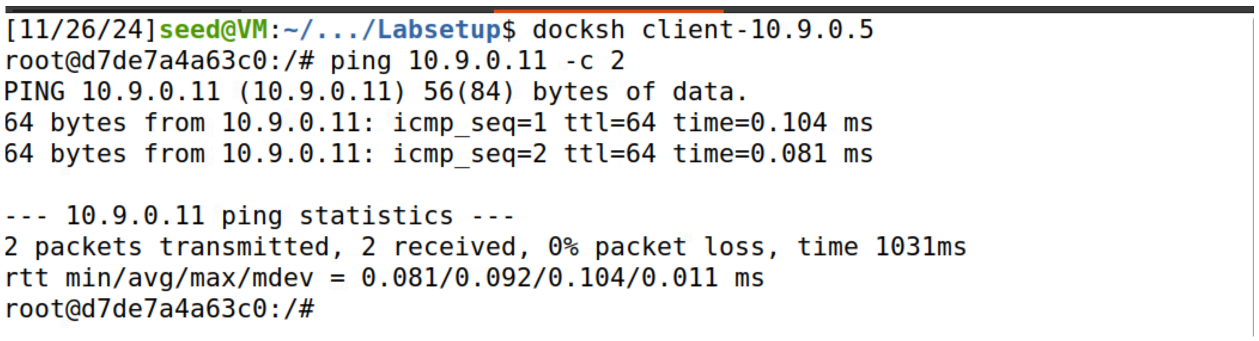
Lab 7

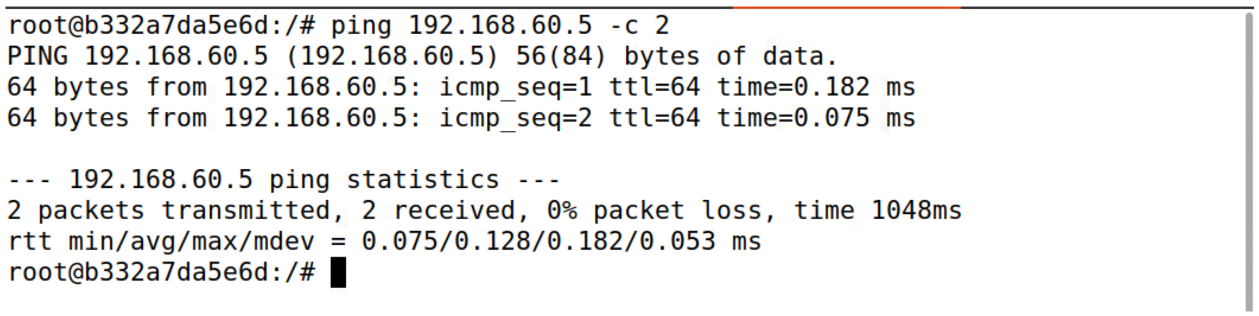
Task 1

- Host, you can communicate with the VPN server because we can see that the ping was successfully received by the server.



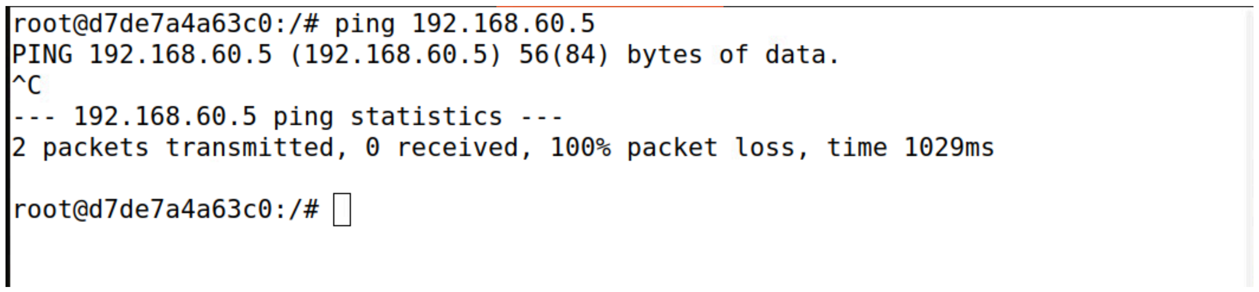
```
[11/26/24]seed@VM:~/.../Labsetup$ docksh client-10.9.0.5
root@d7de7a4a63c0:/# ping 10.9.0.11 -c 2
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
64 bytes from 10.9.0.11: icmp_seq=1 ttl=64 time=0.104 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.081 ms

--- 10.9.0.11 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1031ms
rtt min/avg/max/mdev = 0.081/0.092/0.104/0.011 ms
root@d7de7a4a63c0:/#
```

- VPN Server can communicate with Host V because 2 packets were transmitted and successfully received.



```
root@b332a7da5e6d:/# ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=64 time=0.182 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.075 ms

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1048ms
rtt min/avg/max/mdev = 0.075/0.128/0.182/0.053 ms
root@b332a7da5e6d:/#
```

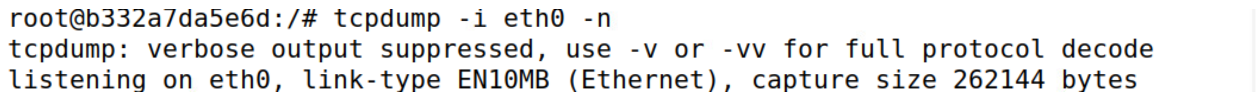
- Host U should not be able to communicate with Host V, as 2 packets are transmitted but none reach the destination. Host U and Host V cannot communicate.



```
root@d7de7a4a63c0:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
--- 192.168.60.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1029ms

root@d7de7a4a63c0:/#
```

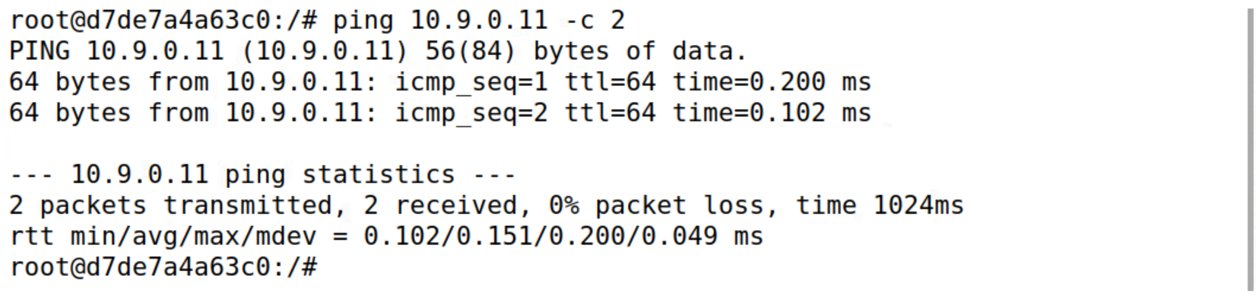
- Sniff the traffic on each of the network:

  1) Run the command *tcpdump -i eth0 -n* on the server to sniff the packets on the eth0 network.



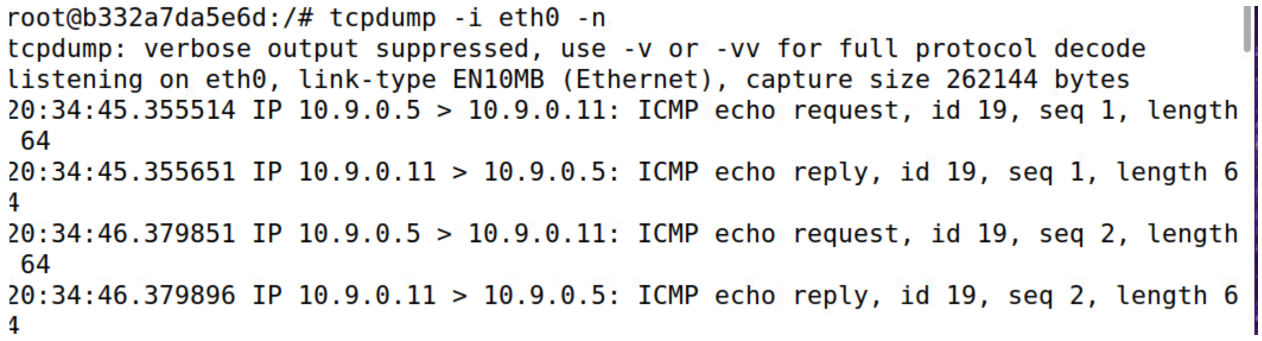
```
root@b332a7da5e6d:/# tcpdump -i eth0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
```

  2) We send packets from Host U to the server.



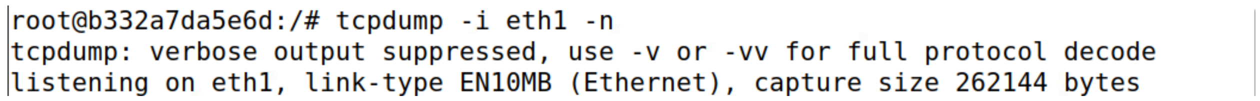
```
root@d7de7a4a63c0:/# ping 10.9.0.11 -c 2
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
64 bytes from 10.9.0.11: icmp_seq=1 ttl=64 time=0.200 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.102 ms

--- 10.9.0.11 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1024ms
rtt min/avg/max/mdev = 0.102/0.151/0.200/0.049 ms
root@d7de7a4a63c0:/#
```

  3) We send packets from Host U to the server.



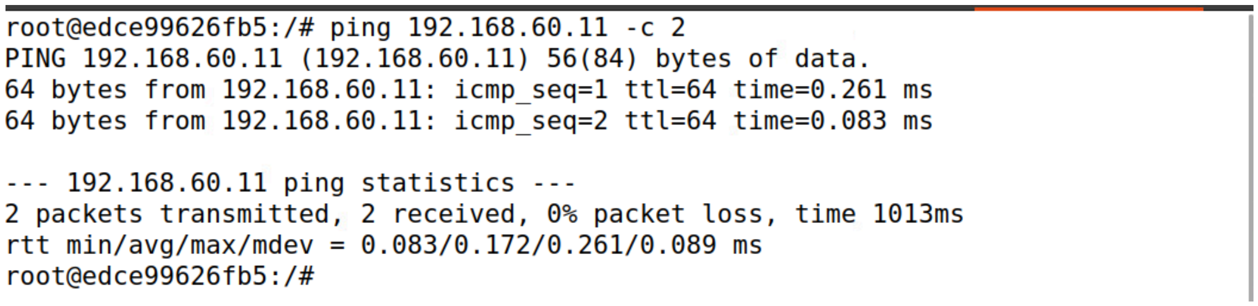
```
root@b332a7da5e6d:/# tcpdump -i eth0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
20:34:45.355514 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 19, seq 1, length
 64
20:34:45.355651 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 19, seq 1, length 6
4
20:34:46.379851 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 19, seq 2, length
 64
20:34:46.379896 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 19, seq 2, length 6
4
```

  1) Run the command *tcpdump -i eth1 -n* on the server to sniff the packets on the eth1 network.



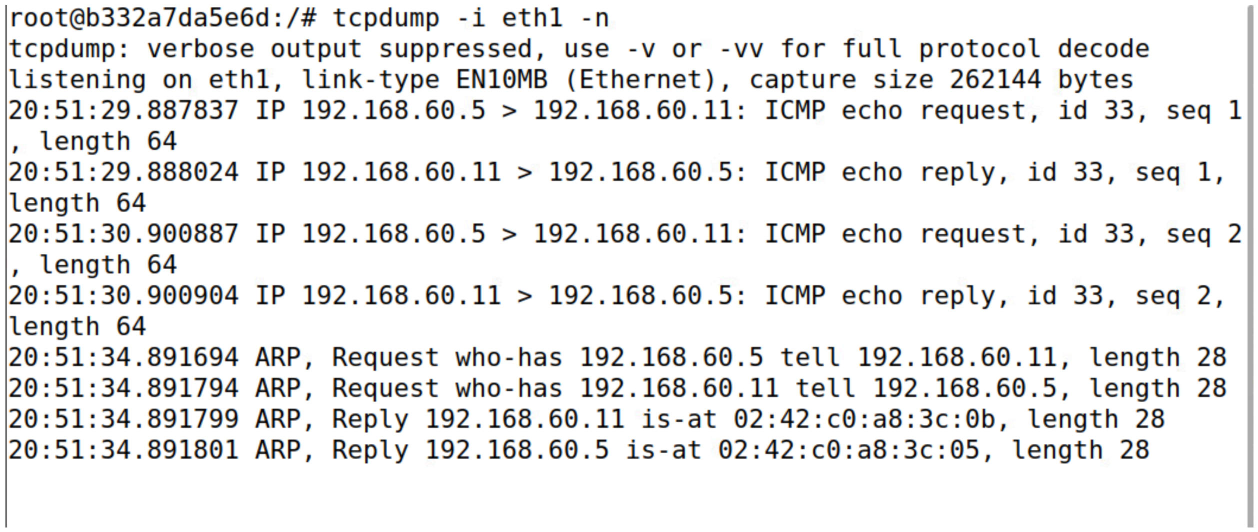
```
root@b332a7da5e6d:/# tcpdump -i eth1 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
```

2) We send packets from Host U to the server.



```
root@edce99626fb5:/# ping 192.168.60.11 -c 2
PING 192.168.60.11 (192.168.60.11) 56(84) bytes of data.
64 bytes from 192.168.60.11: icmp_seq=1 ttl=64 time=0.261 ms
64 bytes from 192.168.60.11: icmp_seq=2 ttl=64 time=0.083 ms

--- 192.168.60.11 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1013ms
rtt min/avg/max/mdev = 0.083/0.172/0.261/0.089 ms
root@edce99626fb5:/#
```

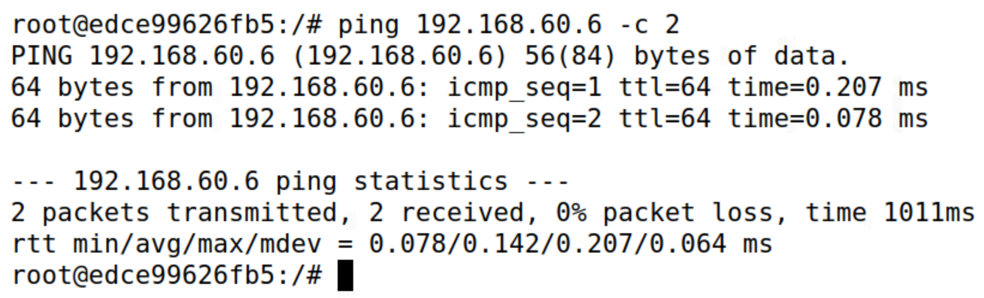
3) We notice that the packets were successfully sniffed by tcpdump on the eth1 network.



```
root@b332a7da5e6d:/# tcpdump -i eth1 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
20:51:29.887837 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 33, seq 1
, length 64
20:51:29.888024 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 33, seq 1,
length 64
20:51:30.900887 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 33, seq 2
, length 64
20:51:30.900904 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 33, seq 2,
length 64
20:51:34.891694 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
20:51:34.891794 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
20:51:34.891799 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
20:51:34.891801 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
```

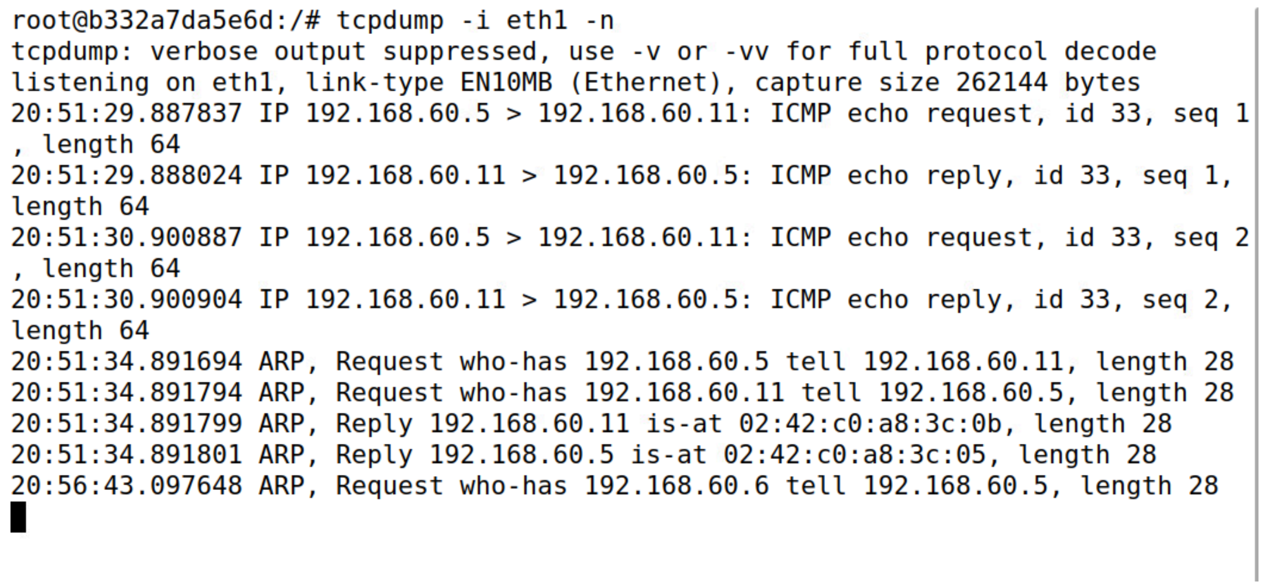
Observation:
If the two clients 192.168.60.5 and 192.168.60.6 communicate, the server cannot sniff the packets.

1) Send the ping.



```
root@edce99626fb5:/# ping 192.168.60.6 -c 2
PING 192.168.60.6 (192.168.60.6) 56(84) bytes of data.
64 bytes from 192.168.60.6: icmp_seq=1 ttl=64 time=0.207 ms
64 bytes from 192.168.60.6: icmp_seq=2 ttl=64 time=0.078 ms

--- 192.168.60.6 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1011ms
rtt min/avg/max/mdev = 0.078/0.142/0.207/0.064 ms
root@edce99626fb5:/# 
```

2) No new packets are sniffed.

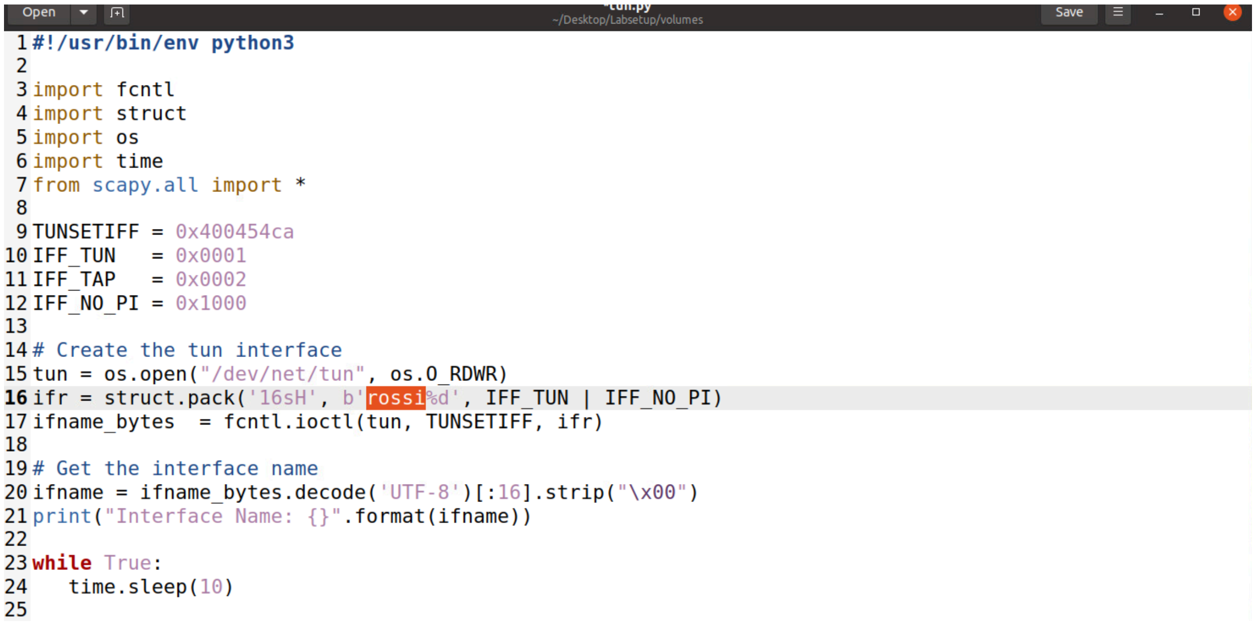


```
root@b332a7da5e6d:/# tcpdump -i eth1 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
20:51:29.887837 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 33, seq 1
, length 64
20:51:29.888024 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 33, seq 1,
length 64
20:51:30.900887 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 33, seq 2
, length 64
20:51:30.900904 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 33, seq 2,
length 64
20:51:34.891694 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
20:51:34.891794 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
20:51:34.891799 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
20:51:34.891801 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
20:56:43.097648 ARP, Request who-has 192.168.60.6 tell 192.168.60.5, length 28
```

Task 2 :

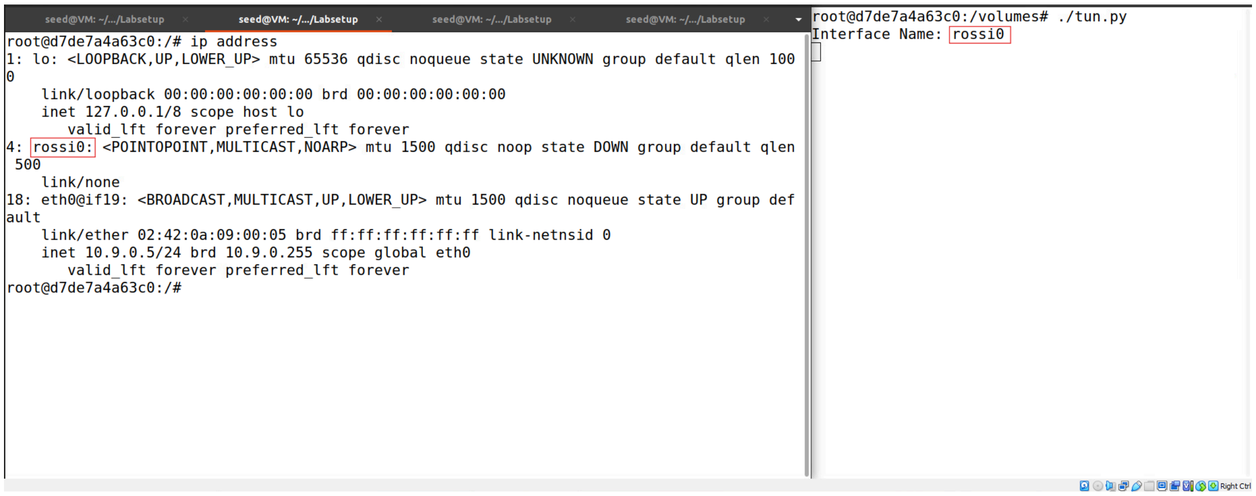
Task 2.a:

We modify the program *tun.py* at line 16 and replace "tun" with my last name.



```
 1 #!/usr/bin/env python3
 2 
 3 import fcntl
 4 import struct
 5 import os
 6 import time
 7 from scapy.all import *
 8 
 9 TUNSETIFF = 0x400454ca
10 IFF_TUN   = 0x0001
11 IFF_TAP   = 0x0002
12 IFF_NO_PI = 0x1000
13 
14 # Create the tun interface
15 tun = os.open("/dev/net/tun", os.O_RDWR)
16 ifr = struct.pack('16sH', b'rossi%d', IFF_TUN | IFF_NO_PI)
17 ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)
18 
19 # Get the interface name
20 ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
21 print("Interface Name: {}".format(ifname))
22 
23 while True:
24    time.sleep(10)
25 
```

We execute *tun.py* on a second terminal to use the *ip address* command. We notice that "rossi0" appears.



```
root@d7de7a4a63c0:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 100
0
    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
4: rossi0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen
 500
    link/none
18: eth0@if19: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group def
ault
    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
root@d7de7a4a63c0:/#
```

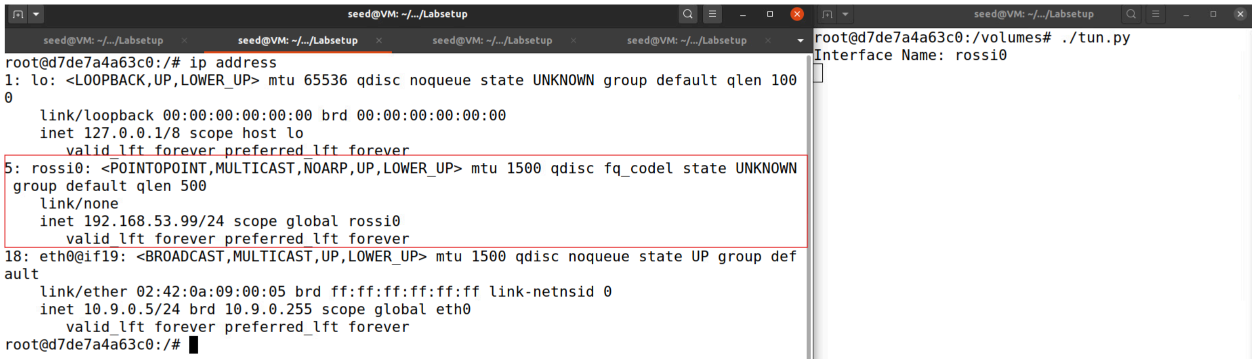
```
root@d7de7a4a63c0:/volumes# ./tun.py
Interface Name: rossi0
```

Task 2.b:

We modify the *tun.py* code by adding the two configuration lines:

```
# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'rossi%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))
```

After execution, we notice the difference between the first and second executions. The two new lines indicate that the configuration has the IP address 192.168.53.99 and it is activated.

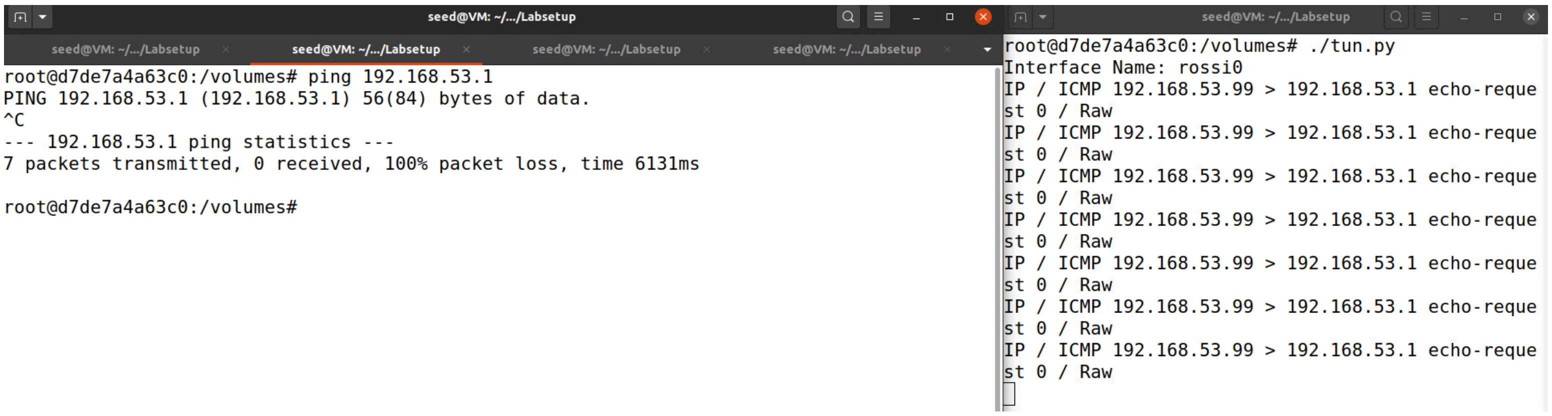


```
root@d7de7a4a63c0:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 100
0
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
5: rossi0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN
 group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global rossi0
       valid_lft forever preferred_lft forever
18: eth0@if19: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group def
ault
    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
root@d7de7a4a63c0:/#
```

```
root@d7de7a4a63c0:/volumes# ./tun.py
Interface Name: rossi0
```

Task 2.c:

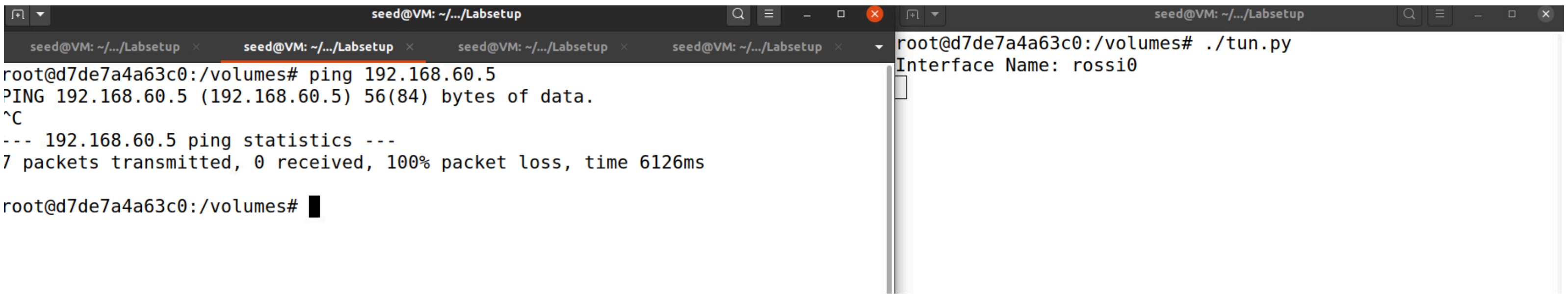
- We decided to send packets to the host 192.168.53.1. We can send the packets because we are on the same network. However, since the configuration has not been set up to receive, the packets cannot yet be received by 192.168.53.1.



```
root@d7de7a4a63c0:/volumes# 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 6131ms

root@d7de7a4a63c0:/volumes#
```

```
root@d7de7a4a63c0:/volumes# ./tun.py
Interface Name: rossi0
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-reque
st 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-reque
st 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-reque
st 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-reque
st 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-reque
st 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-reque
st 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-reque
st 0 / Raw
```

- Now, we send a ping to the network 192.168.60.5 on the internal network. This time, *tun.py* didn't print anything, nothing was sent, and as before, nothing was received. Indeed, our Host U tried to send packets to a network other than the configured one, so it is not accessible.

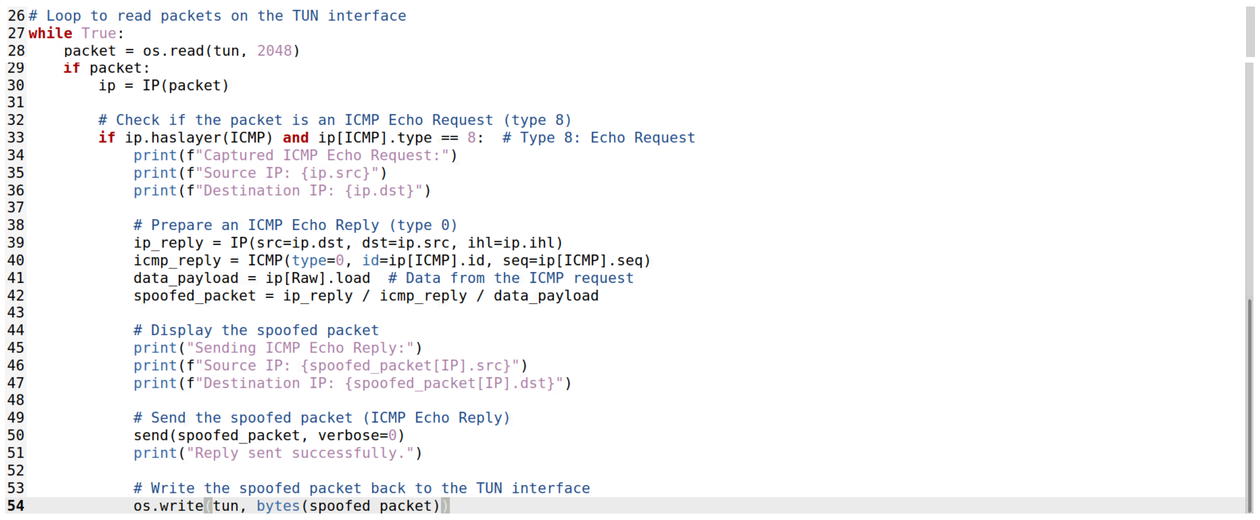


```
root@d7de7a4a63c0:/volumes# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
--- 192.168.60.5 ping statistics ---
7 packets transmitted, 0 received, 100% packet loss, time 6126ms

root@d7de7a4a63c0:/volumes# 
```

```
root@d7de7a4a63c0:/volumes# ./tun.py
Interface Name:  rossi0
```

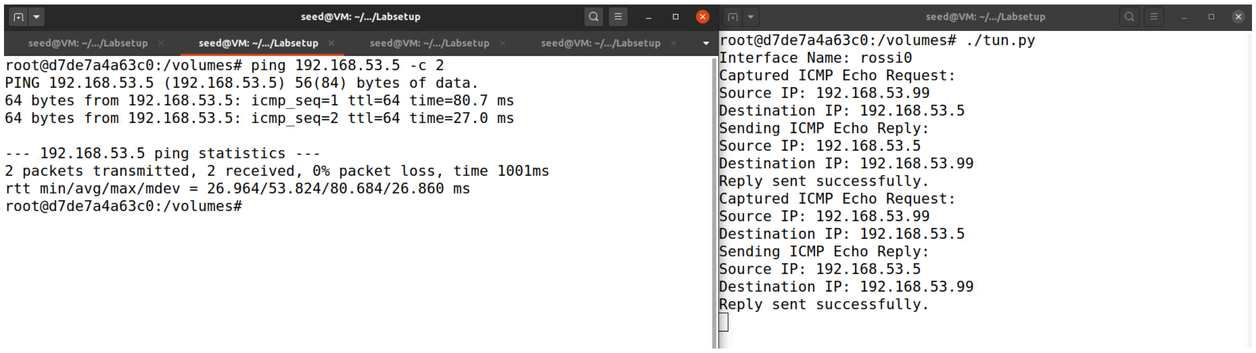
Task 2.d:

This script captures ICMP Echo requests (ping) sent to a TUN interface, then creates an ICMP Echo reply by swapping the source and destination addresses. It first configures the TUN interface with an IP address, then enters an infinite loop where it reads incoming packets. When an ICMP Echo request is detected, the script generates a reply with the same ICMP ID and sequence number, and sends this reply through the network using Scapy. Finally, it writes the reply to the TUN interface so it can be sent back through the VPN tunnel, thus simulating the communication.



```python
# Loop to read packets on the TUN interface
while True:
    packet = os.read(tun, 2048)
    if packet:
        ip = IP(packet)

        # Check if the packet is an ICMP Echo Request (type 8)
        if ip.haslayer(ICMP) and ip[ICMP].type == 8:  # Type 8: Echo Request
            print(f"Captured ICMP Echo Request:")
            print(f"Source IP: {ip.src}")
            print(f"Destination IP: {ip.dst}")

            # Prepare an ICMP Echo Reply (type 0)
            ip_reply = IP(src=ip.dst, dst=ip.src, ihl=ip.ihl)
            icmp_reply = ICMP(type=0, id=ip[ICMP].id, seq=ip[ICMP].seq)
            data_payload = ip[Raw].load  # Data from the ICMP request
            spoofed_packet = ip_reply / icmp_reply / data_payload

            # Display the spoofed packet
            print("Sending ICMP Echo Reply:")
            print(f"Source IP: {spoofed_packet[IP].src}")
            print(f"Destination IP: {spoofed_packet[IP].dst}")

            # Send the spoofed packet (ICMP Echo Reply)
            send(spoofed_packet, verbose=0)
            print("Reply sent successfully.")

            # Write the spoofed packet back to the TUN interface
            os.write(tun, bytes(spoofed_packet))
```

We can see that 2 pings are sent, and 2 are received. The script captures a ping from the source 192.168.53.99 to 192.168.53.5 and sends the response from 192.168.53.5 to 192.168.53.99.

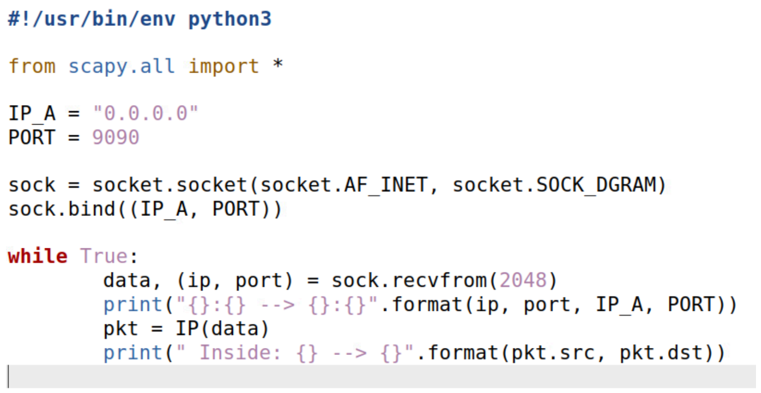


```
root@d7de7a4a63c0:/volumes# ping 192.168.53.5 -c 2
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
64 bytes from 192.168.53.5: icmp_seq=1 ttl=64 time=80.7 ms
64 bytes from 192.168.53.5: icmp_seq=2 ttl=64 time=27.0 ms

--- 192.168.53.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 26.964/53.824/80.684/26.860 ms
root@d7de7a4a63c0:/volumes#
```

```
root@d7de7a4a63c0:/volumes# ./tun.py
Interface Name: rossi0
Captured ICMP Echo Request:
Source IP: 192.168.53.99
Destination IP: 192.168.53.5
Sending ICMP Echo Reply:
Source IP: 192.168.53.5
Destination IP: 192.168.53.99
Reply sent successfully.
Captured ICMP Echo Request:
Source IP: 192.168.53.99
Destination IP: 192.168.53.5
Sending ICMP Echo Reply:
Source IP: 192.168.53.5
Destination IP: 192.168.53.99
Reply sent successfully.
```

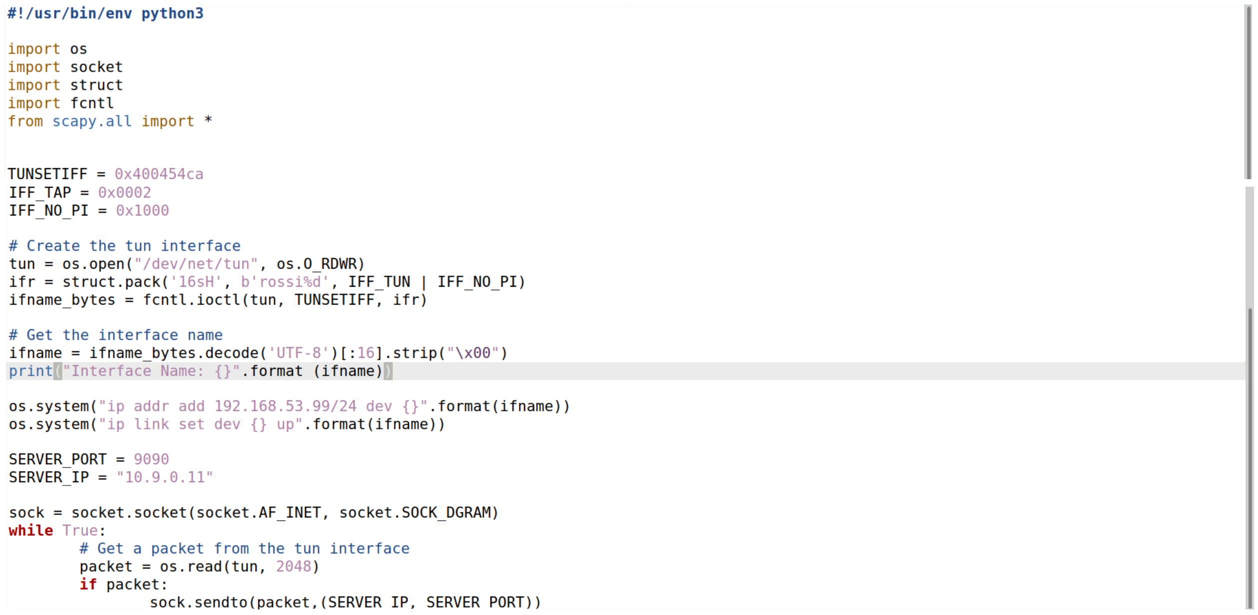
Task 3:

The code server (given in the PDF):



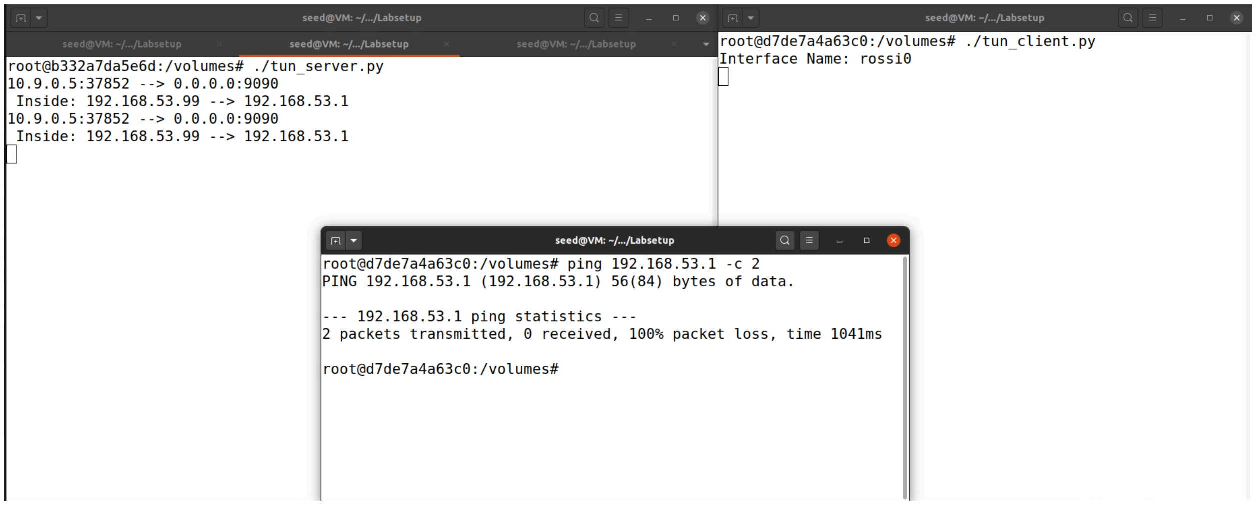
```
#!/usr/bin/env python3

from scapy.all import *

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))
```

The client code:



```
#!/usr/bin/env python3

import os
import socket
import struct
import fcntl
from scapy.all import *


TUNSETIFF = 0x400454ca
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'rossi%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"

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))
```

We run *server.py* and *client.py*, then send pings to the address 192.168.53.1.



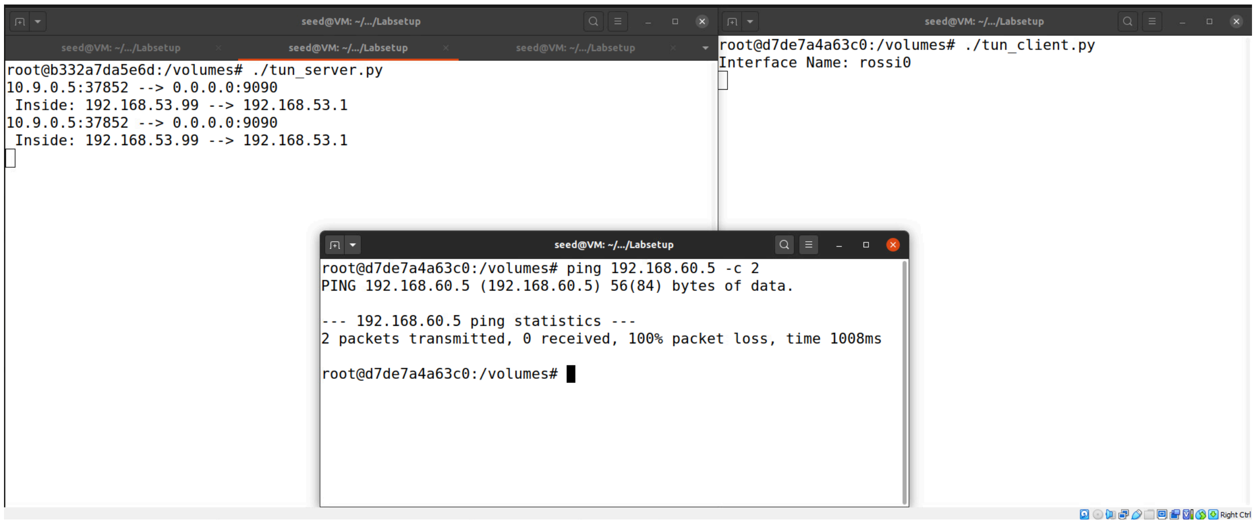
```
root@b332a7da5e6d:/volumes# ./tun_server.py
10.9.0.5:37852 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.1
10.9.0.5:37852 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.1
```

```
root@d7de7a4a63c0:/volumes# ./tun_client.py
Interface Name: rossi0
```

```
root@d7de7a4a63c0:/volumes# ping 192.168.53.1 -c 2
PING 192.168.53.1 (192.168.53.1) 56(84) bytes of data.

--- 192.168.53.1 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1041ms

root@d7de7a4a63c0:/volumes#
```

The server displays information about the source and destination ports, along with the source and destination IP addresses of the received packets. Since the host has a TUN interface with a configured routing route, when a ping is sent to an IP address in this network, the packet passes through the TUN interface and is forwarded via the socket for processing.

Now we send pings to Host V.



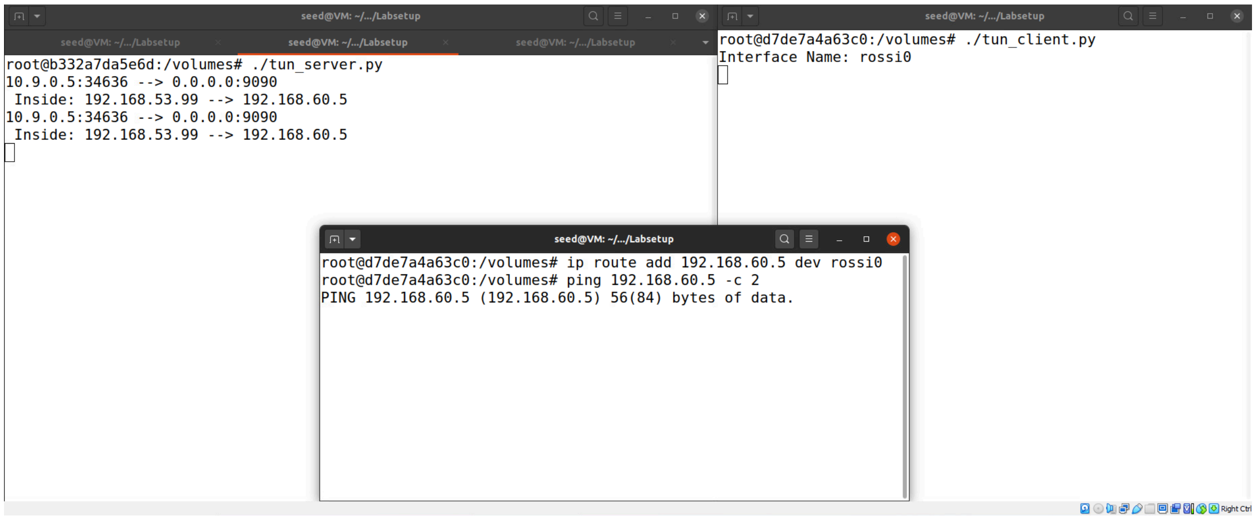
```
root@b332a7da5e6d:/volumes# ./tun_server.py
10.9.0.5:37852 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.1
10.9.0.5:37852 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.1
```

```
root@d7de7a4a63c0:/volumes# ./tun_client.py
Interface Name: rossi0
```

```
root@d7de7a4a63c0:/volumes# ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1008ms

root@d7de7a4a63c0:/volumes#
```

We notice that nothing is displayed on the server side. Indeed, we don't yet have the configuration to route the packets to a private network.

Thanks to the command, packets sent via a ping request to an IP address in the 192.168.60.0/24 network, ICMP packets are received by tun server.py via the tunnel.



```
root@b332a7da5e6d:/volumes# ./tun_server.py
10.9.0.5:34636 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:34636 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
```

```
root@d7de7a4a63c0:/volumes# ./tun_client.py
Interface Name: rossi0
```

```
root@d7de7a4a63c0:/volumes# ip route add 192.168.60.5 dev rossi0
root@d7de7a4a63c0:/volumes# ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
```

Task 4 :

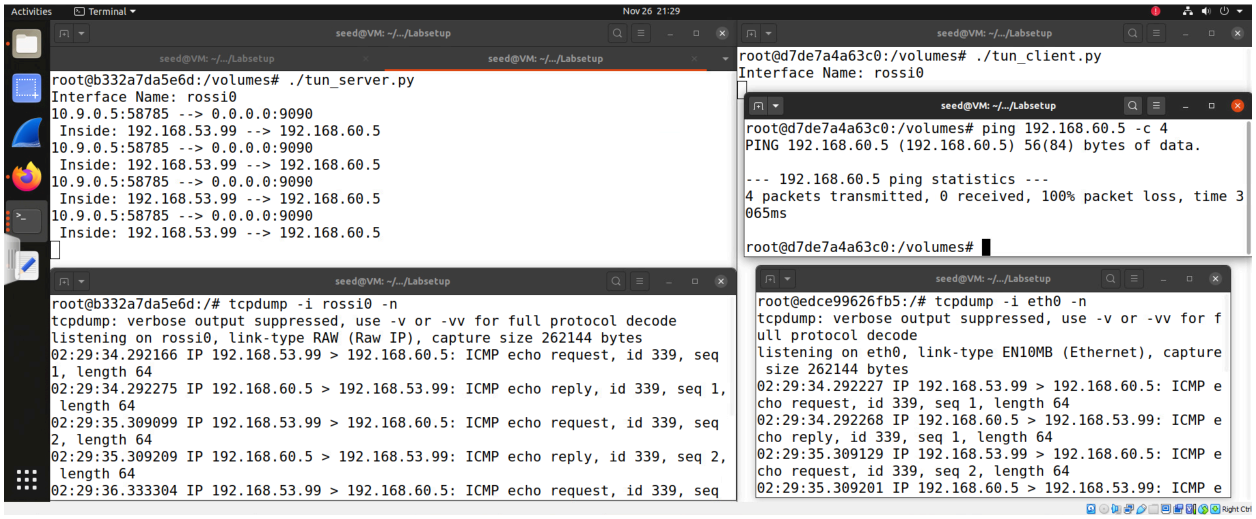
We modified the server.py code to create and configure the interface. We retrieved the data from the socket interface, then processed the received data as an IP packet. Finally, we wrote the packet to the TUN interface.

```
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'rossi%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
```

For the client code, we only added the line that ensures access to the private network via the tunnel, as seen in Task 3.

```python
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)) # We add the private network

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))
```

We then launched the server and client, as well as a ping from Host U to 192.168.60.5 (Host V). We opened two tcpdump windows: one on the server and on the rossi0 network, and another on Host V on the eth0 network.



```
root@b332a7da5e6d:/volumes# ./tun_server.py
Interface Name: rossi0
10.9.0.5:58785 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:58785 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:58785 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:58785 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
```

```
root@b332a7da5e6d:/# tcpdump -i rossi0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on rossi0, link-type RAW (Raw IP), capture size 262144 bytes
02:29:34.292166 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 339, seq
1, length 64
02:29:34.292275 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 339, seq 1,
 length 64
02:29:35.309099 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 339, seq
2, length 64
02:29:35.309209 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 339, seq 2,
 length 64
02:29:36.333304 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 339, seq
```

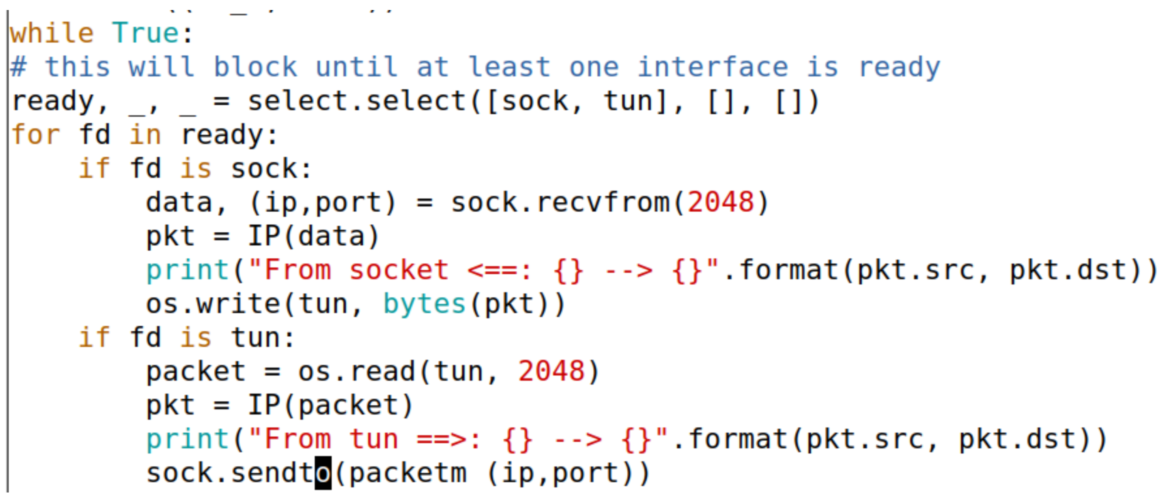
```
root@d7de7a4a63c0:/volumes# ./tun_client.py
Interface Name: rossi0
```

```
root@d7de7a4a63c0:/volumes# ping 192.168.60.5 -c 4
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3
065ms

root@d7de7a4a63c0:/volumes#
```

```
root@edce99626fb5:/# tcpdump -i eth0 -n
tcpdump: verbose output suppressed, use -v or -vv for f
ull protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture
 size 262144 bytes
02:29:34.292227 IP 192.168.53.99 > 192.168.60.5: ICMP e
cho request, id 339, seq 1, length 64
02:29:34.292268 IP 192.168.60.5 > 192.168.53.99: ICMP e
cho reply, id 339, seq 1, length 64
02:29:35.309129 IP 192.168.53.99 > 192.168.60.5: ICMP e
cho request, id 339, seq 2, length 64
02:29:35.309201 IP 192.168.60.5 > 192.168.53.99: ICMP e
```

We notice, thanks to tcpdump, that the ICMP packets successfully reached Host V from Host U. Host V receives and sends a reply. However, we observe that the reply doesn't reach us yet because the configuration has not been set up for that.

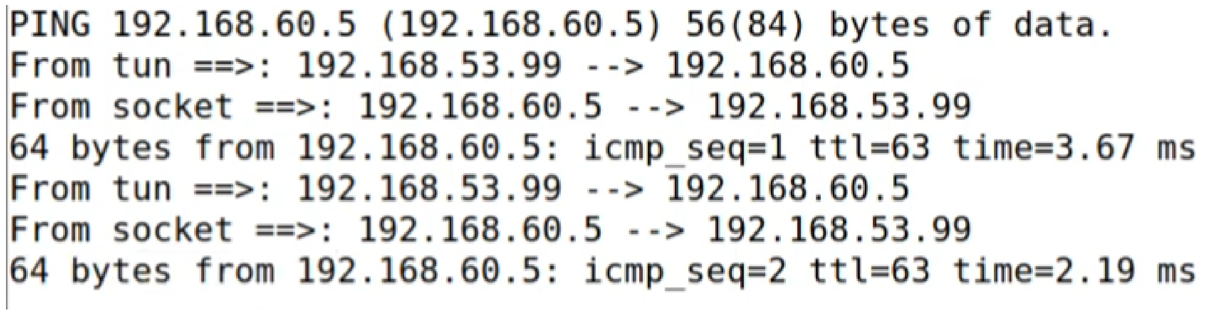
Task 5:



```
while True:
# this will block until at least one interface is ready
ready, _, _ = select.select([sock, tun], [], [])
for fd in ready:
    if fd is sock:
        data, (ip,port) = sock.recvfrom(2048)
        pkt = IP(data)
        print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
        os.write(tun, bytes(pkt))
    if fd is tun:
        packet = os.read(tun, 2048)
        pkt = IP(packet)
        print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
        sock.sendto(packetm (ip,port))
```

Adding this block of code to both server and client files so they are able to handle traffic in both directions. This is achieved through the select command, which will lock machines until it finds data.

Pinging Machine V from Machine U results is in this:

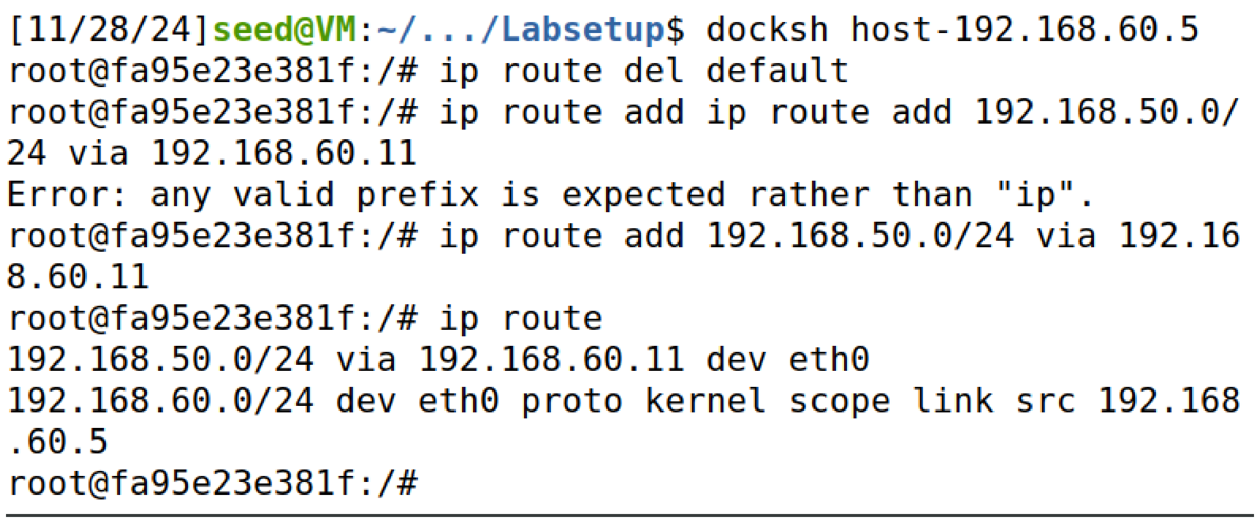


```
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket ==>: 192.168.60.5 --> 192.168.53.99
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=3.67 ms
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket ==>: 192.168.60.5 --> 192.168.53.99
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=2.19 ms
```

Task 6:

After breaking the VPN tunnel we can notice that the telnet session becomes unresponsive, since the VPN packets cannot reach their destination. After reconnection, the telnet session resumes, and the lost packets get retransmitted. If you fail to reconnect too slowly, the connection times out and becomes closed.

Task 7:



```
[11/28/24]seed@VM:~/.../Labsetup$ docksh host-192.168.60.5
root@fa95e23e381f:/# ip route del default
root@fa95e23e381f:/# ip route add ip route add 192.168.50.0/
24 via 192.168.60.11
Error: any valid prefix is expected rather than "ip".
root@fa95e23e381f:/# ip route add 192.168.50.0/24 via 192.16
8.60.11
root@fa95e23e381f:/# ip route
192.168.50.0/24 via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168
.60.5
root@fa95e23e381f:/#
```

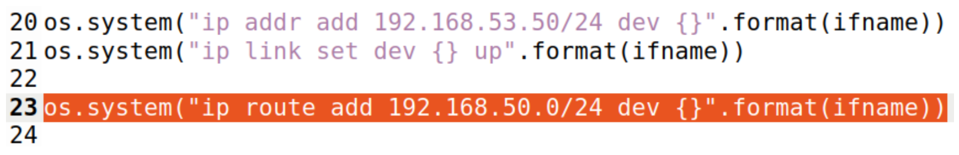
Here we are deleting our default ip route, and adding a more specific entry to the routing table. This will help us prepare for the next task.

Task 8

> The networks are initially set up with the following code:

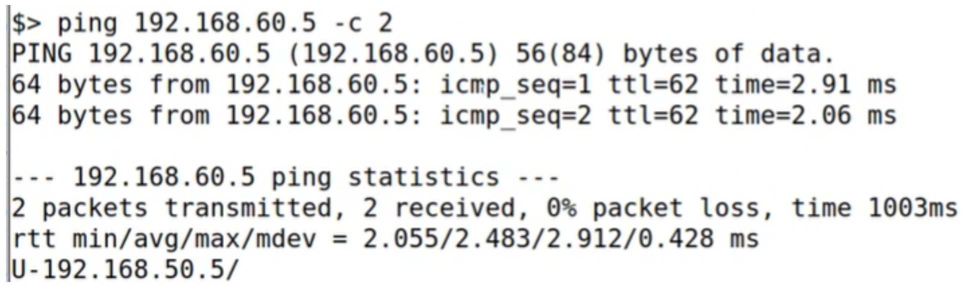
docksh host-192.168.50.5	Host U
docksh client-10.9.0.5	VPN Client
docksh server-router	VPN Server
dock sh host-192.168.60.5	Host V

> By changing the code in *tun_server.py* to the following, it sets up the VPN tunnel by rerouting it through the VPN tunnel, indicated by providing the same interface name in the VPN client and server



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

os.system("ip route add 192.168.50.0/24 dev {}".format(ifname))

```

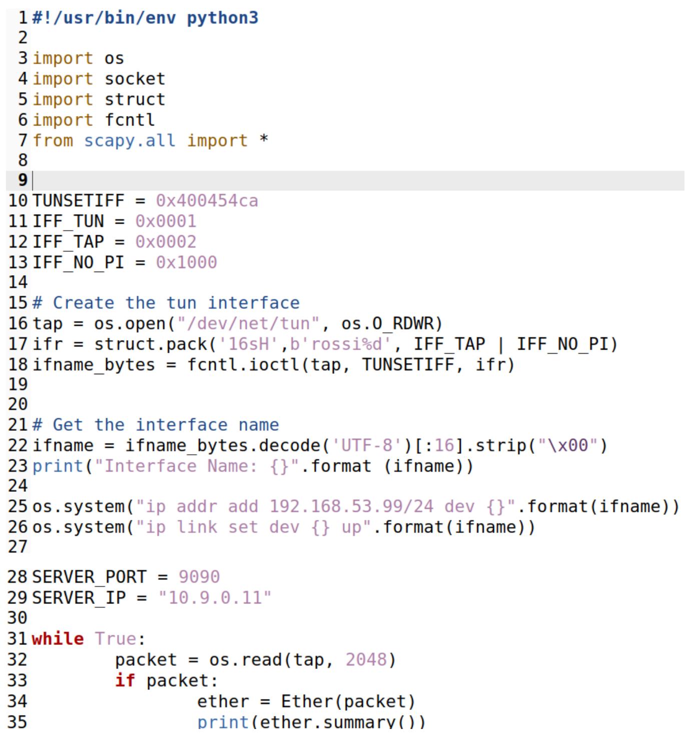
> When pinging User V with User U with the VPN tunnel now set up, User V would receive the packets, as indicated below



```
$> ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=62 time=2.91 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=62 time=2.06 ms

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1003ms
rtt min/avg/max/mdev = 2.055/2.483/2.912/0.428 ms
U-192.168.50.5/
```

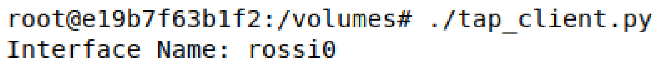
Task 9

> The following code is used in *tap_client.py*. The code is similar to *tap_client.py*, but is used for TAP interfaces



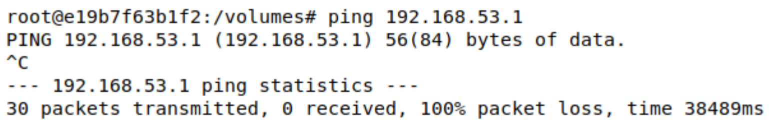
```
#!/usr/bin/env python3

import os
import socket
import struct
import fcntl
from scapy.all import *


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

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

while True:
        packet = os.read(tap, 2048)
        if packet:
                ether = Ether(packet)
                print(ether.summary())
```

> Executing the program shows it's running by printing the interface name



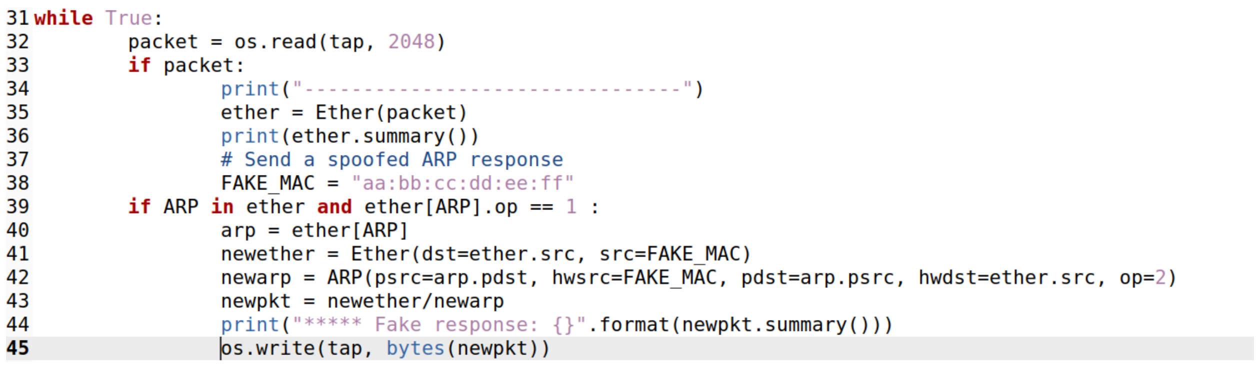
```
root@e19b7f63b1f2:/volumes# ./tap_client.py
Interface Name: rossi0
```

When trying to ping the IP address 192.168.53.1, it results in 0 packets being received. This is because said IP is not corresponding to any machine, this is sending packets to un unreachable destination.



```
root@e19b7f63b1f2:/volumes# 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 ---
30 packets transmitted, 0 received, 100% packet loss, time 38489ms
```

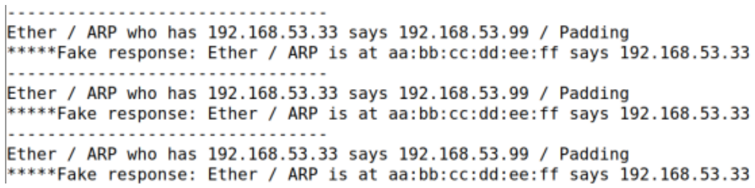
> The code was then updated in the while loops, becoming this



```
while True:
        packet = os.read(tap, 2048)
        if packet:
                print("--------------------------------")
                ether = Ether(packet)
                print(ether.summary())
                # Send a spoofed ARP response
                FAKE_MAC = "aa:bb:cc:dd:ee:ff"
        if ARP in ether and ether[ARP].op == 1 :
                arp = ether[ARP]
                newether = Ether(dst=ether.src, src=FAKE_MAC)
                newarp = ARP(psrc=arp.pdst, hwsrc=FAKE_MAC, pdst=arp.psrc, hwdst=ether.src, op=2)
                newpkt = newether/newarp
                print("***** Fake response: {}".format(newpkt.summary()))
                os.write(tap, bytes(newpkt))
```

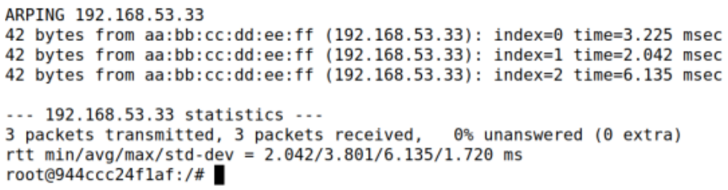
> When running *tap_client.py* and using the command *arping -I rossi0 192.168.53.33*, it responds with the respective outputs, indicating that a spoofed message reply from the MAC address had sent

./tap_client.py



```
--------------------------------
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
```

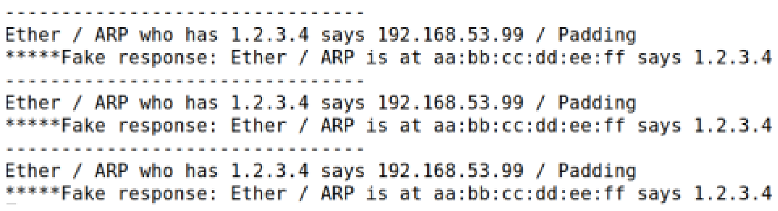
arping -I rossi0 192.168.53.33



```
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:/# 
```

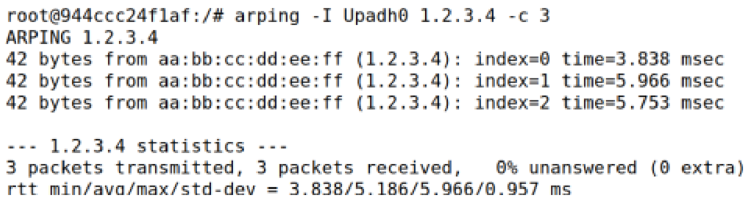
> This works when changing the arping command to *arping -I rossi0 1.2.3.4* as well

./tap_client.py



```
--------------------------------
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
```

arping -I rossi0 1.2.3.4



```
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
```