CSE510 Lab 1

Name: Shreyas Ramesh
Email: ramesh3@buffalo.edu
UBID: ramesh3
UB Number: 50540974

Before You Start:

Please write a detailed lab report, with **screenshots**, to describe what you have **done** and what you have **observed**. You also need to provide **explanation** to the observations that you noticed. Please also show the important **code snippets** followed by explanation. Simply attaching code without any explanation will NOT receive credits.
After you finish, export this report as a **PDF** file and submit it on UBLearns.

Academic Integrity Statement:
I, _Shreyas Ramesh____, have read and understood the course academic integrity policy.
(Your report will not be graded without filling your name in the above AI statement)

Task 1: Network Setup

I setup the lab by following the instructions in the pdf. Ran docker-compose commands and was able to see all the docker process running by running the docker ps command.

```
[02/27/24]seed@VM:~/.../Labsetup$ docker ps
CONTAINER ID        IMAGE                                 COMMAND                  CREATED             STATUS              PORTS
NAMES
1a1dd74dbc25        handsonsecurity/seed-ubuntu:large     "bash -c ' ip route …"   15 minutes ago      Up 15 minutes
host-192.168.60.5
89c9d19adcd6        handsonsecurity/seed-ubuntu:large     "bash -c ' ip route …"   15 minutes ago      Up 15 minutes
host-192.168.60.6
62bfa084b8c2        handsonsecurity/seed-ubuntu:large     "bash -c ' ip route …"   15 minutes ago      Up 15 minutes
server-router
b87586440915        handsonsecurity/seed-ubuntu:large     "bash -c ' tail -f /…"   15 minutes ago      Up 15 minutes
client-10.9.0.5
[02/27/24]seed@VM:~/.../Labsetup$
```

Next step was to check if the client container can ping the router container.

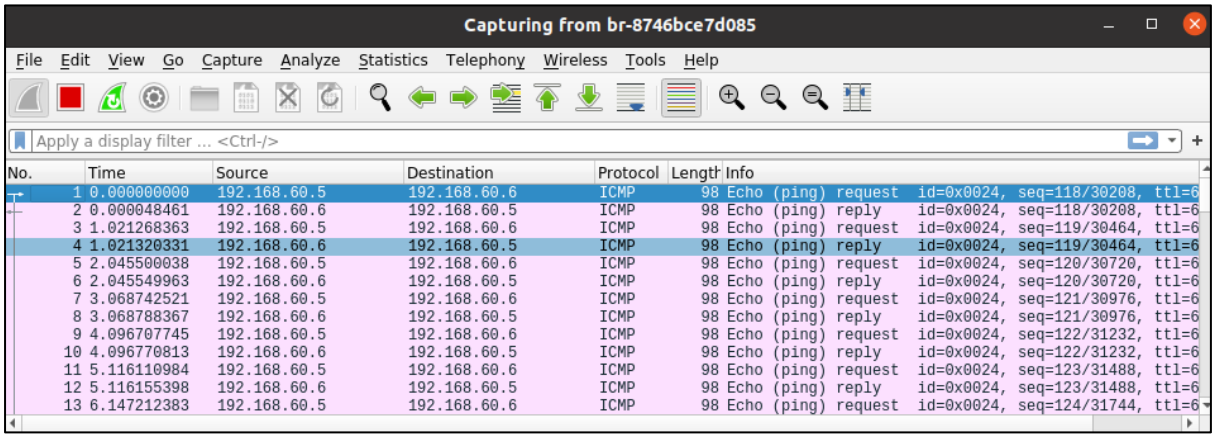
```
root@b87586440915:/# ping 192.168.60.1
PING 192.168.60.1 (192.168.60.1) 56(84) bytes of data.
64 bytes from 192.168.60.1: icmp_seq=1 ttl=64 time=0.082 ms
64 bytes from 192.168.60.1: icmp_seq=2 ttl=64 time=0.498 ms
^C
--- 192.168.60.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1019ms
rtt min/avg/max/mdev = 0.082/0.290/0.498/0.208 ms
root@b87586440915:/#
```

I then verified if the client container was able to ping a host v containers.

```
root@b87586440915:/# 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 ---
5 packets transmitted, 0 received, 100% packet loss, time 4101ms

root@b87586440915:/# ping 192.168.60.6
PING 192.168.60.6 (192.168.60.6) 56(84) bytes of data.
^C
--- 192.168.60.6 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3071ms

root@b87586440915:/#
```

As expected, the client container was not able to ping host v containers.

I used wireshark to capture the packets. It's a very handy tool to troubleshoot network packets.



Task 2: Create and Configure TUN Interface

I used the code snippet given in the pdf and created a tun.py file in the volumes directory. Since it's a shared directory, the docker containers can execute the code in them and I can edit them on the VM saving me a lot of hassle with troubleshooting.

```
root@b87586440915:/# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var  volumes
root@b87586440915:/# cd volumes
root@b87586440915:/volumes# ls
tun.py
root@b87586440915:/volumes# ls
tun.py
root@b87586440915:/volumes# chmod +x tun.py
root@b87586440915:/volumes# sudo tun.py
bash: sudo: command not found
root@b87586440915:/volumes# tun.py
Interface Name: tun0
```

I ran ip a command to look at all the network interfaces and I was able to see a new interface named tun0.

```
[02/27/24]seed@VM:~/.../Labsetup$ docker exec -it b87586440915 /bin/bash
root@b87586440915:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group 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
2: tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
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
root@b87586440915:/#
```

I changed the name of the interface from tun to shrey (my first name). Attached below is the modified code snippet.

```
root@b87586440915:/volumes# tun.py
Interface Name: shrey0
```

Open ~/Documents/Labsetup/volumes tun.py Save

docker-compose.yml | tun.py

```
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'shrey%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
```

I can see that name of the interface was also changed successfully.

```
root@b87586440915:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group 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
4: shrey0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
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
root@b87586440915:/#
```

In order to make the tunnel function as a network interface, it needs to have an ip address and a route through which it can route packets. I added the below lines to the tun.py script:

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

os.system("ip link set dev {} up".format(ifname))

I ran the ip a command again to check the networking status of the tunnel interface.

```
root@b87586440915:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group 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
8: shrey0: <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 shrey0
       valid_lft forever preferred_lft forever
root@b87586440915:/#
```

We can observe that the tunnel interface now has networking details attached to it.

Now that we have a tunnel interface configured, our next step is to fetch and send packets through the tunnel. We use the scapy class in python to achieve this. I replaced the while loop in my original code snippet with the one provided in the lab pdf.

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

After this modification, I ran the code and tried to ping an IP address on the network 192.168.53.0/24 and I saw that the packets were being captured on the tun interface since they were on the same LAN. Since our tunnel is not fully configured yet, we don't get a response back.

```
root@b87586440915:/volumes# tun.py
Interface Name: shrey0
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
```

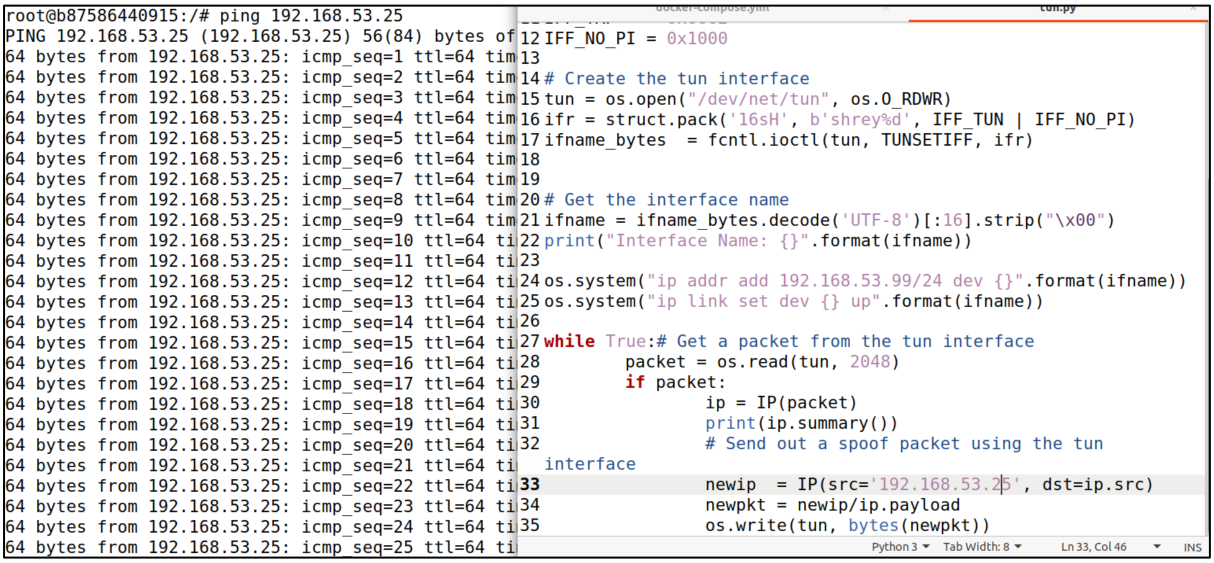
Next, I tried pinging a random IP on the 192.168.60.0/24 network and saw that no packets were being delivered and the tun interface was also not capturing any packets. This means that our tunnel interface is only sending/receiving packets on 192.168.53.0/24 network.

Now that we know our tun interface and accept and receive packets, the next step in our experiment is to see if we can get it to send out any packets as well. While we are not yet making our tunnel interface fully functional yet, we can write some additional code to ensure out tun interface can spoof requests. I added the following lines to code:

```
# Send out a spoof packet using the tun interface
```

```
newip = IP(src='192.168.53.25', dst=ip.src)

newpkt = newip/ip.payload

os.write(tun, bytes(newpkt))
```

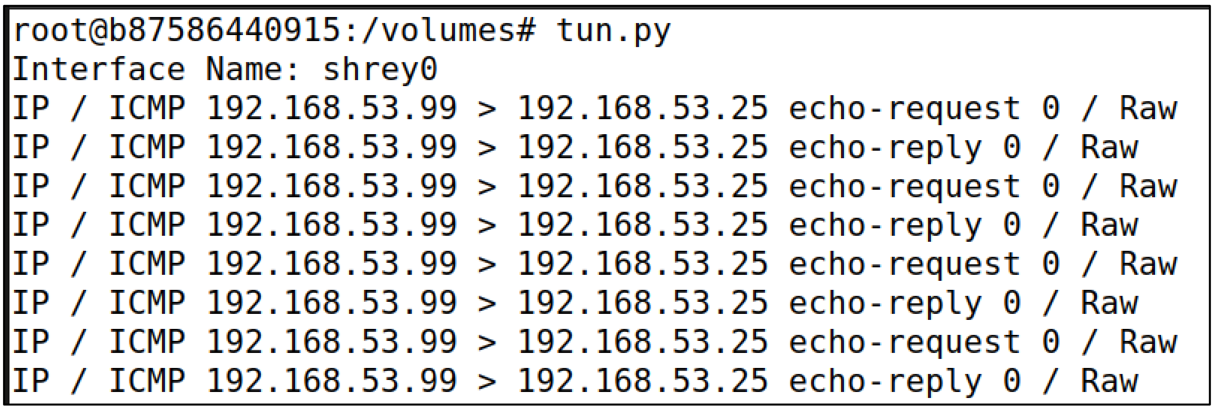
After adding this code, I was able to see that I was getting a response back from the IP address I hardcoded in the code.



```
root@b87586440915:/# ping 192.168.53.25
PING 192.168.53.25 (192.168.53.25) 56(84) bytes of
64 bytes from 192.168.53.25: icmp_seq=1 ttl=64 tim
64 bytes from 192.168.53.25: icmp_seq=2 ttl=64 tim
64 bytes from 192.168.53.25: icmp_seq=3 ttl=64 tim
64 bytes from 192.168.53.25: icmp_seq=4 ttl=64 tim
64 bytes from 192.168.53.25: icmp_seq=5 ttl=64 tim
64 bytes from 192.168.53.25: icmp_seq=6 ttl=64 tim
64 bytes from 192.168.53.25: icmp_seq=7 ttl=64 tim
64 bytes from 192.168.53.25: icmp_seq=8 ttl=64 tim
64 bytes from 192.168.53.25: icmp_seq=9 ttl=64 tim
64 bytes from 192.168.53.25: icmp_seq=10 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=11 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=12 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=13 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=14 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=15 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=16 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=17 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=18 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=19 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=20 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=21 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=22 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=23 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=24 ttl=64 ti
64 bytes from 192.168.53.25: icmp_seq=25 ttl=64 ti
```

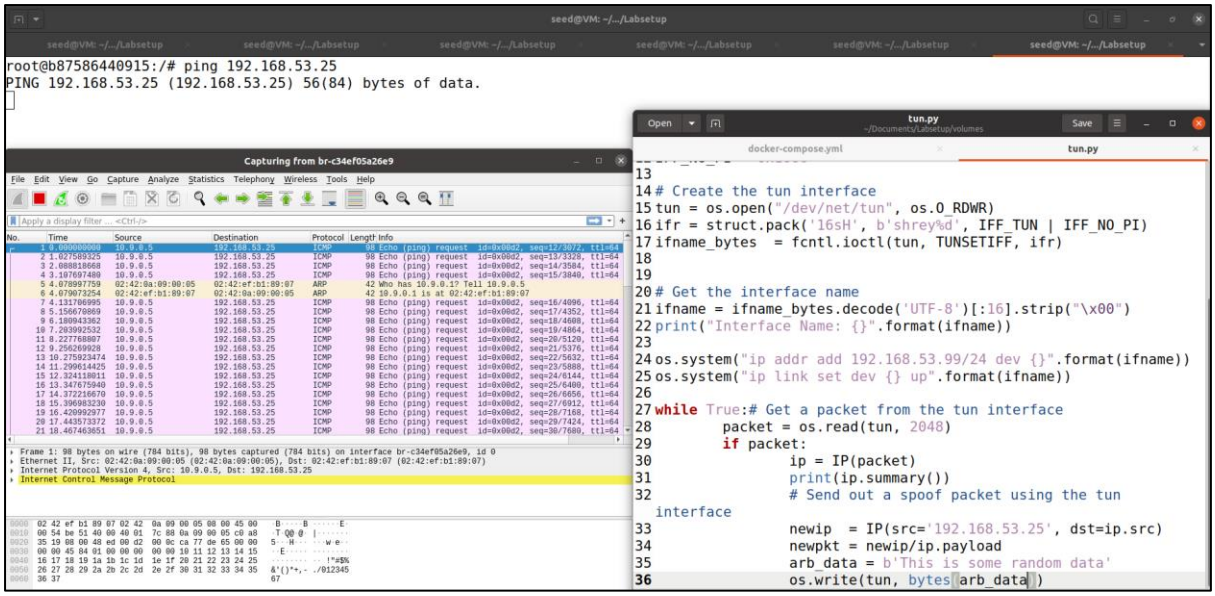
```
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'shrey%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())
                # Send out a spoof packet using the tun
interface
                newip  = IP(src='192.168.53.25', dst=ip.src)
                newpkt = newip/ip.payload
                os.write(tun, bytes(newpkt))
```

Additionally, I was also able to verify that the packets were being read by the tun interface.



```
root@b87586440915:/volumes# tun.py
Interface Name: shrey0
IP / ICMP 192.168.53.99 > 192.168.53.25 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.25 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.25 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.25 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.25 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.25 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.25 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.25 echo-reply 0 / Raw
```

Since we have established that we can spoof IP packets, lets try to see if we will get a response back by sending some arbitrary data instead. Since the data we are returining is not an IP packet, it will not show up as a response to out ICMP ping request. In this case, we can use wireshark or tcpdump to find out if we are getting a response to our request. I used wireshark to find out if we are getting a response.



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

We will now implement a new script called tun_server.py with the following code:

```
#!/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))
```

Lets also rename our tun.py program into tun_client.py and replace the while loop with the one provided in the lab pdf. The code to add is:

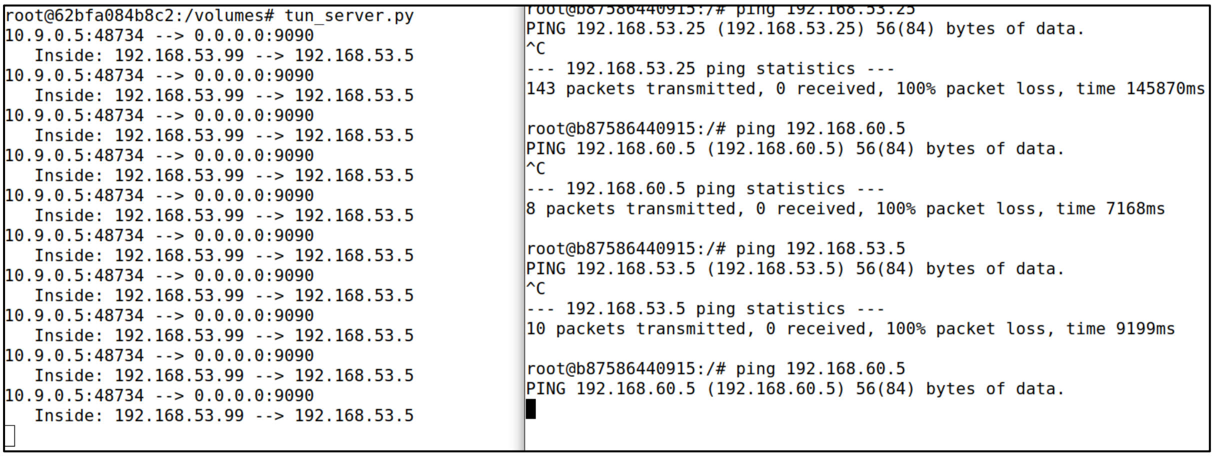
```
SERVER_IP = "10.9.0.11"

SERVER_PORT = 9090

#Create UDP socket
```

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Get a packet from the tun interface

while True:

    packet = os.read(tun, 2048)

    if packet:

    # Send the packet via the tunnel

        sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

The tun_server.py will run on the server router container and the tun_client.py script will run on the client container. After running the code on the containers, I tried pinging a host inside the server router network and saw that packets were being captured by the tun_server interface.

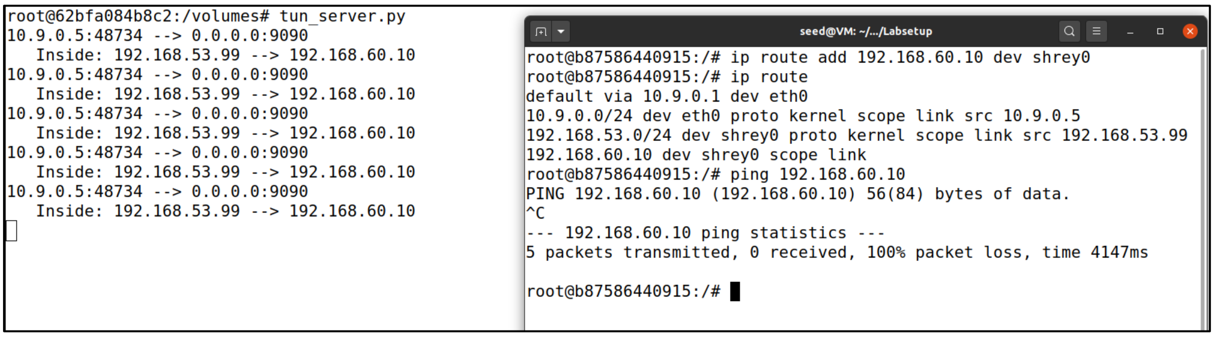
```
root@62bfa084b8c2:/volumes# chmod +x tun_server.py
root@62bfa084b8c2:/volumes# tun_server.py
10.9.0.5:48734 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:48734 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:48734 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:48734 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:48734 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:48734 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:48734 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:48734 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:48734 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:48734 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
```

Since the tunnel does not have the capability to transmit the data packet yet, we can see that the actual host did not receive the data. As my next experiment, I tried pinging a different LAN network to observe what happens.



As witnessed from the screenshot above, the packets are not being captured by the interface.

Next, I tried adding a new IP route and pinged the network.



And sure enough, packets were being captured by the interface.

Task 4: Set Up the VPN Server

In this task, we will try to forward the packets received to the hosts within the server router network. I modified the code within the tun_server.py and documented below is the screenshot which contains the code.

tun_server.py

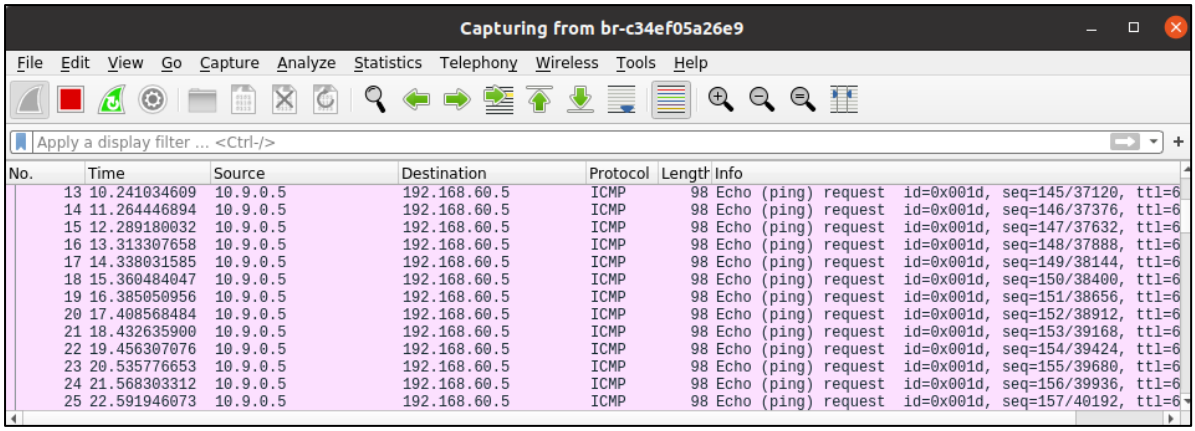
```
#!/usr/bin/env python3
from scapy.all import*
import fcntl
import os
import time
from scapy.all import*

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'shrey%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))

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))
```

After making this modification, we will try to ping a host within the server router from the client container and see if the packet is actually reaching the destination.

```
[02/27/24]seed@VM:~/.../Labsetup$ docker exec -it b87586440915 /bin/bash
root@b87586440915:/# ping 10.9.0.11
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=0.212 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.058 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.061 ms
64 bytes from 10.9.0.11: icmp_seq=4 ttl=64 time=0.209 ms
64 bytes from 10.9.0.11: icmp_seq=5 ttl=64 time=0.063 ms
^C
--- 10.9.0.11 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4089ms
rtt min/avg/max/mdev = 0.058/0.120/0.212/0.073 ms
root@b87586440915:/# 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 ---
15 packets transmitted, 0 received, 100% packet loss, time 14318ms

root@b87586440915:/# cd volumes/
root@b87586440915:/volumes# tun_client.py
Interface Name: shrey0
```

```
seed@VM: ~/.../Labsetup
[02/27/24]seed@VM:~/.../Labsetup$ docker exec -it b87586440915 /bin/bash
root@b87586440915:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
```

Since our tunnel is not configured as bidirectional, we cannot expect a response to our pings. But we can use wireshark to sniff if the traffic reached its destination.



We can see that our packets have successfully reached its destination.

Task 5: Handling Traffic in Both Directions

In order to make our VPN tunnel functional, it has to both send and receive packets. Lets make a few modifications to our tun_client and tun_server codes to add this capability.

Code for tun_server:

```
21 os.system("ip addr add 192.168.53.11/24 dev {}".format(ifname))
22 os.system("ip link set dev {} up".format(ifname))
23
24 IP_A = "0.0.0.0"
25 PORT = 9090
26 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
27 sock.bind((IP_A, PORT))
28
29 #ip = "10.9.0.5"
30 #port = 1030
31
32 while True:
33 # this will block until at least one interface is ready
34         ready, _, _ = select.select([sock, tun], [], [])
35         for fd in ready:
36                 if fd is sock:
37                         data, (ip, port) = sock.recvfrom(2048)
38                         pkt = IP(data)
39                         print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
40                         #forward packet to tun interface
41                         os.write(tun, bytes(pkt))
42
43                 if fd is tun:
44                         packet = os.read(tun, 2048)
45                         pkt = IP(packet)
46                         print("From tun    ==>: {} --> {}".format(pkt.src, pkt.dst))
47                         #forward packet to socket
48                         sock.sendto(packet, (ip, port))
49
```

Code for tun_client:

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

SERVER_PORT = 9090
SERVER_IP = "10.9.0.11"

IP_A = "0.0.0.0"
PORT = 9090
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_A, PORT))


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))
                        #forward packet to tun interface
                        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))
                        #forward packet to socket
                        sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

After making these changes, I ran the code in their respective containers and pinged from client container to host v.

```
[02/28/24]seed@VM:~$ docker exec -it fb98d144f869 /bin/bash
root@fb98d144f869:/# ping 192.168.60.6
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=63 time=5.03 ms
64 bytes from 192.168.60.6: icmp_seq=2 ttl=63 time=5.15 ms
64 bytes from 192.168.60.6: icmp_seq=3 ttl=63 time=3.43 ms
64 bytes from 192.168.60.6: icmp_seq=4 ttl=63 time=3.82 ms
64 bytes from 192.168.60.6: icmp_seq=5 ttl=63 time=3.22 ms
64 bytes from 192.168.60.6: icmp_seq=6 ttl=63 time=3.58 ms
64 bytes from 192.168.60.6: icmp_seq=7 ttl=63 time=3.10 ms
64 bytes from 192.168.60.6: icmp_seq=8 ttl=63 time=4.89 ms
64 bytes from 192.168.60.6: icmp_seq=9 ttl=63 time=3.58 ms
64 bytes from 192.168.60.6: icmp_seq=10 ttl=63 time=3.62 ms
64 bytes from 192.168.60.6: icmp_seq=11 ttl=63 time=3.97 ms
```

We can see that an ICMP response is being sent back from host V. This means that our VPN is functioning as expected. Lets also take a look at the packets being intercepted by the server and client interfaces.

Tun_server interface:

```
root@132758a459de:/volumes# tun_server.py
Interface Name: shrey0
From socket <==: 192.168.53.99 --> 192.168.60.6
From tun    ==>: 192.168.60.6 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.6
From tun    ==>: 192.168.60.6 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.6
From tun    ==>: 192.168.60.6 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.6
From tun    ==>: 192.168.60.6 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.6
From tun    ==>: 192.168.60.6 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.6
From tun    ==>: 192.168.60.6 --> 192.168.53.99
```

Tun_client interface:

```
root@fb98d144f869:/volumes# tun_client.py
Interface Name: shrey0
From tun    ==>: 192.168.53.99 --> 192.168.60.6
From socket <==: 192.168.60.6 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.6
From socket <==: 192.168.60.6 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.6
From socket <==: 192.168.60.6 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.6
From socket <==: 192.168.60.6 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.6
From socket <==: 192.168.60.6 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.6
From socket <==: 192.168.60.6 --> 192.168.53.99
```

Since we established that pinging works as expected, lets try to establish a connection from client container to the host V container using telnet.

```
whoamid376d70d3a15 login:
Password:

Login incorrect
d376d70d3a15 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@d376d70d3a15:~$ whoami
seed
seed@d376d70d3a15:~$
```

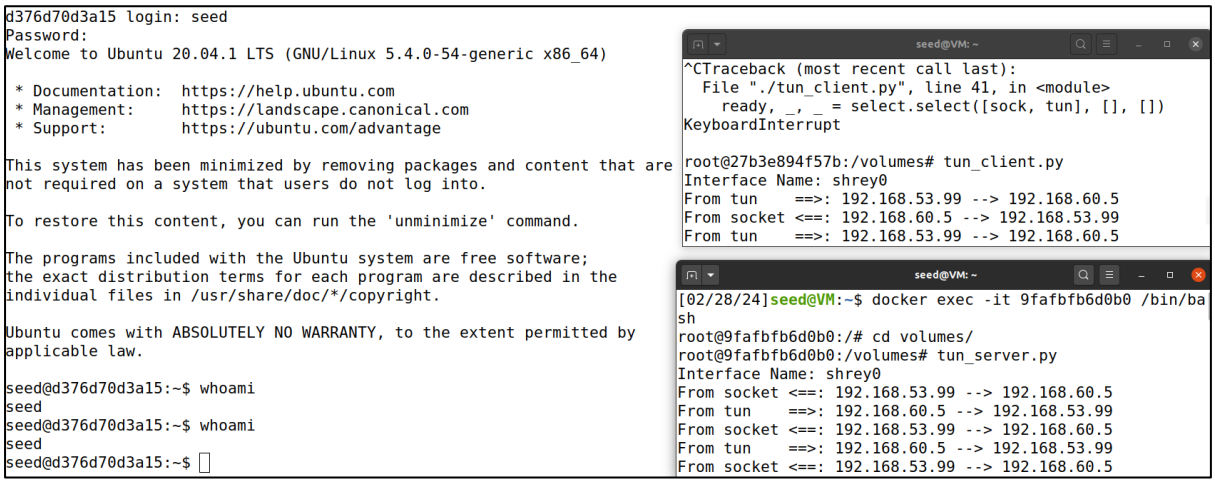
```
[02/28/24]seed@VM:~$ docker exec -it 27b3e894f57b /bin/b
ash
root@27b3e894f57b:/# cd volumes/
root@27b3e894f57b:/volumes# tun_client.py
Interface Name: shrey0
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
```

```
seed@VM: ~
[02/28/24]seed@VM:~$ docker exec -it 9fafbfb6d0b0 /bin/ba
sh
root@9fafbfb6d0b0:/# cd volumes/
root@9fafbfb6d0b0:/volumes# tun_server.py
Interface Name: shrey0
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
```

We can see that telnet also works as expected and we are able to establish a connection without any issues.

Task 6: Tunnel-Breaking Experiment

Since we have already established a telnet connection in our previous experiment, lets try to see what will happen when I suddenly break the VPN connection on either the client or server side. I broke the connection on the client side and saw that whatever I was trying to type on the telnet console was not visible. I then re-established the connection and whatever command I was trying to write showed up on the screen. Since I did not take a long time after breaking and re-establishing the connection, the characters showed up on the console. But if a significant amount of time passes, the connection will timeout on the host v side and we have to establish a telnet connection once again.



Task 7: Routing Experiment on Host V

Lets now experiment with the default routes our traffic uses to reach its destination. In order to start this experiment, I logged into a container on host v and printed out its ip routes using the command 'ip route'.

```
[02/28/24]seed@VM:~$ docker ps
CONTAINER ID        IMAGE                                COMMAND                  CREATED             STATUS              PORTS
NAMES
d376d70d3a15        handsonsecurity/seed-ubuntu:large    "bash -c ' ip route …"   20 minutes ago      Up 20 minutes
host-192.168.60.5
4a2af44bbfaf        handsonsecurity/seed-ubuntu:large    "bash -c ' ip route …"   20 minutes ago      Up 20 minutes
host-192.168.60.6
9fafbfb6d0b0        handsonsecurity/seed-ubuntu:large    "bash -c ' ip route …"   20 minutes ago      Up 20 minutes
server-router
27b3e894f57b        handsonsecurity/seed-ubuntu:large    "bash -c ' tail -f /…"   20 minutes ago      Up 20 minutes
client-10.9.0.5
[02/28/24]seed@VM:~$ docker exec -it d376d70d3a15 /bin/bash
root@d376d70d3a15:/# 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@d376d70d3a15:/#
```

I used the 'ip route del default' command to get rid of the default route on the container.

```
root@d376d70d3a15:/# ip route del default
root@d376d70d3a15:/# ip route
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@d376d70d3a15:/#
```

The next step in this experiment was to check if pinging this container worked after the default route was deleted. Spoiler alert, it wont work but lets try it out.

```
[02/28/24]seed@VM:~$ docker exec -it d376d70d3a15 /bin/bash
root@d376d70d3a15:/# 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@d376d70d3a15:/# ip route del default
root@d376d70d3a15:/# ip route
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@d376d70d3a15:/#
```

```
seed
seed@d376d70d3a15:~$ whoami
seed
seed@d376d70d3a15:~$ exit
logout
Connection closed by foreign host.
root@27b3e894f57b:/# ping 192.168.60.5
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=63 time=6.45 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=3.64 ms
^C
--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 3.644/5.047/6.451/1.403 ms
root@27b3e894f57b:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
```

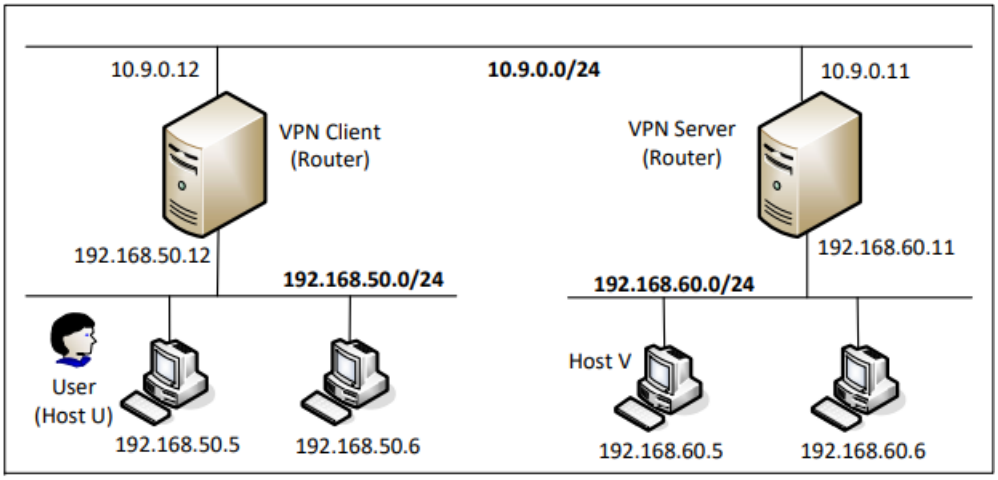
I readded the default route and was able to see that the packets were being sent and received as expected.

```
root@d376d70d3a15:/# ip route del 192.168.53.0/24
root@d376d70d3a15:/# ip route add default via 192.168.60.11
root@d376d70d3a15:/# 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@d376d70d3a15:/#
```

```
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=63 time=3.52 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=2.68 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=3.13 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=2.35 ms
^C
--- 192.168.60.5 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3011ms
```

Task 8: VPN Between Private Networks

In order to emulate a real world scenario, we need to connect two different private networks over a common network (usually the internet). I used the docker-compose2.yaml file to establish the following network:



```
[02/28/24]seed@VM:~/.../Labsetup$ docker-compose -f docker-compose2.yml build
HostA uses an image, skipping
HostB uses an image, skipping
VPN_Client uses an image, skipping
Host1 uses an image, skipping
Host2 uses an image, skipping
Router uses an image, skipping
[02/28/24]seed@VM:~/.../Labsetup$ docker-compose -f docker-compose2.yml up
Creating network "net-192.168.50.0" with the default driver
WARNING: Found orphan containers (oracle-10.9.0.80) for this project. If you removed or renamed this service in your compose file, you can ru
n this command with the --remove-orphans flag to clean it up.
host-192.168.60.6 is up-to-date
server-router is up-to-date
host-192.168.60.5 is up-to-date
Creating host-192.168.50.6 ... done
Recreating client-10.9.0.5 ... done
Creating host-192.168.50.5 ... done
Attaching to host-192.168.60.6, server-router, host-192.168.60.5, host-192.168.50.6, host-192.168.50.5, client-10.9.0.5
host-192.168.50.5 |  * Starting internet superserver inetd              [ OK ]
host-192.168.50.6 |  * Starting internet superserver inetd              [ OK ]
host-192.168.60.5 |  * Starting internet superserver inetd              [ OK ]
host-192.168.60.6 |  * Starting internet superserver inetd              [ OK ]
```

I have successfully established my test environment.

Since these machines also had the ephemeral storage setting set to volumes, I was able to reuse the same code. I just had to figure out the routing aspects in order to be able to send and receive packets over the VPN network. I modified the code in both the tun_server and tun_client and documented them below.

Code for Tun_server:

```python
#!/usr/bin/env python3
from scapy.all import*
import fcntl
import os
import time
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'shrey%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 192.168.53.11/24 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))

IP_A = "0.0.0.0"
PORT = 9090
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_A, PORT))

ip = "10.9.0.12"
```

Code for tun_client:

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

import fcntl
import struct
import os
import time
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'shrey%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))

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

SERVER_PORT = 9090
SERVER_IP = "10.9.0.11"

IP_A = "0.0.0.0"
```

After making these changes, I ran the code in their respective containers to establish the VPN connection. Next part was to test out if the changes works as expected. I logged into a container within the client network and pinged a container within host V.

```
[02/28/24]seed@VM:~/.../Labsetup$ docksh 52117d296d05
root@52117d296d05:/# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group 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
63: eth0@if64: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:c0:a8:32:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.50.5/24 brd 192.168.50.255 scope global eth0
       valid_lft forever preferred_lft forever
root@52117d296d05:/# ping 192.1686.60.5
^C
root@52117d296d05:/# ip route
default via 192.168.50.12 dev eth0
192.168.50.0/24 dev eth0 proto kernel scope link src 192.168.50.5
root@52117d296d05:/# ping 192.1686.60.5
^C
root@52117d296d05:/# ping 192.168.60.5
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=3.96 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=62 time=3.18 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=62 time=2.81 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=62 time=1.82 ms
^C
--- 192.168.60.5 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3006ms
rtt min/avg/max/mdev = 1.823/2.943/3.963/0.769 ms
root@52117d296d05:/#
```

We can see that connection is successful and packets are being sent and received through our VPN from two different private networks.

What would happen if the VPN connection is terminated in either the client or server end? Would we still be able to send and receive packets? We can answer both these questions with a simple demonstration as documented below:

```
root@9fafbfb6d0b0:/volumes# tun_server.py
Interface Name: shrey0
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.50.5
^CTraceback (most recent call last):
  File "./tun_server.py", line 36, in <module>
    ready, _, _ = select.select([sock, tun], [], [])
KeyboardInterrupt

root@9fafbfb6d0b0:/volumes#
```

```
seed@VM: ~/.../Labsetup
                                client-10.9.0.5
52117d296d05        handsonsecurity/seed-ubuntu:large   "bash -c
                                host-192.168.50.5
0ded0a7789c0        handsonsecurity/seed-ubuntu:large   "bash -c
                                host-192.168.50.6
d376d70d3a15        handsonsecurity/seed-ubuntu:large   "bash -c
                                host-192.168.60.5
4a2af44bbfaf        handsonsecurity/seed-ubuntu:large   "bash -c
                                host-192.168.60.6
9fafbfb6d0b0        handsonsecurity/seed-ubuntu:large   "bash -c
                                server-router
[02/29/24]seed@VM:~/.../Labsetup$ docksh 52117d296d05
root@52117d296d05:/# 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 ---
8 packets transmitted, 0 received, 100% packet loss, time 7147ms

root@52117d296d05:/#
```

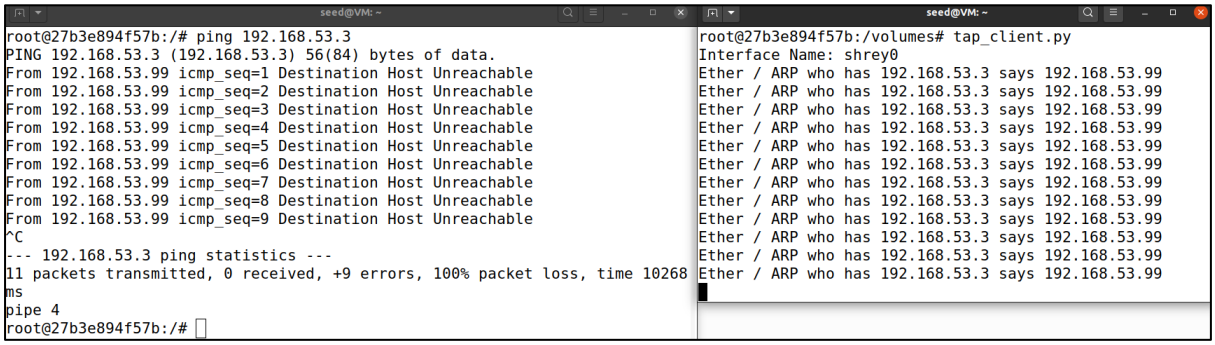
We can see that packets are not being sent once the VPN connection has been severed at the server end. The same would be the case if the connection was terminated from the client side as well. Thus we can prove that packets are only being sent because of the VPN tunnel we have established.

Task 9: Experiment with the TAP Interface

In order to proceed with the TAP interface experiment, I used the previous network as defined in docker-compose.yaml file. I added a new file called tap_client.py and edited it as follows:

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

import fcntl
import struct
import os
import time
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'shrey%d',IFF_TAP| IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tap, TUNSETIFF, ifr)
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:
        packet = os.read(tap, 2048)
        if packet:
                ether = Ether(packet)
                print(ether.summary())
```

I ran the code and pinged a random host on the 192.168.53.0/24 network to check what would be captured at the TAP interface. Documented below is the response I received:



We can see that the interface routes the ARP packets to the tap interface which isn't sure about the IP and hence it is returning the destination host unreachable message.

Lets now try to spoof if we can spoof the ARP packet and send a different MAC address for the non-existent IP address. I edited the code for the tap_client as follows:

```python
# Create the tun interface
tap = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'shrey%d',IFF_TAP| IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tap, TUNSETIFF, ifr)
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:
        packet = os.read(tap, 2048)
        if packet:
                ether = Ether(packet)
                print(ether.summary())
                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))
```

I tried pinging a random IP address after running the code:

```
root@27b3e894f57b:/# ping 192.168.53.3
PING 192.168.53.3 (192.168.53.3) 56(84) bytes of data.
^C
--- 192.168.53.3 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3056ms

root@27b3e894f57b:/#
```

```
root@27b3e894f57b:/volumes# tap_client.py
Interface Name: shrey0
Ether / ARP who has 192.168.53.3 says 192.168.53.99
*****Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1
92.168.53.3
Ether / IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0
/ Raw
Ether / IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0
/ Raw
Ether / IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0
/ Raw
Ether / IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0
/ Raw
```

Since our TAP interface only listens to ARP packets it wont accept an ICMP packet. In order to check if our TAP interface is working we can use the ARPping command to send out an ARP ping.

```
root@27b3e894f57b:/# arping -i shrey0 192.168.53.3
ARPING 192.168.53.3
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.3): index=0 time=3.3
29 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.3): index=1 time=2.3
86 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.3): index=2 time=2.9
79 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.3): index=3 time=84.
542 msec
^C
--- 192.168.53.3 statistics ---
4 packets transmitted, 4 packets received,   0% unanswered (0 ex
tra)
rtt min/avg/max/std-dev = 2.386/23.309/84.542/35.355 ms
```

```
root@27b3e894f57b:/volumes# tap_client.py
Interface Name: shrey0
Ether / ARP who has 192.168.53.3 says 192.168.53.99 / Padding
*****Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1
92.168.53.3
Ether / ARP who has 192.168.53.3 says 192.168.53.99 / Padding
*****Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1
92.168.53.3
Ether / ARP who has 192.168.53.3 says 192.168.53.99 / Padding
*****Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1
92.168.53.3
Ether / ARP who has 192.168.53.3 says 192.168.53.99 / Padding
*****Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1
92.168.53.3
```

We can see that we received the spoofed MAC address. We can see that we were able to successfully spoof the MAC address.