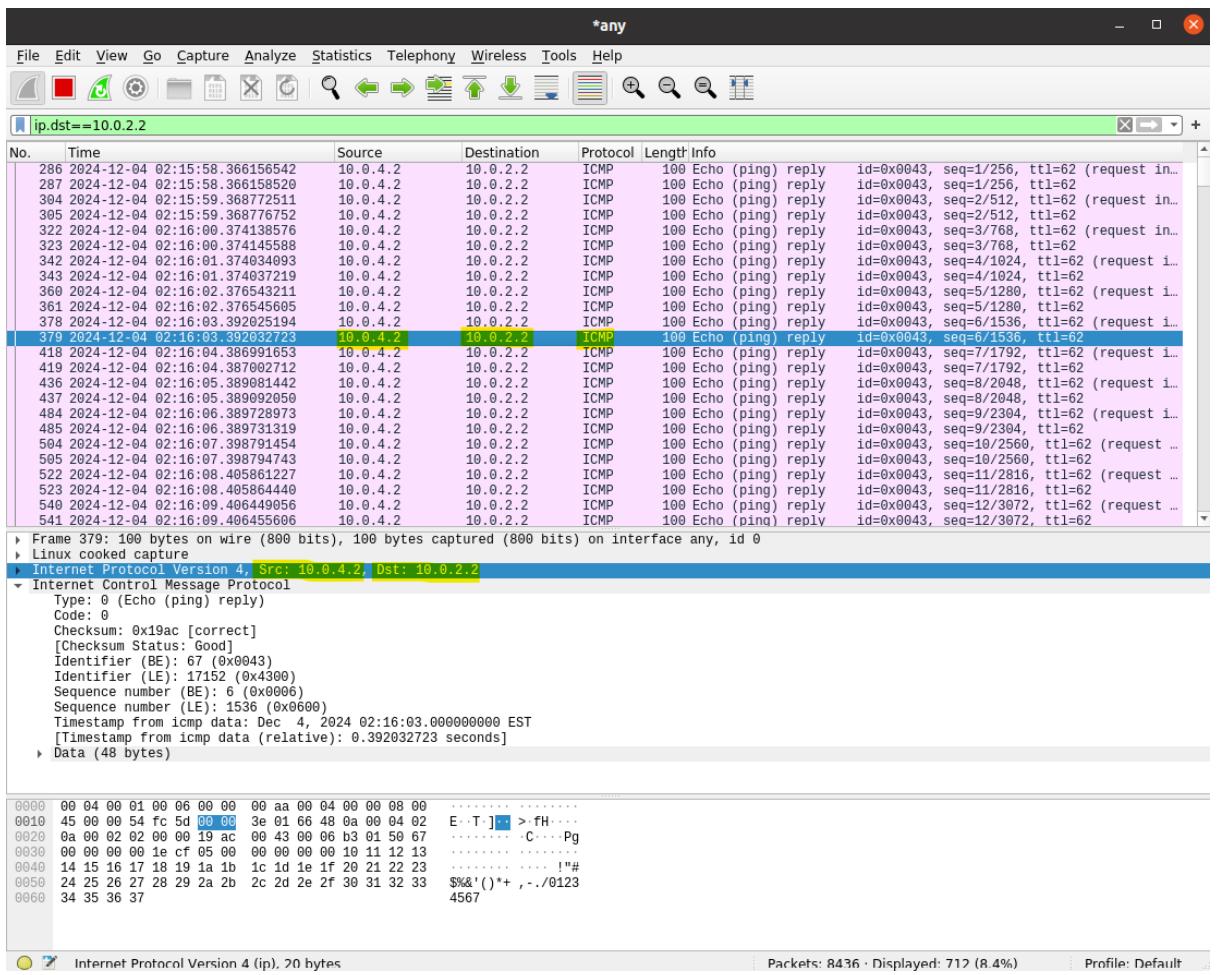
i. Screenshot of ICMP packets arriving at node n4 (i.e., link between n3 and n4). Highlight the source and destination IP addresses

The source IP address is 10.0.2.2 and the destination IP address is 10.0.4.2

The Wireshark packet capture arriving at node 4 (the link between n3 and n4) is shown:



The Wireshark packet capture leaving node 4 (the link between n3 and n4) is shown:

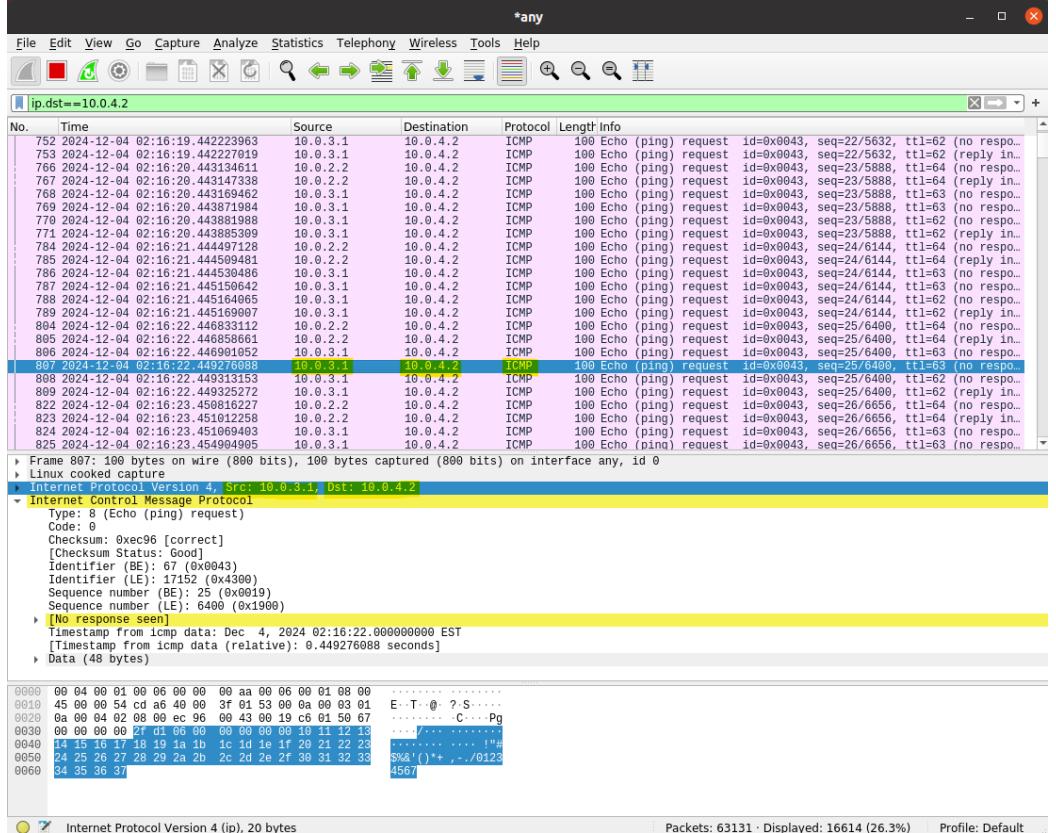


ii. Screenshot of ICMP packets leaving node n4 (i.e., link between n4 and n5). Highlight the source and destination IP addresses

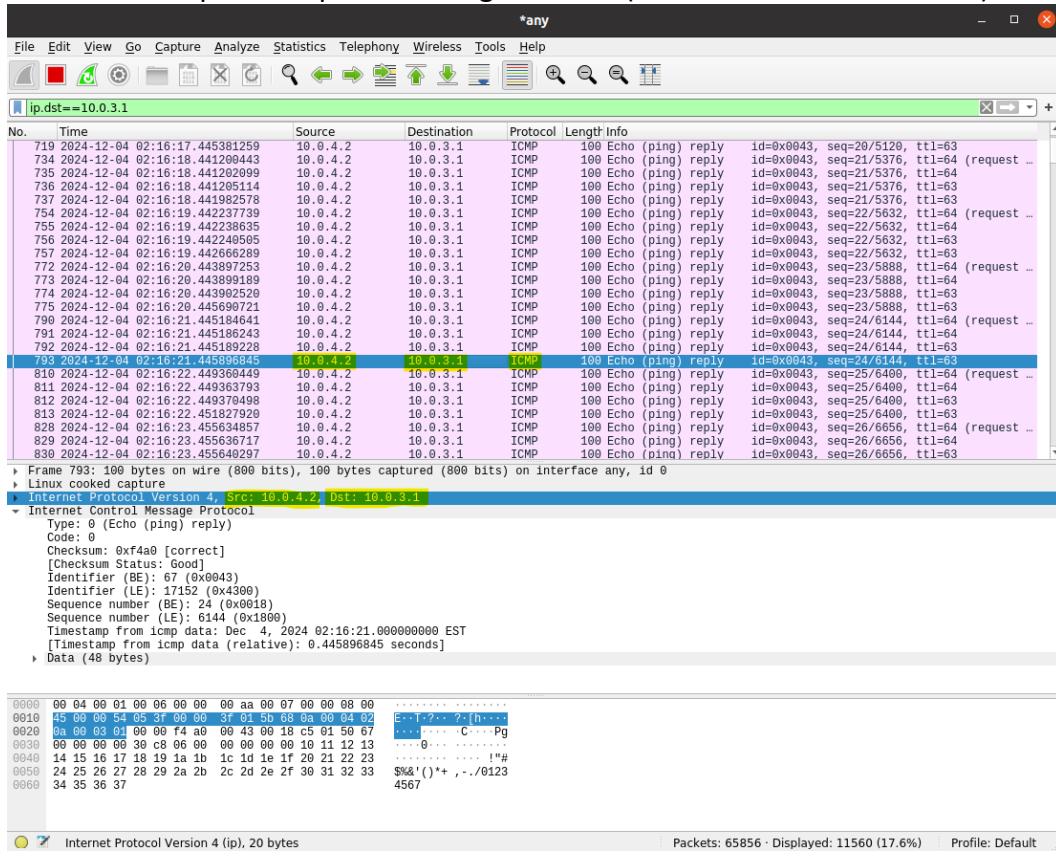
The source IP address is 10.0.3.1 (Due to the masquerading rule)

The destination IP address is 10.0.4.2

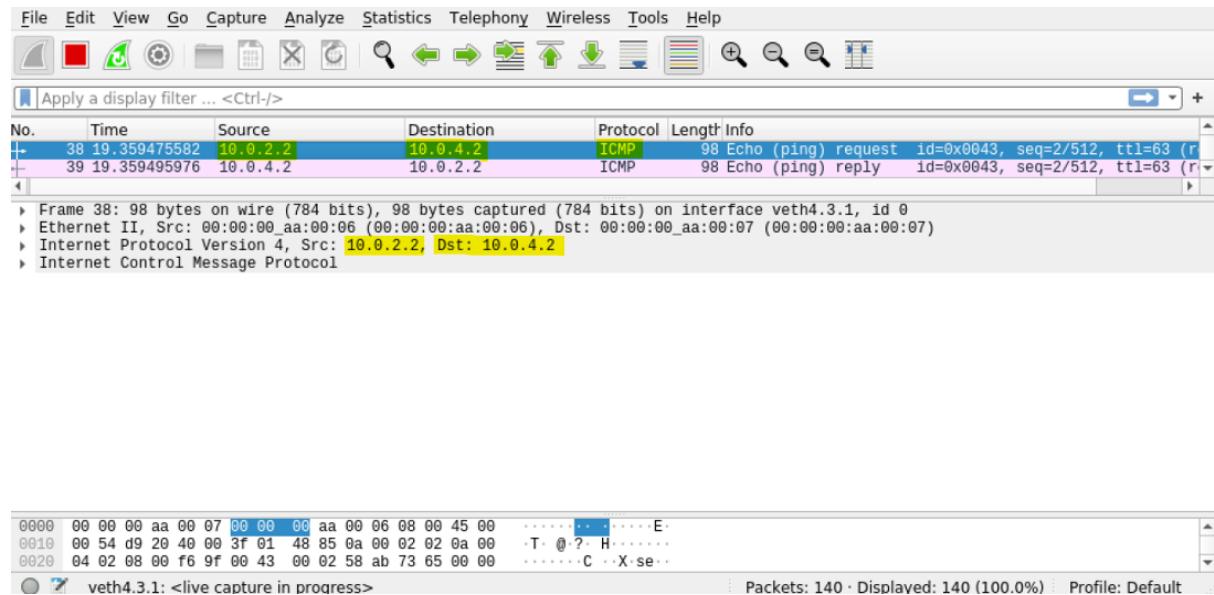
The Wireshark packet capture leaving node 4 (the link between n4 and n5) is shown:



The Wireshark packet capture arriving at node 4 (the link between n4 and n5) is shown:



Without masquerading, the Wireshark packet capture at node n4 (on the link between n4 and n5) shows the source IP address as 10.0.2.2, indicating that the rule is working correctly as shown below:



iii. Show the output of iptables -L -n -v -t NAT

The output of the command **iptables -L -n -v -t nat** is shown below:

```

root@n4:/tmp/pycore.1/n4.conf# iptables -L -n -v -t nat
Chain PREROUTING (policy ACCEPT 5380 packets, 344K bytes)
  pkts bytes target     prot opt in     out    source         destination
                                                     destination

Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
  pkts bytes target     prot opt in     out    source         destination
                                                     destination

Chain OUTPUT (policy ACCEPT 12 packets, 576 bytes)
  pkts bytes target     prot opt in     out    source         destination
                                                     destination

Chain POSTROUTING (policy ACCEPT 12 packets, 576 bytes)
  pkts bytes target     prot opt in     out    source         destination
        1    84 MASQUERADE  all  --  *      eth3    0.0.0.0/0          0.0.0.0/0
        0      0 MASQUERADE  all  --  *      eth3    0.0.0.0/0          0.0.0.0/0

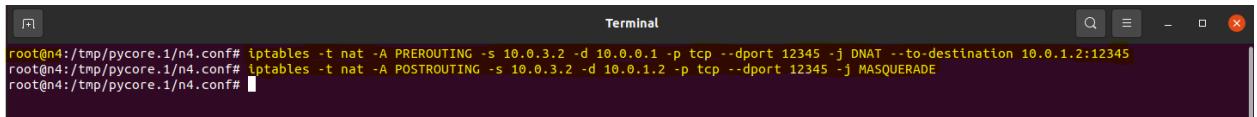
root@n4:/tmp/pycore.1/n4.conf#
  
```

Here, the masquerade rule is being matched whenever messages are initiated from node n4 to n5.

4. (2.5pts) Imagine that nodes n1 and n2 are running the same server: nc -k -l 12345. Nodes n5 and n6 are both running “nc” clients and we want to load balance connections between n1 and n2.

- Assume nc clients on both n5 and n6 initially send traffic to node n1.
- Use a combination of DNAT and NAT (like question 2) to change traffic from n5 so it goes to n2 instead. Ensure bidirectional traffic is possible.

The commands used in combination of DNAT and NAT to change traffic and allow bidirectional traffic to go from n5 to n2 is shown below:



A terminal window titled "Terminal" showing the configuration of iptables. The commands entered are:

```
root@n4:/tmp/pycore.1/n4.conf# iptables -t nat -A PREROUTING -s 10.0.3.2 -d 10.0.0.1 -p tcp --dport 12345 -j DNAT --to-destination 10.0.1.2:12345
root@n4:/tmp/pycore.1/n4.conf# iptables -t nat -A POSTROUTING -s 10.0.3.2 -d 10.0.1.2 -p tcp --dport 12345 -j MASQUERADE
root@n4:/tmp/pycore.1/n4.conf#
```

- Use any combination of screenshots to convince me that your solution works.

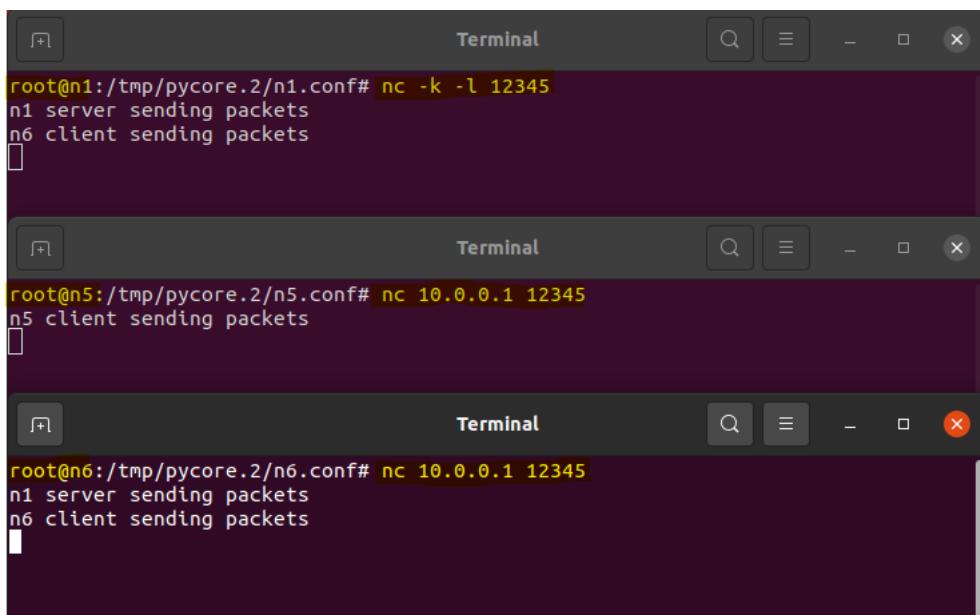
Answers to a, b and c parts:

Before Load Balancing:

Commands used:

- At n1 server: nc -k -l 12345
- At n5 client: nc 10.0.0.1 12345
- At n6 client: nc 10.0.0.1 12345

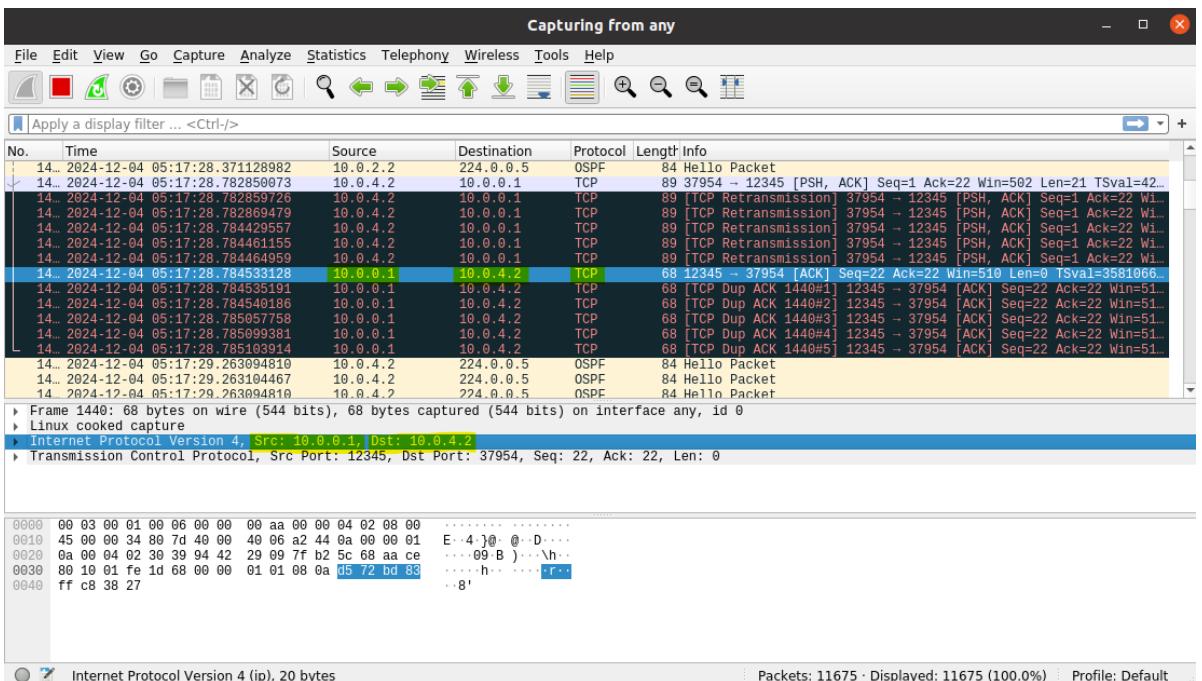
The n1 server can listen for multiple connections but can only maintain a single connection due to its single-threaded design. Consequently, it establishes and maintains a bidirectional connection with the client at node n6, as the command for client at node n6 was executed first, even though it receives packets from the client at node n5. The screenshots below confirm the same:



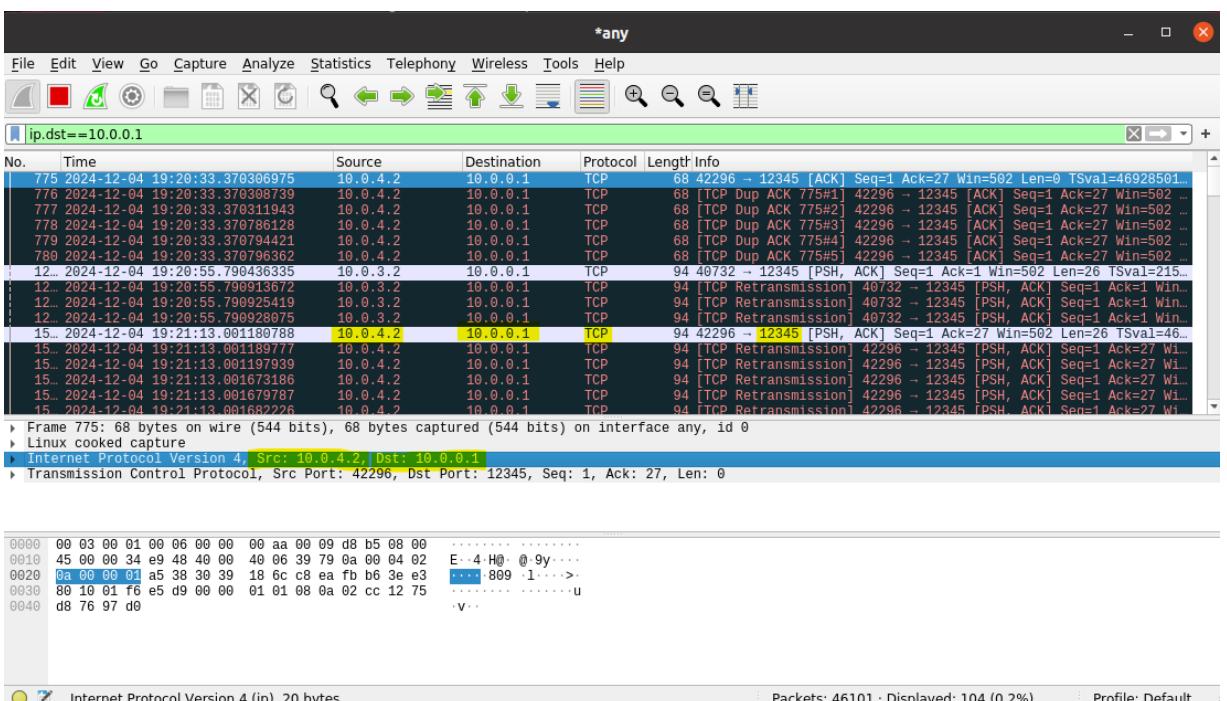
Three terminal windows titled "Terminal" showing bidirectional communication between n1, n5, and n6.

- Top Terminal (n1):** Shows the server listening on port 12345. It prints "n1 server sending packets" and "n6 client sending packets".
- Middle Terminal (n5):** Shows the client connecting to n1 on port 12345. It prints "n5 client sending packets".
- Bottom Terminal (n6):** Shows the client connecting to n1 on port 12345. It prints "n1 server sending packets" and "n6 client sending packets".

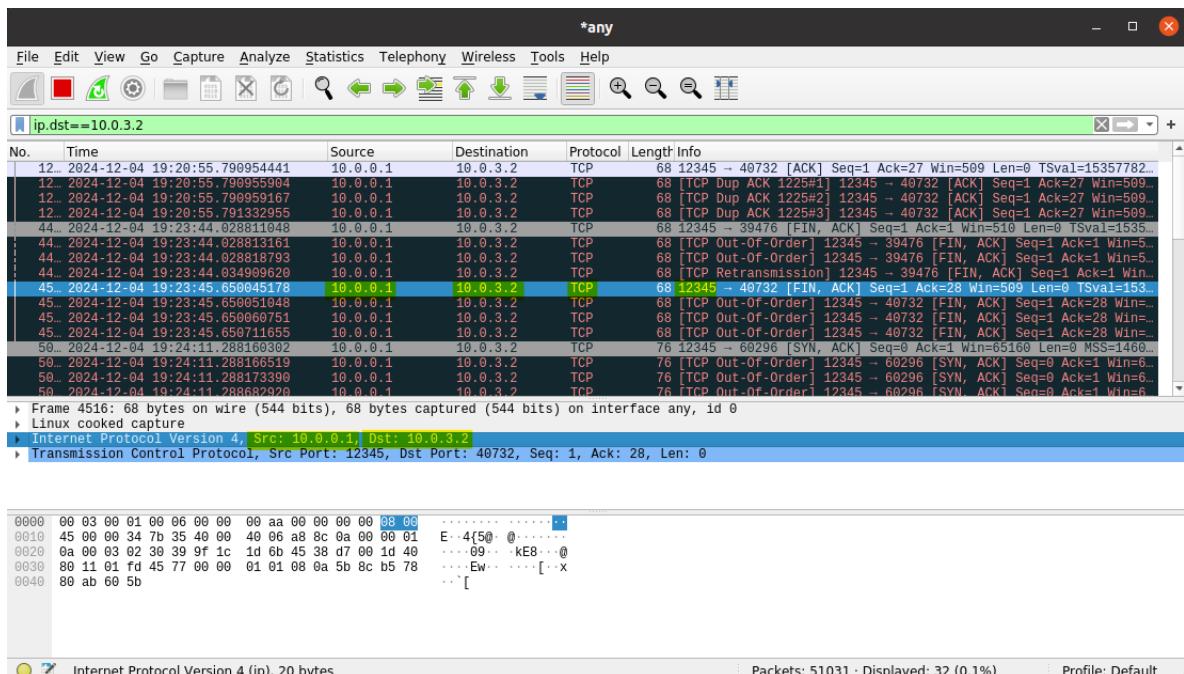
The Wireshark capture showing packet transfer from n1 to n6 is shown below:



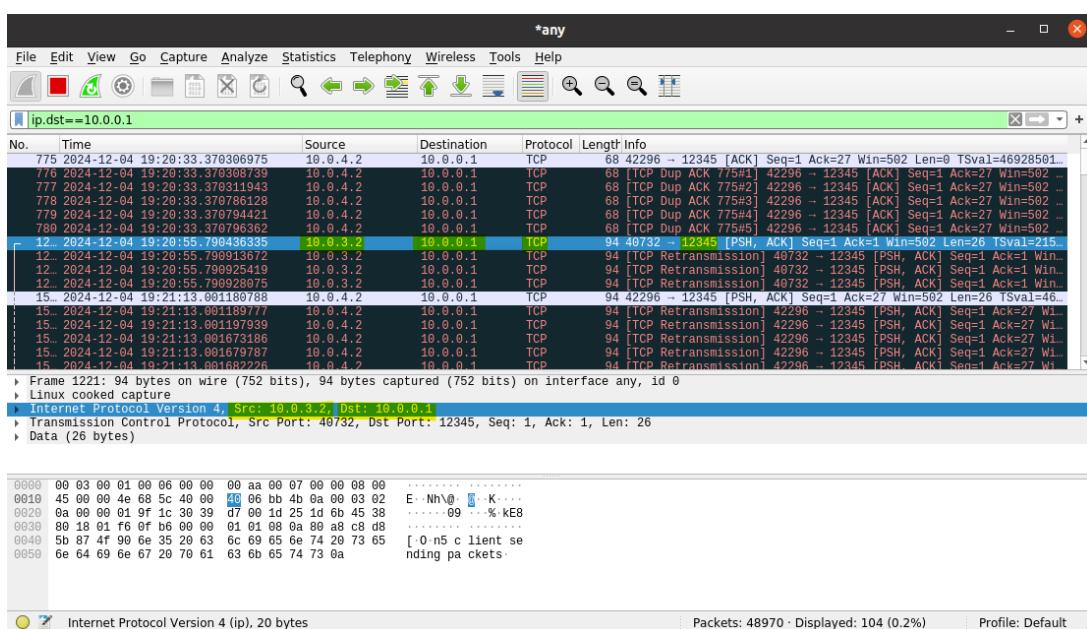
The Wireshark capture showing packet transfer from n6 to n1 proving bidirectional connection is shown below:



The Wireshark capture showing packet transfer from n1 to n5 is shown below:



The Wireshark capture showing packet transfer from n5 to n1 proving bidirectional connection is shown below:



After Load Balancing:

Commands used:

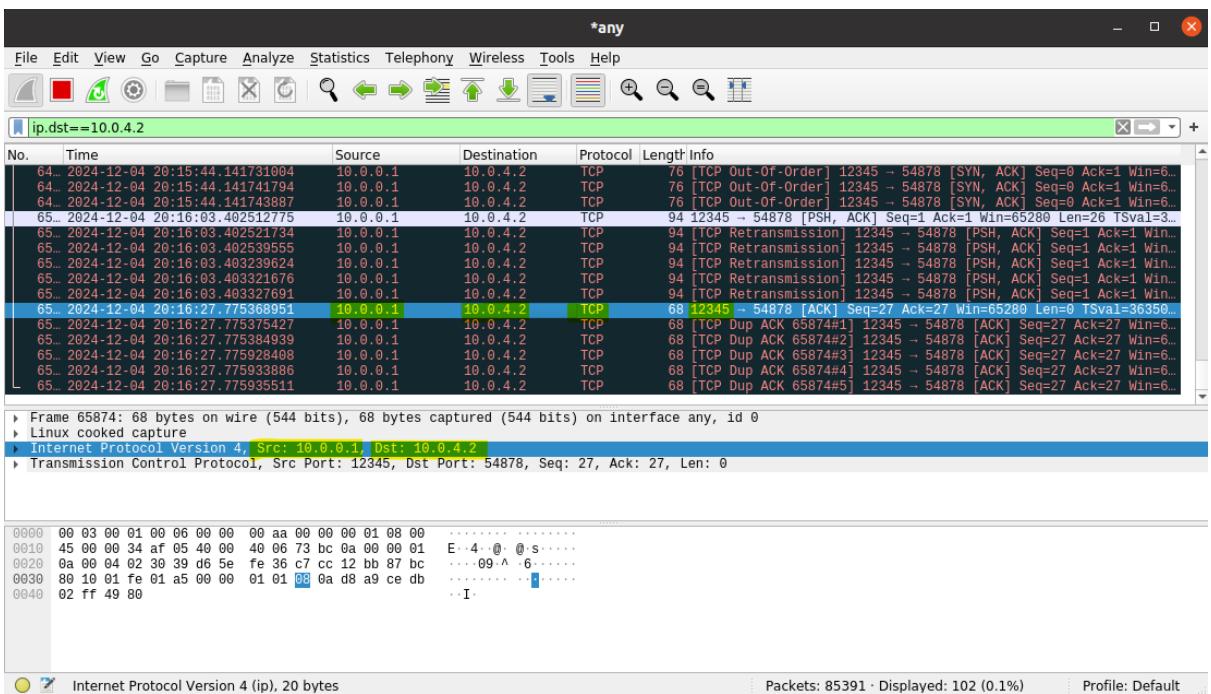
- At n1 server: **nc -k -l 12345**
- At n2 server: **nc -k -l 12345**
- At n5 client: **nc 10.0.0.1 12345**
- At n6 client: **nc 10.0.0.1 12345**

After implementing load balancing, packets from client at node n5 are redirected to server at node n2 instead of n1, while packets from client at node n6 continue to be handled by server at node n1 as shown in the screenshot below:

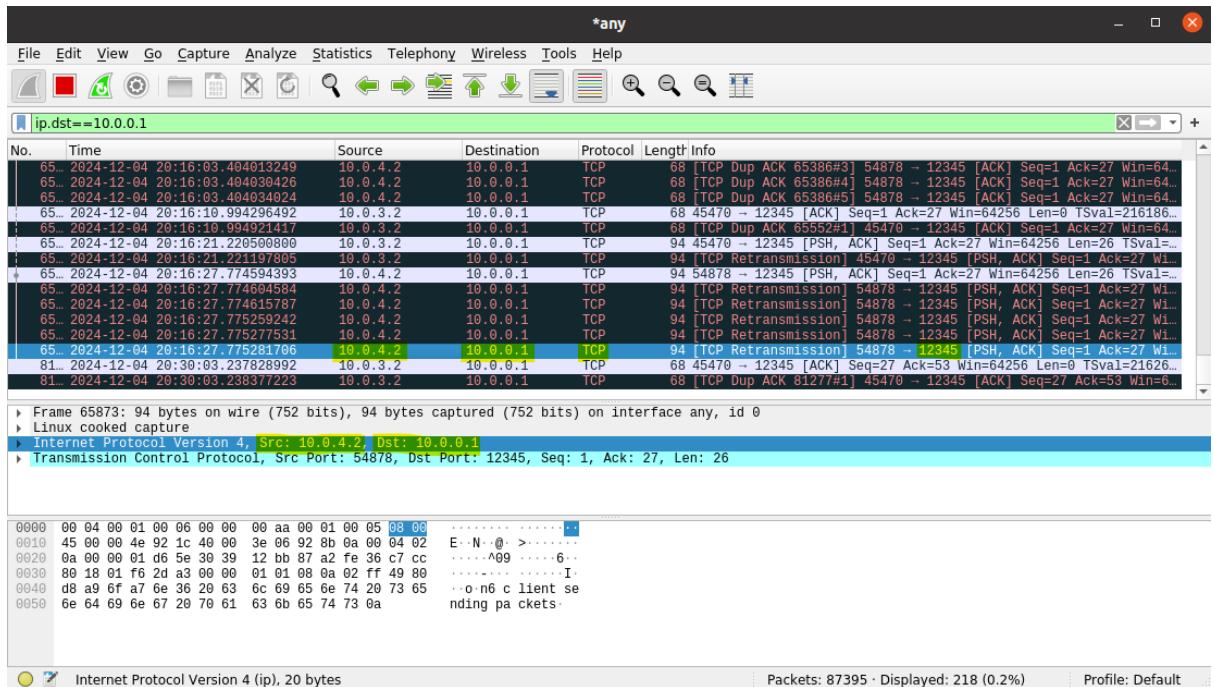
The image shows four terminal windows arranged in a 2x2 grid. Top-left: 'root@n1:/tmp/pycore.2/n1.conf# nc -k -l 12345' with output 'n1 server sending packets' and 'n6 client sending packets'. Top-right: 'root@n2:/tmp/pycore.2/n2.conf# nc -k -l 12345' with output 'n2 server sending packets' and 'n5 client sending packets'. Bottom-left: 'root@n5:/tmp/pycore.2/n5.conf# nc 10.0.0.1 12345' with output 'n2 server sending packets' and 'n5 client sending packets'. Bottom-right: 'root@n6:/tmp/pycore.2/n6.conf# nc 10.0.0.1 12345' with output 'n1 server sending packets' and 'n6 client sending packets'.

- The link between n1 and n4 is indicated via the bidirectional link between n1 to n6:

The Wireshark capture showing n4 as the link between packet transfers from n1 to n6 is shown below:

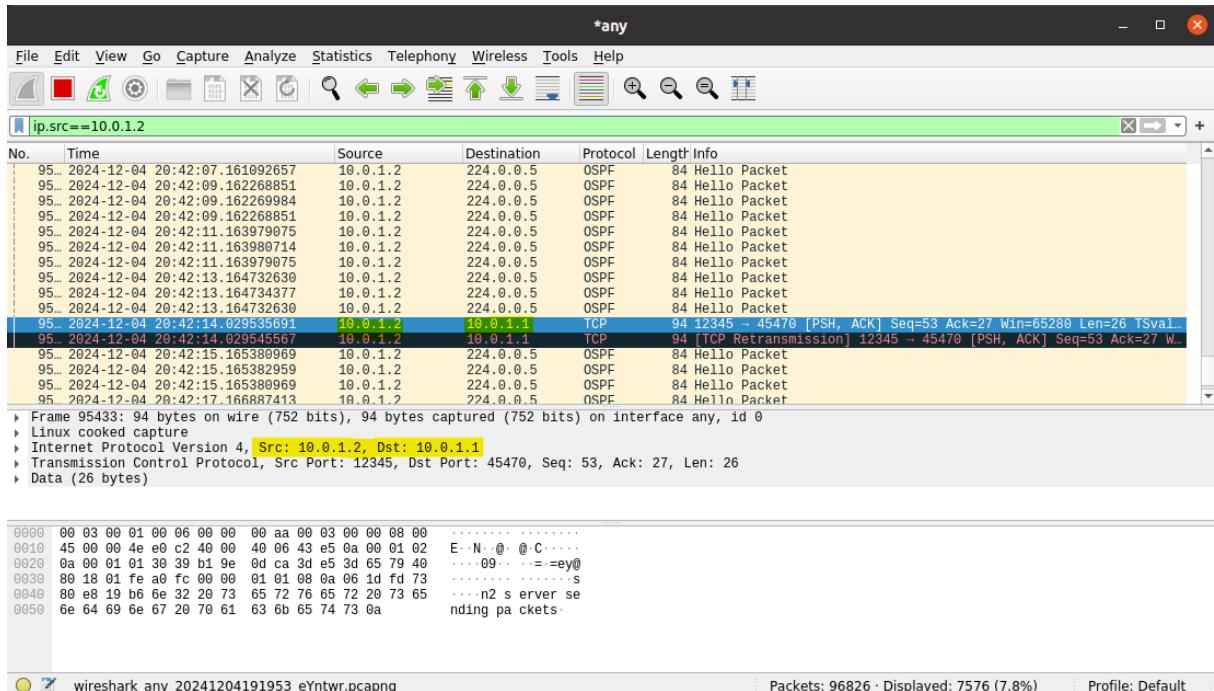


The Wireshark capture showing n4 as the link between packet transfers from n6 to n1 is shown below:

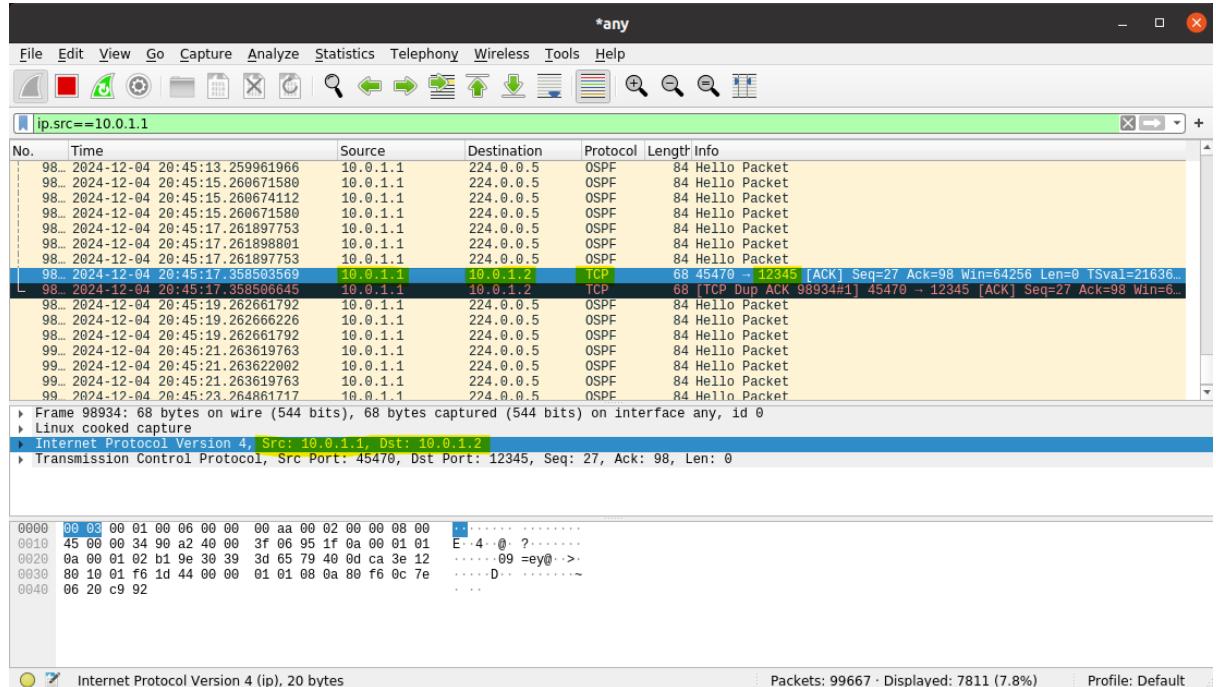


- **The bidirectional link between n2 and n4:**

The Wireshark capturing the link and packet transfers from n2 to n4 is shown below:

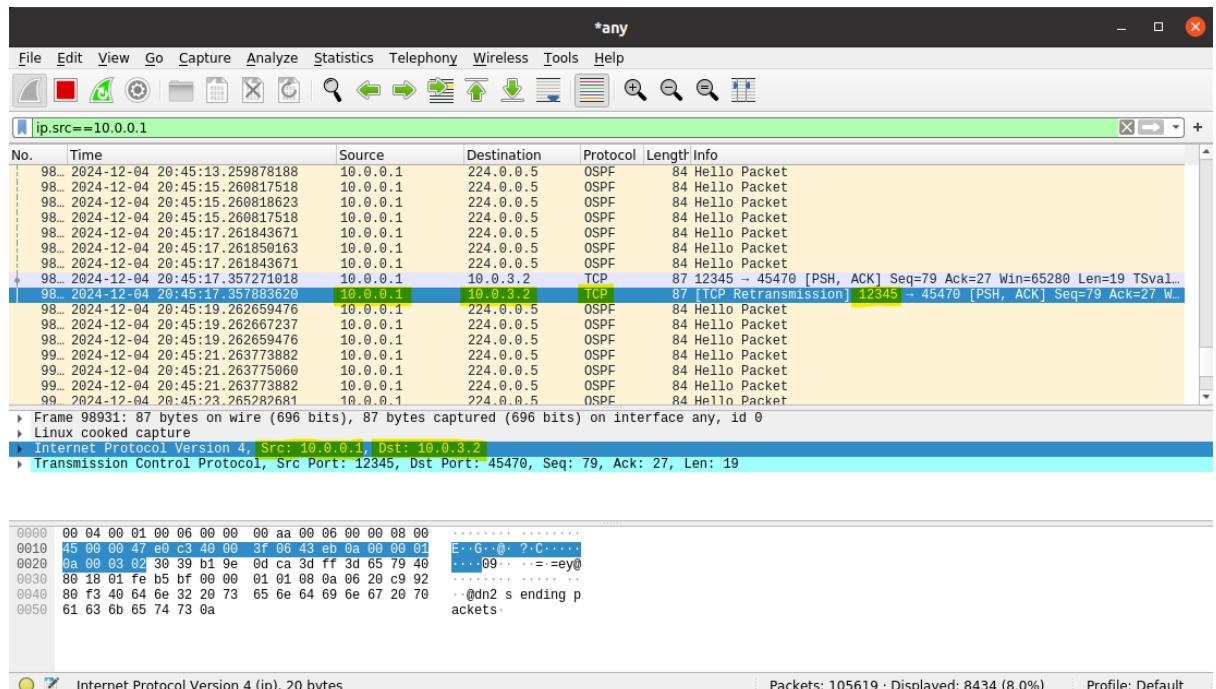


The Wireshark capturing the link and packet transfers from n4 to n2 is shown below:

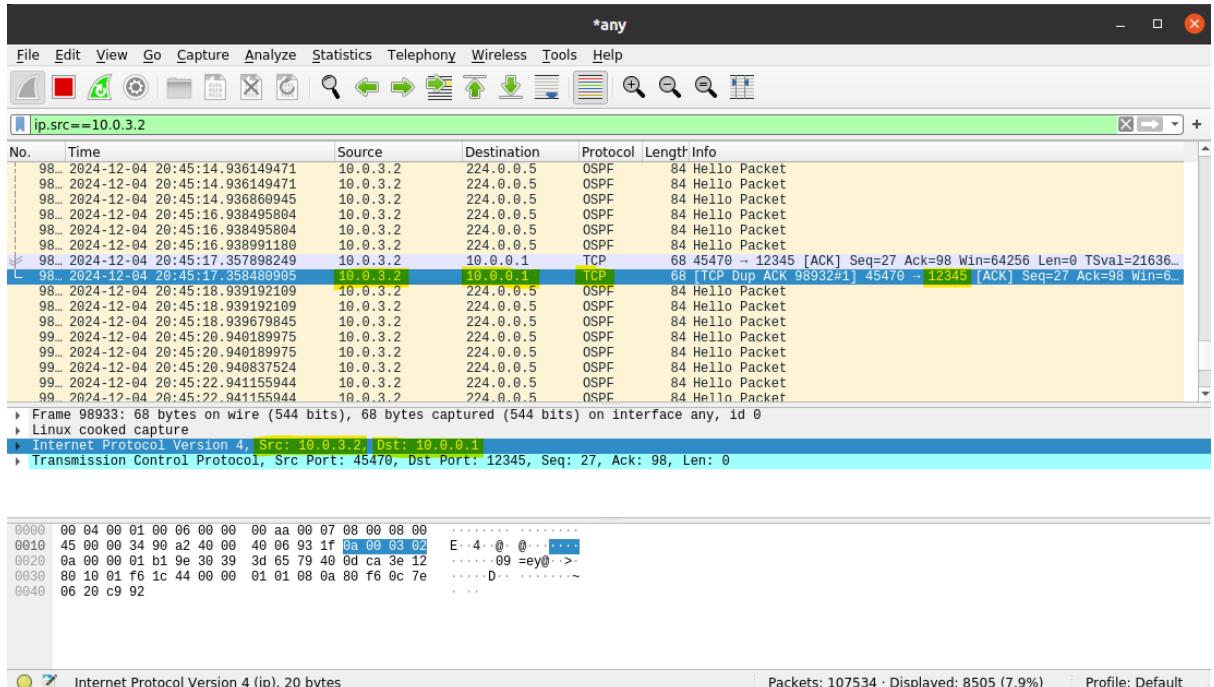


- The link between n4 to n5 is represented via the bidirectional link between n1 to n5:**

The Wireshark capturing the link between n4 to n5 via packet transfers from n1 to n5 is shown below:



The Wireshark capturing the link between n4 to n5 via packet transfers from n5 to n1 is shown below:



A summary of the connections established at roots n1, n2, n5 and n6 are shown below:

The figure shows four terminal windows with the following outputs:

- Terminal 1 (n1):** root@n1:/tmp/pycore.2/n1.conf# nc -k -l 12345
 - n1 server sending packets
 - n6 client sending packets
 - n1 sending packets
 - n6 sending packets to n1
 - n1 sending packets to n6
 - n6 client to n1 server
 - n1 server to n6 client
- Terminal 2 (n2):** root@n2:/tmp/pycore.2/n2.conf# nc -k -l 12345
 - n2 server sending packets
 - n5 client sending packets
 - n2 server sending packets
 - n2 server sending packets
 - n2 sending packets
 - n2 sending packets to n5
 - n5 sending packets to n2
- Terminal 3 (n5):** root@n5:/tmp/pycore.2/n5.conf# nc 10.0.0.1 12345
 - n2 server sending packets
 - n5 client sending packets
 - n2 server sending packets
 - n2 server sending packets
 - n2 sending packets
 - n2 sending packets to n5
 - n5 sending packets to n2
- Terminal 4 (n6):** root@n6:/tmp/pycore.2/n6.conf# nc 10.0.0.1 12345
 - n1 server sending packets
 - n6 client sending packets
 - n1 sending packets
 - n6 sending packets to n1
 - n1 sending packets to n6
 - n6 client to n1 server
 - n1 server to n6 client

d. Provide the iptables commands you used.

The commands used:

- **iptables -t nat -A PREROUTING -s 10.0.3.2 -d 10.0.0.1 -p tcp --dport 12345 -j DNAT --to-destination 10.0.1.2:12345**
- **iptables -t nat -A POSTROUTING -s 10.0.3.2 -d 10.0.1.2 -p tcp --dport 12345 -j MASQUERADE**

Combination of DNAT and NAT commands used to change traffic and allow bidirectional traffic to go from n5 to n2 is shown below:



```
root@n4:/tmp/pycore.1/n4.conf# iptables -t nat -A PREROUTING -s 10.0.3.2 -d 10.0.0.1 -p tcp --dport 12345 -j DNAT --to-destination 10.0.1.2:12345
root@n4:/tmp/pycore.1/n4.conf# iptables -t nat -A POSTROUTING -s 10.0.3.2 -d 10.0.1.2 -p tcp --dport 12345 -j MASQUERADE
root@n4:/tmp/pycore.1/n4.conf#
```

- e. Show the output of “**iptables -L -n -v**”, the output of “**iptables -t NAT -L -n -v**”, and the output of “**iptables -t mangle -L -n -v**”.

Output of the command **iptables -L -n -v** is shown below:



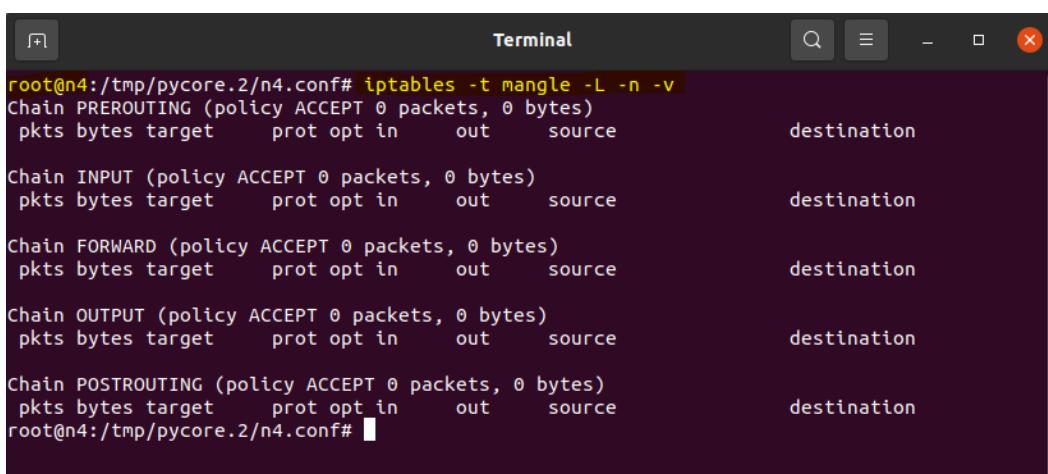
```
root@n4:/tmp/pycore.2/n4.conf# iptables -L -n -v
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out     source          destination
Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out     source          destination
Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out     source          destination
root@n4:/tmp/pycore.2/n4.conf#
```

Output of the command **iptables -t nat -L -n -v** is shown below:



```
root@n4:/tmp/pycore.2/n4.conf# iptables -t nat -L -n -v
Chain PREROUTING (policy ACCEPT 1 packets, 60 bytes)
pkts bytes target     prot opt in     out     source          destination
  1   60 DNAT       tcp  --  *      *      10.0.3.2        10.0.0.1        tcp dpt:12345 to:10.0.1.2:12345
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out     source          destination
Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out     source          destination
Chain POSTROUTING (policy ACCEPT 1 packets, 60 bytes)
pkts bytes target     prot opt in     out     source          destination
  1   60 MASQUERADE  tcp  --  *      *      10.0.3.2        10.0.1.2        tcp dpt:12345
root@n4:/tmp/pycore.2/n4.conf#
```

Output of the command **iptables -t mangle -L -n -v** is shown below:



```
root@n4:/tmp/pycore.2/n4.conf# iptables -t mangle -L -n -v
Chain PREROUTING (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out     source          destination
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out     source          destination
Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out     source          destination
Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out     source          destination
Chain POSTROUTING (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out     source          destination
root@n4:/tmp/pycore.2/n4.conf#
```

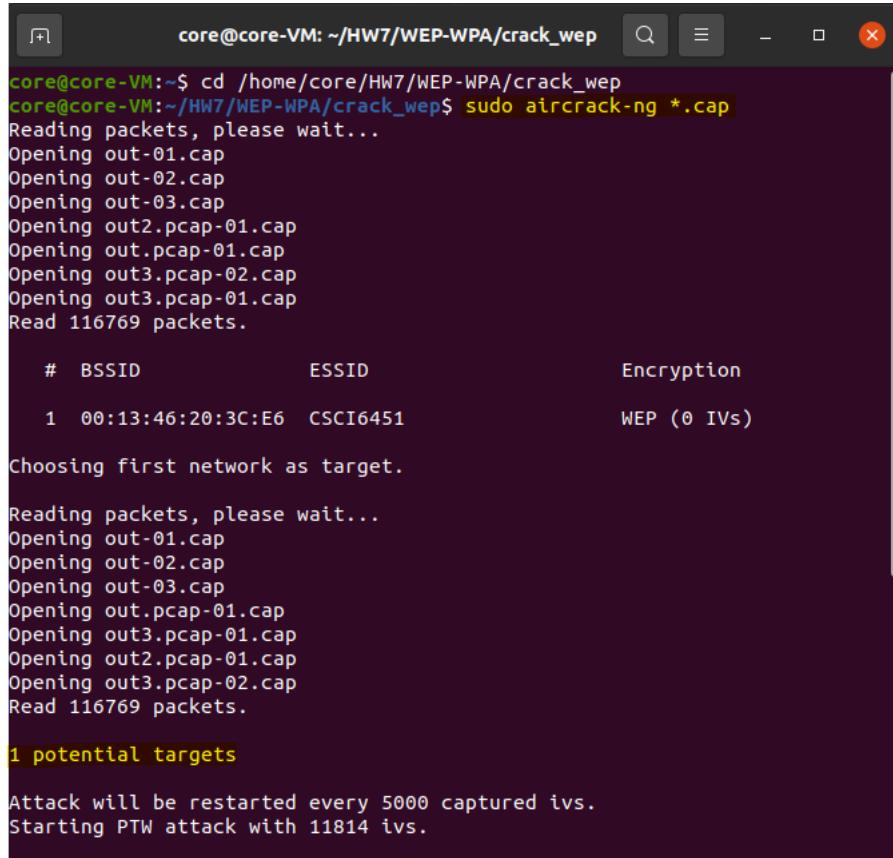
Wireless Section (10pts)

A tar file is included with this exercise. It includes two folders: crack_wpa and crack_wep. You will be using aircrack-ng to crack WPA and WEP. This tool can be installed on any Linux box (using apt-get on Ubuntu for instance) or you can download Kali Linux.

1: WEP

In the directory, there are several captures (*.cap files) from a WLAN network configured for WEP. To crack the password using the run.sh script. All it does is run: aircrack-ng *.cap. The tool will consume all the files and attempt to crack the WEP password.

The output of running the command **sudo aircrack-ng *.cap** is shown below:



```
core@core-VM:~/HW7/WEP-WPA/crack_wep$ sudo aircrack-ng *.cap
Reading packets, please wait...
Opening out-01.cap
Opening out-02.cap
Opening out-03.cap
Opening out2.pcap-01.cap
Opening out.pcap-01.cap
Opening out3.pcap-02.cap
Opening out3.pcap-01.cap
Read 116769 packets.

#   BSSID           ESSID          Encryption
1  00:13:46:20:3C:E6  CSCI6451      WEP (0 IVs)

Choosing first network as target.

Reading packets, please wait...
Opening out-01.cap
Opening out-02.cap
Opening out-03.cap
Opening out.pcap-01.cap
Opening out3.pcap-01.cap
Opening out2.pcap-01.cap
Opening out3.pcap-02.cap
Read 116769 packets.

1 potential targets

Attack will be restarted every 5000 captured ivs.
Starting PTW attack with 11814 ivs.
```

Q1: What is the WEP password? It should take you about a second to crack, but this is because you have all the data already. Show screenshot of the cracked password

The WEP password is **DE:AD:BE:EF:12** as shown below:

```
Aircrack-ng 1.6

[00:00:01] Tested 11043 keys (got 11245 IVs)

KB      depth   byte(vote)
0      11/ 12   DE(14592) 2D(14336) B7(14336) CD(14336) DF(14336)
1      9/ 14    D2(14336) 30(14080) 45(14080) 56(14080) 8B(14080)
2      3/ 11    BE(15616) 7B(15104) 62(14848) 64(14848) 75(14592)
3      0/  2    EF(17408) AC(16384) 26(15104) 46(14848) 99(14592)
4      1/  3    12(16640) 37(16128) 2C(15360) 6D(15104) 1D(14848)

KEY FOUND! [ DE:AD:BE:EF:12 ]
Decrypted correctly: 100%

core@core-VM:~/HW7/WEP-WPA/crack_wep$
```

2: WPA

In this directory, there are several captures (*.cap files) from a WLAN network configured for WPA. To crack the password using the run.sh script. It runs a dictionary attack using the dictionary file provided.

The output of running the command **sudo aircrack-ng *.cap -w dictionary.txt** is shown below:

```
core@core-VM:~/HW7/WEP-WPA$ ls
crack_wep  crack_wpa
core@core-VM:~/HW7/WEP-WPA$ cd crack_wpa
core@core-VM:~/HW7/WEP-WPA/crack_wpa$ sudo aircrack-ng *.cap -w dictionary.txt
[sudo] password for core:
Reading packets, please wait...
Opening out3.pcap-01.cap
Read 1389 packets.

#  BSSID          ESSID           Encryption
1  00:13:46:20:3C:E6  CSCI6451        WPA (1 handshake)

Choosing first network as target.

Reading packets, please wait...
Opening out3.pcap-01.cap
Read 1389 packets.

1 potential targets
```

Q2: What is the WPA password? Show screenshot of the cracked password

The WPA password is **DEADBEEF12** as shown below:

```
core@core-VM: ~/HW7/WEP-WPA/crack_wpa
```

Aircrack-ng 1.6

[00:01:17] 431199/870530 keys tested (5643.91 k/s)

Time left: 1 minute, 17 seconds 49.53%

KEY FOUND! [DEADBEEF12]

Master Key : 72 82 A1 42 41 66 E2 28 53 D7 CA A1 CD 11 6B 58
B9 3C ED 82 27 CF 63 14 70 08 5A AB 58 6D DE 91

Transient Key : 05 B1 F9 7D D3 F8 44 4E 58 89 47 95 36 AB A3 FF
EB B5 1A E6 78 90 B7 77 45 53 E9 5F 96 28 B8 C4
C0 18 10 A8 C1 99 BD 34 C9 71 3C 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

EAPOL HMAC : 84 43 23 FF 43 1C 14 FB 50 42 74 06 62 2E 54 BA