

1. Overview:

Here's what I learned from the lab:

1. **Virtual Private Networks (VPNs):** I understood the concept of a VPN as a secure connection built over public networks, allowing private communication between devices as if they were on a physical private network.
2. **Tunneling Technology:** I explored how VPN tunneling works to encapsulate data packets and route them securely through a public network.
3. **TUN/TAP Interfaces:** I learned about the role of virtual interfaces like TUN and TAP, which are used to simulate network devices and facilitate VPN tunneling.
4. **IP Tunneling:** I gained an understanding of how data packets are encapsulated inside other packets for transmission in a VPN tunnel.
5. **Routing in VPNs:** I learned about the routing process and how traffic is directed through the VPN tunnel to reach the intended destination.

The lab provided hands-on experience with the tunneling aspect of VPNs, which helped solidify these concepts.

Takeaway: From this lab, I learned how VPNs create secure connections over public networks by forming private tunnels for data transfer. I now understand how TUN/TAP virtual interfaces work to simulate network devices and support tunneling. I also got a clearer picture of IP tunneling, which involves wrapping data packets for secure transmission, and how routing ensures the data reaches the correct destination. This lab focused on the tunneling process, giving me a solid understanding of its basics without covering encryption.

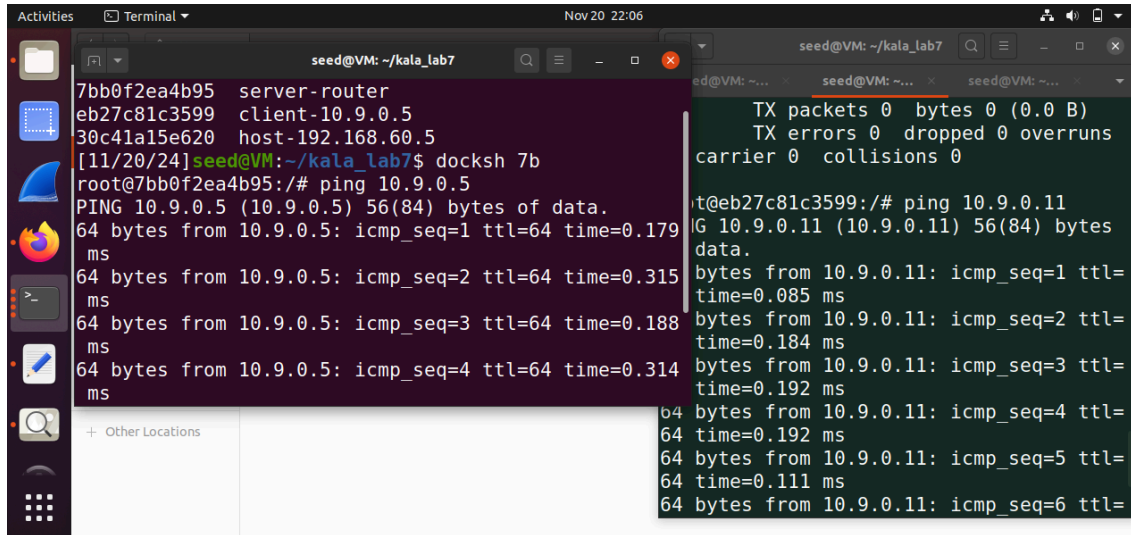
2.Task1: Network Setup

```
Nov 20 20:48
seed@VM: ~/kala_lab7
[11/19/24]seed@VM:~/kala_lab7$ dcbuild
VPN_Client uses an image, skipping
Host1 uses an image, skipping
Host2 uses an image, skipping
Router uses an image, skipping
[11/19/24]seed@VM:~/kala_lab7$ dcup
Starting server-router ... done
Starting host-192.168.60.5 ... done
Starting host-192.168.60.6 ... done
Starting client-10.9.0.5 ... done
Attaching to host-192.168.60.5, host-192.168.60.6, client-10.9.0.5, server-router
host-192.168.60.6 | * Starting internet superserver inetd [ OK ]
host-192.168.60.5 | * Starting internet superserver inetd [ OK ]
```

I successfully setup the lab environment.

```
Nov 19 10:27
seed@VM: ~/kala_lab7
[11/19/24]seed@VM:~/kala_lab7$ dockps
5ef5248be708 host-192.168.60.6
28e504f162b0 host-192.168.60.5
df075bbbf133 server-router
3282ba406426 client-10.9.0.5
[11/19/24]seed@VM:~/kala_lab7$ docksh 28
root@28e504f162b0:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
seed@VM:~/kala_lab7$ docksh 32
root@06426:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
```

I gave ping for client and the host V. we can clearly see that there is not ping between them.



```
Activities Terminal Nov 20 22:06
seed@VM: ~/kala_lab7
7bb0f2ea4b95 server-router
eb27c81c3599 client-10.9.0.5
30c41a15e620 host-192.168.60.5
[11/20/24]seed@VM:~/kala_lab7$ docksh 7b
root@7bb0f2ea4b95:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.179 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.315 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.188 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.314 ms
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns
carrier 0 collisions 0
t@eb27c81c3599:/# ping 10.9.0.11
IG 10.9.0.11 (10.9.0.11) 56(84) bytes
data.
bytes from 10.9.0.11: icmp_seq=1 ttl=
time=0.085 ms
bytes from 10.9.0.11: icmp_seq=2 ttl=
time=0.184 ms
bytes from 10.9.0.11: icmp_seq=3 ttl=
time=0.192 ms
64 bytes from 10.9.0.11: icmp_seq=4 ttl=
64 time=0.192 ms
64 bytes from 10.9.0.11: icmp_seq=5 ttl=
64 time=0.111 ms
64 bytes from 10.9.0.11: icmp_seq=6 ttl=
```

Then after i ping the server and client you can see that there is ping going on between them.

Takeaway: In this lab, I learned to simulate a VPN setup where Host U communicates with Host V via a VPN Server while maintaining network isolation. I used Docker Compose commands and aliases to build, manage, and access containers efficiently. The shared folder setup with Docker volumes made file sharing between the VM and containers seamless. I also practiced packet sniffing with `tcpdump` and Wireshark to analyze network traffic and debug connectivity issues.

3.Task2: Create and Configure TUN Interface

```

11/20/24]seed@VM: ~/kala_lab7$ docksh eb
root@eb27c81c3599:~# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UN
KNOWN 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
7: kala0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 q
disc fq_codel state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global kala0
        valid lft forever preferred_lft forever
8: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qd
isc fq_codel state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global tun0
        valid lft forever preferred_lft forever
10: eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdis
c noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-ne
tnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
        valid lft forever preferred_lft forever
root@eb27c81c3599:~#

```

I went to the client container and ran the tun.py code i got the interface name. I went to ip address of client i can see the tun0.

Takeaway: I learned that TUN/TAP technologies simulate virtual network devices, with TUN handling IP packets (layer 3) and TAP handling Ethernet frames (layer 2). These interfaces enable user-space programs to exchange packets with the OS network stack. Using the provided Python code, I practiced reading and writing packets through TUN/TAP interfaces, understanding their role in virtual networking.

Task2.a:NameoftheInterface:

```

root@3282ba406426:~# cd volumes
root@3282ba406426:/volumes# ls
tun.py  tun_client.py  tun_server.py
root@3282ba406426:/volumes# chmod a+x tun.py
root@3282ba406426:/volumes# tun.py
Interface Name: kala0

```

Now i changed the interface name to my last name. And ran code. And got expected result.

```

Text Editor Nov 19 10:31
tun.py tun_server.py
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'kala%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 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
23 os.system("ip link set dev {} up".format(ifname))
24
25 while True:
26     # Get a packet from the tun interface
27     packet = os.read(tun, 2048)
28     if packet:
29         ip = IP(packet)
30         print(ip.summary())
31
32     # Send out a spoof packet using the tun interface
33     newip = IP(src='192.168.53.3', dst=ip.src)
34     newpkt = newip/ip.payload
35     arb_data = b'Any arbitrary data'
36     os.write(tun, bytes(newpkt))
37

```

This is where i updated my code.

```

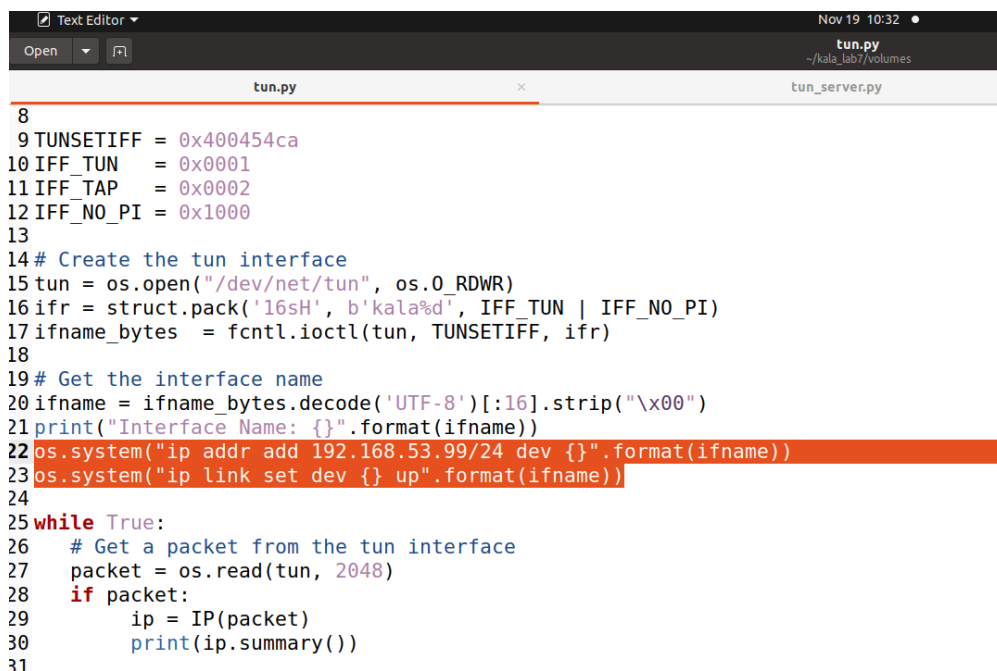
seed@VM: ~/kala_lab7
[11/20/24] seed@VM:~/kala_lab7$ docksh eb
root@eb27c81c3599:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UN
KNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
7: kala0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 q
disc fq_codel state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global kala0
        valid_lft forever preferred_lft forever
8: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qd
isc fq_codel state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global tun0
        valid_lft forever preferred_lft forever
10: eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdis
c noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-ne
tsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
        valid_lft forever preferred_lft forever
root@eb27c81c3599:/#

```

I successfully added interface with my name.

Takeaway: In this task, I learned how to work with the tun.py program to create a virtual TUN interface. After running the program with root privileges, I observed the creation of the tun0 interface using the ip address command. My task was to modify the program to use my last name (or its first five characters) as the prefix for the interface name instead of the default tun. This helped me understand how to customize TUN interfaces and verify their creation on the system.

Task2.b:SetuptheTUNInterface



```
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'kala%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 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
23 os.system("ip link set dev {} up".format(ifname))
24
25 while True:
26     # Get a packet from the tun interface
27     packet = os.read(tun, 2048)
28     if packet:
29         ip = IP(packet)
30         print(ip.summary())
31
```

I have an interface but it needs an inet so to do that i added the inet in the code.

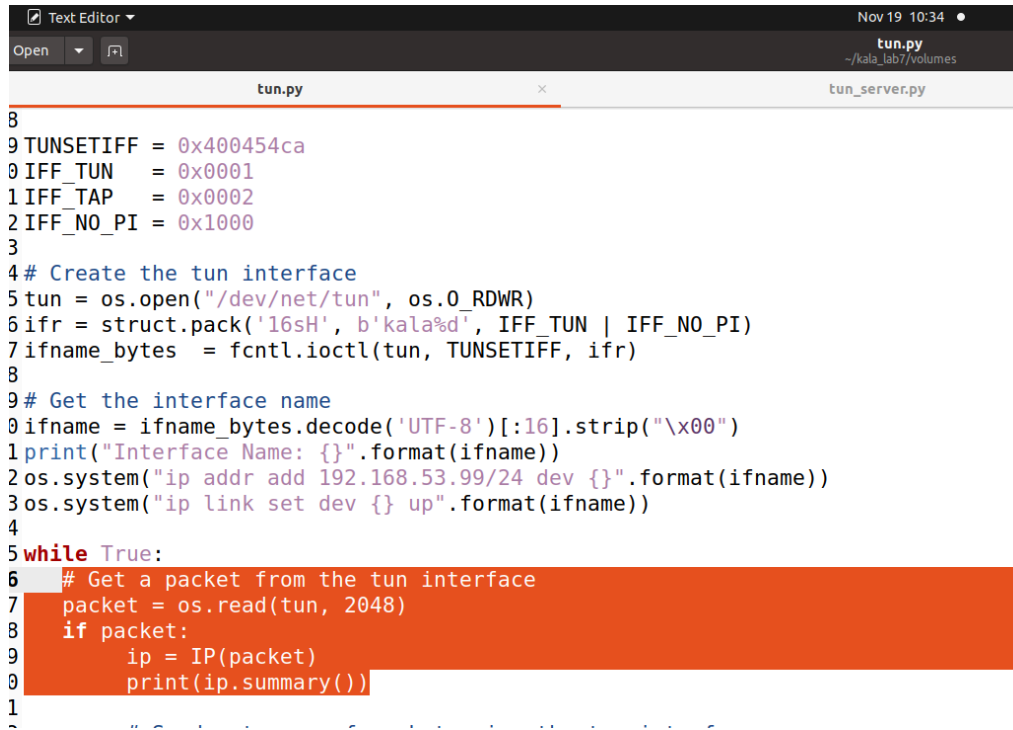
```
Nov 19 10:33 •
seed@VM: ~/kala_lab7
seed@VM: ~/kala_lab7 x seed@VM: ~/kala_lab7 x seed@VM: ~/kala_lab7 x
^C
--- 192.168.60.5 ping statistics ---
82 packets transmitted, 0 received, 100% packet loss, time 83769ms

root@3282ba406426:/# exit
exit
[11/19/24]seed@VM:~/kala_lab7$ docksh 32
root@3282ba406426:/# 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
3: kala0: <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 kala0
        valid_lft forever preferred_lft forever
173: eth0@if174: <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@3282ba406426:/#
```

Now i have an inet to my interface.

Takeaway: In this task, I learned that after creating a TUN interface, it is not immediately usable until it is assigned an IP address and brought up. I used the `ip addr add` command to assign an IP address and the `ip link set` command to bring the interface up. To automate this process, I modified the `tun.py` program to include these commands. After running the configuration, I observed that the `tun0` interface was no longer in the down state and had an assigned IP address. This made the interface active and usable, unlike before when it was inactive and without an IP.

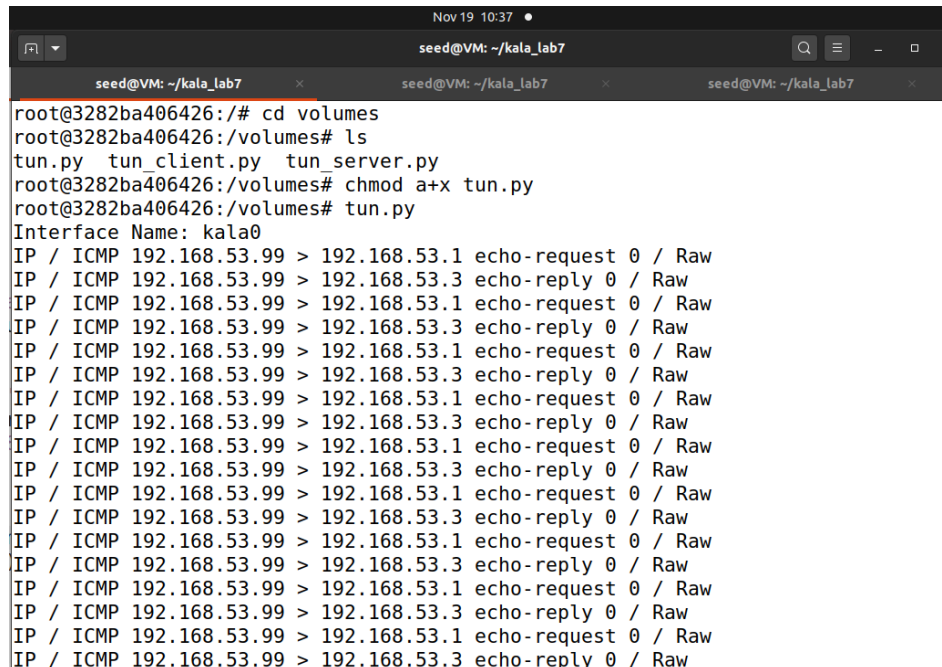
Task 2.c: Read from the TUN Interface



```
Text Editor Nov 19 10:34
tun.py
~/kala_lab7/volumes

8
9 TUNSETIFF = 0x400454ca
9 IFF_TUN = 0x0001
1 IFF_TAP = 0x0002
2 IFF_NO_PI = 0x1000
3
4 # Create the tun interface
5 tun = os.open("/dev/net/tun", os.O_RDWR)
6 ifr = struct.pack('16sH', b'kala%d', IFF_TUN | IFF_NO_PI)
7 ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
8
9 # Get the interface name
9 ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
1 print("Interface Name: {}".format(ifname))
2 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
3 os.system("ip link set dev {} up".format(ifname))
4
5 while True:
6     # Get a packet from the tun interface
7     packet = os.read(tun, 2048)
8     if packet:
9         ip = IP(packet)
9         print(ip.summary())
1
~
```

I updated the code to read the packets



```
Nov 19 10:37
seed@VM: ~/kala_lab7
seed@VM: ~/kala_lab7
seed@VM: ~/kala_lab7

root@3282ba406426:/# cd volumes
root@3282ba406426:/volumes# ls
tun.py tun_client.py tun_server.py
root@3282ba406426:/volumes# chmod a+x tun.py
root@3282ba406426:/volumes# tun.py
Interface Name: kala0
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
```

And my interface is reading the packets


```
Nov 19 10:39 •
seed@VM: ~/kala_lab7
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@3282ba406426:/# ping 192.168.53.1
PING 192.168.53.1 (192.168.53.1) 56(84) bytes of data.
64 bytes from 192.168.53.3: icmp_seq=1 ttl=64 time=8.51 ms
64 bytes from 192.168.53.3: icmp_seq=2 ttl=64 time=6.60 ms
64 bytes from 192.168.53.3: icmp_seq=3 ttl=64 time=9.41 ms
64 bytes from 192.168.53.3: icmp_seq=4 ttl=64 time=12.6 ms
64 bytes from 192.168.53.3: icmp_seq=5 ttl=64 time=5.16 ms
64 bytes from 192.168.53.3: icmp_seq=6 ttl=64 time=3.22 ms
64 bytes from 192.168.53.3: icmp_seq=7 ttl=64 time=5.60 ms
64 bytes from 192.168.53.3: icmp_seq=8 ttl=64 time=4.48 ms
64 bytes from 192.168.53.3: icmp_seq=9 ttl=64 time=4.39 ms
64 bytes from 192.168.53.3: icmp_seq=10 ttl=64 time=4.31 ms
64 bytes from 192.168.53.3: icmp_seq=11 ttl=64 time=4.60 ms
64 bytes from 192.168.53.3: icmp_seq=12 ttl=64 time=4.13 ms
64 bytes from 192.168.53.3: icmp_seq=13 ttl=64 time=4.16 ms
64 bytes from 192.168.53.3: icmp_seq=14 ttl=64 time=3.74 ms
64 bytes from 192.168.53.3: icmp_seq=15 ttl=64 time=3.29 ms
64 bytes from 192.168.53.3: icmp_seq=16 ttl=64 time=3.22 ms
```

For ip address 192.168.53.1 i got the ping

```
Nov 19 10:40 •
seed@VM: ~/kala_lab7
64 bytes from 192.168.53.3: icmp_seq=23 ttl=64 time=1.85 ms
^C
--- 192.168.53.1 ping statistics ---
23 packets transmitted, 23 received, 0% packet loss, time 22088ms
rtt min/avg/max/mdev = 1.761/4.422/12.626/2.643 ms
root@3282ba406426:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
```

But for the host v i didnt got the ping because there is no tunnel.

Takeaway: In this task, I learned how to read IP packets from the TUN interface and turn them into Scapy IP objects. By changing the `tun.py` program, I was able to capture packets from the interface and print their details, like the source and destination addresses.

Task 2.d: Write to the TUN Interface

```
Text Editor Nov 19 10:41
tun.py tun_server.py

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'kala%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 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
23 os.system("ip link set dev {} up".format(ifname))
24
25 while True:
26     # Get a packet from the tun interface
27     packet = os.read(tun, 2048)
28     if packet:
29         ip = IP(packet)
30         print(ip.summary())
31
32     # Send out a spoof packet using the tun interface
33     newip = IP(src='192.168.53.3', dst=ip.src)
34     newpkt = newip/ip.payload
35     arb_data = b'Any arbitrary data'
36     os.write(tun, bytes(newpkt))
37
```

I added some code and gave the ip address to write in the interface.

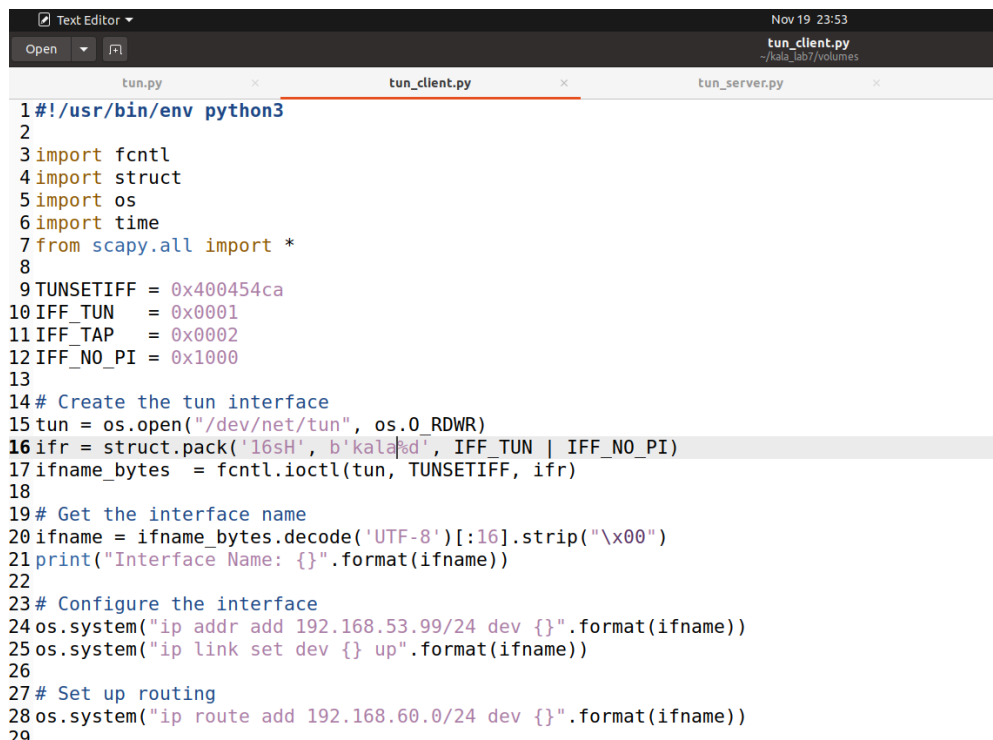
```
seed@VM: ~/kala_lab7
seed@VM: ~/kala_lab7 x seed@VM: ~/kala_lab7 x seed@VM: ~/kala_lab7 x

root@3282ba406426:/volumes# chmod a+x tun.py
root@3282ba406426:/volumes# tun.py
Interface Name: kala0
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
```

I got the output.

Takeaway: In this task, I learned how to modify the `tun.py` program to write data to the TUN interface. I created an ICMP echo reply for incoming echo requests and experimented with writing arbitrary data to the interface. This helped me understand how to send custom packets and observe the system's response.

Task3: Send the IP Packet to VPN Server Through a Tunnel



```
Text Editor Nov 19 23:53
Open tun_client.py ~/kala_lab7/volumes
tun.py tun_client.py tun_server.py
1#!/usr/bin/env python3
2
3import fcntl
4import struct
5import os
6import time
7from scapy.all import *
8
9TUNSETIFF = 0x400454ca
10IFF_TUN = 0x0001
11IFF_TAP = 0x0002
12IFF_NO_PI = 0x1000
13
14# Create the tun interface
15tun = os.open("/dev/net/tun", os.O_RDWR)
16ifr = struct.pack('16sH', b'kalad', IFF_TUN | IFF_NO_PI)
17ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
18
19# Get the interface name
20ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
21print("Interface Name: {}".format(ifname))
22
23# Configure the interface
24os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
25os.system("ip link set dev {} up".format(ifname))
26
27# Set up routing
28os.system("ip route add 192.168.60.0/24 dev {}".format(ifname))
29
```

For task3, i used the tun_clinet and the tun_server code.

```

Text Editor Nov 19 23:53
tun_server.py
~/kala_lab7/volumes

tun.py tun_client.py tun_server.py
9 PORT = 9090
10
11 TUNSETIFF = 0x400454ca
12 IFF_TUN = 0x0001
13 IFF_TAP = 0x0002
14 IFF_NO_PI = 0x1000
15
16 # Create a tun interface
17 tun = os.open("/dev/net/tun", os.O_RDWR)
18 ifr = struct.pack('16sH', b'kala%d', IFF_TUN | IFF_NO_PI)
19 ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
20 ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
21 print("Interface Name: {}".format(ifname))
22
23 # Set up the tun interface
24 os.system("ip addr add 192.168.53.1/24 dev {}".format(ifname))
25 os.system("ip link set dev {} up".format(ifname))
26
27 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
28 sock.bind((IP_A, PORT))
29
30 while True:
31     data, (ip, port) = sock.recvfrom(2048)
32     pkt = IP(data)
33     print("{}:{}".format(ip, port, IP_A, PORT))
34     print("    Inside: {} --> {}".format(pkt.src, pkt.dst))
35     os.write(tun, bytes(pkt))
36
37

```

After running the server code and i opened the interface

```

Nov 19 23:56
seed@VM: ~/kala_lab7

seed@VM: ~/kala_lab7
[11/19/24]seed@VM:~/kala_lab7$ dockps
30fbd41c61e4 host-192.168.60.6
7bb0f2ea4b95 server-router
eb27c81c3599 client-10.9.0.5
30c41a15e620 host-192.168.60.5
[11/19/24]seed@VM:~/kala_lab7$ docksh 7b
root@7bb0f2ea4b95:/# cd volumes
root@7bb0f2ea4b95:/volumes# chmod a+x tun_server.py
root@7bb0f2ea4b95:/volumes# tun_server.py
Interface Name: kala0
10.9.0.5:33375 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:33375 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:33375 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:33375 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:33375 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:33375 --> 0.0.0.0:9090

```

```
Nov 19 23:57
seed@VM: ~/kala_lab7
packet = os.read(tun, 2048)
KeyboardInterrupt

root@eb27c81c3599:/volumes# chmod a+x tun_client.py
root@eb27c81c3599:/volumes# tun_client.py
Interface Name: kala0
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
```

And i did the same for the client program.

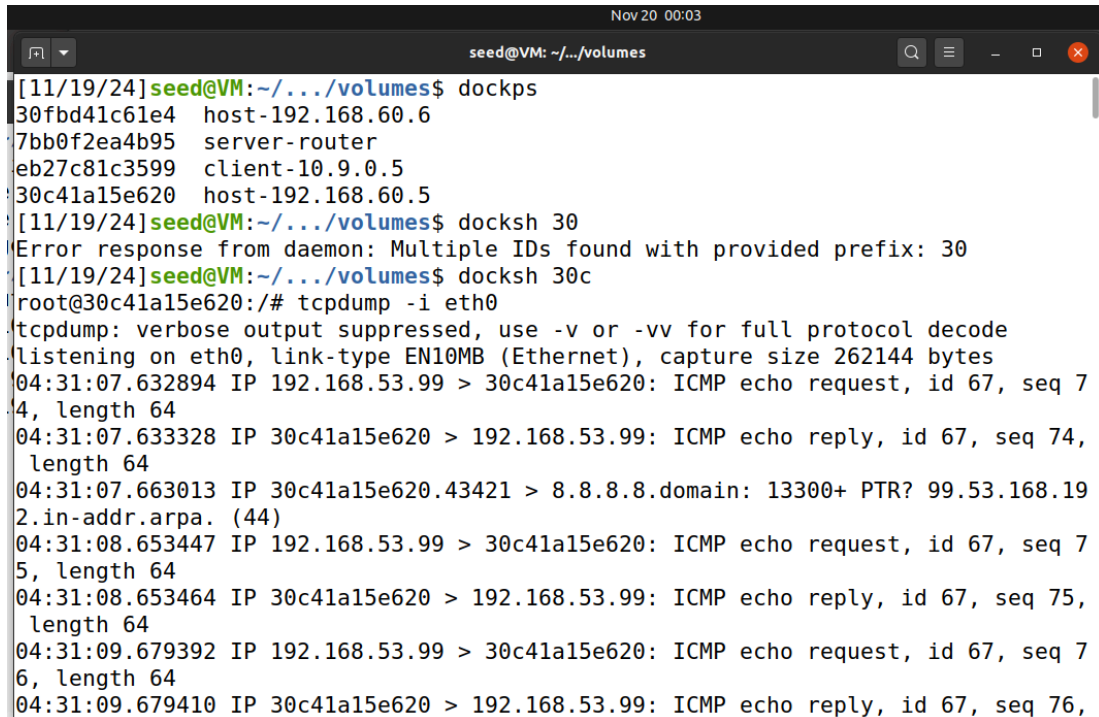
```
Nov 19 23:58
seed@VM: ~/kala_lab7
[11/19/24]seed@VM:~/kala_lab7$ dockkps
30fbd41c61e4 host-192.168.60.6
7bb0f2ea4b95 server-router
eb27c81c3599 client-10.9.0.5
30c41a15e620 host-192.168.60.5
[11/19/24]seed@VM:~/kala_lab7$ docksh eb
root@eb27c81c3599:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
--- 192.168.60.5 ping statistics ---
14 packets transmitted, 0 received, 100% packet loss, time 13296ms

root@eb27c81c3599:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
```

And i gave the host v ip address in client and i got the desired output.

Takeaway: In this task, I learned how to tunnel IP packets using UDP. The `tun_server.py` receives and prints tunneled IP packets, while `tun_client.py` sends packets from the TUN interface through UDP. I tested the tunnel by pinging the 192.168.53.0/24 network and adjusted the routing table to route packets to the 192.168.60.0/24 network. This helped me understand IP tunneling and routing through a tunnel.

Task4: Set Up the VPNServer

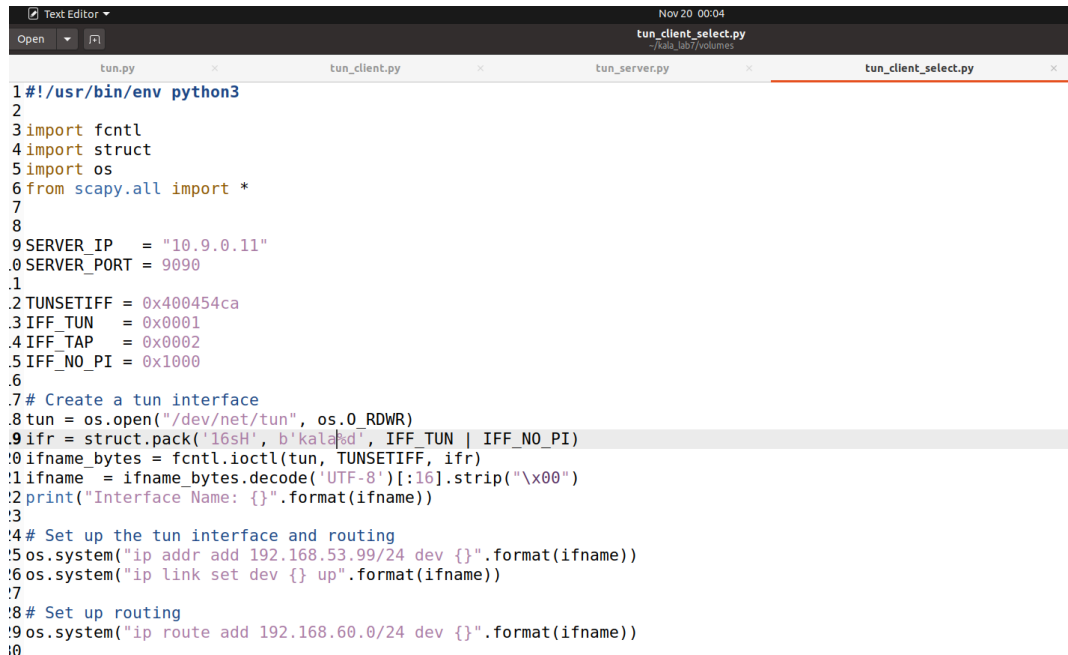


```
Nov 20 00:03
seed@VM: ~/.../volumes
[11/19/24]seed@VM:~/.../volumes$ dockps
30fbd41c61e4  host-192.168.60.6
7bb0f2ea4b95  server-router
eb27c81c3599  client-10.9.0.5
30c41a15e620  host-192.168.60.5
[11/19/24]seed@VM:~/.../volumes$ docksh 30
Error response from daemon: Multiple IDs found with provided prefix: 30
[11/19/24]seed@VM:~/.../volumes$ docksh 30c
root@30c41a15e620:/# tcpdump -i eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
04:31:07.632894 IP 192.168.53.99 > 30c41a15e620: ICMP echo request, id 67, seq 74, length 64
04:31:07.633328 IP 30c41a15e620 > 192.168.53.99: ICMP echo reply, id 67, seq 74, length 64
04:31:07.663013 IP 30c41a15e620.43421 > 8.8.8.8.domain: 13300+ PTR? 99.53.168.192.in-addr.arpa. (44)
04:31:08.653447 IP 192.168.53.99 > 30c41a15e620: ICMP echo request, id 67, seq 75, length 64
04:31:08.653464 IP 30c41a15e620 > 192.168.53.99: ICMP echo reply, id 67, seq 75, length 64
04:31:09.679392 IP 192.168.53.99 > 30c41a15e620: ICMP echo request, id 67, seq 76, length 64
04:31:09.679410 IP 30c41a15e620 > 192.168.53.99: ICMP echo reply, id 67, seq 76,
```

I went to the Host V and checked the ICMP packets using the tcpdump

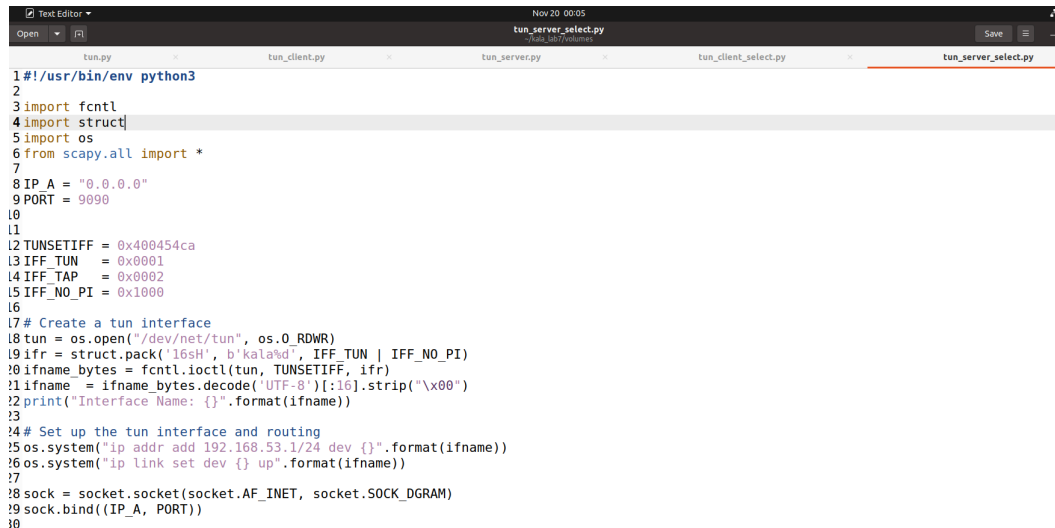
Takeaway: In this task, I learned how to modify `tun_server.py` to create a TUN interface, configure it, and forward packets received from the tunnel to the kernel for routing. I also learned how to enable IP forwarding to allow the VPN server to act as a gateway. After setting everything up, I tested the system by pinging Host V from Host U. I observed that the ICMP echo request packets successfully reached Host V, as shown using Wireshark or tcpdump. However, the reply didn't reach Host U yet, as further setup was still needed.

Task5: Handling Traffic in Both Directions



```
1#!/usr/bin/env python3
2
3import fcntl
4import struct
5import os
6from scapy.all import *
7
8
9SERVER_IP = "10.9.0.11"
10SERVER_PORT = 9090
11
12TUNSETIFF = 0x400454ca
13IFF_TUN = 0x0001
14IFF_TAP = 0x0002
15IFF_NO_PI = 0x1000
16
17# Create a tun interface
18tun = os.open("/dev/net/tun", os.O_RDWR)
19ifr = struct.pack('16sH', b'kalad', IFF_TUN | IFF_NO_PI)
20ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
21ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
22print("Interface Name: {}".format(ifname))
23
24# Set up the tun interface and routing
25os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
26os.system("ip link set dev {} up".format(ifname))
27
28# Set up routing
29os.system("ip route add 192.168.60.0/24 dev {}".format(ifname))
30
```

Till now we have only one directional, to do bidirectional, i used the code for tun_client_select and server_select code.



```
1#!/usr/bin/env python3
2
3import fcntl
4import struct
5import os
6from scapy.all import *
7
8IP_A = "0.0.0.0"
9PORT = 9090
10
11TUNSETIFF = 0x400454ca
12IFF_TUN = 0x0001
13IFF_TAP = 0x0002
14IFF_NO_PI = 0x1000
15
16# Create a tun interface
17tun = os.open("/dev/net/tun", os.O_RDWR)
18ifr = struct.pack('16sH', b'kalad', IFF_TUN | IFF_NO_PI)
19ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
20ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
21print("Interface Name: {}".format(ifname))
22
23# Set up the tun interface and routing
24os.system("ip addr add 192.168.53.1/24 dev {}".format(ifname))
25os.system("ip link set dev {} up".format(ifname))
26
27sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
28sock.bind((IP_A, PORT))
29
30
```

This the server_select code.


```
Nov 20 00:10
seed@VM: ~/kala_lab7
seed@VM: ~/kala_lab7
seed@VM: ~/kala_lab7
--- 192.168.60.5 ping statistics ---
126 packets transmitted, 0 received, 100% packet loss, time 128055ms

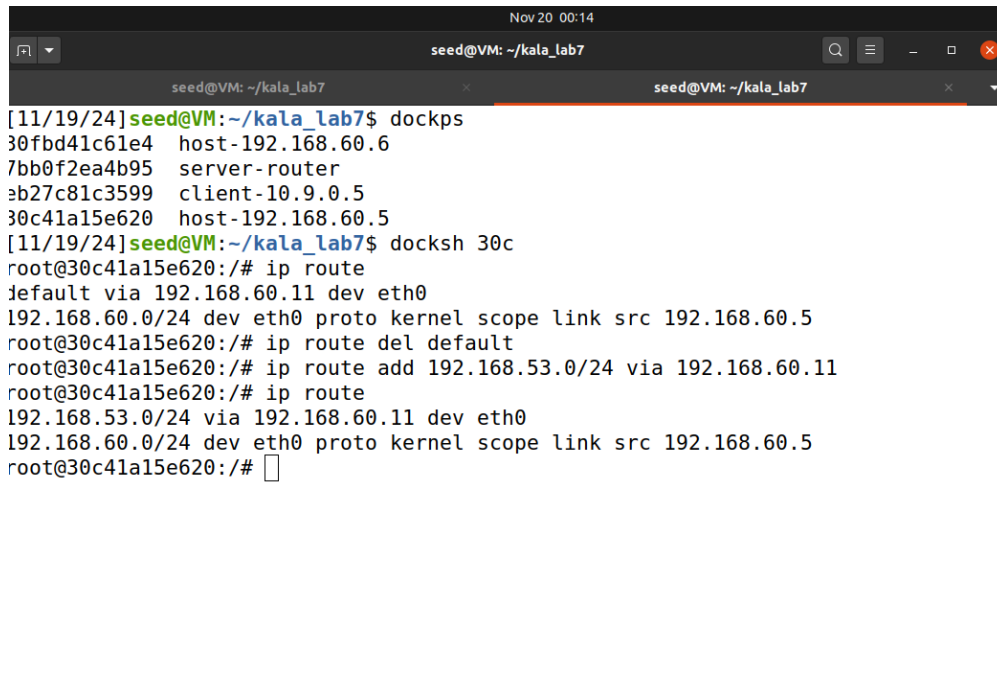
root@eb27c81c3599:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
i4 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=9.61 ms
i4 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=4.81 ms
i4 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=4.40 ms
i4 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=2.11 ms
i4 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=2.89 ms
i4 bytes from 192.168.60.5: icmp_seq=6 ttl=63 time=3.25 ms
i4 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=3.83 ms
i4 bytes from 192.168.60.5: icmp_seq=8 ttl=63 time=5.33 ms
i4 bytes from 192.168.60.5: icmp_seq=9 ttl=63 time=3.85 ms
i4 bytes from 192.168.60.5: icmp_seq=10 ttl=63 time=2.40 ms
i4 bytes from 192.168.60.5: icmp_seq=11 ttl=63 time=4.81 ms
i4 bytes from 192.168.60.5: icmp_seq=12 ttl=63 time=3.78 ms
i4 bytes from 192.168.60.5: icmp_seq=13 ttl=63 time=2.25 ms
i4 bytes from 192.168.60.5: icmp_seq=14 ttl=63 time=4.28 ms
i4 bytes from 192.168.60.5: icmp_seq=15 ttl=63 time=2.76 ms
i4 bytes from 192.168.60.5: icmp_seq=16 ttl=63 time=3.18 ms
```

I pinged the 198.168.60.5 and i got the output

You can see that i successfully logged into the Host V

Takeaway: I learned how to make the tunnel bidirectional by modifying the TUN client and server programs to read data from both the TUN interface and the socket interface. By polling these interfaces, the programs can handle incoming data from either source without wasting CPU resources. After these changes, I tested the tunnel with ping and Telnet, using Wireshark to confirm that the tunnel worked for both sending and receiving traffic.

Task7: Routing Experiment on Host V

A terminal window titled 'Nov 20 00:14' with two tabs labeled 'seed@VM: ~/kala_lab7'. The active tab shows a series of commands and their outputs. The user 'seed@VM' runs 'dockps', listing four containers: '30fbd41c61e4 host-192.168.60.6', '7bb0f2ea4b95 server-router', 'eb27c81c3599 client-10.9.0.5', and '30c41a15e620 host-192.168.60.5'. Then, 'docksh 30c' is run, spawning a shell on container '30c41a15e620'. Inside this shell, the user 'root' runs 'ip route', showing the default route 'default via 192.168.60.11 dev eth0'. Then, 'ip route del default' is run, removing the default route. Finally, 'ip route add 192.168.53.0/24 via 192.168.60.11' is run, adding a specific route for the 192.168.53.0/24 network. The final 'ip route' command shows the new configuration: '192.168.53.0/24 via 192.168.60.11 dev eth0' and '192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5'.

```
[11/19/24]seed@VM:~/kala_lab7$ dockps
30fbd41c61e4  host-192.168.60.6
7bb0f2ea4b95  server-router
eb27c81c3599  client-10.9.0.5
30c41a15e620  host-192.168.60.5
[11/19/24]seed@VM:~/kala_lab7$ docksh 30c
root@30c41a15e620:/# 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@30c41a15e620:/# ip route del default
root@30c41a15e620:/# ip route add 192.168.53.0/24 via 192.168.60.11
root@30c41a15e620:/# ip route
192.168.53.0/24 via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@30c41a15e620:/#
```

I went to the host V and deleted the default ip address and added the new ip address.

Takeaway: I learned that in a real VPN setup, ensuring return traffic reaches the VPN server requires proper routing configuration. In the lab, the default route on Host V was removed, and a more specific route was added to direct return traffic to the VPN server. This simulates how private networks must configure their routing tables to ensure packets reach the correct destination, especially when the VPN server is not the default gateway.