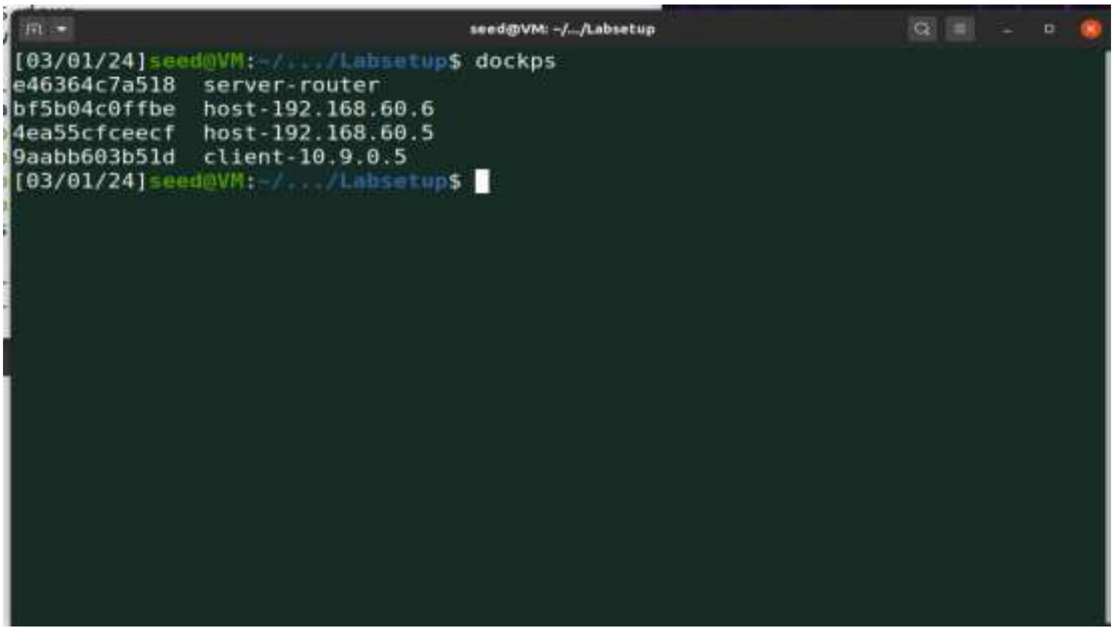
VPN

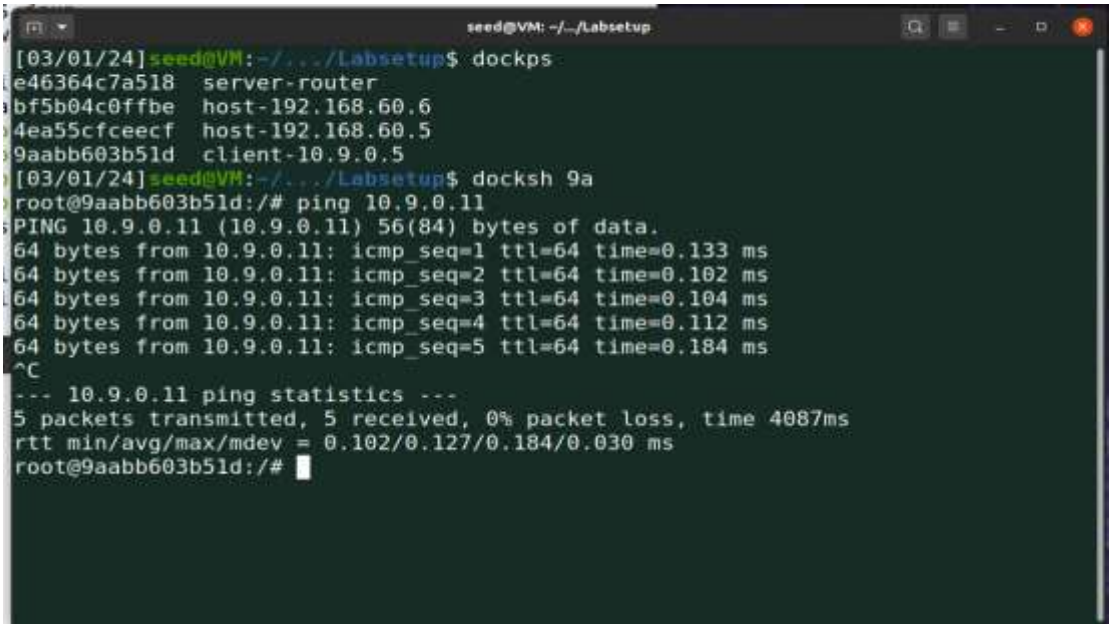
In this lab, we're creating Virtual Private Networks (VPNs) to understand how they work. VPNs are crucial for ensuring secure communication over public networks like the Internet. They essentially create private networks over public infrastructure, allowing for protected data transmission as if you were on a physically isolated network. Our focus here is on one vital aspect of VPN technology: tunneling. Tunneling sets up virtual network paths, but unlike real VPNs, we won't be dealing with encryption in this lab. Instead, we'll concentrate on grasping the basics of tunneling to lay a solid foundation for understanding more complex VPN setups.

Task 1 - Network Setup

We use dcbuild and dcup to setup the required containers with hosts, VPN server/Router, client



We are able to reach from Host u(10.9.0.5) to VPN server(10.9.0.11)



We are able to reach from VPN server(192.168.60.11) to Host v (192.168.60.5)

```
[03/01/24]seed@VM:~/.../Labsetup$ docksh e4
root@e46364c7a518:/# 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=64 time=0.067 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.060 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=64 time=0.157 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=64 time=0.139 ms
^C
--- 192.168.60.5 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3082ms
rtt min/avg/max/mdev = 0.060/0.105/0.157/0.042 ms
root@e46364c7a518:/#
```

We are not able to reach from Host u(10.9.0.5) to Host v(192.168.60.5)

```
root@9aabb603b51d:/# 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 4083ms
```

To check all the interfaces on router/vpn server

```
root@e46364c7a518:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.11  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:0b  txqueuelen 0  (Ethernet)
        RX packets 93  bytes 13574 (13.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 7  bytes 574 (574.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.60.11  netmask 255.255.255.0  broadcast 192.168.60.255
        ether 02:42:c0:a8:3c:0b  txqueuelen 0  (Ethernet)
        RX packets 89  bytes 12667 (12.6 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 6  bytes 476 (476.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
```

We are able to capture packets using tcpdump on eth0 of router.
As we can see ICMP request and replies between client and router are captured.

```
root@e46364c7a518:/# 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
18:38:29.654730 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 15, seq 1, length
 64
18:38:29.654748 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 15, seq 1, length 6
4
18:38:30.678825 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 15, seq 2, length
 64
18:38:30.678842 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 15, seq 2, length 6
4
18:38:31.709551 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 15, seq 3, length
 64
18:38:31.709603 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 15, seq 3, length 6
4
18:38:32.727569 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 15, seq 4, length
 64
18:38:32.727603 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 15, seq 4, length 6
4
18:38:33.751273 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 15, seq 5, length
```

Although we see packets captured from Host u to server but it does not capture packets from host u to host v

Now we listen to interface eth1 and we see that the ICMP packets are captured from Router/VPN server to Host v

```
seed@VM: ~/.../Labsetup
root@e46364c7a518:/# 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
18:44:09.829809 IP 192.168.60.11 > 192.168.60.5: ICMP echo request, id 26, seq 1
, length 64
18:44:09.829867 IP 192.168.60.5 > 192.168.60.11: ICMP echo reply, id 26, seq 1,
length 64
18:44:10.844732 IP 192.168.60.11 > 192.168.60.5: ICMP echo request, id 26, seq 2
, length 64
18:44:10.844791 IP 192.168.60.5 > 192.168.60.11: ICMP echo reply, id 26, seq 2,
length 64
18:44:11.866642 IP 192.168.60.11 > 192.168.60.5: ICMP echo request, id 26, seq 3
, length 64
18:44:11.866684 IP 192.168.60.5 > 192.168.60.11: ICMP echo reply, id 26, seq 3,
length 64
18:44:12.900141 IP 192.168.60.11 > 192.168.60.5: ICMP echo request, id 26, seq 4
, length 64
18:44:12.900235 IP 192.168.60.5 > 192.168.60.11: ICMP echo reply, id 26, seq 4,
length 64
18:44:13.913497 IP 192.168.60.11 > 192.168.60.5: ICMP echo request, id 26, seq 5
, length 64
18:44:13.913549 IP 192.168.60.5 > 192.168.60.11: ICMP echo reply, id 26, seq 5,
length 64
18:44:14.943233 IP 192.168.60.11 > 192.168.60.5: ICMP echo request, id 26, seq 6
```

At this instance we can see that environment is ready for further tasks.

<u>Task 2- Create and configure TUN Interface</u>

The TUN interface serves as a virtual network device that allows for the creation of secure, point-to-point connections over a public network, such as the Internet. We'll be creating and configuring the TUN interface, defining its parameters such as IP addresses and routes, and exploring how it facilitates secure communication between endpoints. By understanding how to set up and configure the TUN interface, we'll lay the groundwork for building our own basic VPN infrastructure.

Task 2.a : Name of the interface

We run it using root privilege, so executing after providing necessary access. We see the tun0 named interface.

```
root@9aabb603b51d:/# cd volumes
root@9aabb603b51d:/volumes# chmod a+x tun.py
root@9aabb603b51d:/volumes# tun.py
Interface Name: tun0
```

The program gets blocked so we open another window and type command ip address to check if the interface was created. We see it has been created.

```
[03/01/24]seed@VM:~/.../Labsetup$ docksh 9a
root@9aabb603b51d:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defaul
t 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 defa
ult qlen 500
    link/none
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP g
roup 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@9aabb603b51d:/#
```

We open the code and change the TUN interface name to my last name menon.

```
#!/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'menon%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))

while True:
    time.sleep(10)
```

Executing after the change we see that it displays the new name which is my last name.

```
root@9aabb603b51d:/volumes# tun.py
Interface Name: menon0
```

We can check that the TUN interface with name menon0 has been created on client 10.9.0.5.

```
root@9aabb603b51d:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defaul
t 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
3: menon0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group de
fault qlen 500
    link/none
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP g
roup 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@9aabb603b51d:/#
```

Task 2.b : Set up the TUN Interface

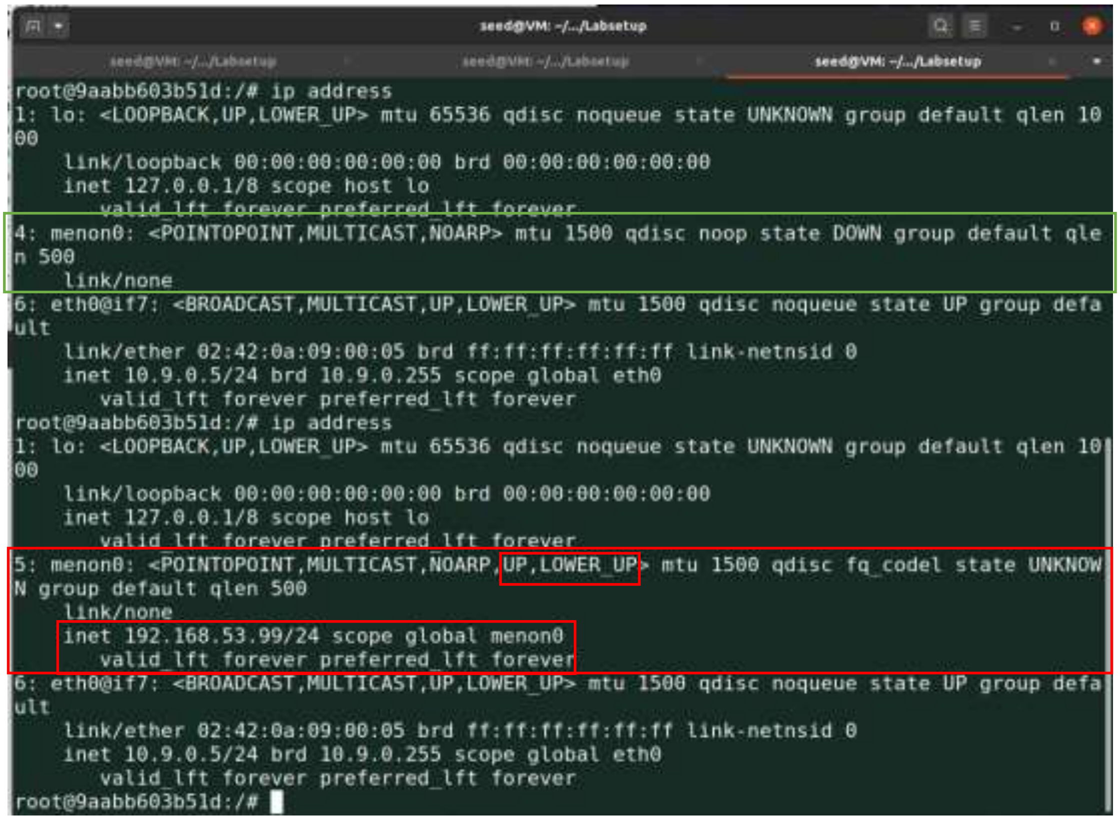
The interface is still not usable as we need to assign it an ip address and also turn it up.
We add the following 2 lines of code to tun.py
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))
Thus ip address is assigned to TUN interface and its brought up

```
#!/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'menon%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))

while True:
   time.sleep(10)
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))
```

On executing we see that the interface name is displayed.

```
root@9aabb603b51d:/volumes# tun.py
Interface Name: menon0
```

Next, we compare the older ip address details and the current after assigning the ip to TUN interface.

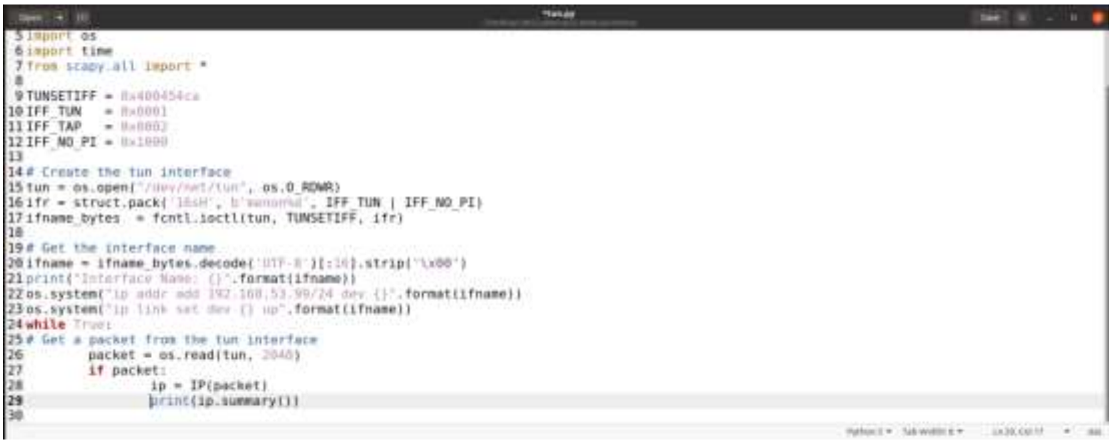


Here we see that in green box, the older configuration we see that no ip address is assigned to menon0 interface created. The state is down for interface for now.

Once we add the lines in code and execute we see that in red box ip address is assigned and the 'UP' State is displayed.

Task 2.c : Read from TUN Interface

Here, we try to read packets with TUN Interface. We add the following script in while loop in tun.py.



1.We try to ping 192.168.53.0/24 hosts, We ping 192.168.53.3 and 192.168.53.1. We see that even though the packets are not received, interface is able to read the packets. But when we ping 192.168.53.99 that is the interface itself nothing is dispalyed on interface it maybe because beuing the self ip it gets sent to loopback address.

```
root@9aabb603b51d:/# tcpdump -i lo -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
19:57:43.551030 IP 192.168.53.99 > 192.168.53.99: ICMP echo request, id 68, seq 1, length
 64
19:57:43.551041 IP 192.168.53.99 > 192.168.53.99: ICMP echo reply, id 68, seq 1, length 6
4
19:57:44.567262 IP 192.168.53.99 > 192.168.53.99: ICMP echo request, id 68, seq 2, length
 64
19:57:44.567278 IP 192.168.53.99 > 192.168.53.99: ICMP echo reply, id 68, seq 2, length 6
4
19:57:45.604997 IP 192.168.53.99 > 192.168.53.99: ICMP echo request, id 68, seq 3, length
 64
19:57:45.605031 IP 192.168.53.99 > 192.168.53.99: ICMP echo reply, id 68, seq 3, length 6
4
^C
6 packets captured
12 packets received by filter
0 packets dropped by kernel
root@9aabb603b51d:/#
```

Here we ping 192.168.53.3, 192.168.53.1, 192.168.53.99

```
root@9aabb603b51d:/# 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 ---
8 packets transmitted, 0 received, 100% packet loss, time 7178ms

root@9aabb603b51d:/# 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 6171ms

root@9aabb603b51d:/# ping 192.168.53.99
PING 192.168.53.99 (192.168.53.99) 56(84) bytes of data.
64 bytes from 192.168.53.99: icmp_seq=1 ttl=64 time=0.028 ms
64 bytes from 192.168.53.99: icmp_seq=2 ttl=64 time=0.060 ms
64 bytes from 192.168.53.99: icmp_seq=3 ttl=64 time=0.088 ms
64 bytes from 192.168.53.99: icmp_seq=4 ttl=64 time=0.041 ms
64 bytes from 192.168.53.99: icmp_seq=5 ttl=64 time=0.107 ms
64 bytes from 192.168.53.99: icmp_seq=6 ttl=64 time=0.051 ms
^C
--- 192.168.53.99 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5102ms
rtt min/avg/max/mdev = 0.028/0.062/0.107/0.027 ms
root@9aabb603b51d:/#
```

This is the displayed data read by TUN interface on pinging hosts in 192.168.53.0/24 network.

```
root@9aabb603b51d:/volumes# tun.py
Interface Name: menon0
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 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
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
```

2.Now we try pinging 192.168.60.0/24 internal network. So we ping host 192.168.60.5 and we see that no packet is received as expected and the TUN interface doesn’t read any packet info.

```
root@9aabb603b51d:/# 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 6193ms

root@9aabb603b51d:/#
```

We try pinging but no packets received because of being in internal network and the TUN interface also does not read it.

```
root@9aabb603b51d:/volumes# tun.py
Interface Name: menon0
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 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
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
```

Task 2.d : Write to TUN Interface

1.In this part of the code, we simulate an echo server using the TUN interface. Initially, a crafted IP packet with source address '1.2.3.4' and destination address '192.168.53.5' is created, encapsulating an ICMP message with the content 'Menon'. This packet is then written to the TUN interface. Subsequently, the program enters an infinite loop to continuously read packets from the TUN interface. Upon receiving a packet, it parses it into an IP object, prints a summary of the packet and its payload, and constructs an ICMP echo reply packet by swapping the source and destination addresses. This reply packet, containing the message 'Menon', is then written back to the TUN interface. Thus, this segment of the code serves as a basic echo server, echoing back any ICMP messages it receives on the TUN interface.

```
# Receiving packet using the tun interface
newip = IP(src='1.2.3.4', dst='192.168.53.5')
newpkt = newip/ICMP()/b'Menon'
os.write(tun, bytes(newpkt))
while True:
# Get a packet from the tun interface
        packet = os.read(tun, 2048)
        if packet:
                ip = IP(packet)
                print(ip.summary())
                print(ip.payload)
                #sending packet to TUN Interface
                newip = IP(src=ip.dst, dst=ip.src)
                newpkt = newip/ICMP()/b'Hi Menon'
                os.write(tun, bytes(newpkt))
                print('Reply')
                print(IP(bytes(newpkt)).summary())
                print(IP(bytes(newpkt)).payload)
```

On executing we get to see that the packet is sent out and replied back with the payload

```
root@3c392a6f80b9:/volumes# tun.py
Interface Name: menon0
IP / ICMP 1.2.3.4 > 192.168.53.5 echo-request 0 / Raw
b'\x08\x00\xce*\x00\x00\x00\x00Menon'
Reply
IP / ICMP 192.168.53.5 > 1.2.3.4 echo-request 0 / Raw
b'\x08\x00\xbal\x00\x00\x00\x00Hi Menon'
```

2. We try to return some arbitrary data in place of the reply packet we see that an enccoding error is displayed

```
# Receiving packet using the tun interface
newip = IP(src='1.2.3.4', dst='192.168.53.5')
newpkt = newip/ICMP()/b'Menon'
os.write(tun, bytes(newpkt))
while True:
# Get a packet from the tun interface
        packet = os.read(tun, 2048)
        if packet:
                ip = IP(packet)
                print(ip.summary())
                print(ip.payload)
                os.write(tun, bytes('hello'))
```

On executing we see that ‘String argument without an encoding’ message is displayed if we try to reply with arbitrary data.

```
root@3c392a6f80b9:/volumes# tun.py
Interface Name: menon0
IP / ICMP 1.2.3.4 > 192.168.53.5 echo-request 0 / Raw
b'\x08\x00\xce*\x00\x00\x00\x00Menon'
Traceback (most recent call last):
  File "./tun.py", line 35, in <module>
    os.write(tun, bytes('hello'))
TypeError: string argument without an encoding
root@3c392a6f80b9:/volumes# 
```

Task 3 - Send IP packet to VPN Server through Tunnel

We try to enclose our IP Packet in UDP Payload so that we can show successful tunneling
First, We set up UDP server by writing tun_server.py which is UDP server program that recieves and displays received packets.

```
#!/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))
```

We also set the tun_client.py to send data to another computer using UDP socket program.

```
#!/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'menon%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))
# Create UDP socket
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:
                # Send the packet via the tunnel
                sock.sendto(packet, (10.9.0.11, 9090))
```

1. We run tun_client and server program in their respective shells

```
root@3c392a6f80b9:/volumes# tun_client.py
Interface Name: menon0
```

```
[03/02/24]seed@VM:~/.../volumes$ docksh 3c
root@3c392a6f80b9:/# ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
^C^C
--- 192.168.53.5 ping statistics ---
11 packets transmitted, 0 received, 100% packet loss, time 10221ms
```

We start pinging 192.168.53.5 and we see that server displays the packets.

```
root@d3f2298c65ed:/volumes# chmod a+x tun_server.py
root@d3f2298c65ed:/volumes# tun_server.py
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:38784 --> 0.0.0.0:9090
```

We see that the packet sent to 192.168.53.0/24 used menon0 interface and thus we see the packets sent from 192.168.53.99 ie TUN interface we set earlier.

2. When we try pinging hosts from 192.168.60.0/24 network packets are sent but nothing is shown on server screen because we have not set the routing configuration

We configure the routing table on client

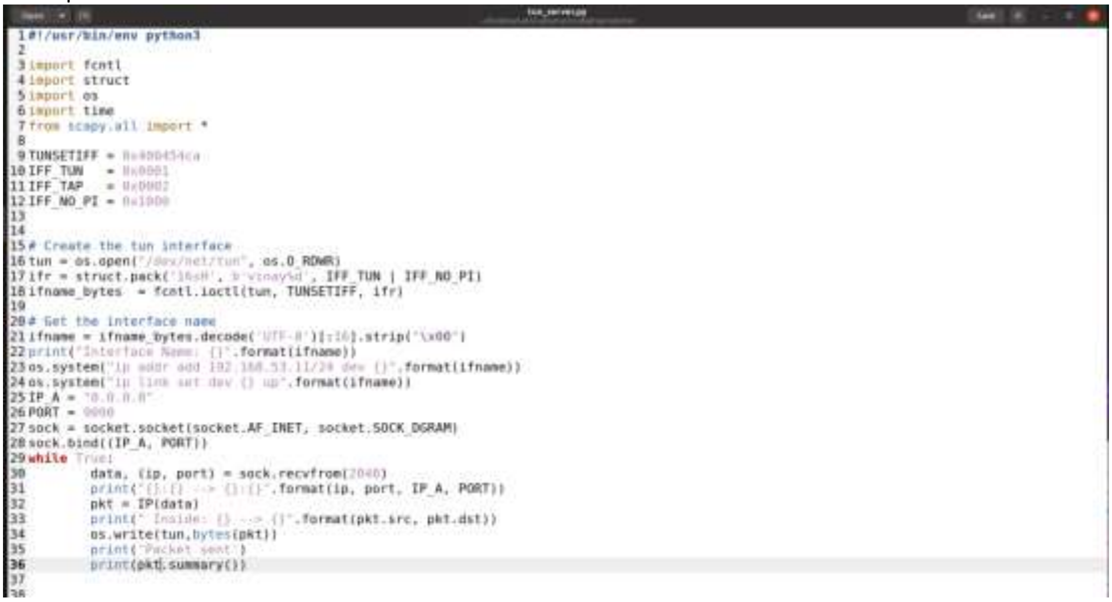
```
root@3c392a6f80b9:/#  ip route add 192.168.60.0/24  dev menon0 via 192.168.53.99
root@3c392a6f80b9:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
```

We see that the packets are displayed on server screen when we ping 192.168.60.5 but the packets are not received back.

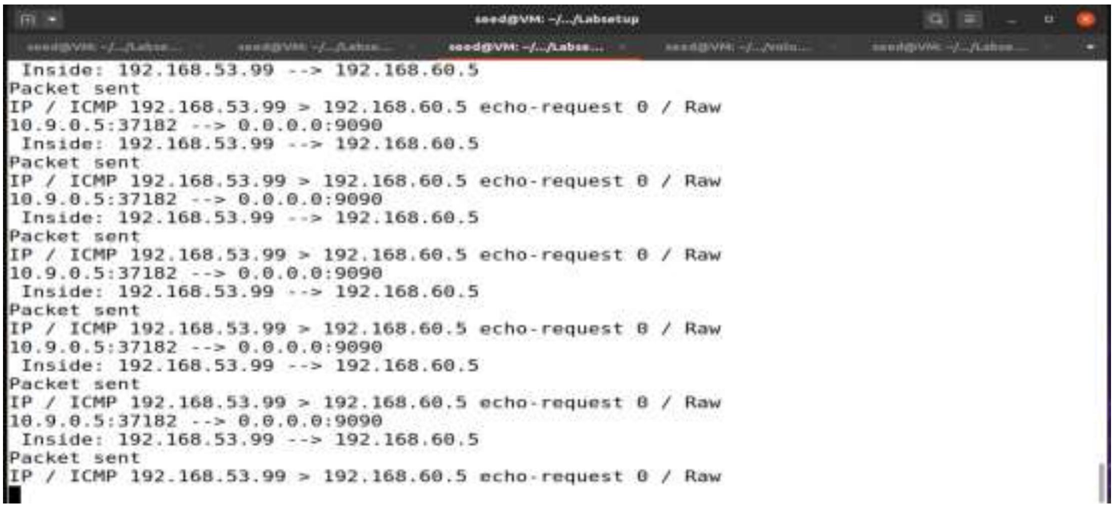
```
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38784 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
```

Task 4 - Set up VPN Server

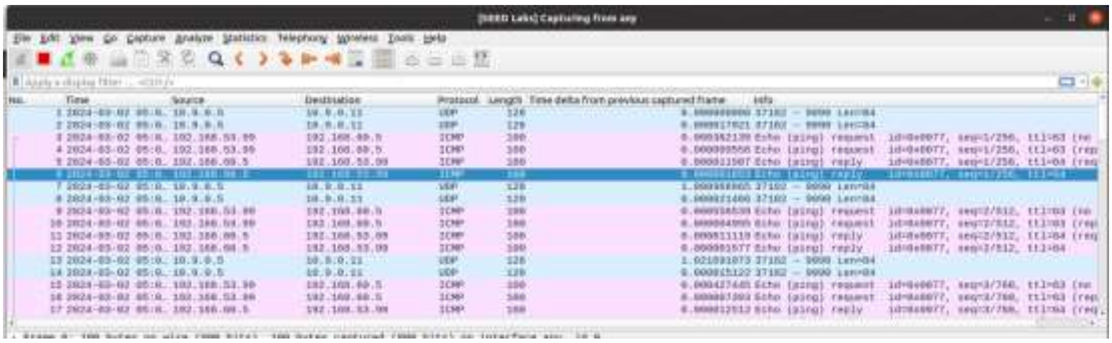
We set the ip forwarding to 1 so that packet can be sent from router and change the server code with creating TUN interface and configuring it, get data from socket,treat received data as ip packet and write packet to tun interface



We see that TUN interface displays the packets sent from Host u to v

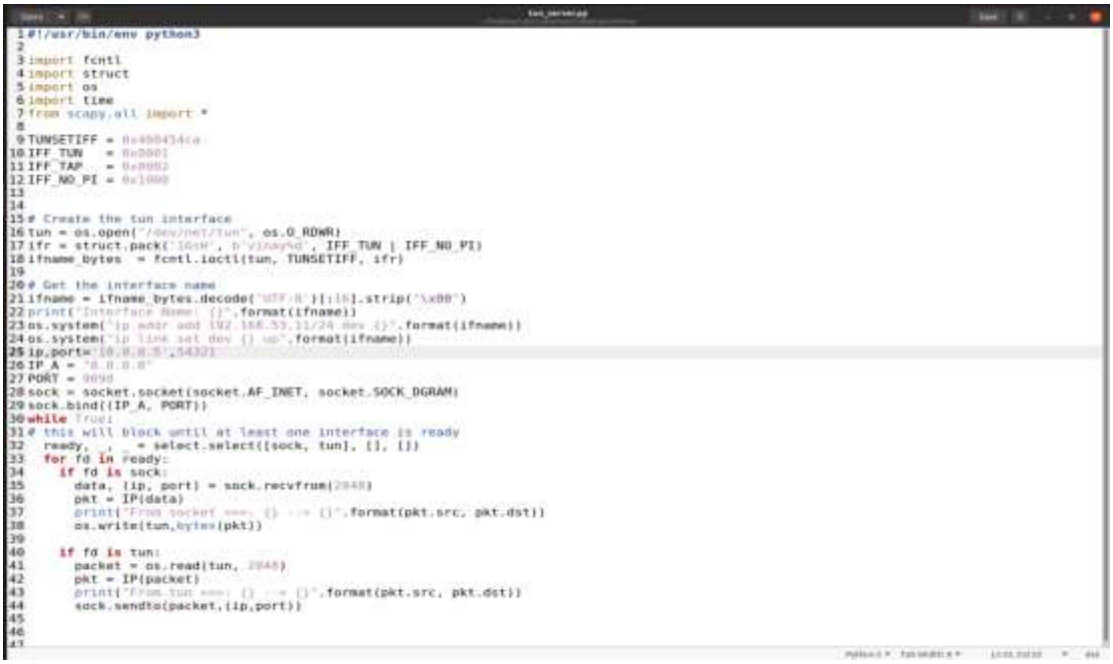


We see that packets arent received when we ping but the requests and replies can be seen through wireshark.

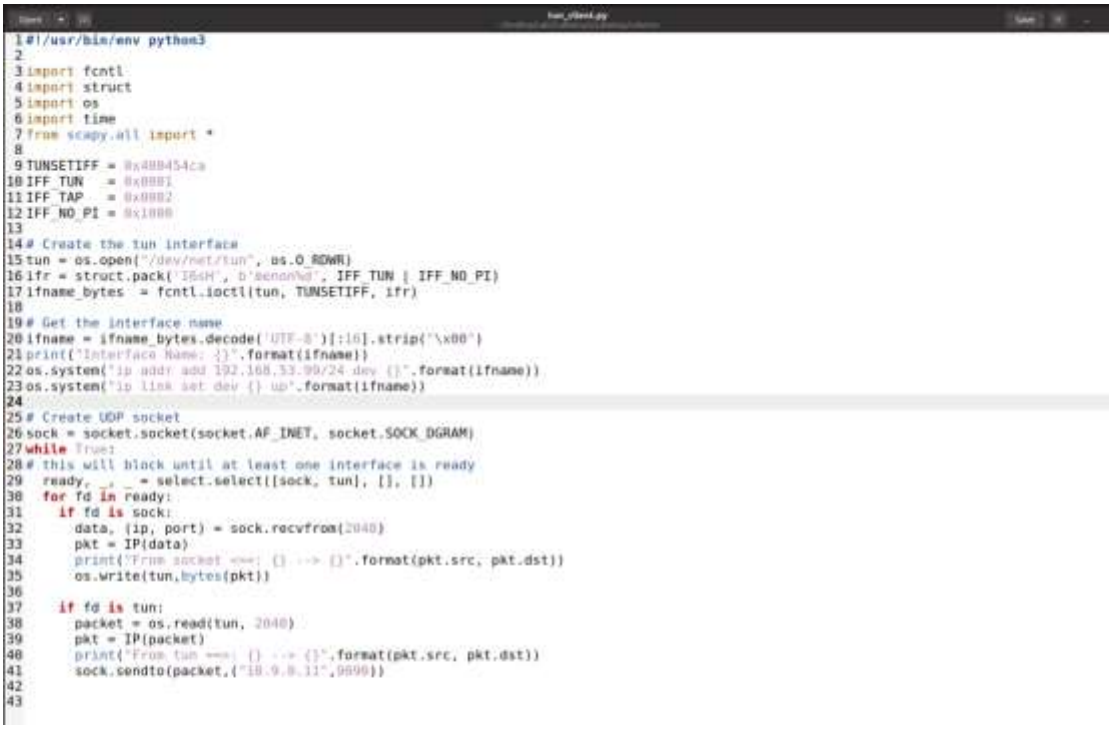


Task 5- Handling traffic in both directions

To handle traffic in both sides we program TUN client and server to read data from 2 interfaces, the TUN interface and socket interface.

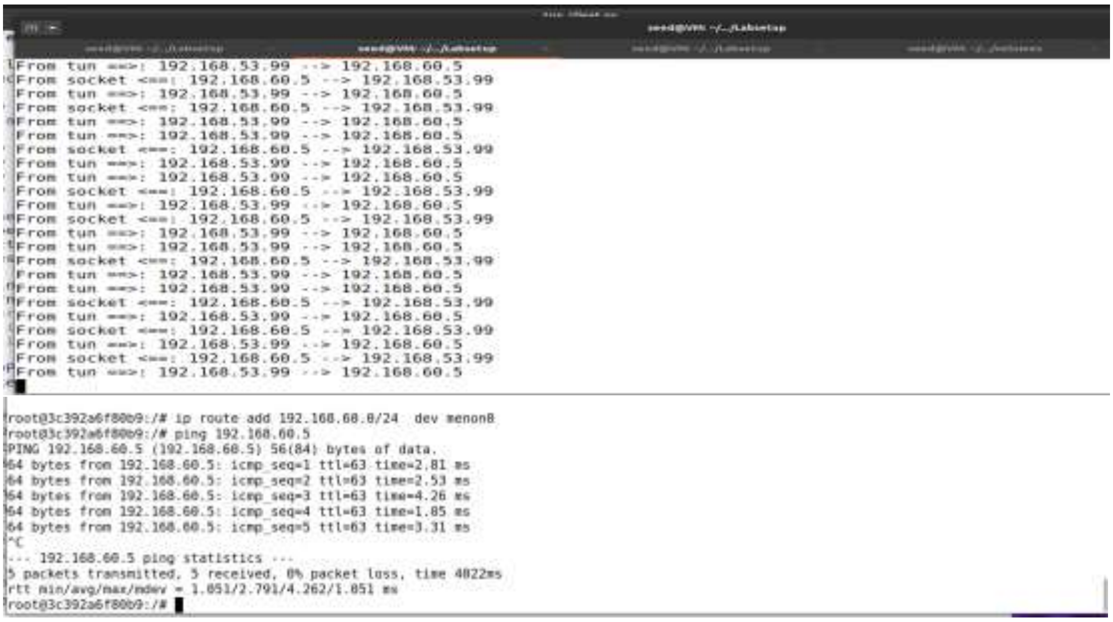


```python
#!/usr/bin/env python3

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
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'vinay%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))
ip,port='10.0.0.5',54321
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))
      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(packet,(ip,port))
```

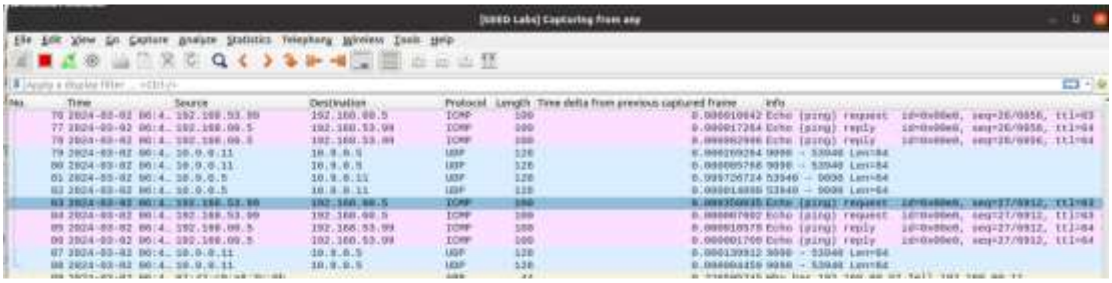


```python
#!/usr/bin/env python3

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
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'menon%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))

# Create UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
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(packet,("10.9.0.11",9090))
```

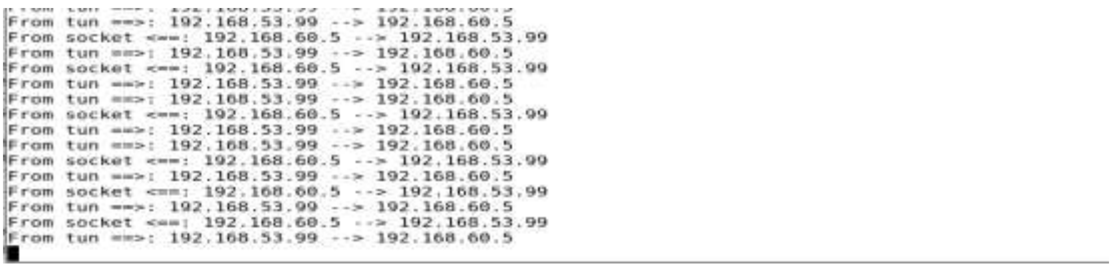
The code remains same for both client and server. Minor changes in source address and port.
Now after doing this when we run TUN client and server, we can successfully ping 192.168.60.5 from 10.9.0.5 successfully.

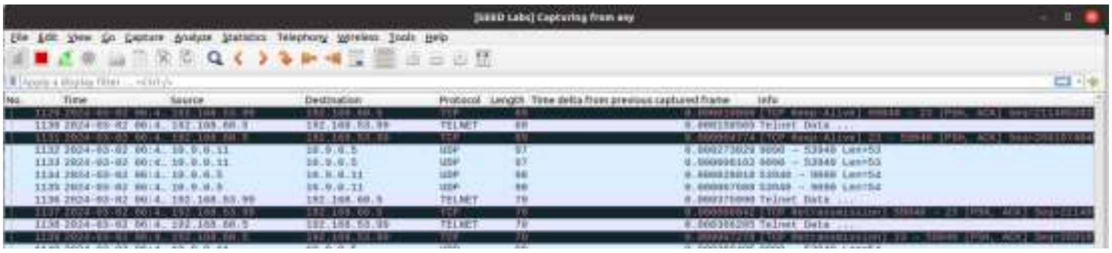


We can verify using wireshark.



We can do telnet and see connection is successful. Shows that remote login is possible.





Task 6 - Tunnel-Breaking Experiment

When we break tunnel by stopping TUN client or server while logged in with telnet. The text pauses and we are not able to type. Telnet gets stuck.

```
seed@891ab46e65c9:~$
```

Once we resume the tunnel we see that all the message cache gets displayed one by one in the shell when the telnet was stuck. Everything comes up together.

```
seed@891ab46e65c9:~$ pwd
/home/seed
seed@891ab46e65c9:~$ pwd
/home/seed
seed@891ab46e65c9:~$ pwd
/home/seed
seed@891ab46e65c9:~$
seed@891ab46e65c9:~$
seed@891ab46e65c9:~$
seed@891ab46e65c9:~$  fdffffffffff
-bash: fdffffffffff: command not found
seed@891ab46e65c9:~$ f
-bash: f: command not found
seed@891ab46e65c9:~$ f
-bash: f: command not found
seed@891ab46e65c9:~$ f
-bash: f: command not found
seed@891ab46e65c9:~$ f
```

Task 7 - Routing Experiment on Host V

We delete the default route entry and add the new path which can connect private network directly to VPN server by default.
This is done to showcase that private network can be few hops away from VPN server outside this lab. So we need to configure the routing rules.

```
root@891ab46e65c9:/# ip route show
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@891ab46e65c9:/# ip route del default
root@891ab46e65c9:/# ip route show
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@891ab46e65c9:/#
```

Now we add more specific rules to reach VPN server

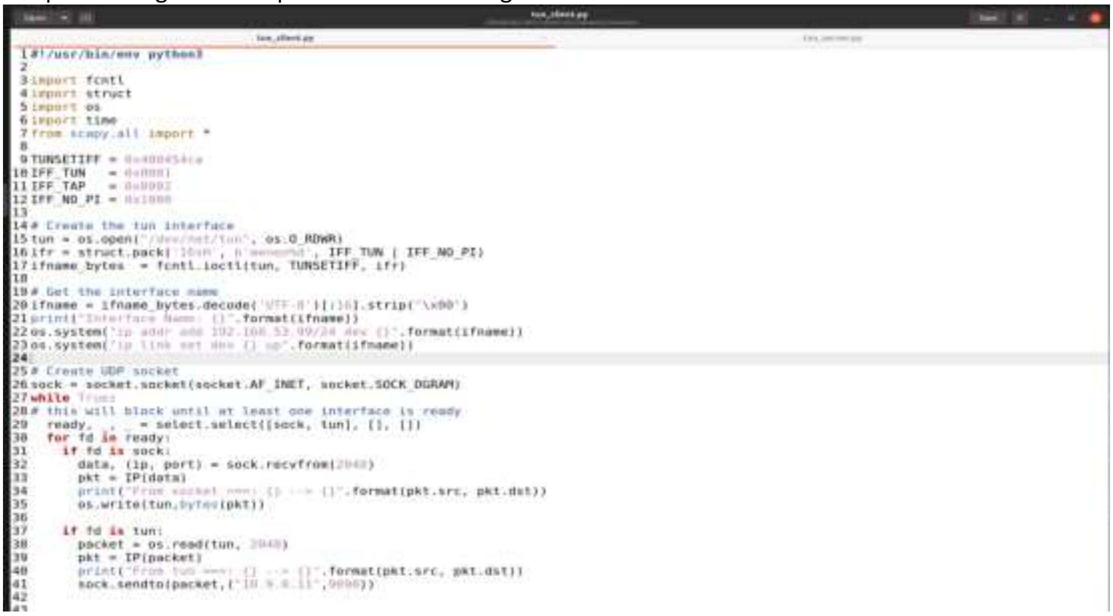
```
root@891ab46e65c9:/# ip route add 192.168.53.0/24 via 192.168.60.11
root@891ab46e65c9:/# ip route show
192.168.53.0/24 via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@891ab46e65c9:/#
```

Task 8 - VPN between private networks

We first setup all the new containers and try pinging Host v from Host u.

We see that the packets are not able to reach the other side of network.

So we write the tun_ client and tun_server code. Checking the sending ip address and cross verifying the ports along with tun ip and interface configuration.



We don't need to make any major changes to the previous codes. Now setting up the router settings in VPN Client and server containers.

We start the client and server programs first,

```
root@d3409e0cbad8:/# cd volumes
root@d3409e0cbad8:/volumes# tun_client.py
```

```
root@e1f7a8e2959c:/volumes# tun_server.py
Interface Name: vinay0
```

Now setting up router settings in VPN Client and server

```
seed@VM: ~/.../Labsetup
[03/03/24]seed@VM:~/.../Labsetup$ docksh d3
root@d3409e0cbad8:/# ip route add 192.168.60.0/24 dev menon0
root@d3409e0cbad8:/#
```

```
seed@VM: ~/.../Labsetup
[03/03/24]seed@VM:~/.../Labsetup$ docksh e1
root@e1f7a8e2959c:/# ip route add 192.168.50.0/24 dev vinay0
root@e1f7a8e2959c:/#
```

We use the following interfaces to reach the other network. Ie vinay0 for 192.168.50.0/24 and menon0 to reach 192.168.60.0/24.

We try pinging from host u 192.168.50.5 to host v 192.168.60.5

```
root@44e9bc40ef0b:/# 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=1.87 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=62 time=1.21 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=62 time=2.02 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=62 time=2.62 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=62 time=1.23 ms
^C
--- 192.168.60.5 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4016ms
rtt min/avg/max/mdev = 1.212/1.789/2.617/0.527 ms
root@44e9bc40ef0b:/#
```

We see the packets are sent and received which means tunneling was successful

Now looking at the interfaces

```
root@d3409e0cbad8:/volumes# tun_client.py
Interface Name: menon0
From tun ==>: 192.168.50.6 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.6
From tun ==>: 192.168.50.6 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.6
From tun ==>: 192.168.50.6 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.6
From tun ==>: 192.168.50.6 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.6
From tun ==>: 192.168.50.6 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.6
```

We see that packet sent from host u is routed to tun interface and from there it is routed to the server tun interface and then routed to host v.

```
root@e1f7a8e2959c:/volumes# tun_server.py
Interface Name: vinay0
From socket <==: 192.168.50.6 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.6
From socket <==: 192.168.50.6 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.6
From socket <==: 192.168.50.6 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.6
From socket <==: 192.168.50.6 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.6
From socket <==: 192.168.50.6 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.6
```

The same goes on the server side and thus the tunneling works between two private networks.

Task 9 - Experiment with the TAP Interface

We try to experiment with tap interface. So we write the following code with adding few changes. Making tap interface and making it read tap interface.

```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 tap interface
tap = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'menon%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))


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

Executing tap.py in client and seeing the interface name.

```
[03/04/24]seed@VM:~/.../Labsetup$ cd volumes
[03/04/24]seed@VM:~/.../volumes$ gedit tap.py
[03/04/24]seed@VM:~/.../volumes$ docksh 9c
root@9c701564c651:/# cd volumes
root@9c701564c651:/volumes# chmod a+x tap.py
root@9c701564c651:/volumes# tap.py
Interface Name: menon0
```

We try pinging 192.168.50.0/24 network and try to see if it is read in tap interface.

```
root@9c701564c651:/# ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 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
--- 192.168.53.5 ping statistics ---
7 packets transmitted, 0 received, +6 errors, 100% packet loss, time 6152ms
pipe 4
root@9c701564c651:/#
```

The packets are not received. And there is a ARP request sent on interface but no response.
So we need to make a spoof reponse to see if it works.

```
seed@VM: ~/.../volumes
[03/04/24]seed@VM:~/.../Labsetup$ cd volumes
[03/04/24]seed@VM:~/.../volumes$ gedit tap.py
[03/04/24]seed@VM:~/.../volumes$ docksh 9c
root@9c701564c651:/# cd volumes
root@9c701564c651:/volumes# chmod a+x tap.py
root@9c701564c651:/volumes# tap.py
Interface Name: menon0
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
```

So we add the following code to read and display the packet summary from tap interface and then display the fake response

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


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))
```

No we do arping multiple ip addresses after setting up the interface

```
root@9c701564c651:/volumes# tap.py
Interface Name: menon0
```

Arping to ip addresses 192.168.53.33

```
root@9c701564c651:/# arping -I menon0 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=941.585 usec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=1 time=1.447 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=2 time=879.204 usec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=3 time=1.060 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=4 time=1.225 msec
^C
--- 192.168.53.33 statistics ---
5 packets transmitted, 5 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 0.879/1.111/1.447/0.206 ms
root@9c701564c651:/#
```

We see that arping is successful as the responses are received. Now looking at the tap interface.

```
root@9c701564c651:/volumes# tap.py
Interface Name: menon0
--------------------------------
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
--------------------------------
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
```

Now for 1.2.3.4

```
root@9c701564c651:/# arping -I menon0 1.2.3.4
ARPING 1.2.3.4
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=0 time=7.182 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=1 time=1.322 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=2 time=2.799 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=3 time=1.004 msec
^C
--- 1.2.3.4 statistics ---
4 packets transmitted, 4 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 1.004/3.077/7.182/2.465 ms
root@9c701564c651:/#
```

We see packets are received
Looking at tap interface to see the responses generated

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

We are able to successfully read the ARPING from tap interface and generate spoofed responses and verify it using TAP interface. Thus experiment was successful.