CPS 472/572 Fall 2024
Prof: Zhongmei Yao
Lab 7 Report: VPN Tunneling Lab

Dinesh Kumar Kala (kalad1@udayton.edu) Student ID:101745354

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.

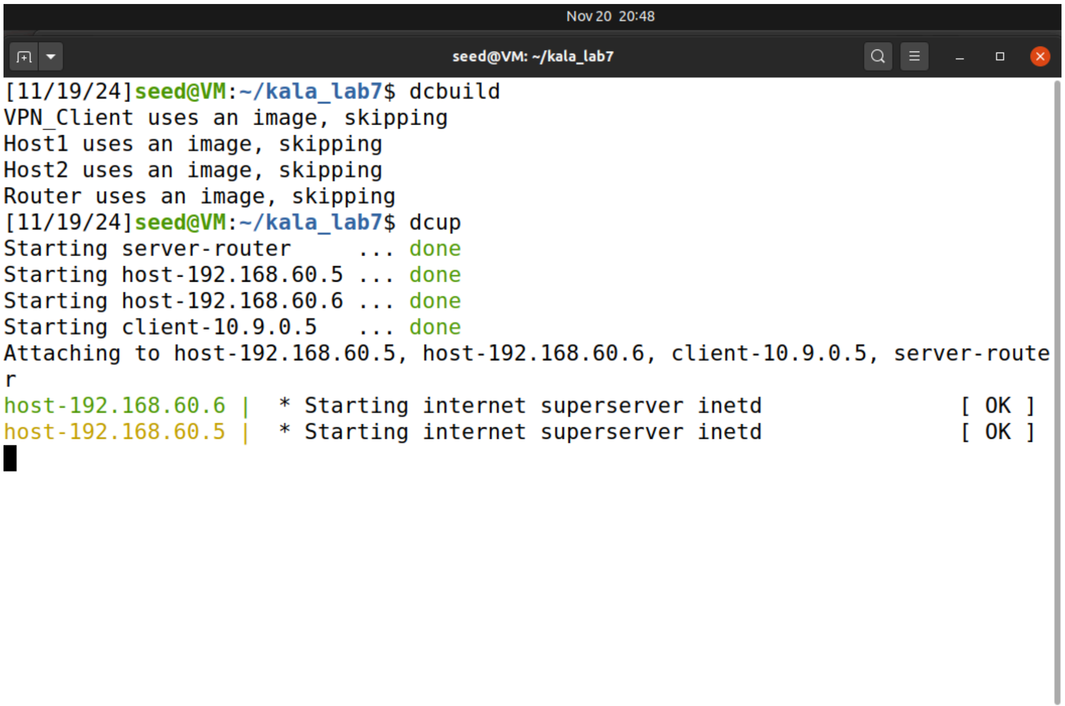
4. **IP Tunneling:** I gained an understanding of how data packets are encapsulated inside other packets for transmission in a VPN tunnel.

5. **Routing in VPNs:** I learned about the routing process and how traffic is directed through the VPN tunnel to reach the intended destination.

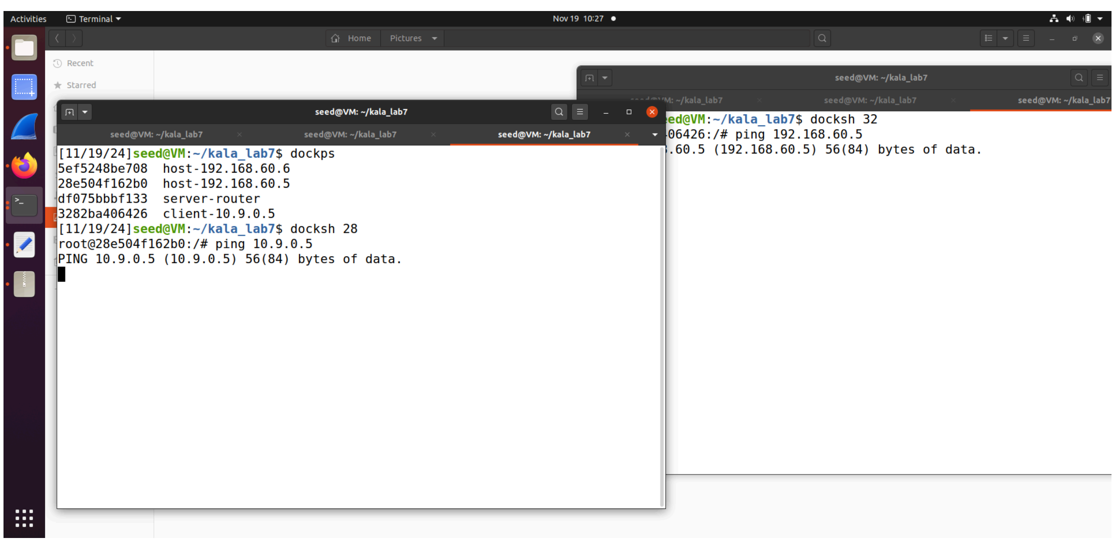
The lab provided hands-on experience with the tunneling aspect of VPNs, which helped solidify these concepts.

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.

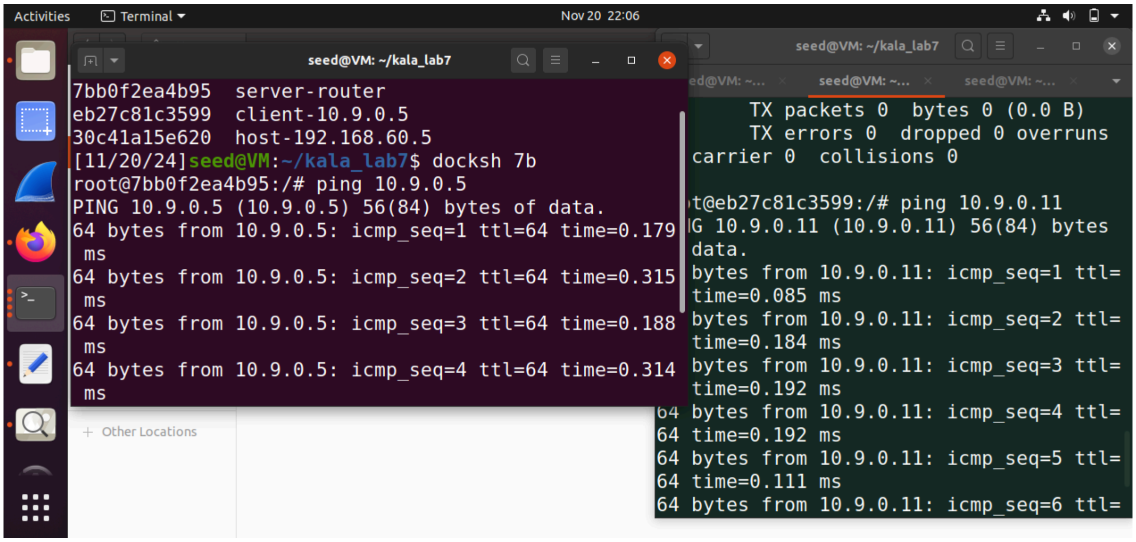
2. Task1: Network Setup



I successfully setup the lab environment.



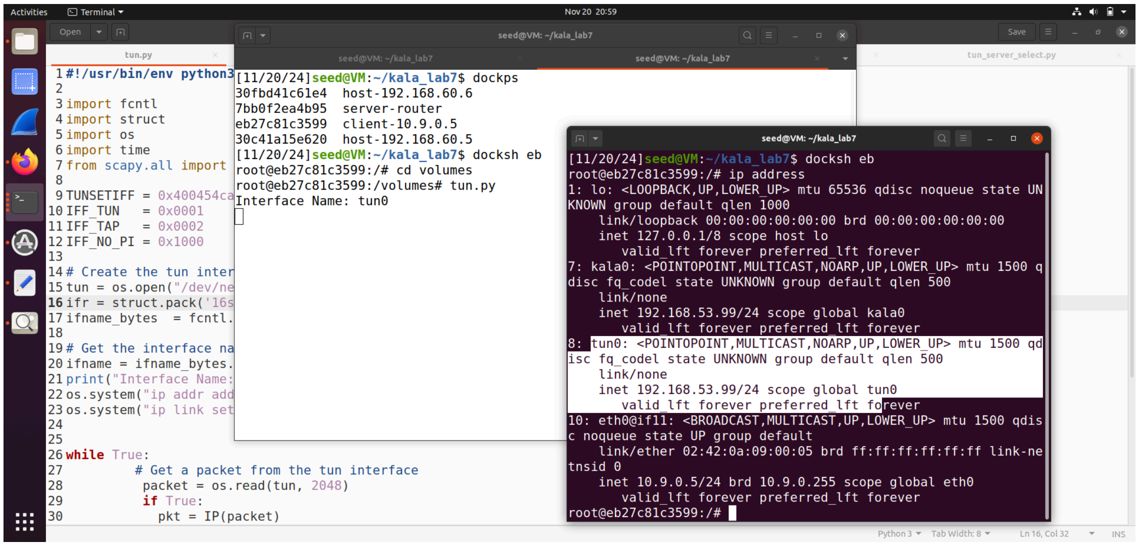
I gave ping for client and the host V. we can clearly see that there is not ping between them.



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.

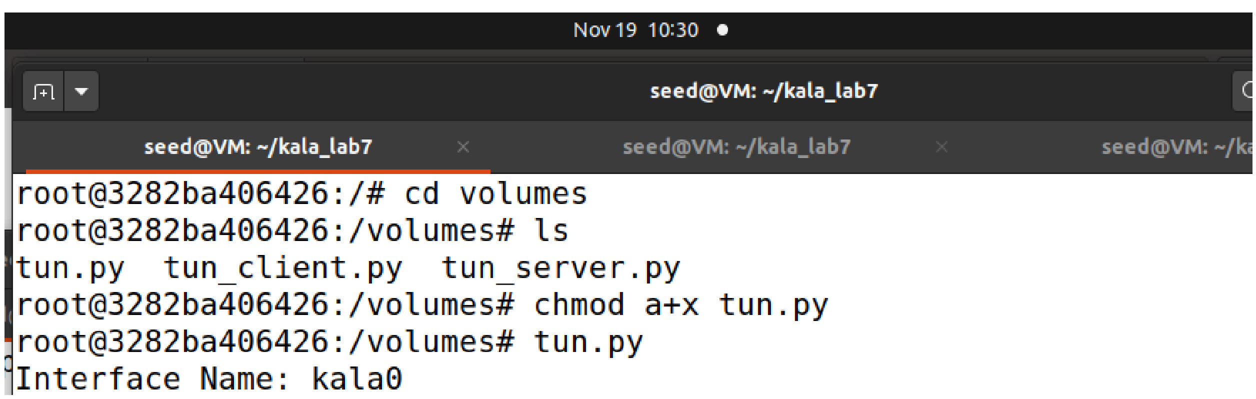
3. Task2: Create and Configure TUN Interface



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:



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

```
Text Editor                                   Nov 19 10:31
Open                                          tun.py
                                              ~/kala_lab7/volumes
tun.py                                        tun_server.py

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'kala%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='192.168.53.3', dst=ip.src)
        newpkt = newip/ip.payload
        arb_data = b'Any arbitrary data'
        os.write(tun, bytes(newpkt))
```

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 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
7: 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
8: tun0: <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 tun0
       valid_lft forever preferred_lft forever
10: eth0@if11: <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@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 thetun0 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

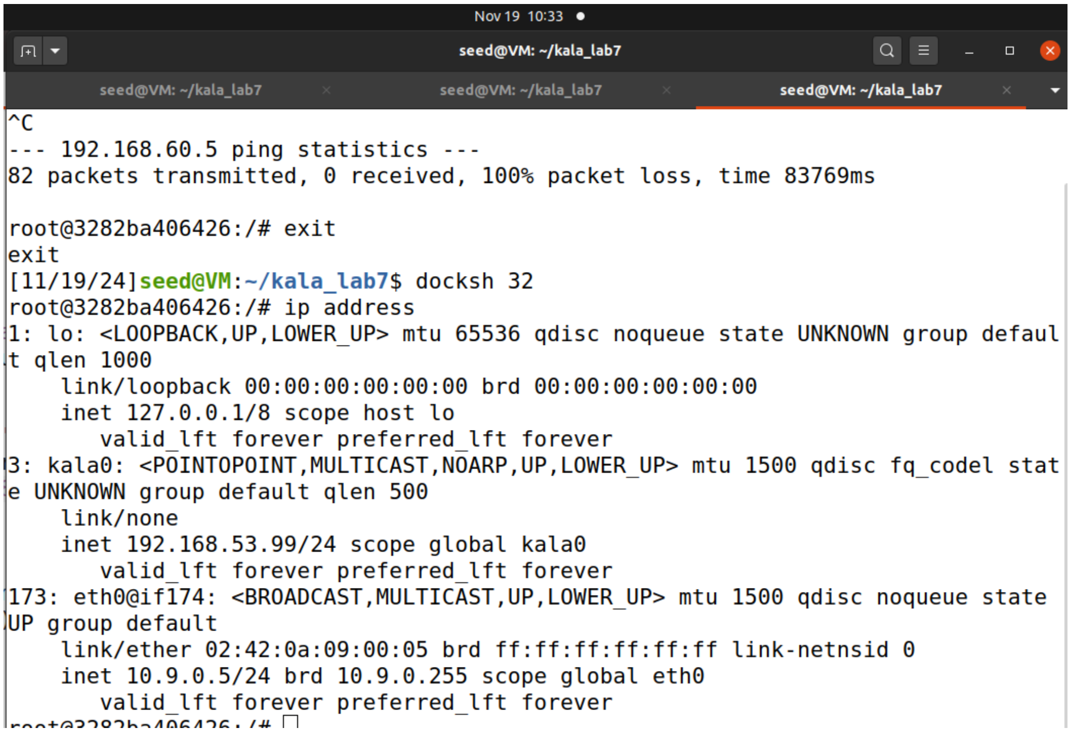
Text Editor Nov 19 10:32

Open tun.py ~/kala_lab7/volumes

tun.py tun_server.py

```
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'kala%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())
```

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    seed@VM: ~/kala_lab7    seed@VM: ~/kala_lab7
^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 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: kala0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel stat
e 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.

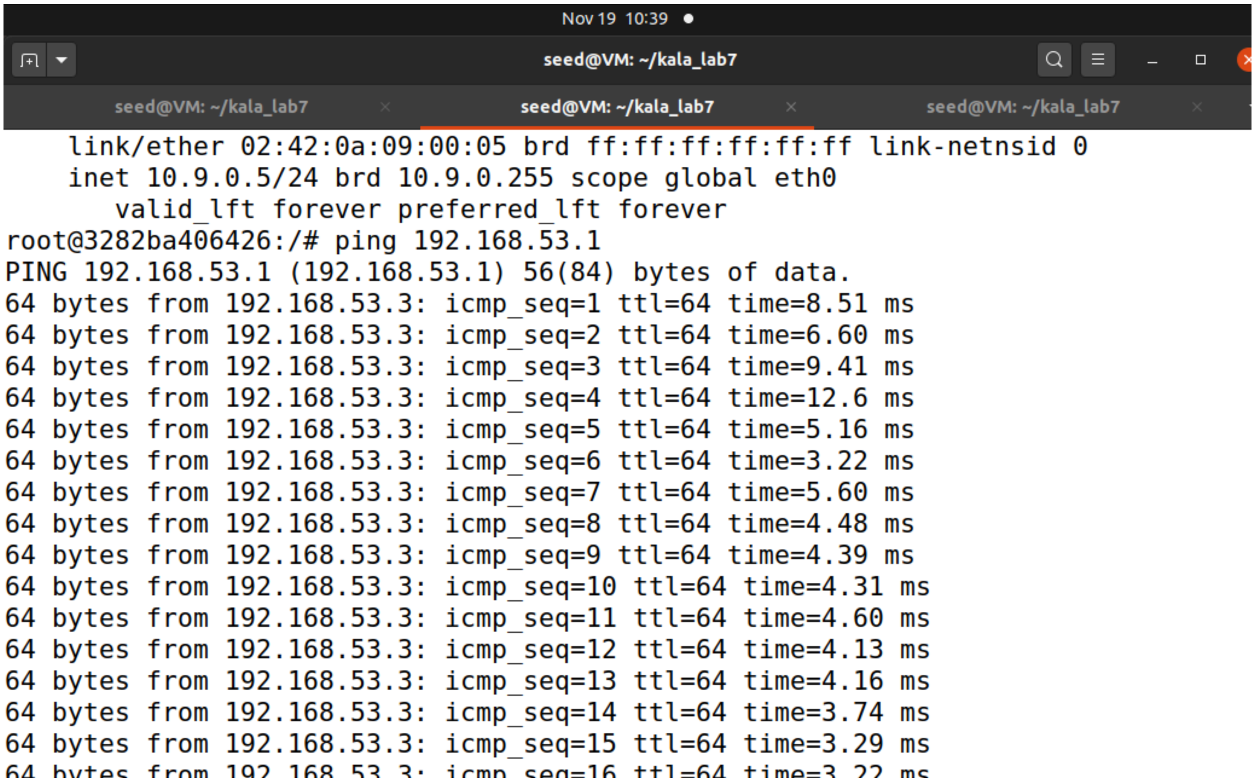
Task 2.c: Read from the TUN Interface

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

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'kala%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())
```

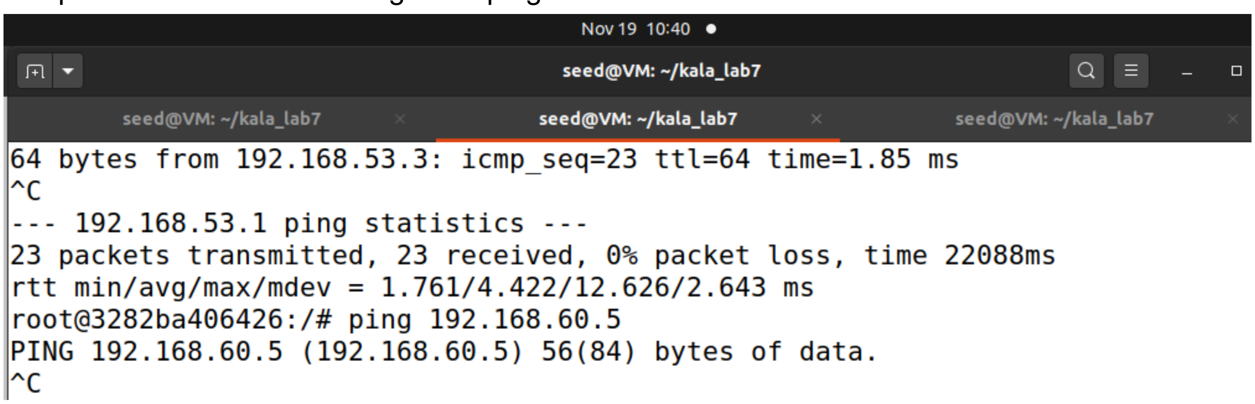
I updated the code to read the packets

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



For ip address 192.168.53.1 i got the ping



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

Open tun.py ~/kala_lab7/volumes

tun.py | tun_server.py

```python

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'kala%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='192.168.53.3', dst=ip.src)
        newpkt = newip/ip.payload
        arb_data = b'Any arbitrary data'
        os.write(tun, bytes(newpkt))

```

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

seed@VM: ~/kala_lab7

seed@VM: ~/kala_lab7 | seed@VM: ~/kala_lab7 | seed@VM: ~/kala_lab7

```
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: Sendthe 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

```
#!/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'kala%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))

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

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

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

```python
PORT = 9090

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

# Create a tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'kala%d', IFF_TUN | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
ifname  = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

# Set up the tun interface
os.system("ip addr add 192.168.53.1/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

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

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

After running the server code and i opened the interface

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

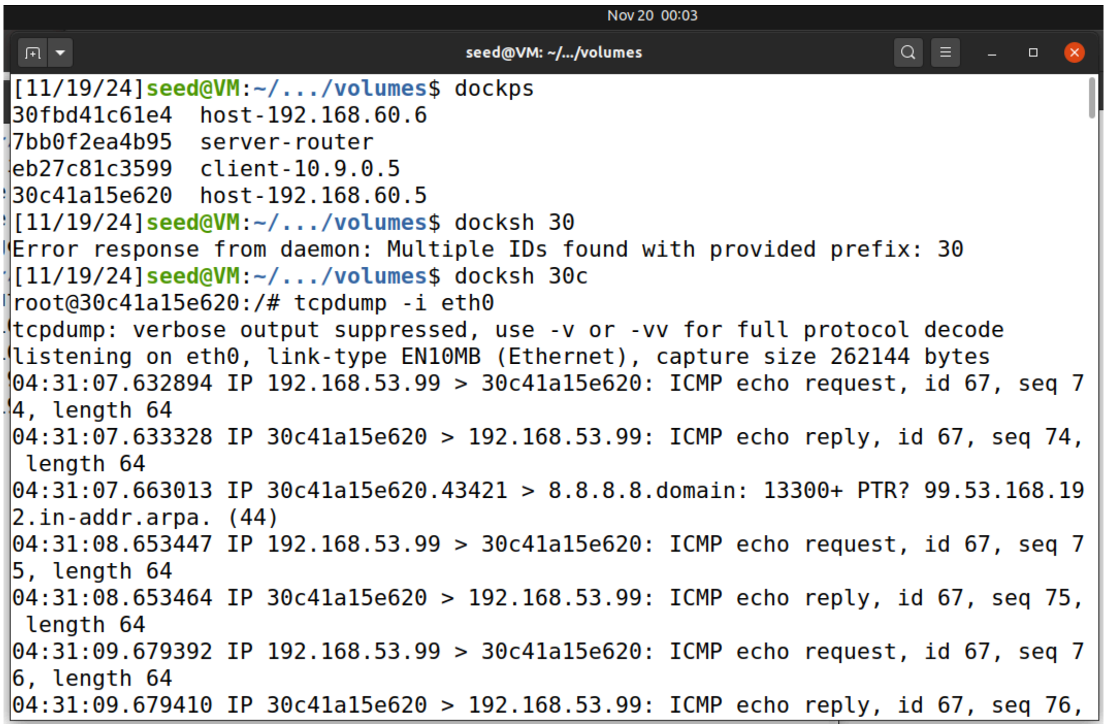
And i did the same for the client program.

```
Nov 19  23:58
seed@VM: ~/kala_lab7
seed@VM: ~/kala_lab7    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 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




```
[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 7
4, 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? 99.53.168.19
2.in-addr.arpa. (44)
04:31:08.653447 IP 192.168.53.99 > 30c41a15e620: ICMP echo request, id 67, seq 7
5, 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 7
6, 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

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

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


SERVER_IP   = "10.9.0.11"
SERVER_PORT = 9090

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

# Create a tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'kala%d', IFF_TUN | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
ifname  = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

# Set up the tun interface and routing
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

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

```

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

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

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

IP_A = "0.0.0.0"
PORT = 9090


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

# Create a tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'kala%d', IFF_TUN | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
ifname  = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

# Set up the tun interface and routing
os.system("ip addr add 192.168.53.1/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

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

```

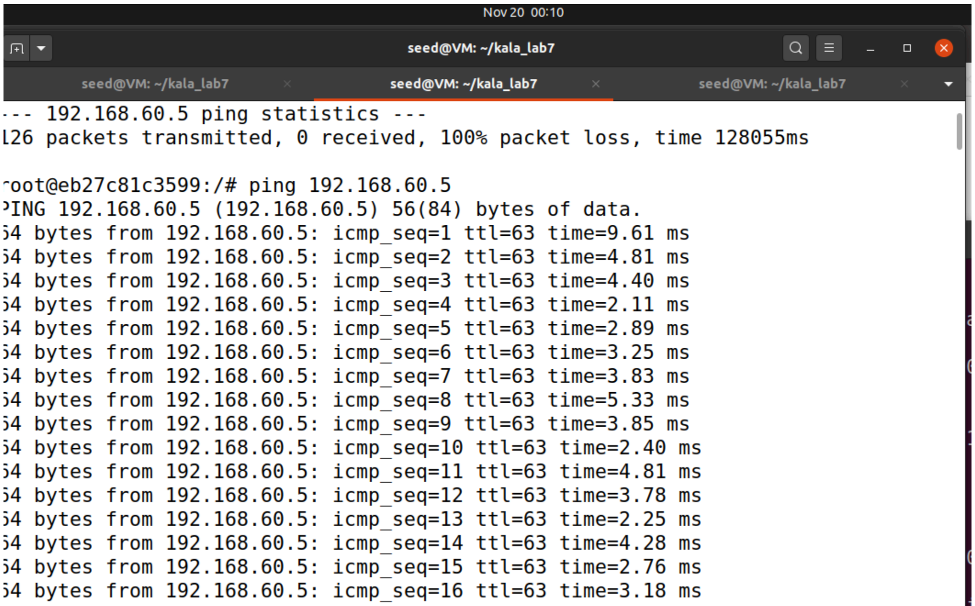
This the server_select code.

```
    ready, _, _ = select.select(fds, [], [])
(eyboardInterrupt

root@eb27c81c3599:/volumes# tun_client_select.py
Interface Name: kala0
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
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
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
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
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
```

Then i ping again and you see i got response in the client_select.

```
root@7bb0f2ea4b95:/volumes# ls
tun.py          tun_client_select.py  tun_server_select.py
tun_client.py   tun_server.py
root@7bb0f2ea4b95:/volumes# chmod a+x tun_server_select.py
root@7bb0f2ea4b95:/volumes# tun_server_select.py
Interface Name: kala0
sock ....
10.9.0.5:59993 --> 0.0.0.0:9090
   Inside Tunnel: 192.168.53.99 --> 192.168.60.5
tun ....
Return: 192.168.60.5 --> 192.168.53.99
sock ....
10.9.0.5:59993 --> 0.0.0.0:9090
   Inside Tunnel: 192.168.53.99 --> 192.168.60.5
tun ....
Return: 192.168.60.5 --> 192.168.53.99
sock ....
```

And this the response from the server.



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

```
Nov 20 00:14
seed@VM: ~/kala_lab7
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 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.