VPN Lab: The Container Version

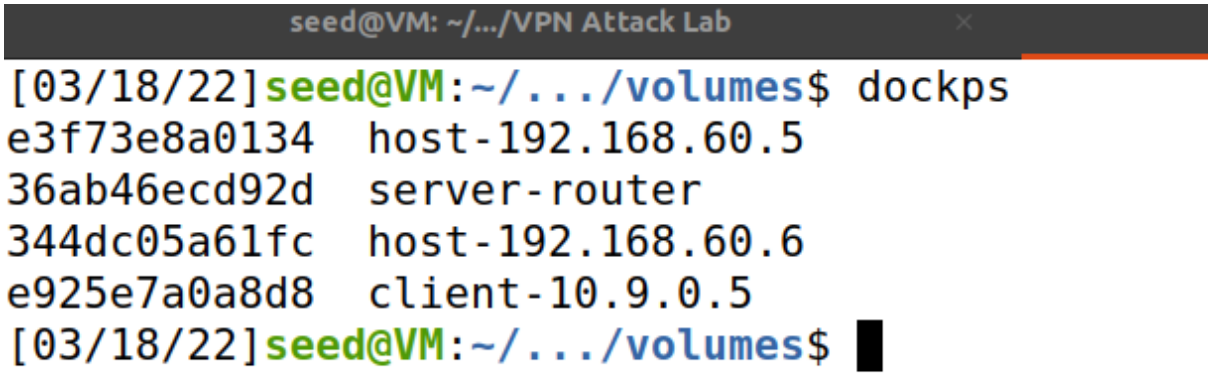
Internet Security

Name: Gaurav Upadhyay

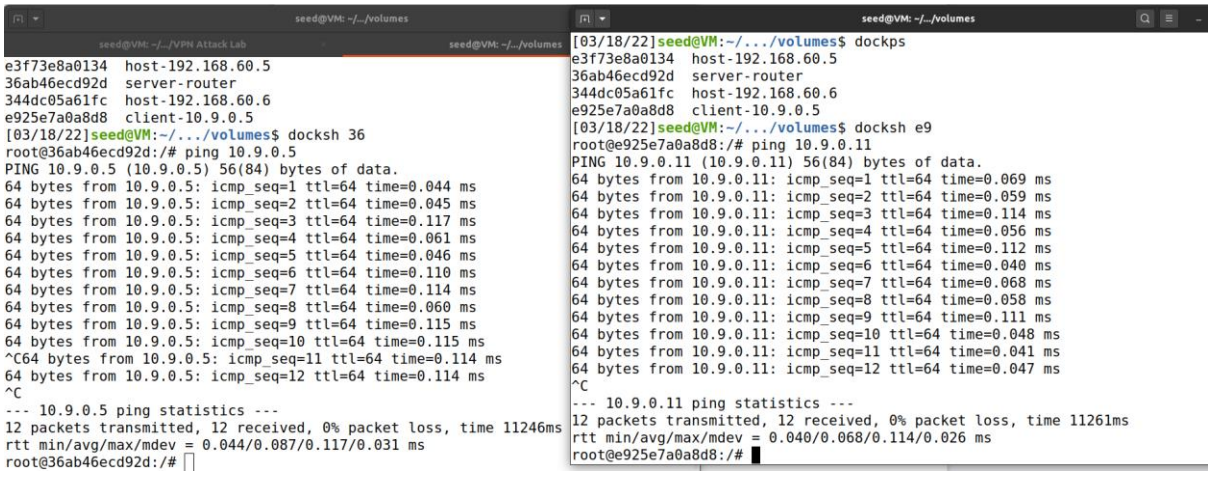
Email: gsupadhy@syr.edu

Task 1: Network Setup

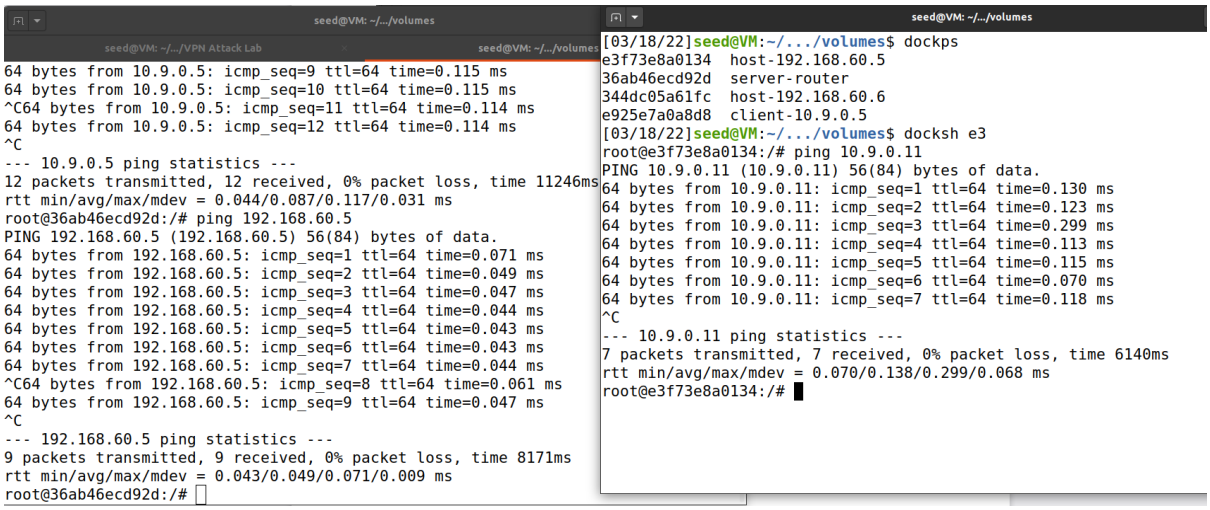
We set up the environment as follows:



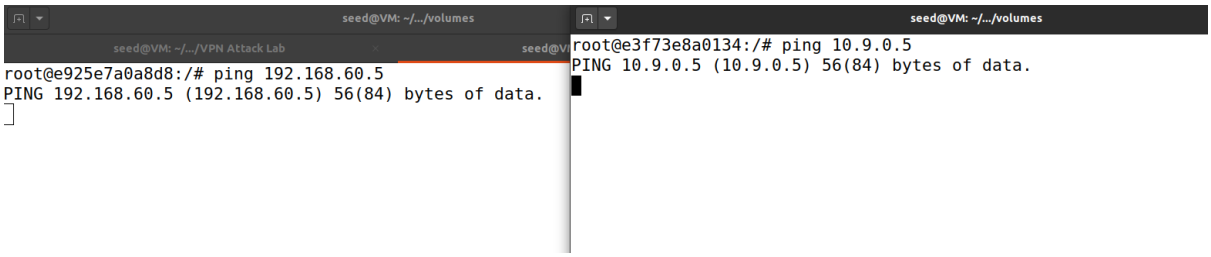
Host U can communicate with VPN Server:



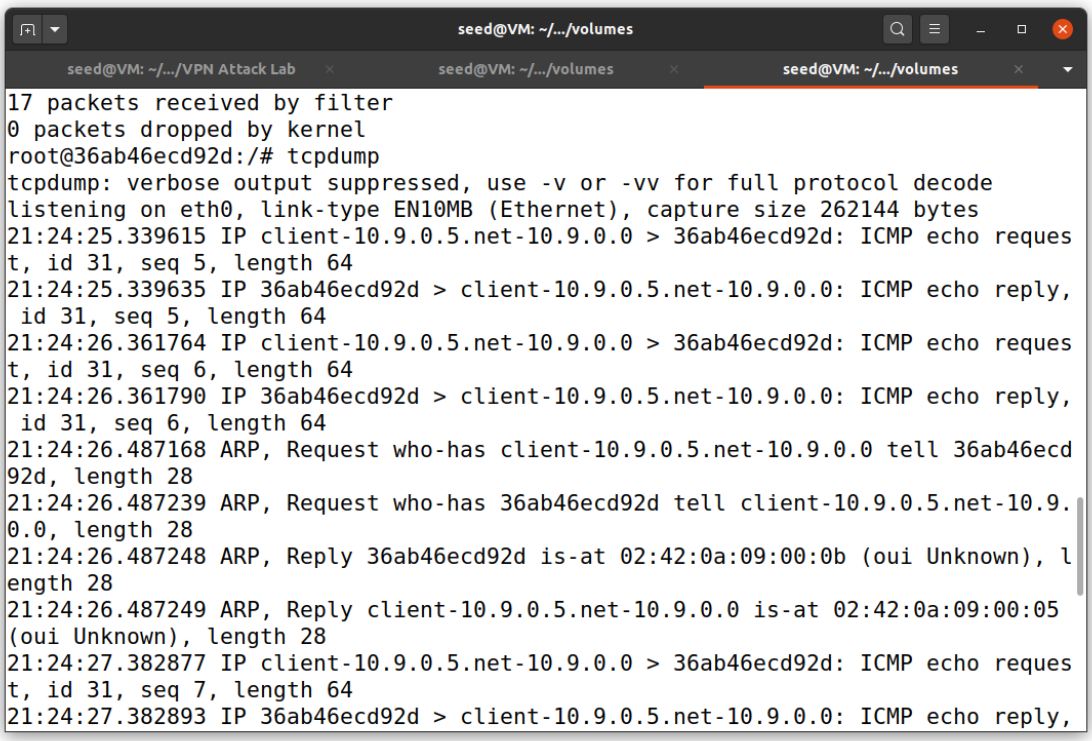
VPN Server can communicate with Host V:



Host U should not be able to communicate with Host V:



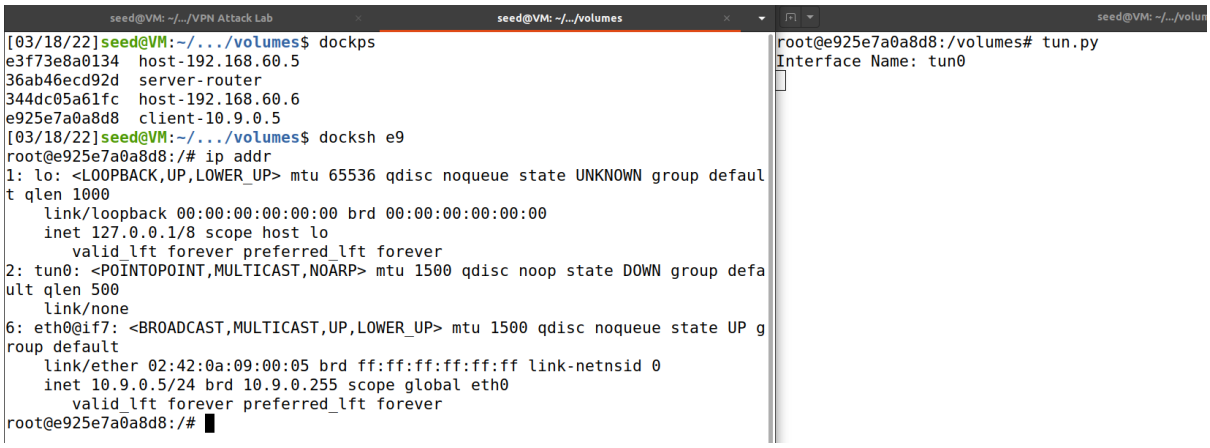
Run tcpdump on the router, and sniff the traffic on each of the network. We showed that we can capture packets:



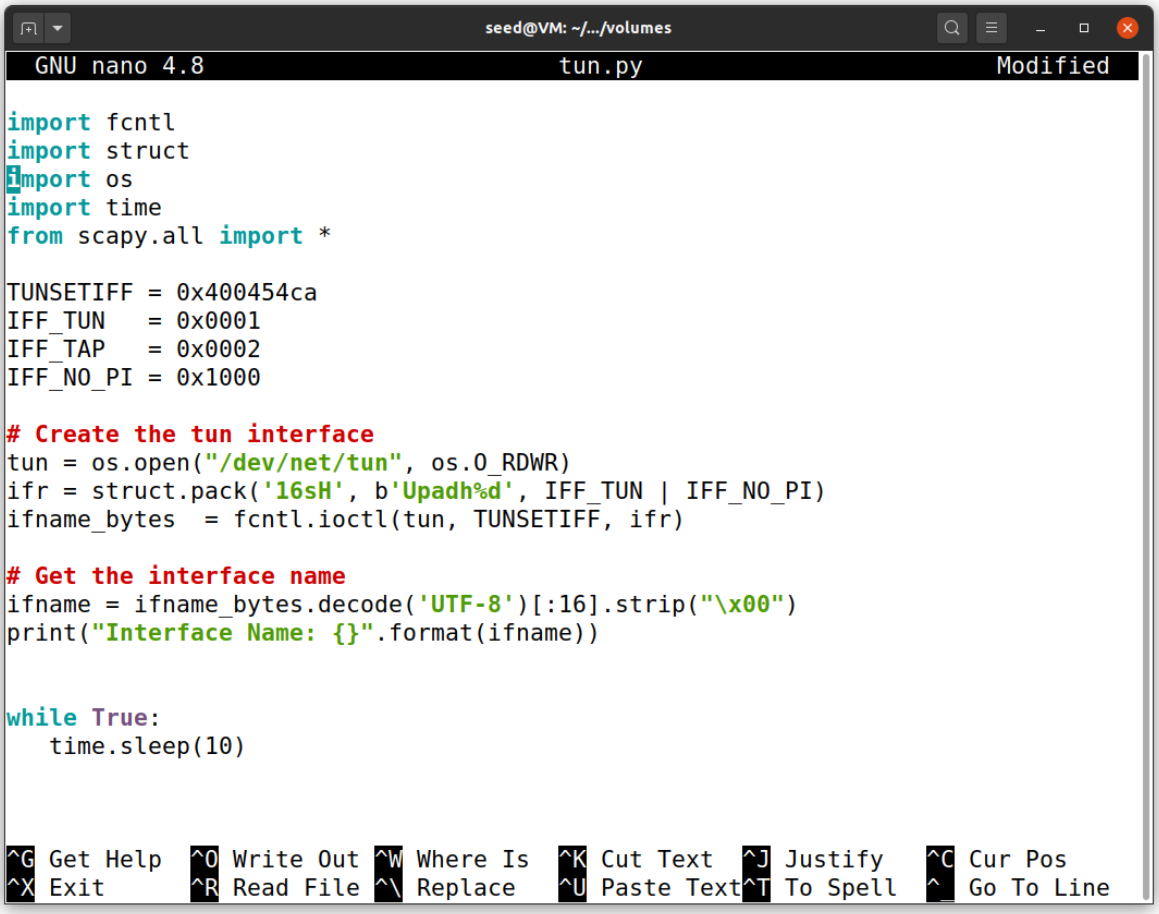
Task 2: Create and Configure TUN Interface:

Task 2.a: Name of the Interface:

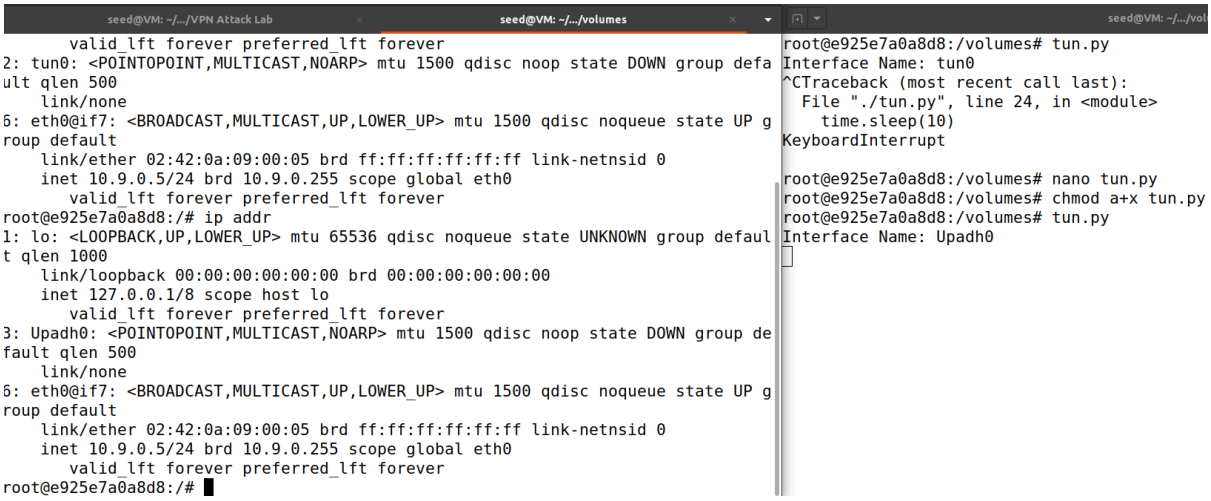
We run the code given to us to check the interface created:



We make changes in the code to create an own interface with last name Upadh:

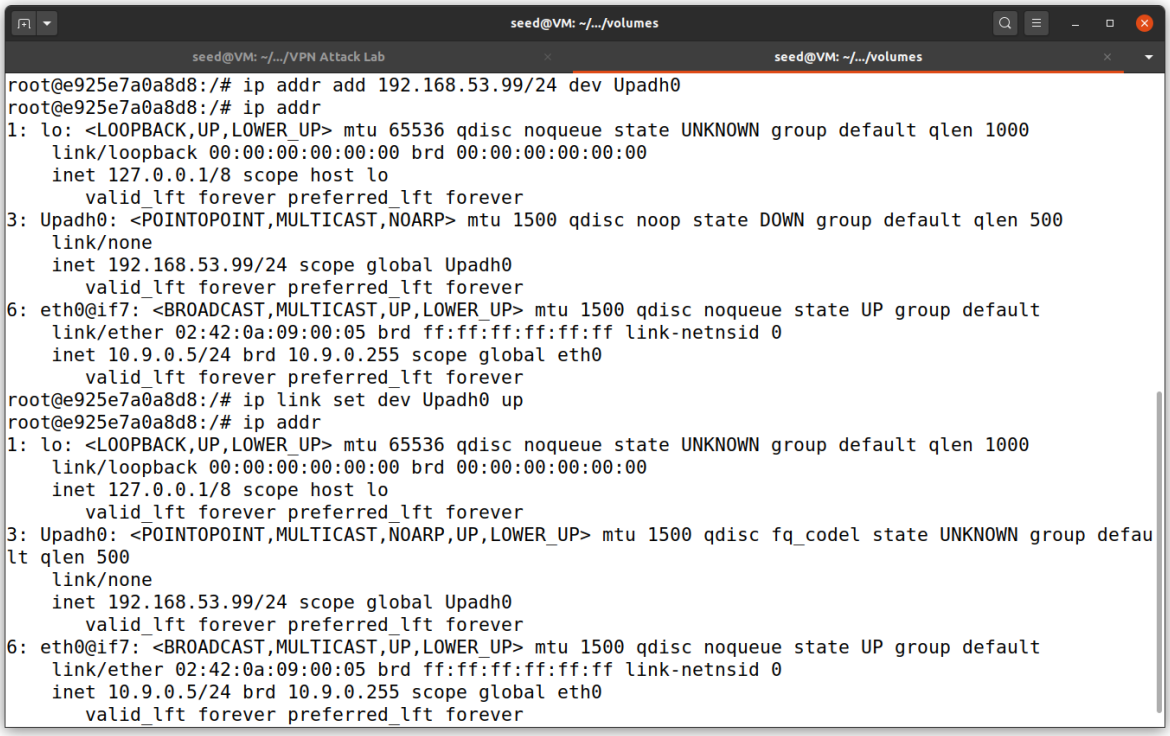


WE run the code again to see that the interface Upadh0 is created:

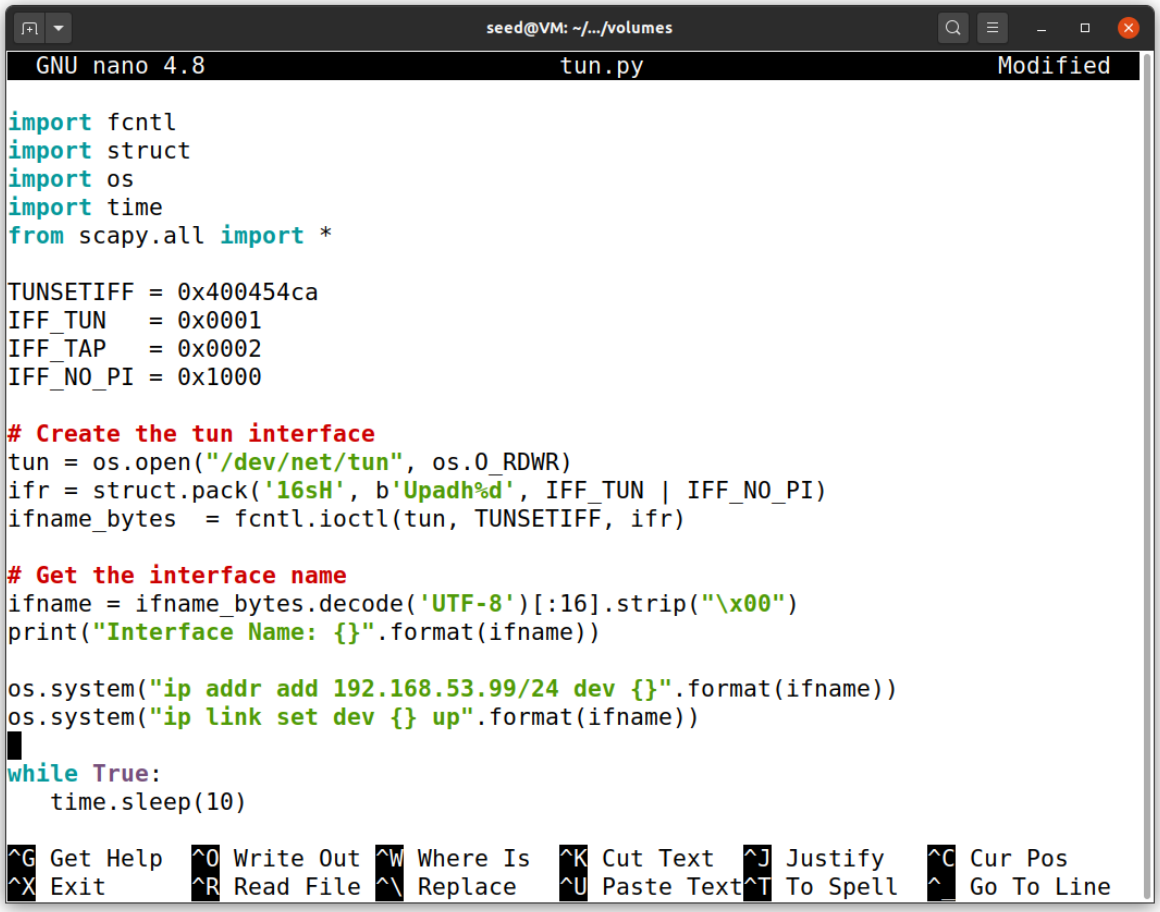


Task 2.b: Set up the TUN Interface:

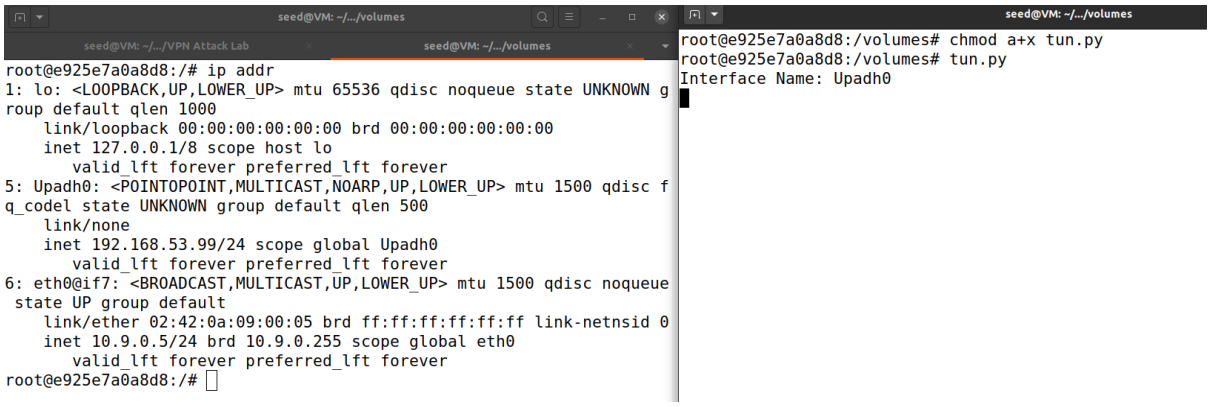
First, we added the ip address manually using the command- **ip addr add** and command – **ip link set up** to bring up the interface:



Also, this is done automatically by adding the commands to the code:



We run the code again to see that the interface has been added automatically:



Task 2.c: Read from the TUN Interface:

Code:

```
seed@VM: ~/.../volumes
seed@VM: ~/.../volumes  seed@VM: ~/.../volumes
 GNU nano 4.8                    tun.py                         Modified
from scapy.all import *

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'Upadh%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

while True:
        # Get a packet from the tun interface
        packet = os.read(tun, 2048)
        if packet:
                ip = IP(packet)
                print(ip.summary())


^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Paste Text^T To Spell  ^_ Go To Line
```

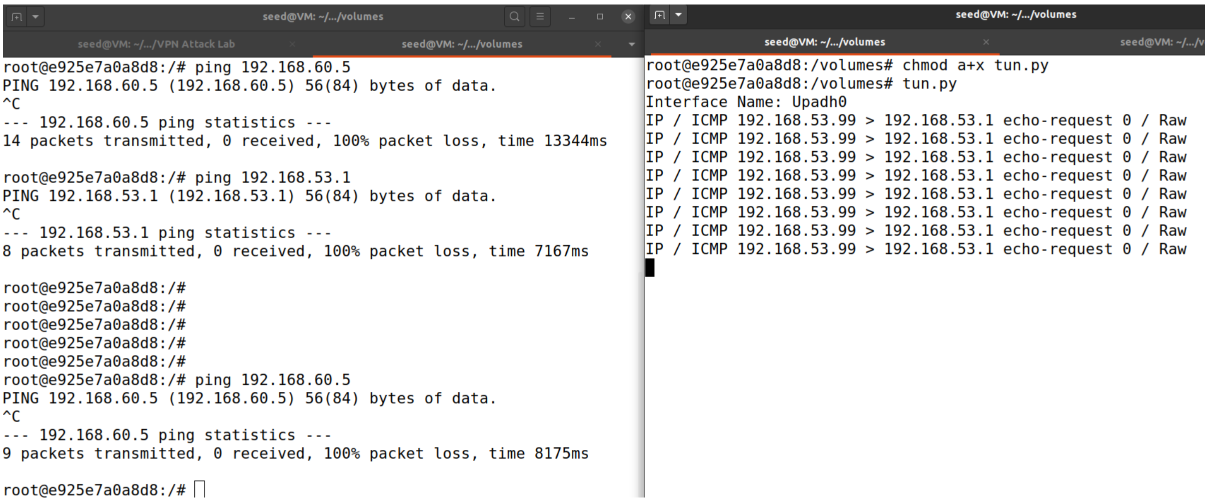
First, On Host U, we ping a host in the 192.168.53.0/24 network. Specifically, we ping the 192.168.53.1 network. As the interface is not yet configured to get a reply back, we can see that packets were not received back. However, being in the same LAN network, the packets get sent successfully.

```
seed@VM: ~/.../VPN Attack Lab  seed@VM: ~/.../volumes
root@e925e7a0a8d8:/# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN g
roup default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
 state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
7: Upadh0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc f
q_codel state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global Upadh0
       valid_lft forever preferred_lft forever
root@e925e7a0a8d8:/# ping 192.168.53.1
PING 192.168.53.1 (192.168.53.1) 56(84) bytes of data.
^C
--- 192.168.53.1 ping statistics ---
7 packets transmitted, 0 received, 100% packet loss, time 6133ms

root@e925e7a0a8d8:/#
```

```
seed@VM: ~/.../volumes  seed@VM: ~/.../volumes
root@e925e7a0a8d8:/volumes# chmod a+x tun.py
root@e925e7a0a8d8:/volumes# tun.py
Interface Name: Upadh0
^CTraceback (most recent call last):
  File "./tun.py", line 27, in <module>
    # Get a packet from the tun interface
KeyboardInterrupt

root@e925e7a0a8d8:/volumes# chmod a+x tun.py
root@e925e7a0a8d8:/volumes# tun.py
Interface Name: Upadh0
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
```

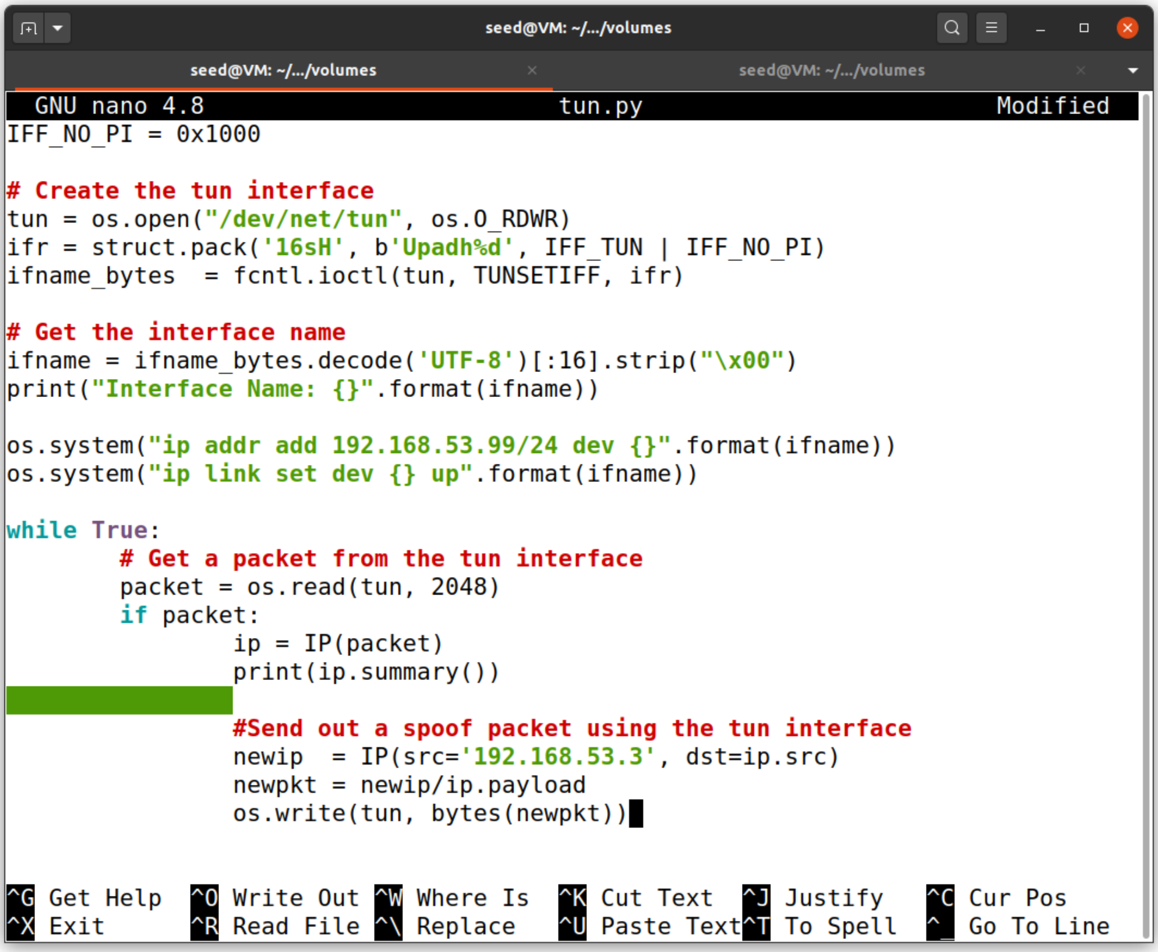
Now, on Host U, we ping a host in the internal network 192.168.60.0/24. Specifically, we ping the 192.168.60.5 network. Similar to before, we can see that no packets are received.

However, in this case, no packets were sent out as well due to which the client side screen is blank as it tried sending out packets to network outside of the host network which is unreachable.

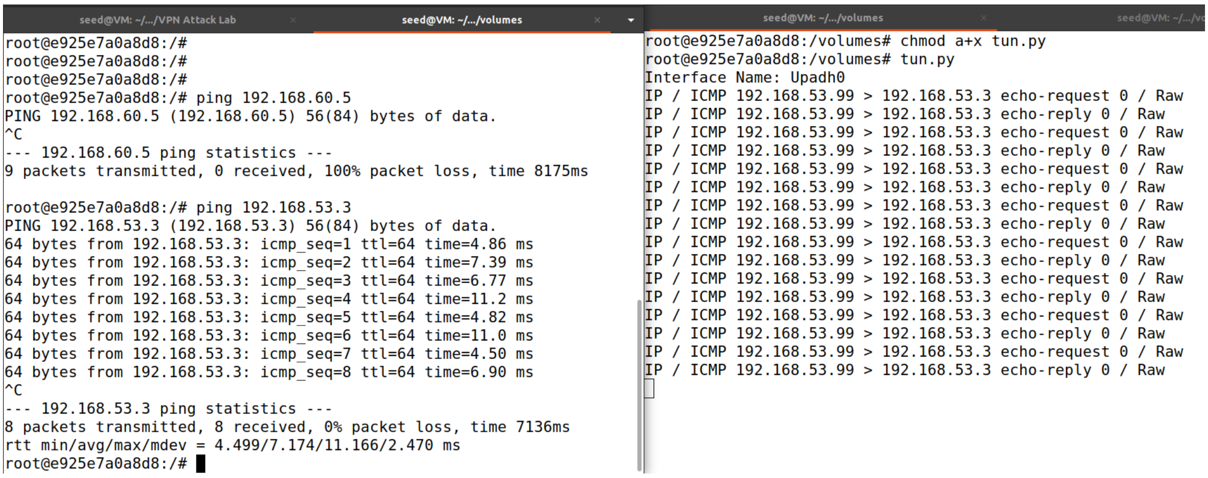


Task 2.d: Write to the TUN Interface

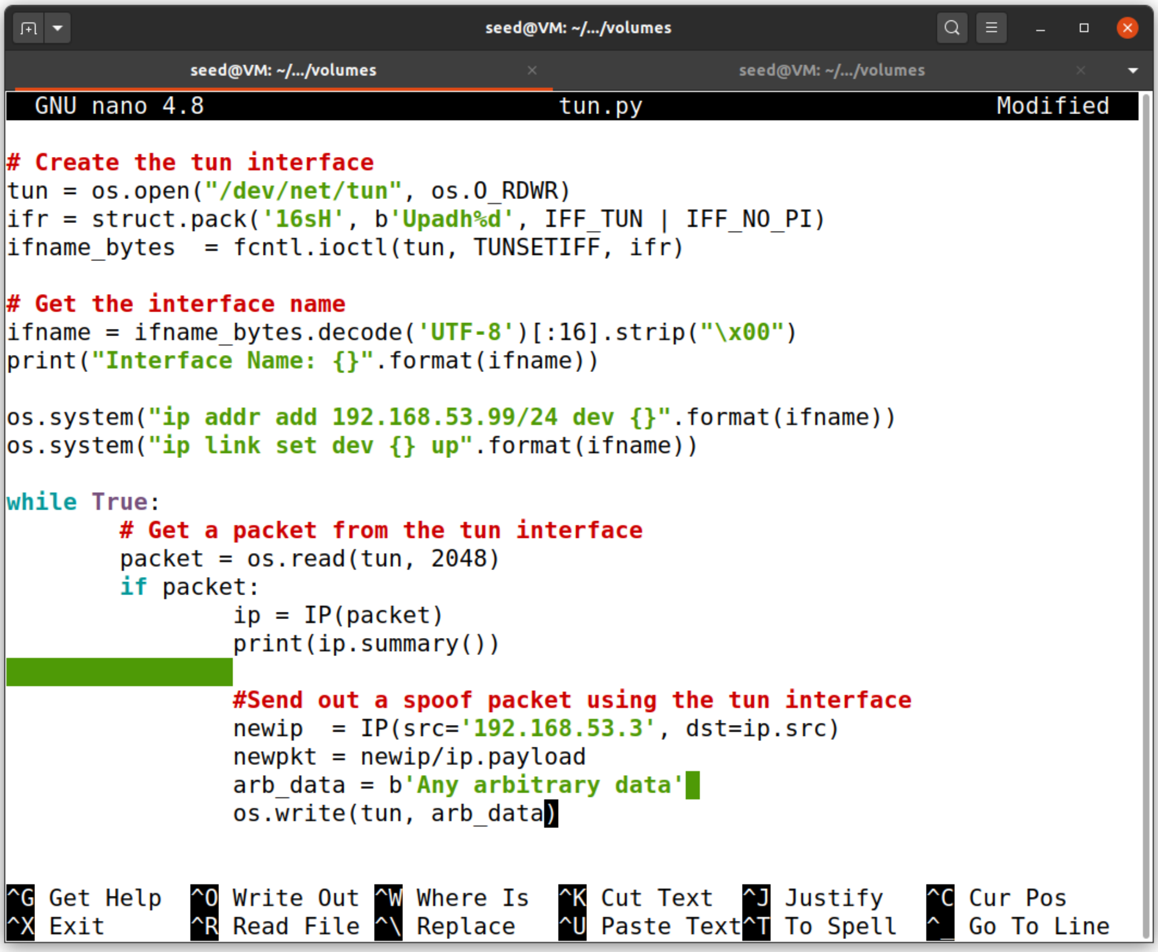
We modified the code so that a spoofed packet is sent out with src IP as 192.168.53.3 :



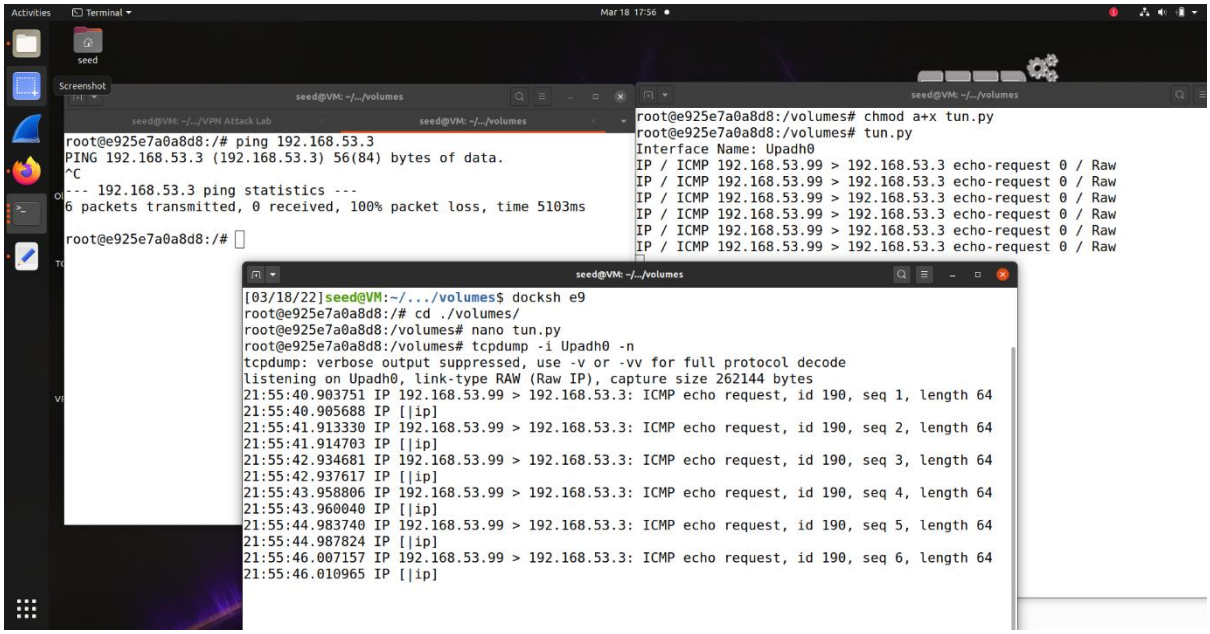
We run the code to see that packets were sent and received successfully:



Instead of writing an IP packet to the interface, we modify the code to write some arbitrary data to the interface to send it as bytes:



We run the code to see the output as follows:



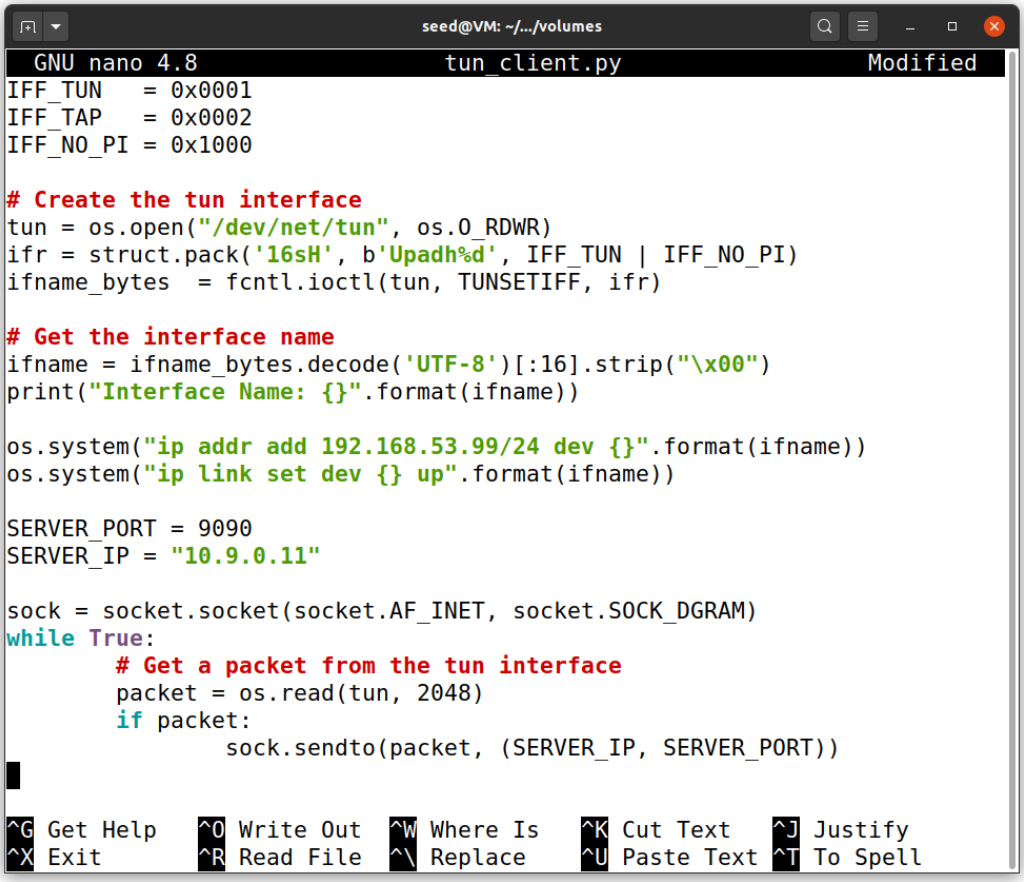
Task 3: Send the IP Packet to VPN Server Through a Tunnel

Tun-server.py code:

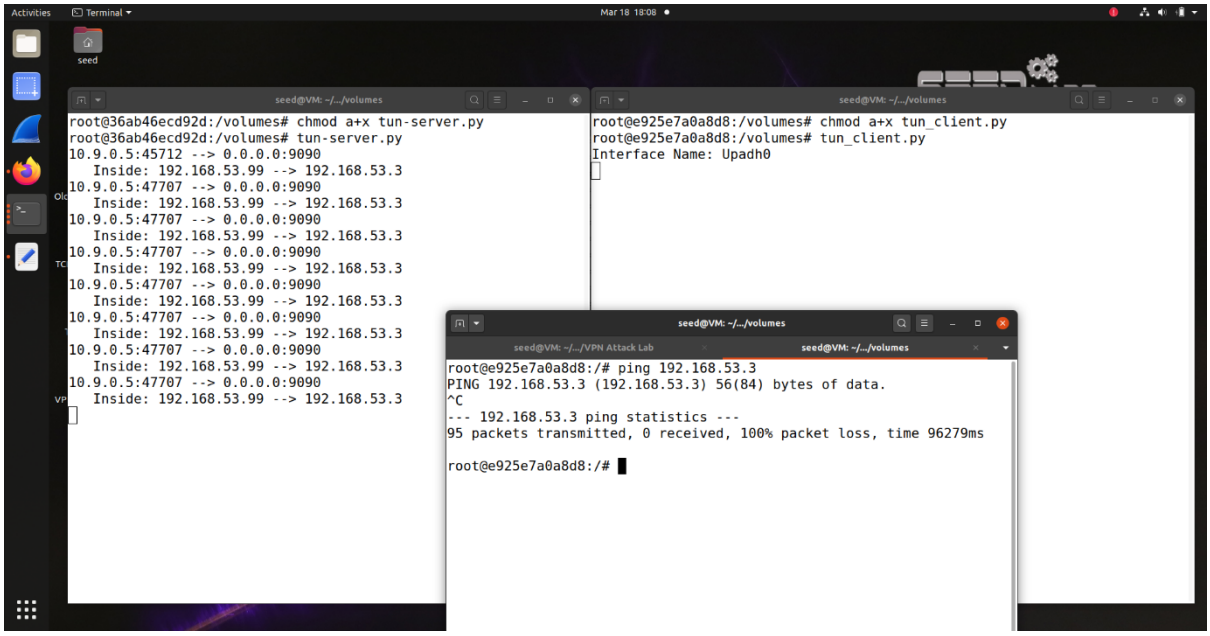
```
IP_A = "0.0.0.0"
PORT = 9090
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_A, PORT))
while True:
        data, (ip, port) = sock.recvfrom(2048)
        print("{}:{} --> {}:{}".format(ip, port, IP_A, PORT))
        pkt = IP(data)
        print("   Inside: {} --> {}".format(pkt.src, pkt.dst))
        os.write(tun, bytes(pkt))


^G Get Help  ^O Write Out^W Where Is  ^K Cut Text  ^J Justify
^X Exit      ^R Read File^\ Replace   ^U Paste Tex^T To Spell
```

We modified the tun.py code to tun_client.py while making changes in the code. We also added the server IP and server Port in the code.

Tun_client.py code:

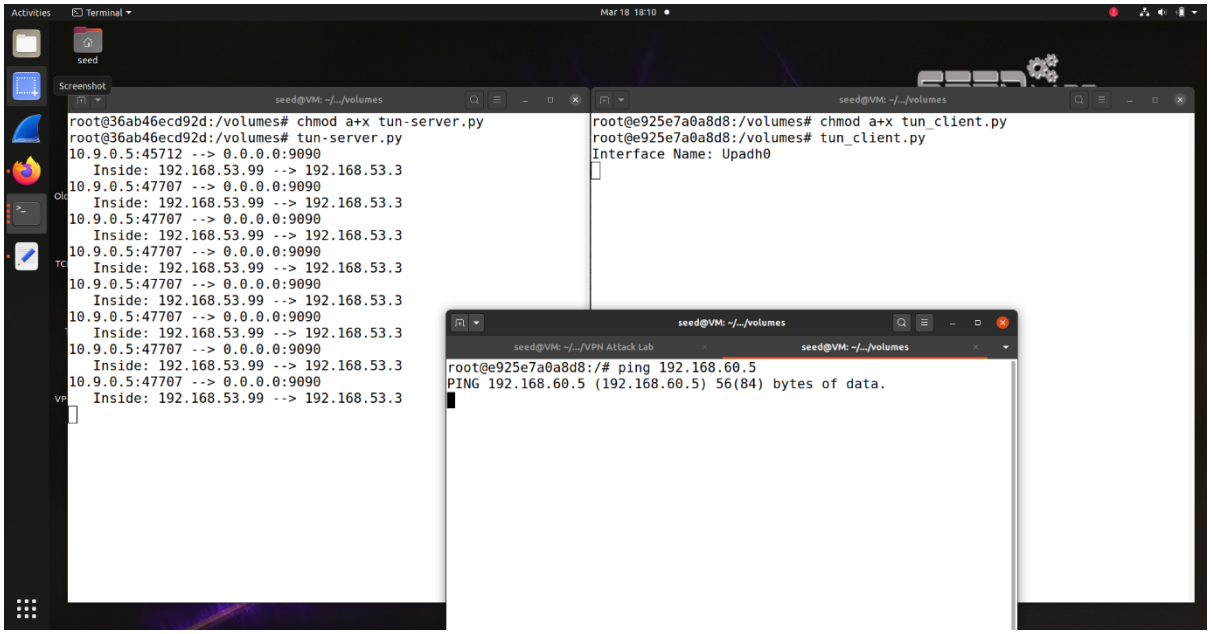


We run the server code and client code and ping 192.168.53.3 from client side:



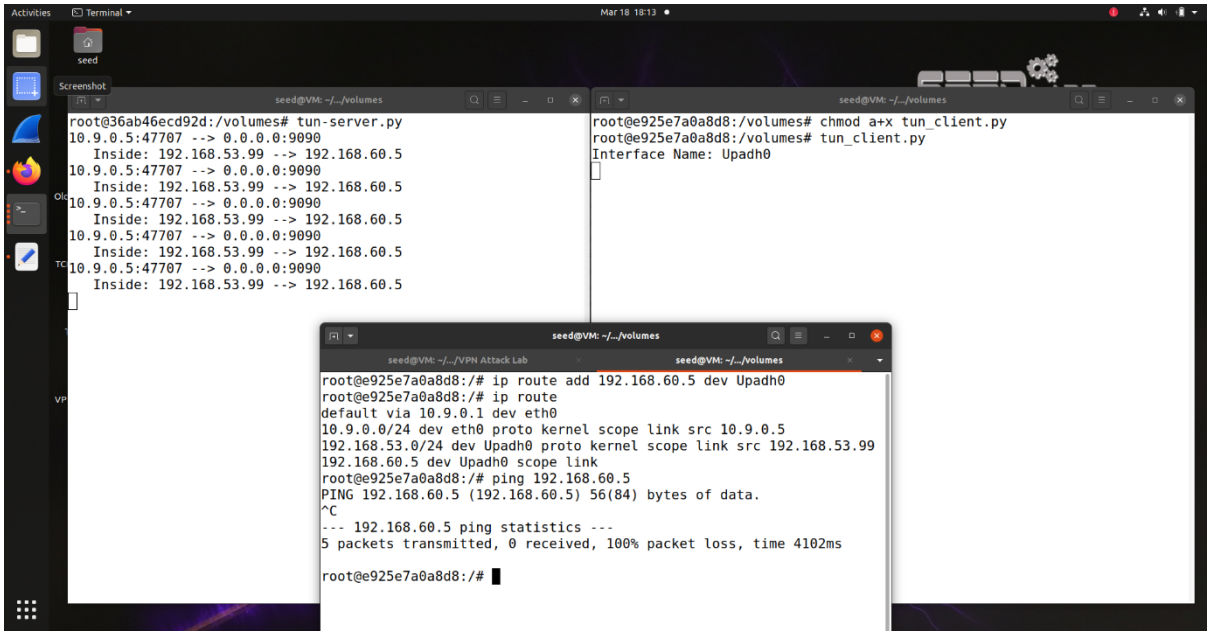
We can see that the server prints out server and destination port with the packet source and destination information. As the host has a tun interface with routing route configured, pinging an IP in this network passes through the tun interface and is sent for listening through the socket.

Now, we try to ping a host in another private network:



Nothing is printed as the interface is not configured to pass the packets through it to a private network.

After adding routing commands, we ping to a private network again to see that packets now can pass through the Upadh0 interface which then gets printed on server side:



Task 4: Set Up the VPN Server:

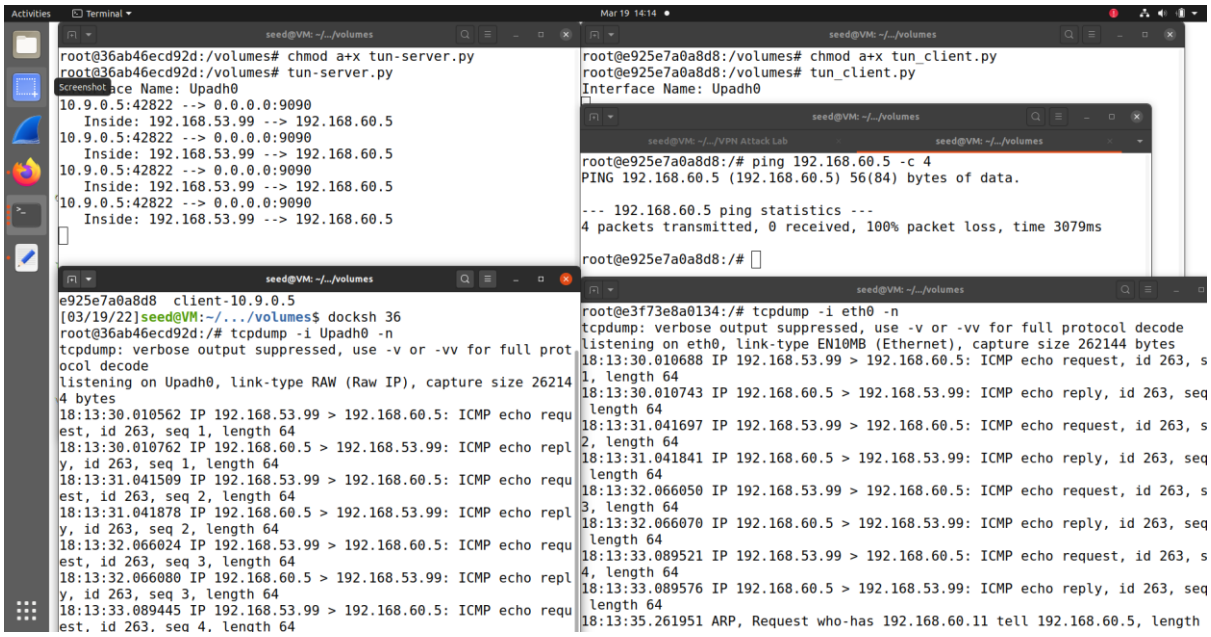
Tun-server.py code:

```
seed@VM: ~/.../volumes
  GNU nano 4.8                tun-server.py                Modified
from scapy.all import*
TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'Upadh%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

os.system("ip addr add 192.168.53.50/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))


IP_A = "0.0.0.0"
PORT = 9090
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_A, PORT))
while True:
        data, (ip, port) = sock.recvfrom(2048)
        print("{}:{} --> {}:{}".format(ip, port, IP_A, PORT))
        pkt = IP(data)
        print("   Inside: {} --> {}".format(pkt.src, pkt.dst))
        os.write(tun, bytes(pkt))


^G Get Help  ^O Write Out^W Where Is  ^K Cut Text  ^J Justify
^X Exit      ^R Read File^\ Replace   ^U Paste Tex^T To Spell
```

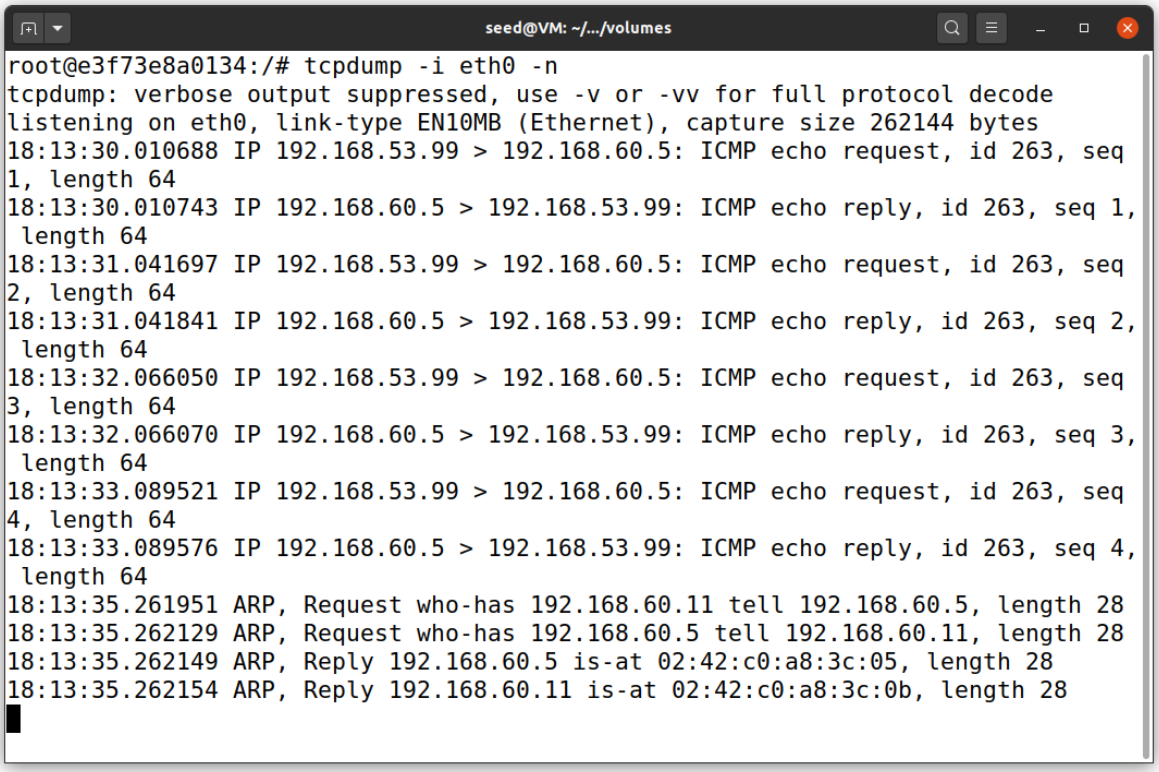
Tun_client.py code:

```
seed@VM: ~/.../volumes
  GNU nano 4.8                tun_client.py                Modified
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'Upadh%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

SERVER_PORT = 9090
SERVER_IP = "10.9.0.11"

os.system("ip route add 192.168.60.0/24 dev {}".format(ifname))

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while True:
        # Get a packet from the tun interface
        packet = os.read(tun, 2048)
        if packet:
                sock.sendto(packet, (SERVER_IP, SERVER_PORT))


^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify
^X Exit        ^R Read File   ^\ Replace     ^U Paste Text  ^T To Spell
```

We try to ping to a private network host and notice that we were able to ping a private network host and receive replies in the server. However, it is not showing on our host terminal:

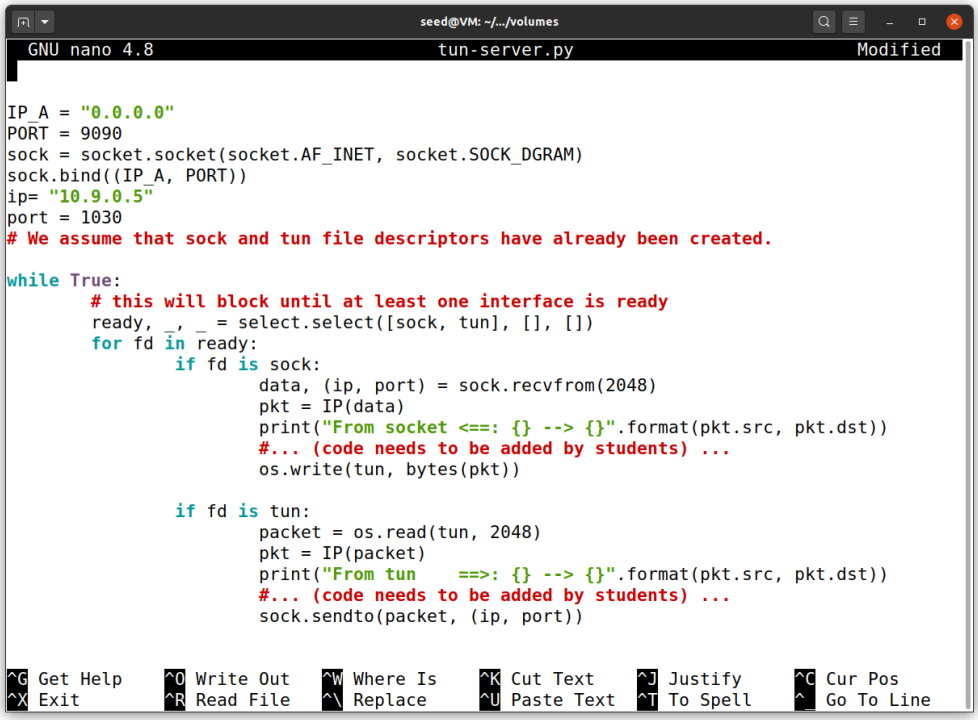


We use tcpdump to show that that the ICMP packets have arrived at Host V:

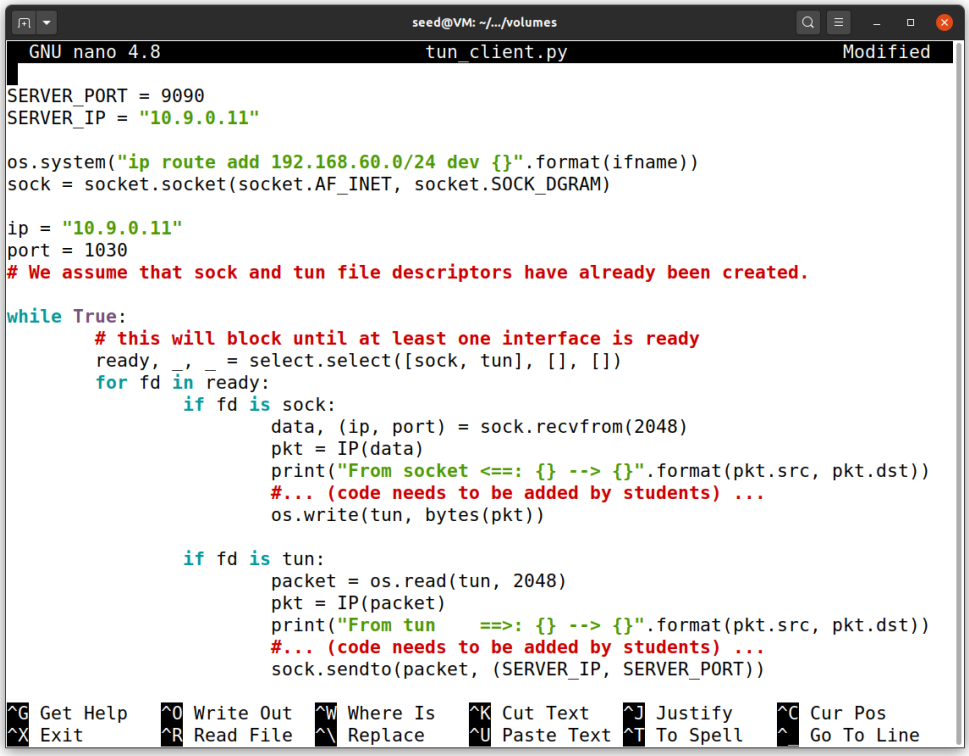


Task 5: Handling Traffic in Both Directions

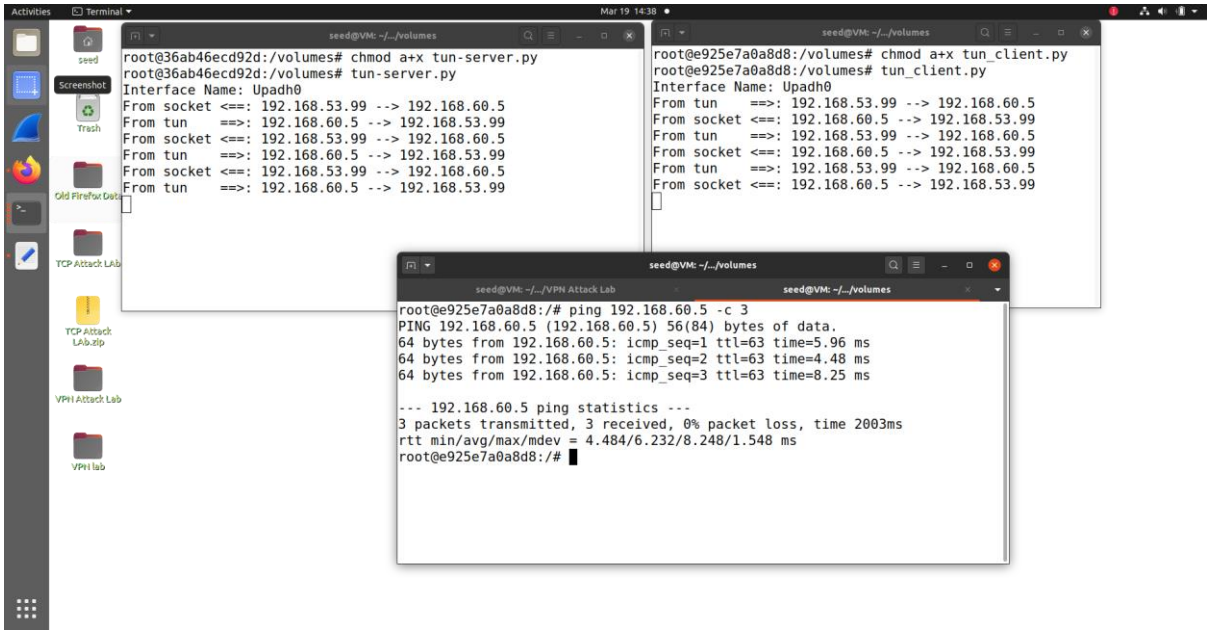
Tun-server.py code:



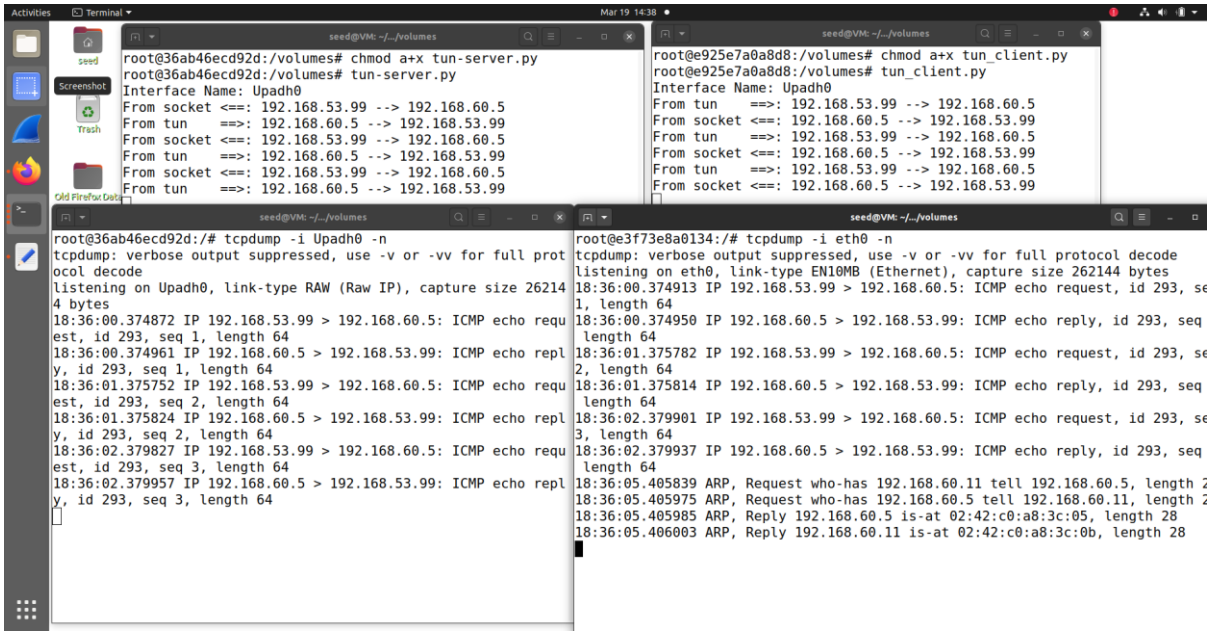
Tun_client.py code:



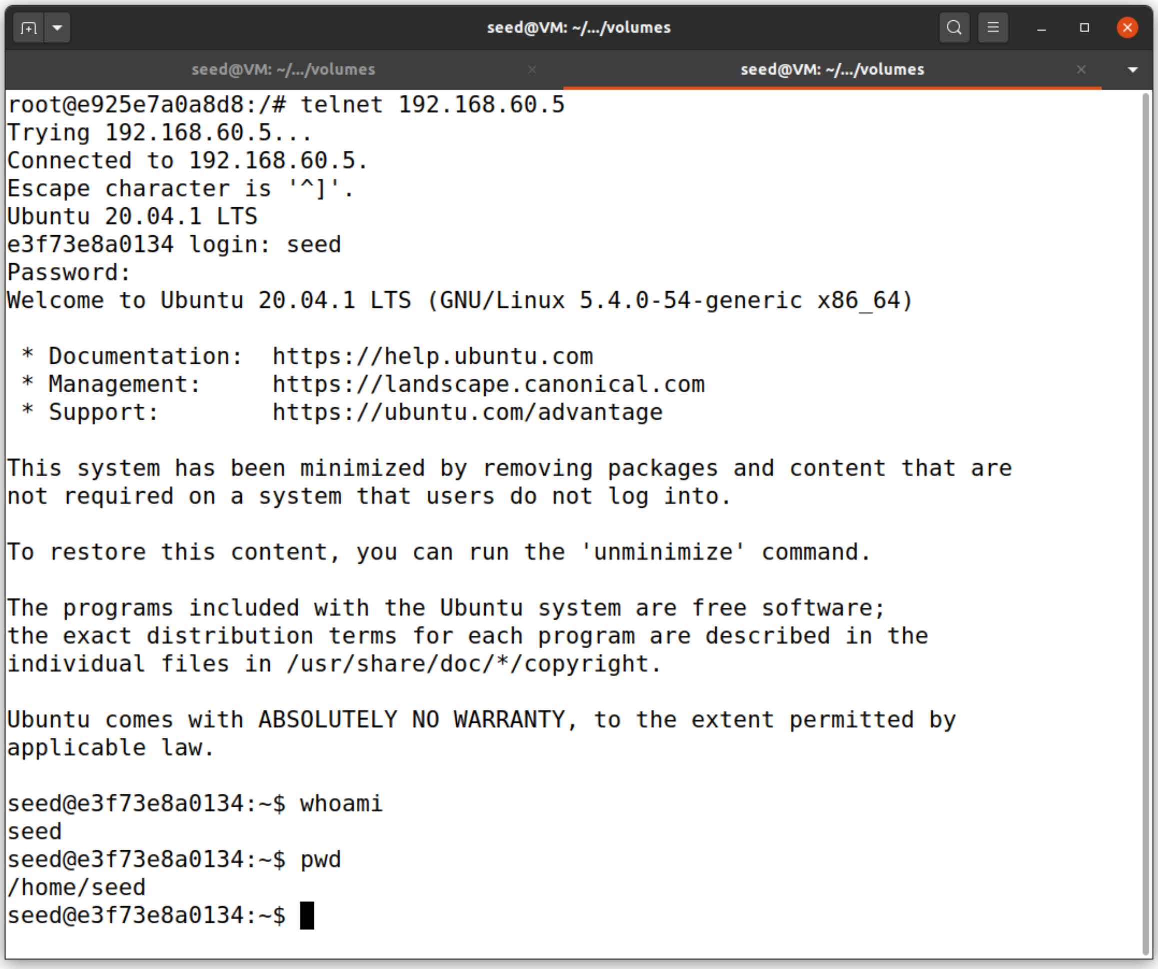
We try to run the code and can see that the ping was successful to a private network and Host U and V were both getting packets and sending back replies:



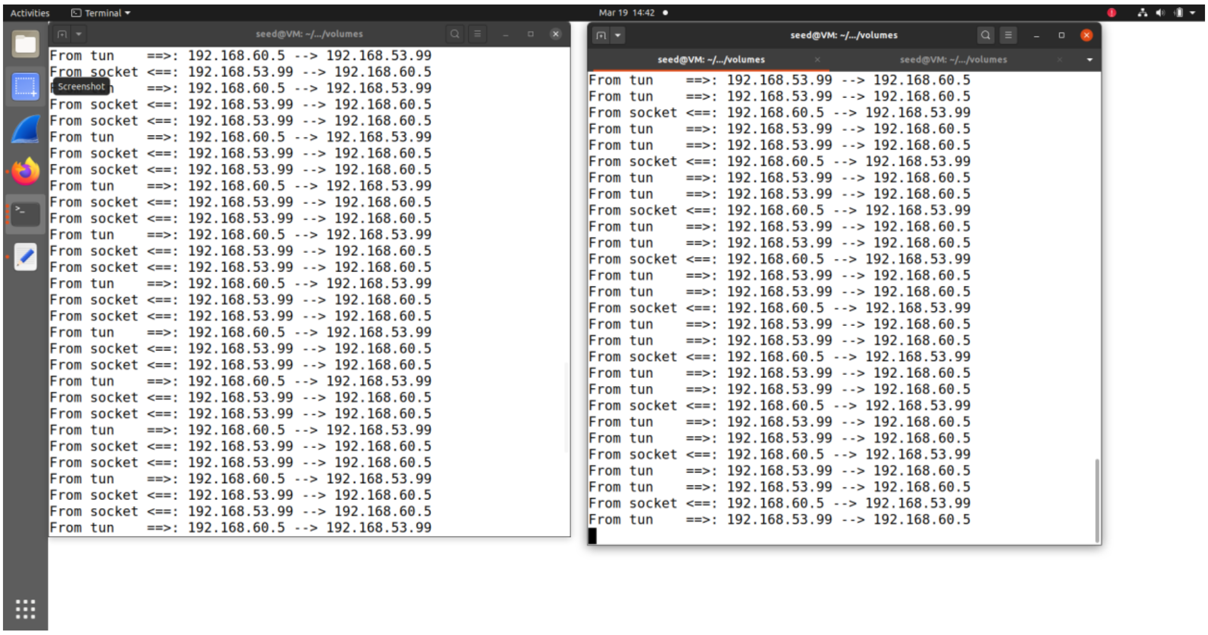
We proved this using tcpdump on server as well as the Host V side:



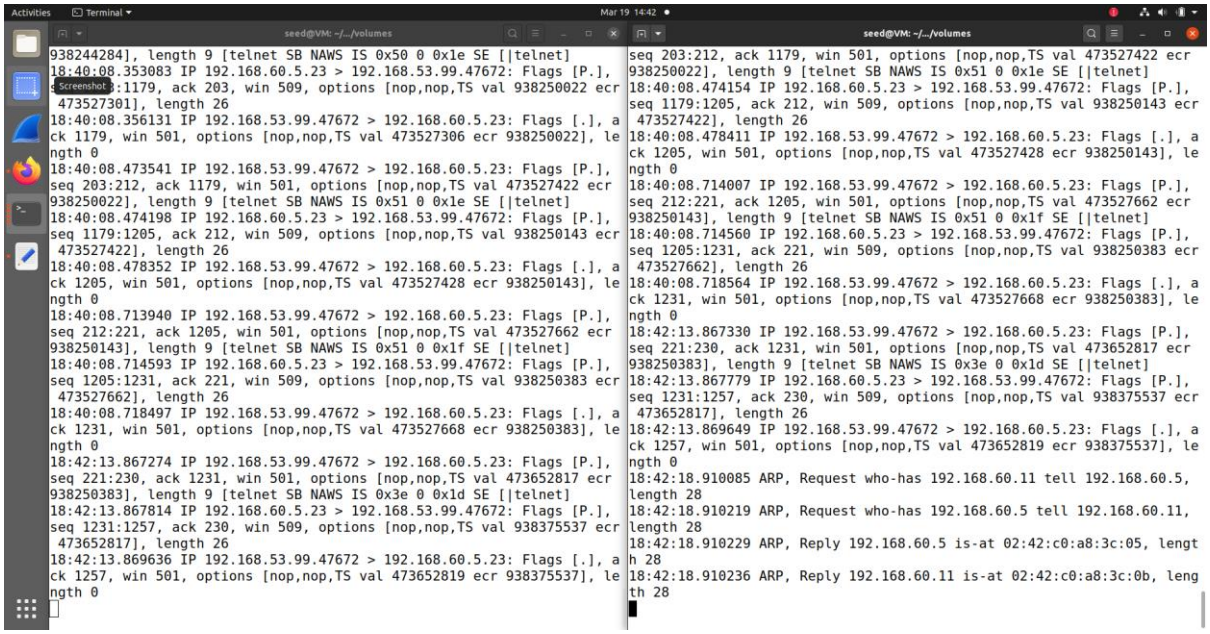
We make use of telnet for proof of the connection. We telnet from Host U to Host V to see the connection being established:



The connection was successful as a series of messages and replies as shown:

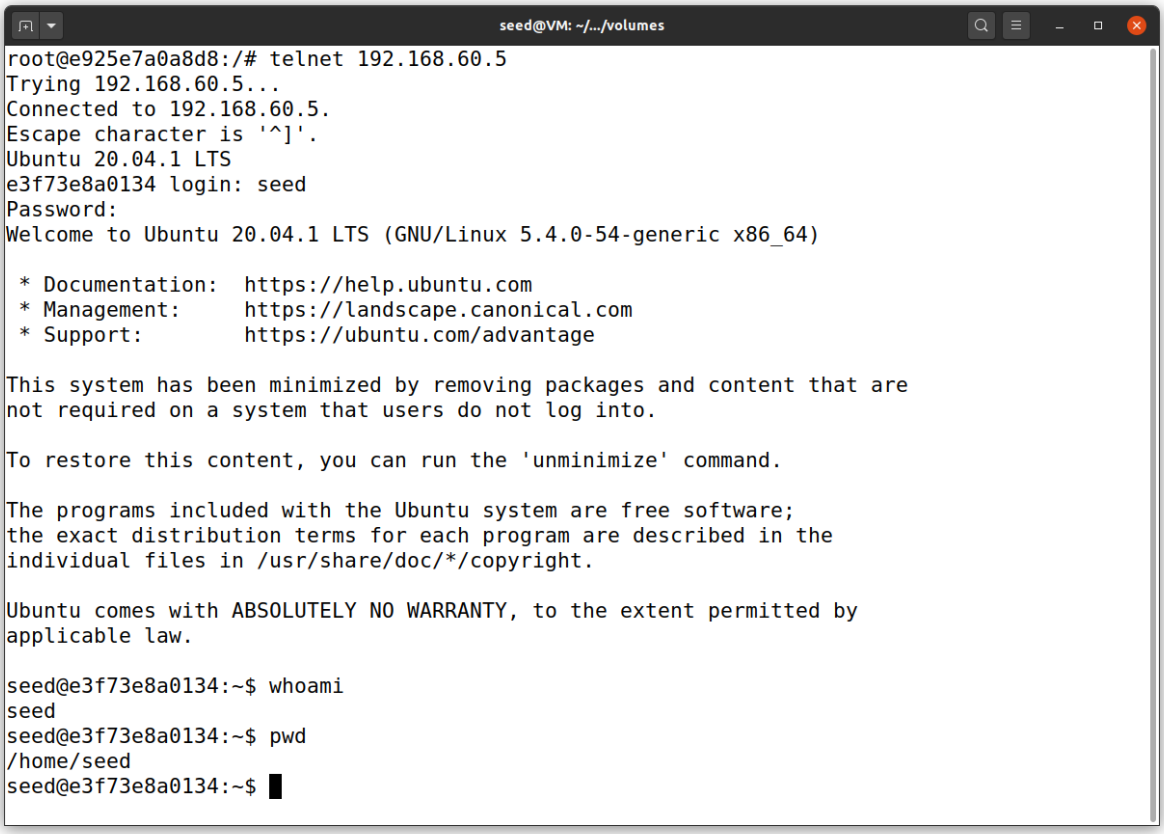


A tcpdump was done in order to show the successful transmission of data:

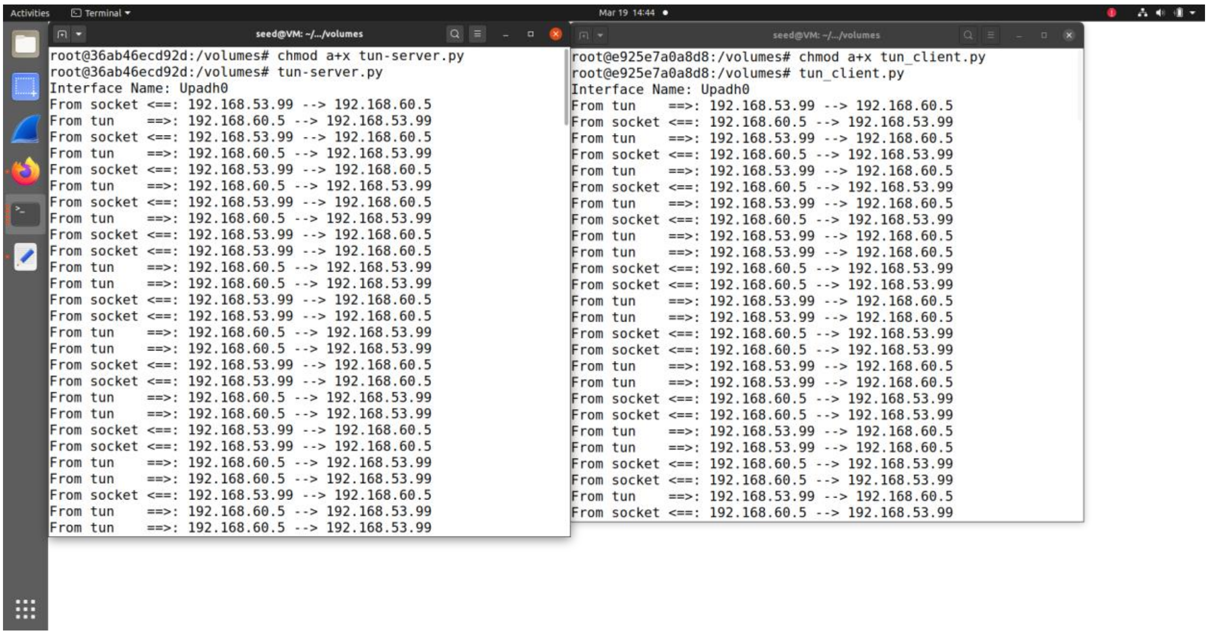


Task 6: Tunnel-Breaking Experiment

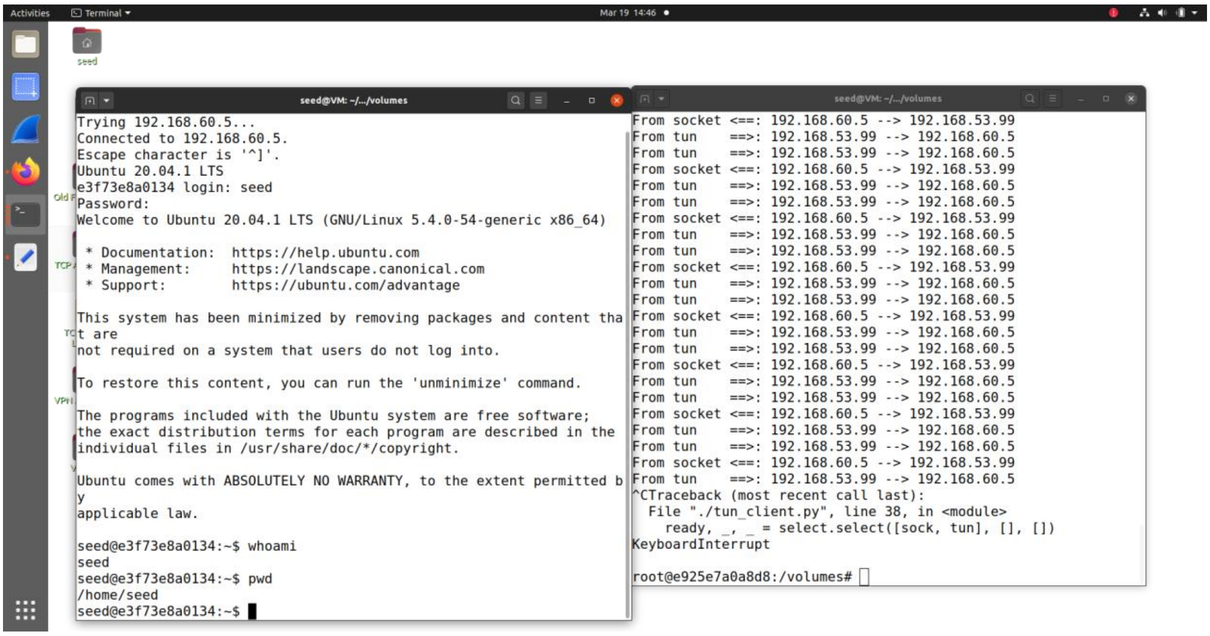
First, we set up a connection between Host U and Host V via a telnet:



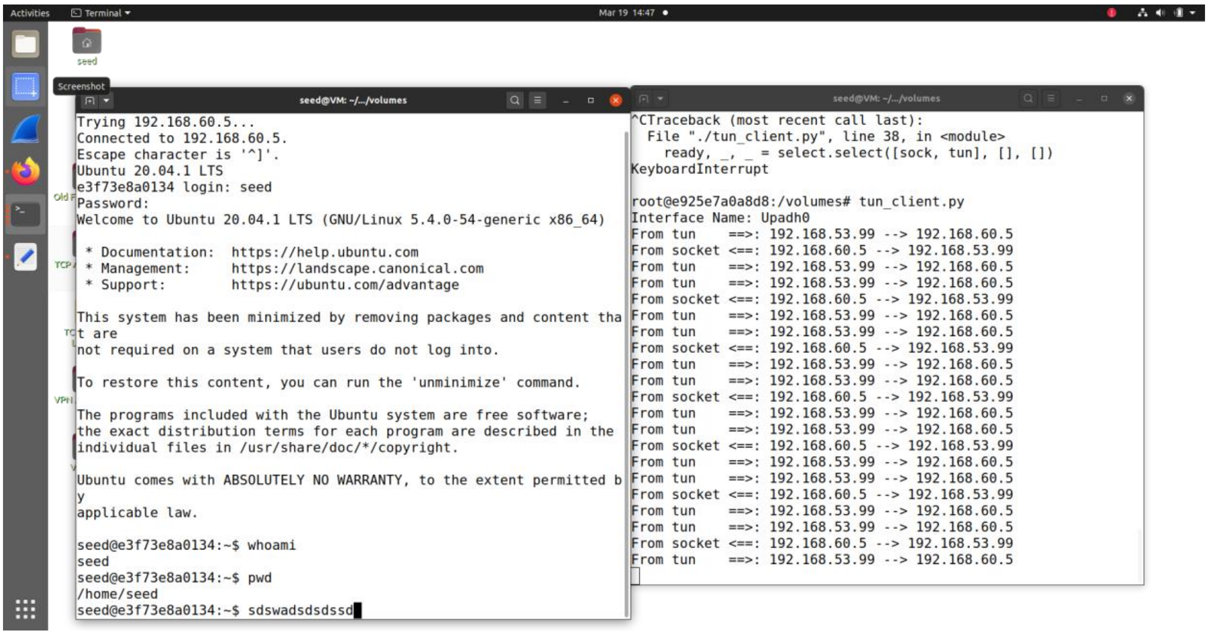
The connection was shown as follows:



Now we broke the connection on the Host U side. WE can observe that on the telnet, we were not able to type anything as the connection was broken.

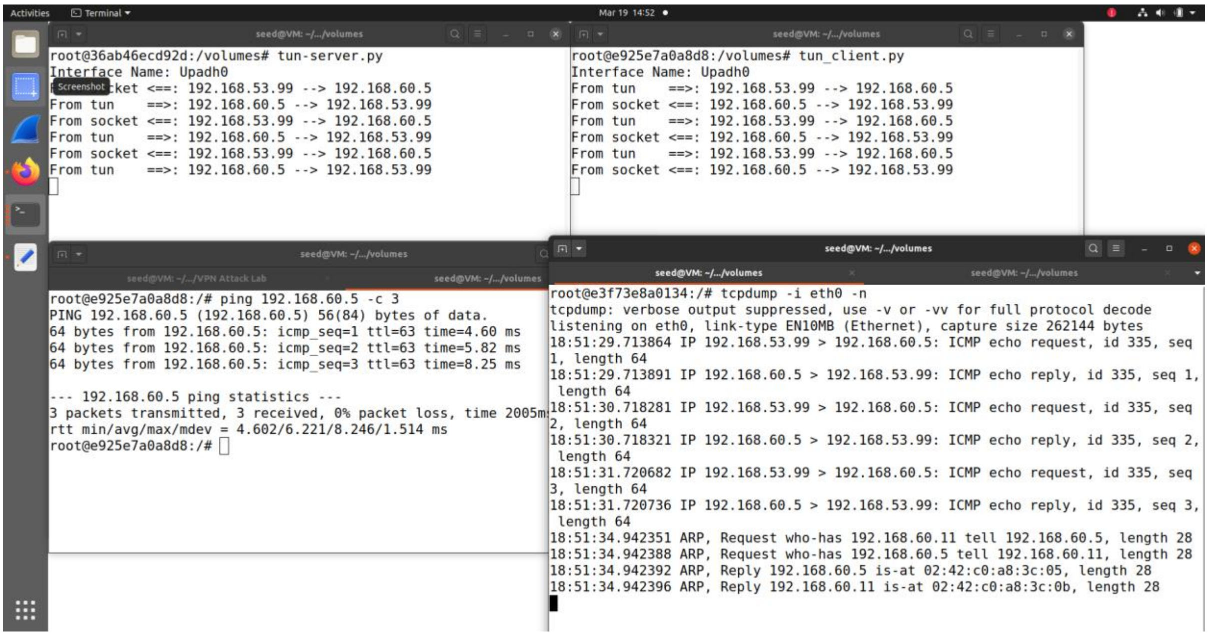


We rerun the code on Host U side and can see that the typed data was pasted as the connection got established again:



Task 7: Routing Experiment on Host V

We set up the connection on Host U and Host V side and make use of tcpdump to make sure Host U was getting data:



We can see that the default route is via 192.168.60.11.

```
root@e3f73e8a0134:/# ip route
default via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@e3f73e8a0134:/#
```

We delete this route and try to ping from Host V to Host U. We can observe that a specific route is needed to make the packets flow.

```
root@e3f73e8a0134:/# ip route
default via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@e3f73e8a0134:/# ip route del default
root@e3f73e8a0134:/# ip route
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@e3f73e8a0134:/#
```

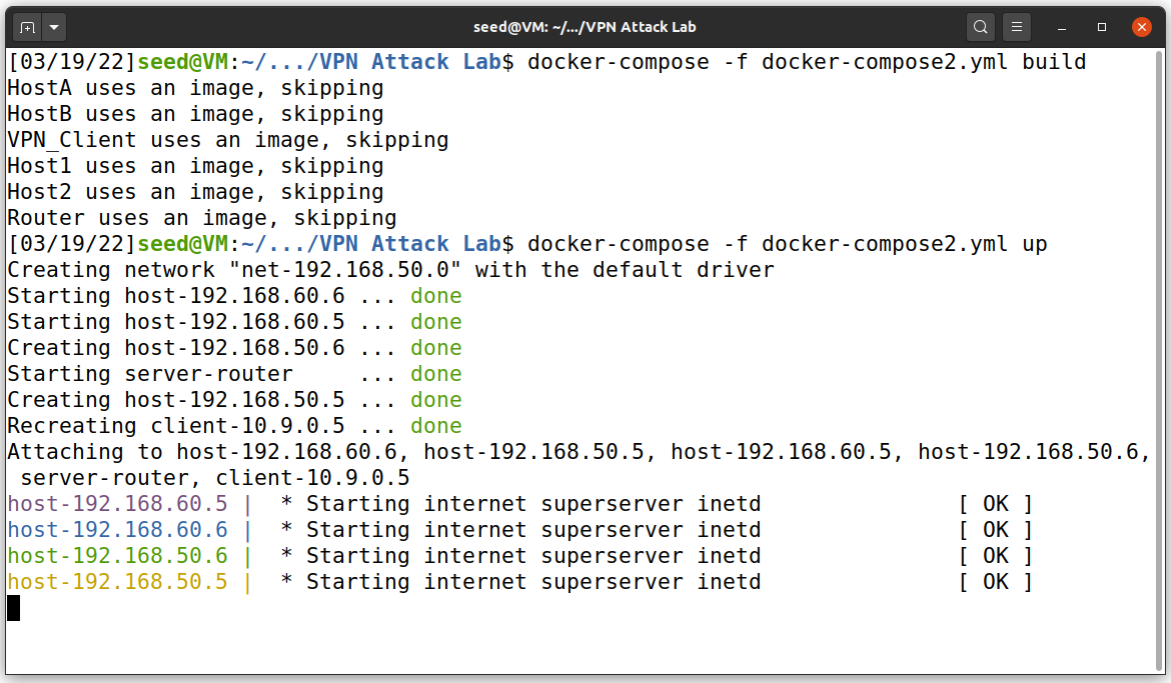
We run the command “ip route add” to add a specific route. We run the code again to check for the route as follows:

```
root@e3f73e8a0134:/# ip route add 192.168.53.0/24 via 192.168.60.11
root@e3f73e8a0134:/# ip route
192.168.53.0/24 via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@e3f73e8a0134:/#
```

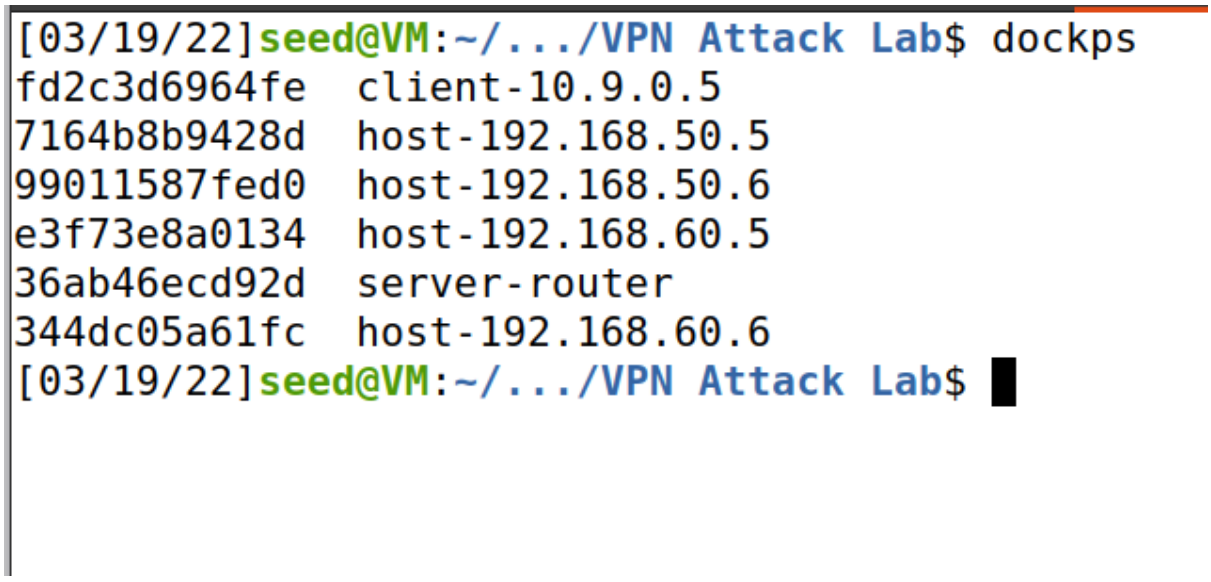
This will make sure that the ping works.

Task 8: VPN Between Private Networks

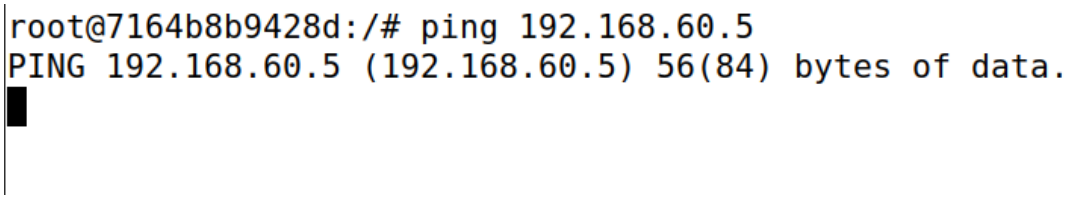
For the following task, we set up a new environment using the new docker file as follows:



Environment:

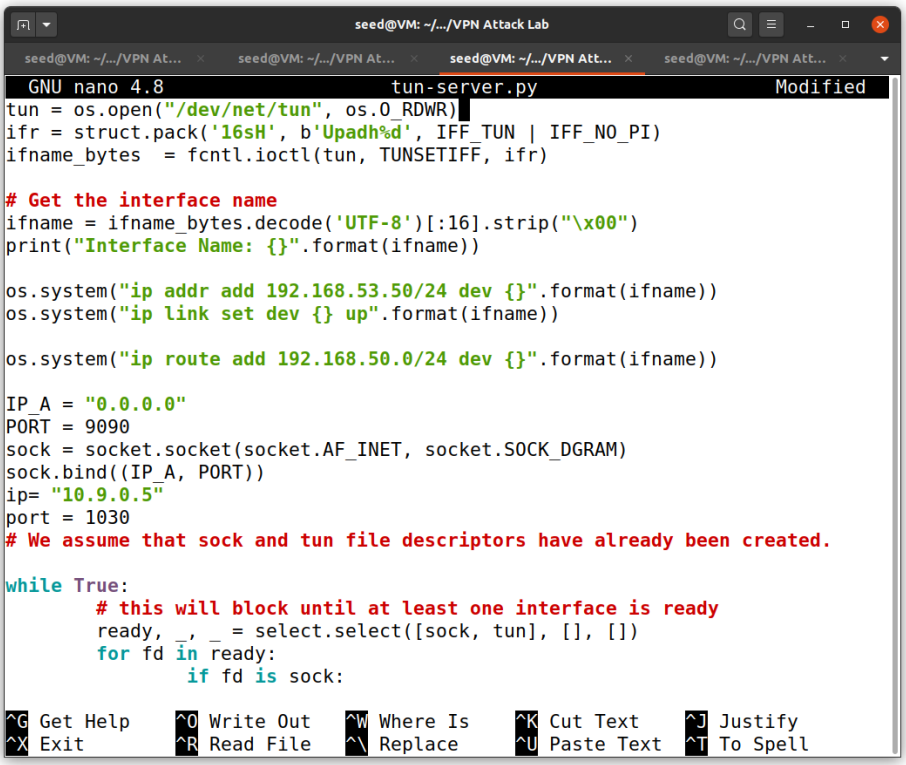


As the VPN is not setup, a ping to a host in a private network does not works:

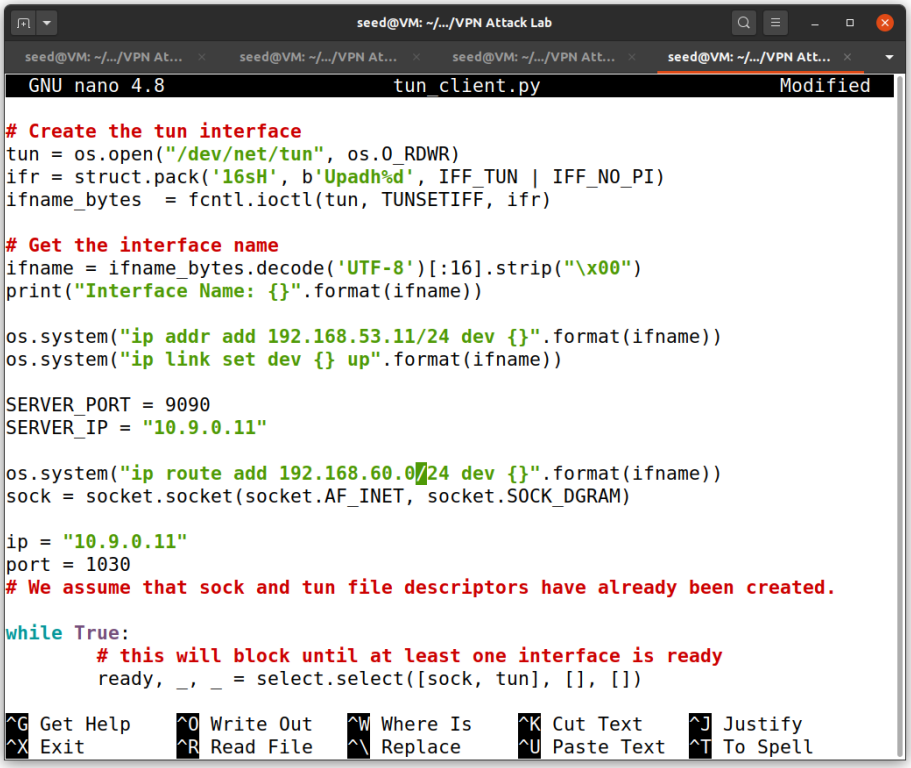


We modify the code as follows:

Tun-server.py code:

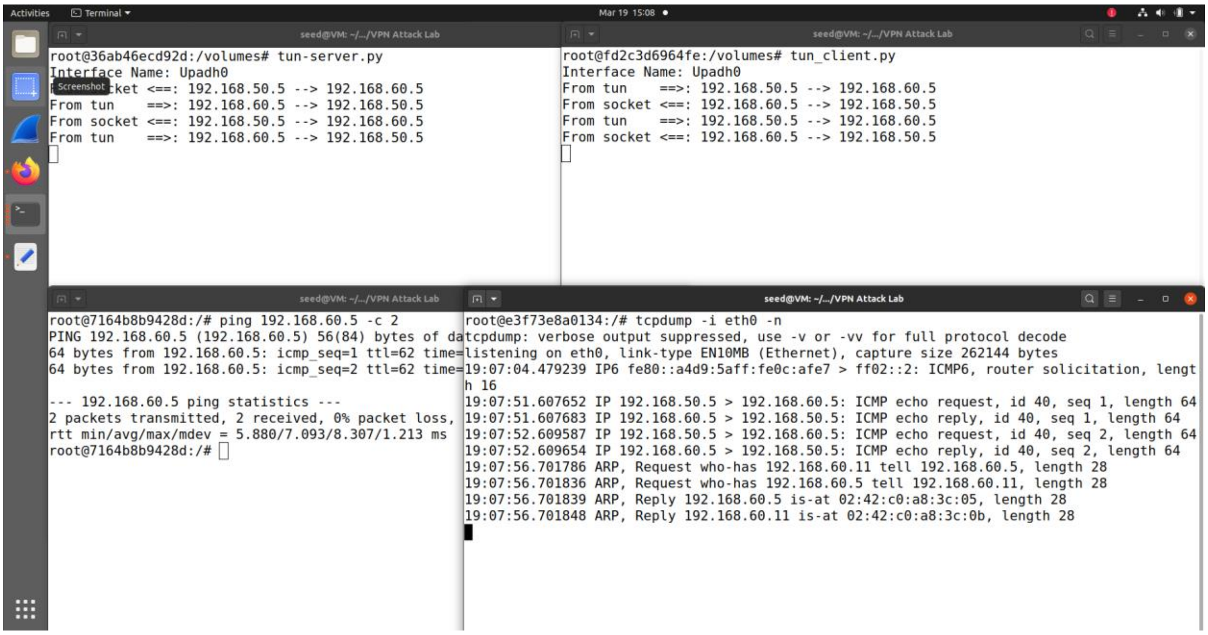


Tun_client.py code:

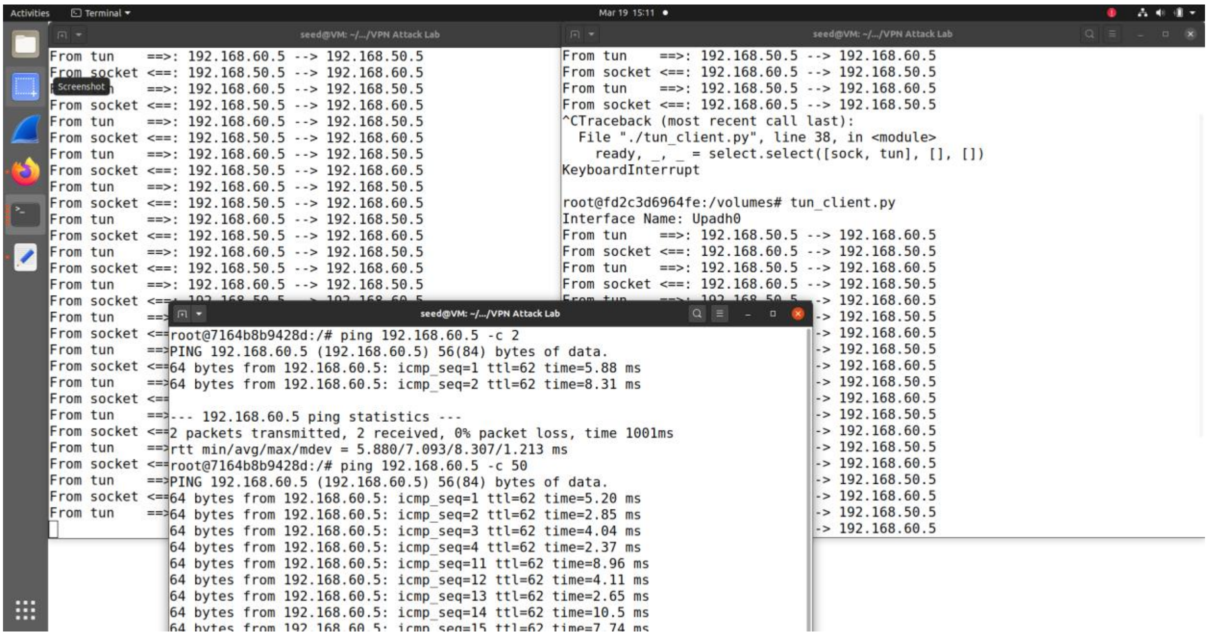


In the client code, we route the packet so that they can be routed through a private network as well such as 192.168.60.0/24.

When we run the code, we can see that while pinging from Host U to Host V on a private network, we receive an ICMP reply back:

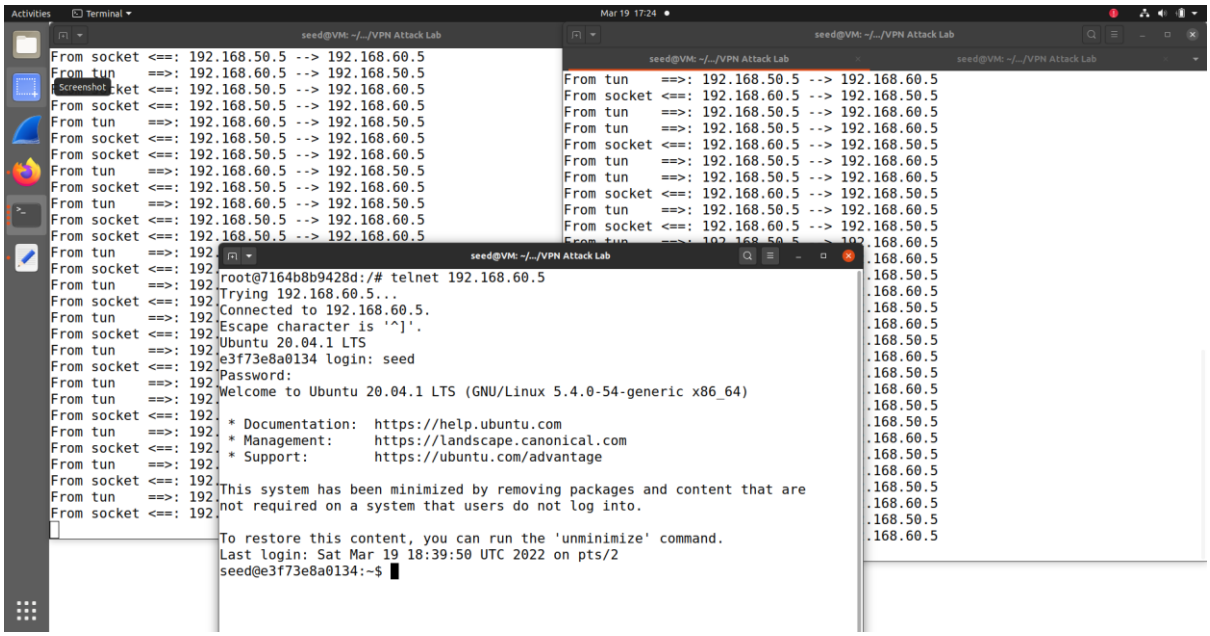


To test the connection if done properly, we break the connection as done in one of the previous tasks and connect again to see that connection successfully starts again:



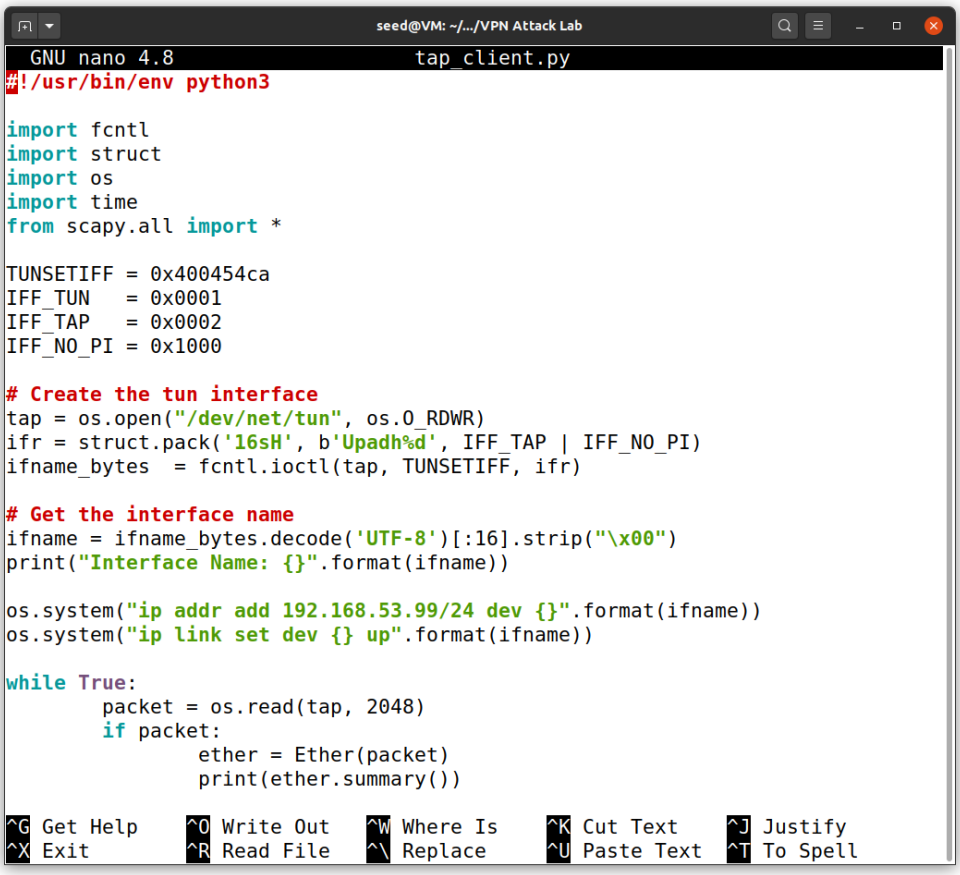
This can be seen using the ICMP sequence number gap during connection loss.

As our connection is ready, we telnet from Host V to Host U to see a connection being established which shows our VPN was configured properly:



Task 9: Experiment with the TAP Interface

Tap_client.py code:



We can see that the interface has been successfully created:

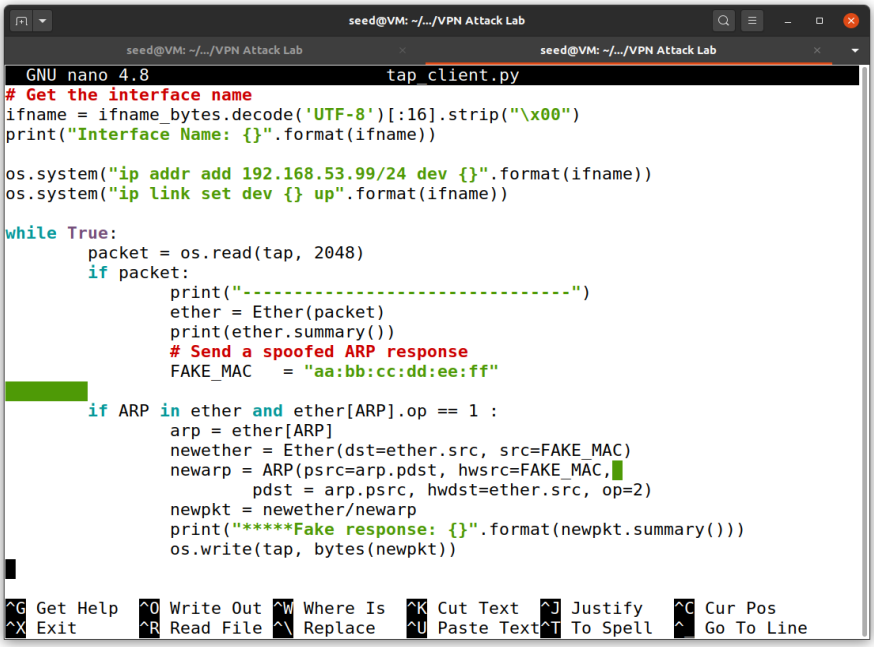
```
root@944ccc24f1af:/volumes# nano tap_client.py
root@944ccc24f1af:/volumes# chmod a+x tap_client.py
root@944ccc24f1af:/volumes# tap_client.py
Interface Name: Upadh0
```

We try to ping the IP address 192.168.53.1 network. The interface routes the ARP packets to tunnel interface which isn't sure which machine as that IP address hence sending the destination host unreachable message.

```
root@944ccc24f1af:/volumes# nano tap_client.py
root@944ccc24f1af:/volumes# chmod a+x tap_client.py
root@944ccc24f1af:/volumes# tap_client.py
Interface Name: Upadh0
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
```

```
root@944ccc24f1af:/# ping 192.168.53.1
PING 192.168.53.1 (192.168.53.1) 56(84) bytes of data.
From 192.168.53.99 icmp_seq=1 Destination Host Unreachable
From 192.168.53.99 icmp_seq=2 Destination Host Unreachable
From 192.168.53.99 icmp_seq=3 Destination Host Unreachable
From 192.168.53.99 icmp_seq=4 Destination Host Unreachable
From 192.168.53.99 icmp_seq=5 Destination Host Unreachable
From 192.168.53.99 icmp_seq=6 Destination Host Unreachable
^C
--- 192.168.53.1 ping statistics ---
7 packets transmitted, 0 received, +6 errors, 100% packet loss, time 6140ms
pipe 4
root@944ccc24f1af:/#
```

We modify the code to send spoofed ARP packets from a fake MAC address:



We run the code and try to arping to IP address 192.168.53.33 first. Here, we received a spoofed message replies from the MAC address sent for each request sent.

```
root@944ccc24f1af:/volumes# chmod a+x tap_client.py
root@944ccc24f1af:/volumes# tap_client.py
Interface Name: Upadh0
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
*****Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.33
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
*****Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.33
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
*****Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.33
```

seed@VM: ~/.../VPN Attack Lab

```
root@944ccc24f1af:/# arping -I Upadh0 192.168.53.33 -c 3
ARPING 192.168.53.33
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=0 time=3.225 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=1 time=2.042 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=2 time=6.135 msec

--- 192.168.53.33 statistics ---
3 packets transmitted, 3 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 2.042/3.801/6.135/1.720 ms
root@944ccc24f1af:/#
```

Now we try to arping to IP address 1.2.3.4. Here, we received a spoofed message replies from the MAC address sent for each request sent:

```
root@944ccc24f1af:/volumes# chmod a+x tap_client.py
root@944ccc24f1af:/volumes# tap_client.py
Interface Name: Upadh0
--------------------------------
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
*****Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1.2.3.4
--------------------------------
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
*****Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1.2.3.4
--------------------------------
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
*****Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1.2.3.4
```

seed@VM: ~/.../VPN Attack Lab

```
root@944ccc24f1af:/# arping -I Upadh0 1.2.3.4 -c 3
ARPING 1.2.3.4
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=0 time=3.838 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=1 time=5.966 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=2 time=5.753 msec

--- 1.2.3.4 statistics ---
3 packets transmitted, 3 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 3.838/5.186/5.966/0.957 ms
root@944ccc24f1af:/#
```