CPS 633 Section 09

VPN Lab: The Container Version Lab

Group 18

Roxie Reginold (501087897)
Hetu Virajkumar Patel (501215707)
Sayyada Aisha Mehvish (501106795)

Task 1: Network Setup

The lab environment uses Docker containers to simulate a real-world network setup.

volumes$ dockps
bf8e4ab74dc1 host-192.168.60.5 --> Host V
0dca28f6cab9 host-192.168.60.6 --> another Host in the same private network at Host V
0cf123042294 server-router --> VPN Server (Router) connecting Host U and Host V.
d6687e788233 client-10.9.0.5 --> VPN client/User (Host U)

We need a separate machine to simulate the VPN setup and ensure realistic network isolation, where the VPN server acts as a gateway between two networks (Internet and private network).

Verifying Network Configuration:
We perform this to check testing connectivity between these components and validate the network topology and configuration to ensure proper routing, identify misconfigurations and to verify isolation.

HostUcan communicate with VPN Server:
We perform this to test connectivity to the VPN Server, verifying HostU can reach the server in its subnet.

```
root@d6687e788233:/# ping server-router
PING server-router (10.9.0.11) 56(84) bytes of data.
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=1 ttl=64 time=0.166 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=2 ttl=64 time=0.216 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=3 ttl=64 time=0.185 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=4 ttl=64 time=0.379 ms
^C
--- server-router ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3053ms
rtt min/avg/max/mdev = 0.166/0.236/0.379/0.084 ms
root@d6687e788233:/#
```

```
root@0cf123042294:/# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
22:19:45.411286 IP client-10.9.0.5.net-10.9.0.0 > 0cf123042294: ICMP echo request, id 17, seq 1, length 64
22:19:45.411336 IP 0cf123042294 > client-10.9.0.5.net-10.9.0.0: ICMP echo reply, id 17, seq 1, length 64
22:19:46.413087 IP client-10.9.0.5.net-10.9.0.0 > 0cf123042294: ICMP echo request, id 17, seq 2, length 64
22:19:46.413169 IP 0cf123042294 > client-10.9.0.5.net-10.9.0.0: ICMP echo reply, id 17, seq 2, length 64
22:19:47.438577 IP client-10.9.0.5.net-10.9.0.0 > 0cf123042294: ICMP echo request, id 17, seq 3, length 64
22:19:47.438627 IP 0cf123042294 > client-10.9.0.5.net-10.9.0.0: ICMP echo reply, id 17, seq 3, length 64
22:19:48.464582 IP client-10.9.0.5.net-10.9.0.0 > 0cf123042294: ICMP echo request, id 17, seq 4, length 64
22:19:48.464728 IP 0cf123042294 > client-10.9.0.5.net-10.9.0.0: ICMP echo reply, id 17, seq 4, length 64
22:19:50.576174 ARP, Request who-has client-10.9.0.5.net-10.9.0.0 tell 0cf123042294, length 28
22:19:50.577074 ARP, Request who-has 0cf123042294 tell client-10.9.0.5.net-10.9.0.0, length 28
22:19:50.577110 ARP, Reply 0cf123042294 is-at 02:42:0a:09:00:0b (oui Unknown), length 28
22:19:50.577132 ARP, Reply client-10.9.0.5.net-10.9.0.0 is-at 02:42:0a:09:00:05 (oui Unknown), length 28
^C
12 packets captured
12 packets received by filter
0 packets dropped by kernel
```

VPN Server can communicate with HostV:
To verify that the VPN Server can communicate with HostV within the private network.

The output replies with a message that confirms the private network is reachable from the VPN Server.

```
root@0cf123042294:/# 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=0.103 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.085 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=64 time=0.101 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=64 time=0.112 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=64 time=0.172 ms
^C
--- 192.168.60.5 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4102ms
rtt min/avg/max/mdev = 0.085/0.114/0.172/0.030 ms
root@0cf123042294:/#
```

```
root@bf8e4ab74dc1:/# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
22:25:40.744972 IP server-router.net-192.168.60.0 > bf8e4ab74dc1: ICMP echo request, id 17, seq 1, length 64
22:25:40.745019 IP bf8e4ab74dc1 > server-router.net-192.168.60.0: ICMP echo reply, id 17, seq 1, length 64
22:25:41.775361 IP server-router.net-192.168.60.0 > bf8e4ab74dc1: ICMP echo request, id 17, seq 2, length 64
22:25:41.775387 IP bf8e4ab74dc1 > server-router.net-192.168.60.0: ICMP echo reply, id 17, seq 2, length 64
22:25:42.798690 IP server-router.net-192.168.60.0 > bf8e4ab74dc1: ICMP echo request, id 17, seq 3, length 64
22:25:42.798722 IP bf8e4ab74dc1 > server-router.net-192.168.60.0: ICMP echo reply, id 17, seq 3, length 64
22:25:43.823196 IP server-router.net-192.168.60.0 > bf8e4ab74dc1: ICMP echo request, id 17, seq 4, length 64
22:25:43.823231 IP bf8e4ab74dc1 > server-router.net-192.168.60.0: ICMP echo reply, id 17, seq 4, length 64
22:25:44.847052 IP server-router.net-192.168.60.0 > bf8e4ab74dc1: ICMP echo request, id 17, seq 5, length 64
22:25:44.847109 IP bf8e4ab74dc1 > server-router.net-192.168.60.0: ICMP echo reply, id 17, seq 5, length 64
^C
10 packets captured
10 packets received by filter
0 packets dropped by kernel
root@bf8e4ab74dc1:/#
```

HostU should not be able to communicate with HostV:
To ensure that HostU cannot directly reach HostV, demonstrating proper network isolation.
In the output we get a message that confirms that HostU and HostV are isolated unless traffic is routed through the VPN Server.

```
root@d6687e788233:/# 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 ---
16 packets transmitted, 0 received, 100% packet loss, time 15344ms

root@d6687e788233:/#
```

```
root@bf8e4ab74dc1:/# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
^C
0 packets captured
0 packets received by filter
0 packets dropped by kernel
```

Run tcpdump on the router, and sniff the traffic on each of the networks. Show that you can capture packets:

Tcpdump captures and analyzes network traffic, which is crucial for verifying communication paths.

Capturing packets confirms if routing and firewall rules are working as intended.

```
root@d6687e788233:/# ping server-router
PING server-router (10.9.0.11) 56(84) bytes of data.
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=1 ttl=64 time=0.185 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=2 ttl=64 time=0.109 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=3 ttl=64 time=0.092 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=4 ttl=64 time=0.349 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=5 ttl=64 time=0.131 ms
^C
--- server-router ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4102ms
rtt min/avg/max/mdev = 0.092/0.173/0.349/0.093 ms
```

```
root@0cf123042294:/# 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
02:13:15.347069 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 23, seq 1, length 64
02:13:15.347100 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 23, seq 1, length 64
02:13:16.377099 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 23, seq 2, length 64
02:13:16.377132 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 23, seq 2, length 64
02:13:17.401392 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 23, seq 3, length 64
02:13:17.401417 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 23, seq 3, length 64
02:13:18.424637 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 23, seq 4, length 64
02:13:18.424689 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 23, seq 4, length 64
02:13:19.448737 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 23, seq 5, length 64
02:13:19.448799 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 23, seq 5, length 64
02:13:20.440387 ARP, Request who-has 10.9.0.5 tell 10.9.0.11, length 28
02:13:20.441540 ARP, Request who-has 10.9.0.11 tell 10.9.0.5, length 28
02:13:20.441579 ARP, Reply 10.9.0.11 is-at 02:42:0a:09:00:0b, length 28
02:13:20.441584 ARP, Reply 10.9.0.5 is-at 02:42:0a:09:00:05, length 28
^C
14 packets captured
14 packets received by filter
0 packets dropped by kernel
```

```
root@bf8e4ab74dc1:/# ping 192.168.60.11
PING 192.168.60.11 (192.168.60.11) 56(84) bytes of data.
64 bytes from 192.168.60.11: icmp_seq=1 ttl=64 time=0.187 ms
64 bytes from 192.168.60.11: icmp_seq=2 ttl=64 time=0.134 ms
64 bytes from 192.168.60.11: icmp_seq=3 ttl=64 time=0.099 ms
^C
--- 192.168.60.11 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2040ms
rtt min/avg/max/mdev = 0.099/0.140/0.187/0.036 ms
```

```
root@0cf123042294:/# 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
02:25:47.297750 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 47, seq 1, length 64
02:25:47.297823 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 47, seq 1, length 64
02:25:48.312541 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 47, seq 2, length 64
02:25:48.312583 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 47, seq 2, length 64
02:25:49.337398 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 47, seq 3, length 64
02:25:49.337421 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 47, seq 3, length 64
^C
6 packets captured
6 packets received by filter
0 packets dropped by kernel
```

We run commands tcpdump -i eth0 -n (VPN network) and tcpdump -i eth1 -n(private network), to observe traffic on each interface.

To generate traffic:
From HostU, we ping 10.9.0.5 (VPN Server) and get output of eth0 i.e. packets from Host U.
From VPN Server, we ping 192.168.60.5 (Host V) and get output of eth1 i.e. packets from HostV

Task 2: Create and Configure TUN Interface

Task 2.a: Name of the Interface

run the tun.py program on HostU.

```
root@d6687e788233:/volumes# nano tun.py
root@d6687e788233:/volumes# chmod a+x tun.py
root@d6687e788233:/volumes# tun.py
Interface Name: tun0
```

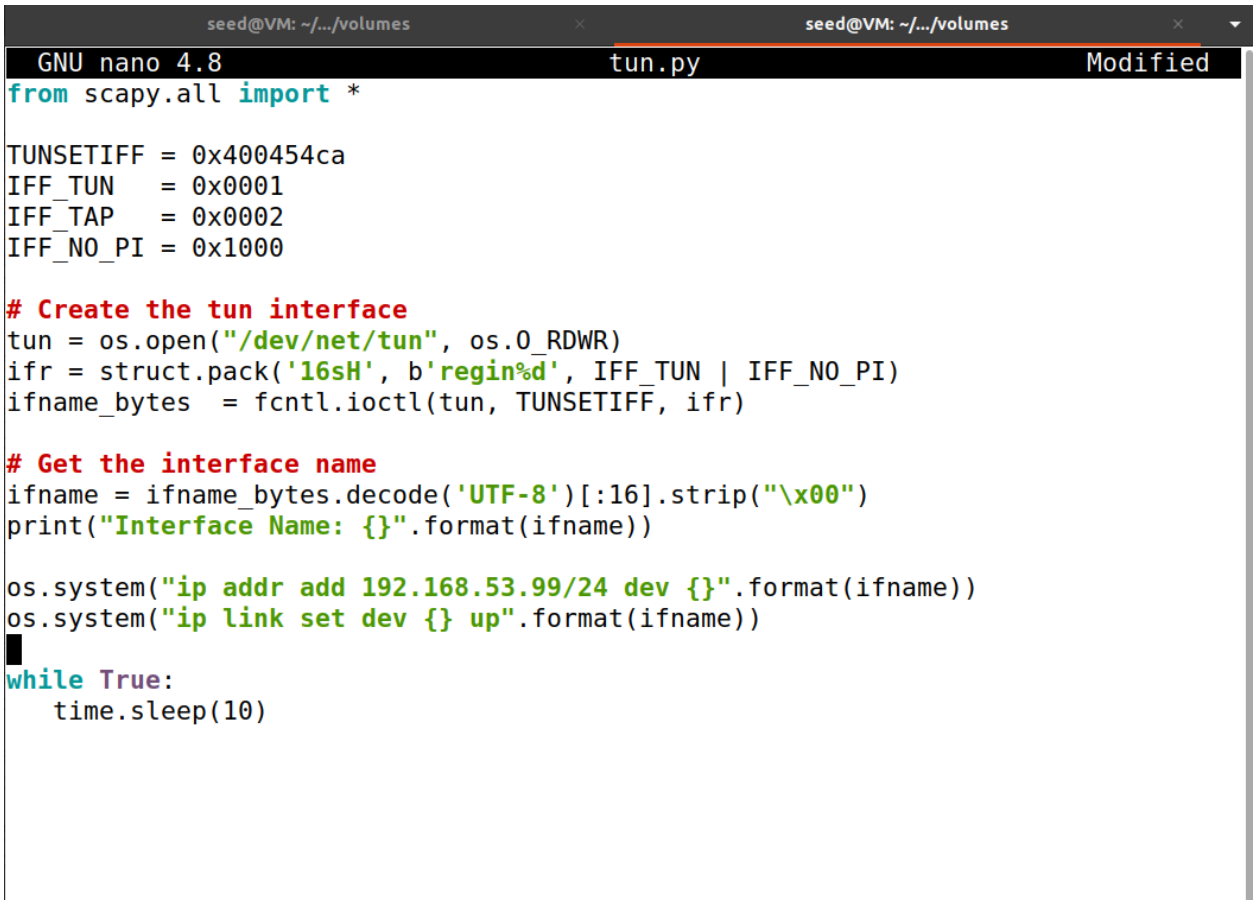
```
[11/25/24]seed@VM:~/.../volumes$ docksh d6
root@d6687e788233:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defaul
t qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group defa
ult qlen 500
    link/none
50: eth0@if51: <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
```

When modifying the interface name using the code line:
ifr = struct.pack('16sH', b'regin%d', IFF_TUN | IFF_NO_PI)

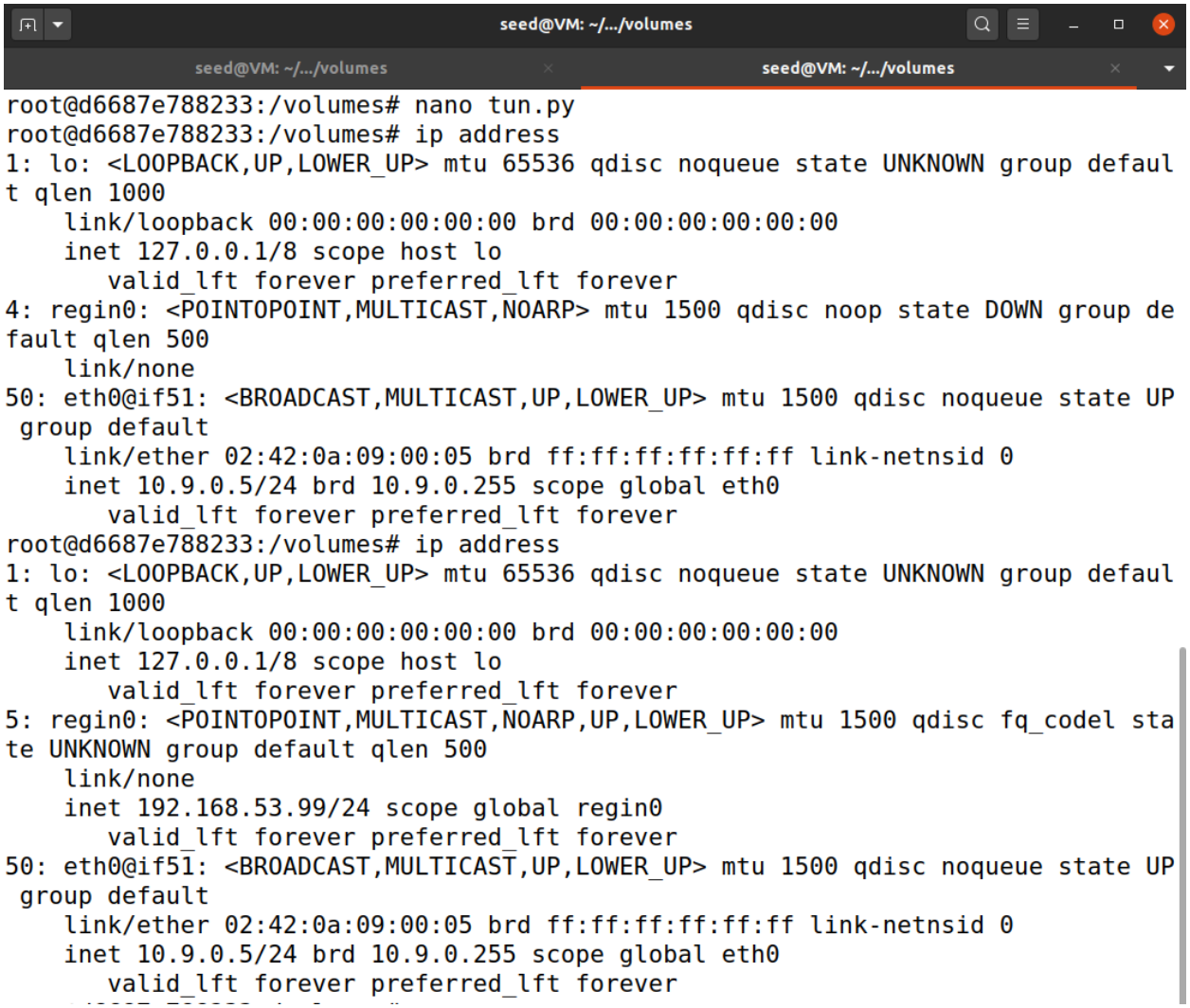
```
root@d6687e788233:/volumes# tun.py
Interface Name: regin0
```

```
root@d6687e788233:/# 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: regin0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group de
fault qlen 500
    link/none
50: eth0@if51: <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
```

Task 2.b: Set up the TUN Interface




```
root@d6687e788233:/volumes# tun.py
Interface Name: regin0
```



Here is what I had before:
4: regin0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
link/none

This is what is different:
5: regin0: <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 regin0
valid_lft forever preferred_lft forever

UP and LOWER_UP: Confirms that the interface is active and ready to send/receive packets.
The interface has been assigned the IP address 192.168.53.99 with a subnet mask of /24

1. State of the Interface
Before: state DOWN
The interface was inactive and could not transmit or receive packets.
After: state UNKNOWN
The interface is now active (UP) and ready for use, even though its state is "UNKNOWN," which is common for virtual interfaces like TUN.

2. Flags
Before: <POINTOPOINT,MULTICAST,NOARP>
These flags indicate that the interface is point-to-point, supports multicast traffic, and does not use ARP.
After: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP>
The UP and LOWER_UP flags were added:
UP: The interface is enabled and administratively active.
LOWER_UP: Indicates that the link layer (virtual in this case) is operational.

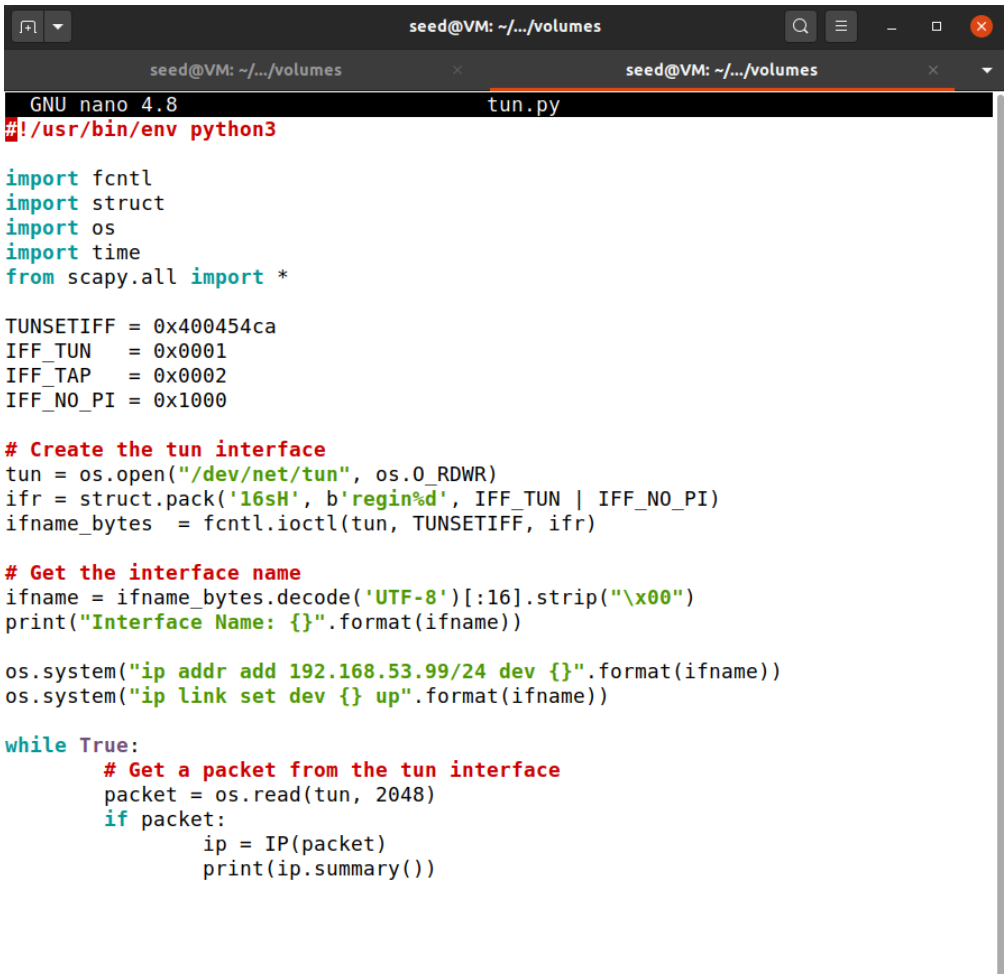
3. IP Address
Before: No inet address assigned.
After: Assigned the IP address 192.168.53.99/24 with global scope.
This enables communication with devices in the 192.168.53.0/24 subnet.

4. Queueing Discipline
Before: qdisc noop
A placeholder queuing discipline, meaning no traffic shaping or advanced queue management.
After: qdisc fq_codel
Changed to fq_codel (Fair Queuing Controlled Delay), which helps reduce network latency and manage packet queues efficiently.

5. New Fields
Before: No inet or valid_lft fields present.
After: Added:
inet: Displays the IP address assigned to the interface.
valid_lft forever preferred_lft forever: Indicates the IP address is valid and has no expiration.

Key Differences in Summary:
The interface is now UP and operational (state UNKNOWN, flags UP and LOWER_UP).
An IP address (192.168.53.99/24) has been assigned.
The queueing discipline has been updated to fq_codel.
Additional fields (inet, valid_lft) confirm the interface configuration.

Task 2c: Rewriting the while loop



Rerunning the revised program on Host U:

```
root@d6687e788233:/volumes# tun.py
Interface Name: regin0
```

```
root@d6687e788233:/volumes# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group d
efault qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
6: regin0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_code
l state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global regin0
       valid_lft forever preferred_lft forever
50: eth0@if51: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue sta
te 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
```

On HostU, ping a host in the 192.168.53.0/24 network. What is printed out by the run.py program? What has happened? Why?

```
root@d6687e788233:/volumes# ping 192.168.53.1
PING 192.168.53.1 (192.168.53.1) 56(84) bytes of data.
^C
--- 192.168.53.1 ping statistics ---
15 packets transmitted, 0 received, 100% packet loss, time 14331ms

root@d6687e788233:/volumes#
```

```
root@d6687e788233:/volumes# tun.py
Interface Name: regin0
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
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
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
```

When you ping a host in the same subnet (192.168.53.0/24), the ICMP echo request packet is routed through the TUN interface. The tun.py program captures and displays these packets. The TUN interface is configured to handle traffic for external networks (like 192.168.53.0/24). Packets to 192.168.53.0/24 are routed through the tunnel, where tun.py reads them as raw IP packets and processes them with Scapy.

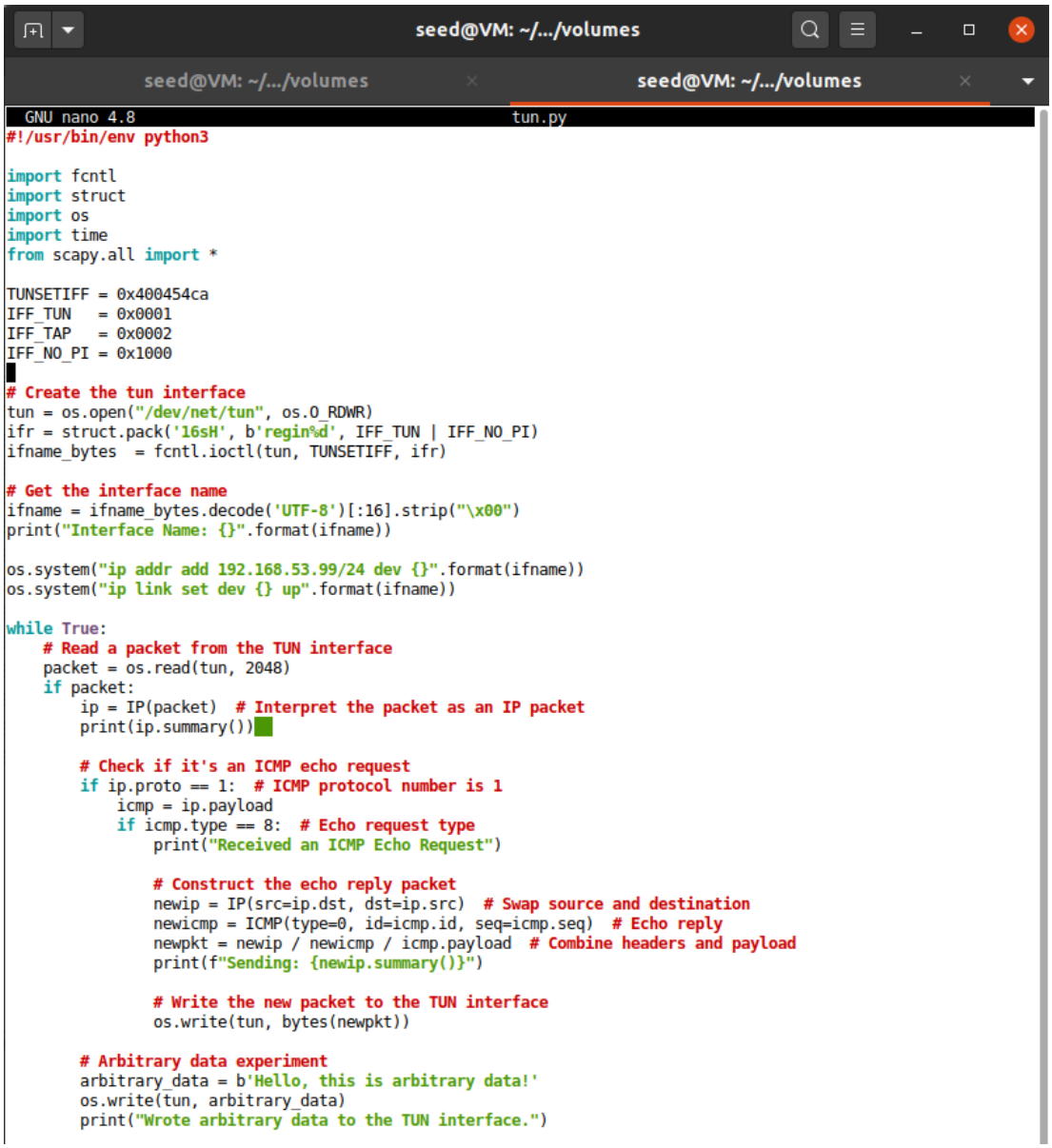
On HostU, ping a host in the internal network 192.168.60.0/24, Does tun.py print out anything? Why?

```
root@d6687e788233:/volumes# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
--- 192.168.60.5 ping statistics ---
11 packets transmitted, 0 received, 100% packet loss, time 10222ms
```

```
root@d6687e788233:/volumes# tun.py
Interface Name: regin0
```

Packets destined for the 192.168.60.0/24 network are not routed through the TUN interface. Instead, they are forwarded to the VPN router, which handles communication between the 192.168.53.0/24 and 192.168.60.0/24 networks. The TUN interface only captures packets directly routed to it, which excludes traffic meant for other subnets.

Task 2.d: Write to the TUN Interface



```
root@d6687e788233:/volumes# ping 192.168.53.3
PING 192.168.53.3 (192.168.53.3) 56(84) bytes of data.
64 bytes from 192.168.53.3: icmp_seq=1 ttl=64 time=5.80 ms
64 bytes from 192.168.53.3: icmp_seq=2 ttl=64 time=7.56 ms
64 bytes from 192.168.53.3: icmp_seq=3 ttl=64 time=3.13 ms
64 bytes from 192.168.53.3: icmp_seq=4 ttl=64 time=4.91 ms
64 bytes from 192.168.53.3: icmp_seq=5 ttl=64 time=4.22 ms
64 bytes from 192.168.53.3: icmp_seq=6 ttl=64 time=12.2 ms
^C
--- 192.168.53.3 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5007ms
rtt min/avg/max/mdev = 3.126/6.306/12.221/2.978 ms
root@d6687e788233:/volumes#
```

```
root@d6687e788233:/volumes# tun.py
Interface Name: regin0
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
Received an ICMP Echo Request
Sending: 192.168.53.3 > 192.168.53.99 hopopt
Wrote arbitrary data to the TUN interface.
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
Received an ICMP Echo Request
Sending: 192.168.53.3 > 192.168.53.99 hopopt
Wrote arbitrary data to the TUN interface.
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
Received an ICMP Echo Request
Sending: 192.168.53.3 > 192.168.53.99 hopopt
Wrote arbitrary data to the TUN interface.
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
Received an ICMP Echo Request
Sending: 192.168.53.3 > 192.168.53.99 hopopt
Wrote arbitrary data to the TUN interface.
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
Received an ICMP Echo Request
Sending: 192.168.53.3 > 192.168.53.99 hopopt
Wrote arbitrary data to the TUN interface.
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
Received an ICMP Echo Request
Sending: 192.168.53.3 > 192.168.53.99 hopopt
Wrote arbitrary data to the TUN interface.
```

ICMP Echo Request and Reply:

When the TUN interface receives an ICMP echo request, the modified tun.py code:
Detects the request,
Constructs a valid ICMP echo reply,
Writes the reply back to the TUN interface.
This process mimics how a router or server would respond to ping requests.

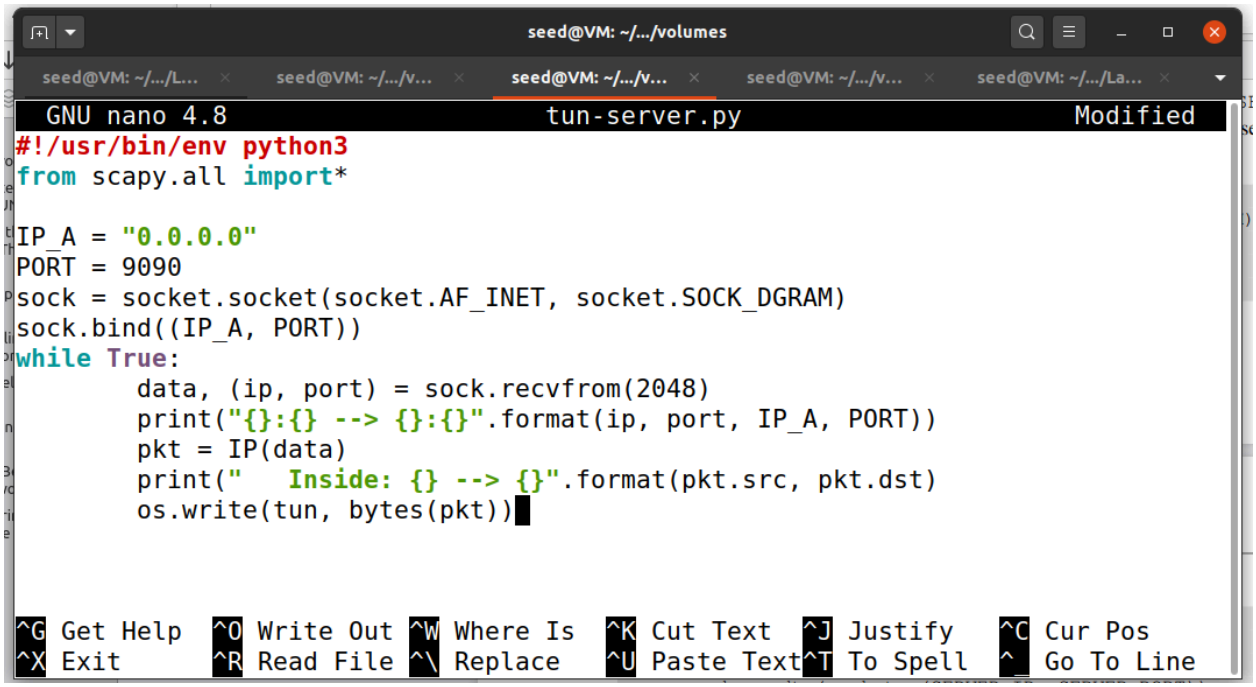
Arbitrary Data:

Writing arbitrary data to the TUN interface demonstrates that the TUN interface is simply a pipe for data.
However, if the data does not conform to the expected format of an IP packet, the kernel will treat it as invalid and drop it.

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

The server receives packets sent by tunclient.py over UDP, unpacks them, and prints the encapsulated IP packets.

We do this to enable communication between Host U and Host V through the VPN Server by securely routing packets through the tunnel.



The client program (tunclient.py) will read packets from the TUN interface and send them over a UDP socket to the server.

We run tunserver.py on the VPN server and run tunclient.py on HostU.

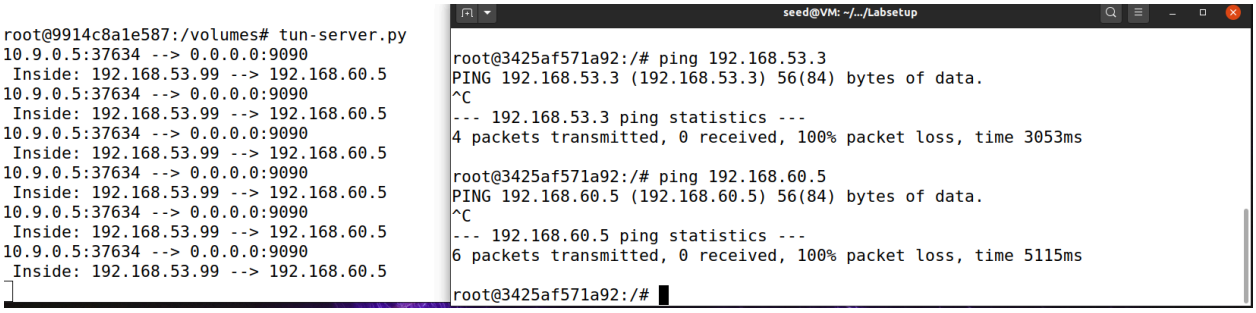
```
root@3425af571a92:/volumes# nano tun-client.py
root@3425af571a92:/volumes# chmod a+x tun-client.py
root@3425af571a92:/volumes# tun-client.py
Interface Name: mehvi0
```

This shows the source IPs of encapsulated and decapsulated packets.

Next, we add the following entry to the routing table:

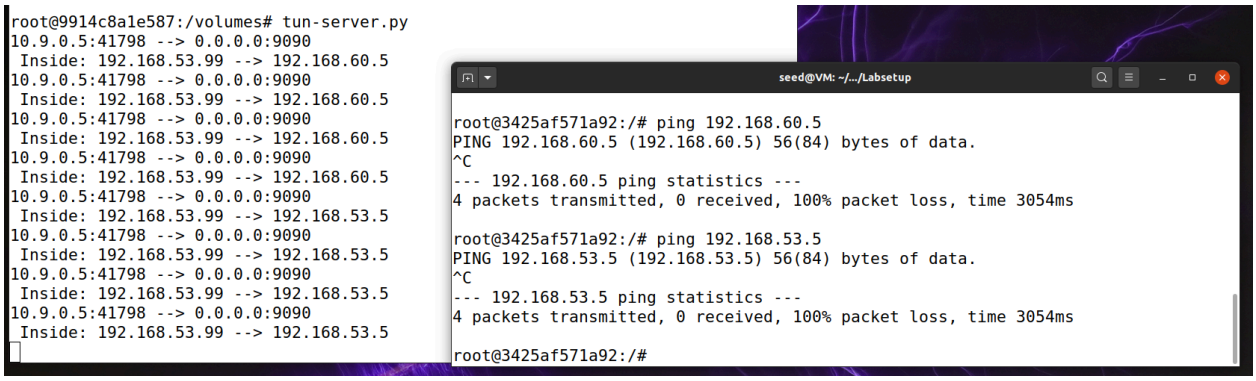
```
root@3425af571a92:/# ip route add 192.168.60.5 dev mehvi0
root@3425af571a92:/# ip route
default via 10.9.0.1 dev eth0
10.9.0.0/24 dev eth0 proto kernel scope link src 10.9.0.5
192.168.53.0/24 dev mehvi0 proto kernel scope link src 192.168.53.99
192.168.60.0/24 dev mehvi0 scope link
192.168.60.5 dev mehvi0 scope link
root@3425af571a92:/#
```

On HostU with ping 192.168.53.1 to test sending packets:

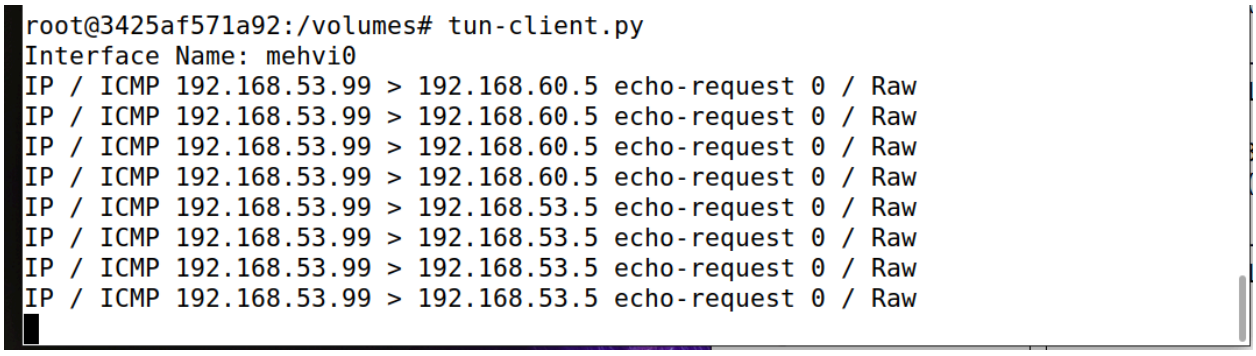


tunserver.py prints the encapsulated packets it receives, showing the source and destination IPs.

Next, we test reaching the private network (192.168.60.0/24) by pinging HostV:



This successful ping reply confirms that ICMP packets traverse the tunnel to Host V.



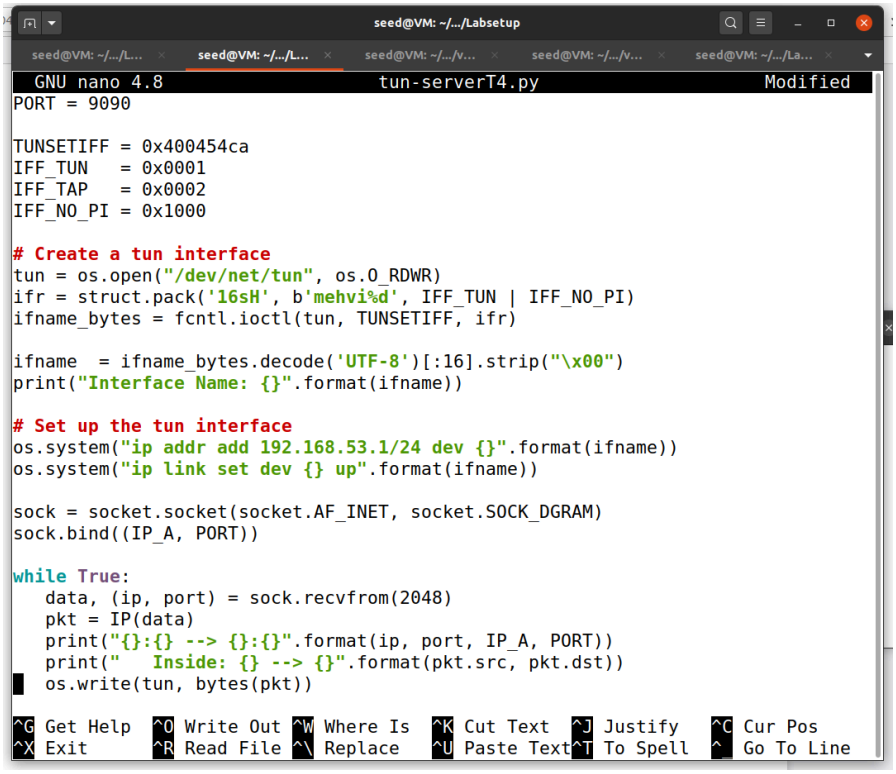
Output from tun-server.py demonstrates that encapsulated packets are received, decapsulated and interpreted correctly.

And we use the ip route command to verify the routing rules and to ensure that entries direct traffic to 192.168.60.0/24 through the TUN interface.

Task 4: Set Up the VPN Server

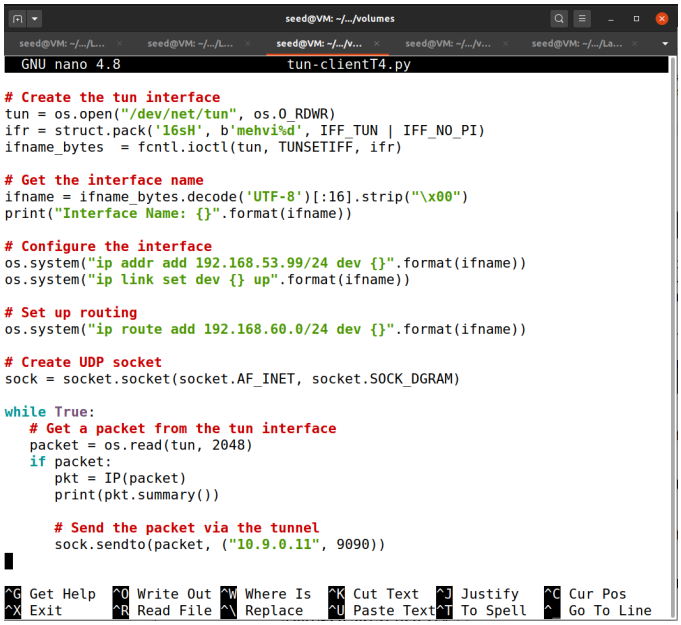
We want to use the VPN Server as a gateway for routing packets from the tunnel (Host U) to their final destination in the private network (Host V). By writing packets to the TUN interface, the operating system can process the packet as if it were received on a real network interface.

We modify tunserver.py to the lab requirements:



We receive data from the UDP socket, treat it as an IP packet and write it to the TUN interface.

- This is achieved by reading the payload from the UDP packet, casting it to a Scapy IP object to confirm it is a valid IP packet and then writing the IP packet to the TUN interface for routing by the kernel.
- When tun_server.py receives packets, it will correctly write them to the TUN interface, enabling the kernel to forward the packets to their final destination.



From Host U, we ping host V:

```
root@3425af571a92:/# ping 192.168.60.5 -c 4
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
--- 192.168.60.5 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3135ms

root@3425af571a92:/#
```

This sends ICMP (Internet Control Message Protocol) echo requests to Host V through the VPN Server.

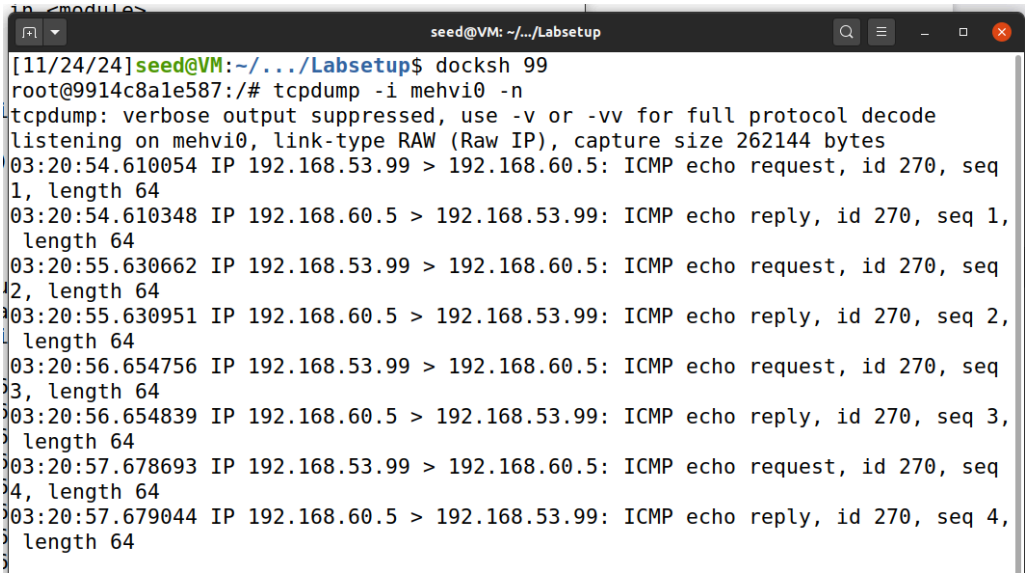
Running updated tun-client.py file with results:

```
root@3425af571a92:/volumes# nano tun-clientT4.py
root@3425af571a92:/volumes# chmod a+x tun-clientT4.py
root@3425af571a92:/volumes# tun-clientT4.py
Interface Name: mehvi0
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
```

Running updated tun-server.py files with results:

```
root@9914c8a1e587:/volumes# chmod a+x tun-serverT4.py
root@9914c8a1e587:/volumes# tun-serverT4.py
Interface Name: mehvi0
10.9.0.5:42083 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:42083 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:42083 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:42083 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
```

On the VPN server's interface, we run tcpdump, the results of which show that packets are being forwarded.



On Host V, the output displays the source (192.168.53.99) and destination (192.168.60.5) addresses of the packets.

```
root@74725494cb33:/# 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
03:20:54.610223 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 270, seq
1, length 64
03:20:54.610334 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 270, seq 1,
 length 64
03:20:55.630896 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 270, seq
2, length 64
03:20:55.630935 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 270, seq 2,
 length 64
03:20:56.654804 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 270, seq
3, length 64
03:20:56.654830 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 270, seq 3,
 length 64
03:20:57.678735 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 270, seq
4, length 64
03:20:57.679025 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 270, seq 4,
 length 64
03:20:59.695804 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
03:20:59.695947 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
03:20:59.695963 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
03:20:59.695965 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
```

Therefore, we configured the VPN Server to function as a gateway between the tunneled network and the private network. By utilizing TUN interfaces and enabling IP forwarding, we ensured that packets sent from Host U through the tunnel were accurately routed to their destination, Host V. This setup underscored essential concepts for building a VPN Server, such as packet encapsulation, routing, and kernel-level forwarding.

While the VPN Server successfully transmitted ICMP echo requests from Host U to Host V, reverse communication (ICMP replies from Host V to Host U) was not operational, highlighting the need for additional routing configurations to enable bidirectional traffic in a fully functional VPN setup.

Task 5 Handling Traffic in Both Directions

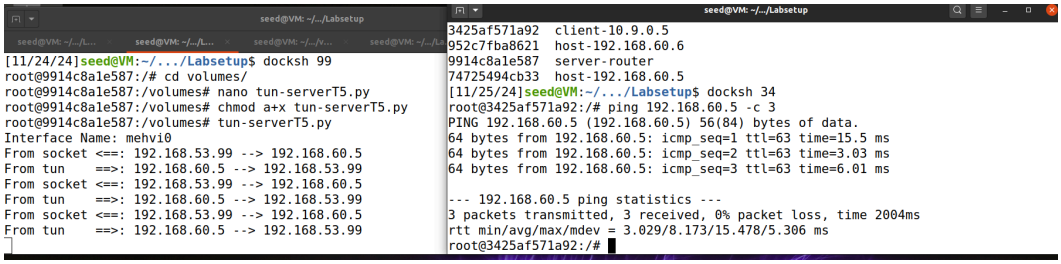
Here, we aim to create two-way communication within the VPN tunnel. In the previous steps we were able to send traffic from Host U to Host V successfully. However, the reverse communication, where Host V tries to send a response back to Host U, is not functioning. This is because the tunnel was set up to handle traffic in only one direction. To complete the setup, we need to enable the reverse flow of packets, so Host V can send a response back to Host U through the tunnel.

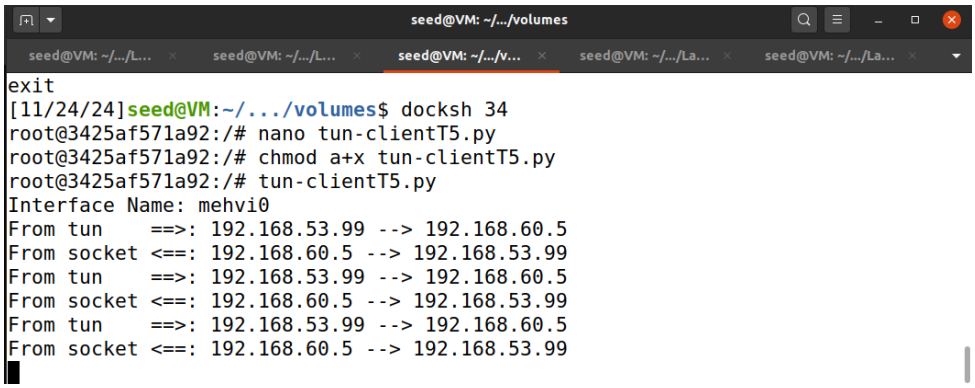
Code tun-server.py

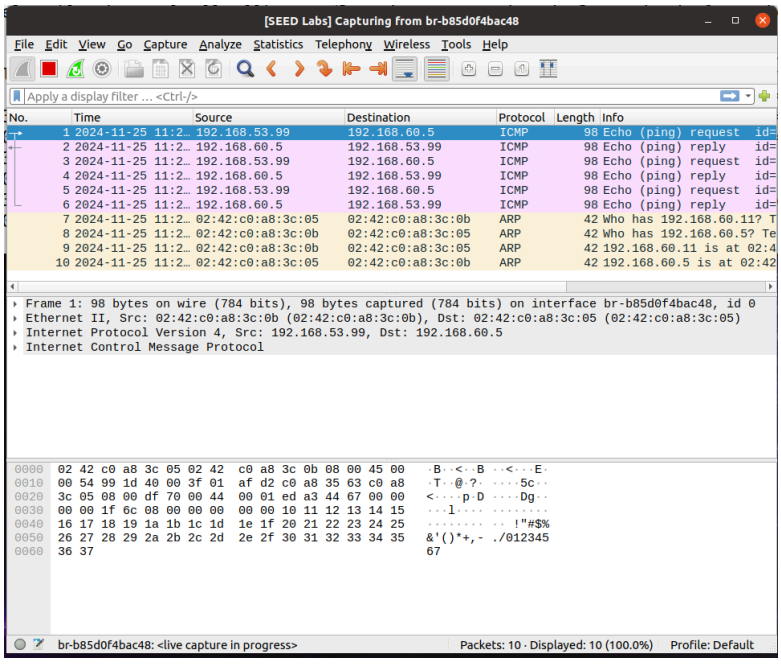
```
#!/usr/bin/python3
#import select
import fcntl
import struct
import os
from scapy.all import *

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'mehvi%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))

ip   = '10.9.0.5'
port = 10000
while True:
  # this will block until at least one socket 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, data)

    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))
```

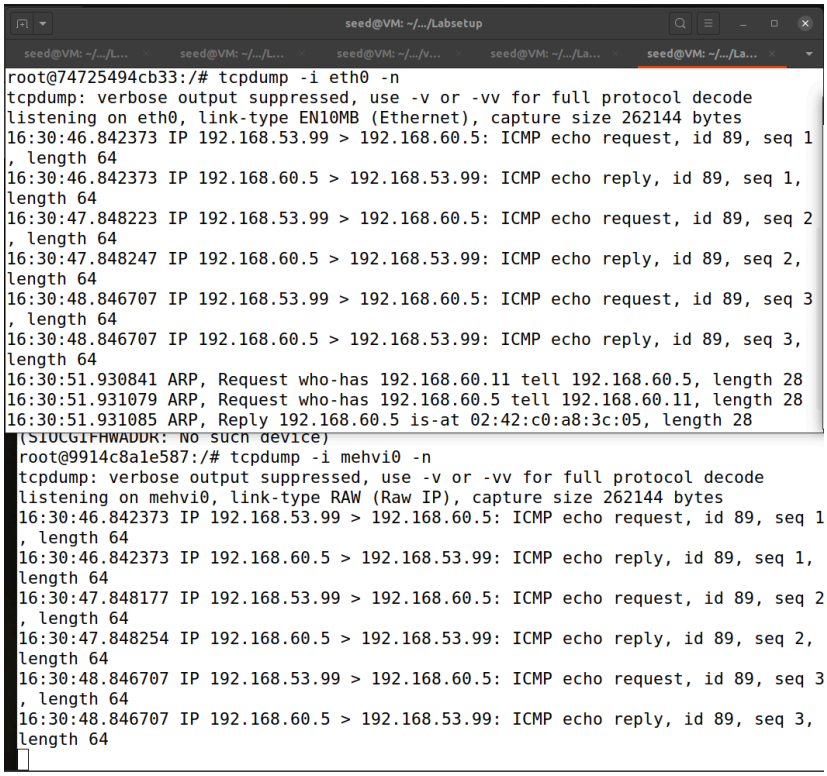
Code tun-client.py

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

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'mehvi%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))

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

# Create UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

while True:
  # this will block until at least one socket 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, data)

    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))
```


seed@VM: ~/.../Labsetup
[11/24/24]seed@VM:~/.../Labsetup$ docksh 99
root@9914c8a1e587:/# cd volumes/
root@9914c8a1e587:/volumes# nano tun-serverT5.py
root@9914c8a1e587:/volumes# chmod a+x tun-serverT5.py
root@9914c8a1e587:/volumes# tun-serverT5.py
Interface Name: mehvi0
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
seed@VM: ~/.../Labsetup
3425af571a92 client-10.9.0.5
952c7fba8621 host-192.168.60.6
9914c8a1e587 server-router
74725494cb33 host-192.168.60.5
[11/25/24]seed@VM:~/.../Labsetup$ docksh 34
root@3425af571a92:/# ping 192.168.60.5 -c 3
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=15.5 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=3.03 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=6.01 ms
--- 192.168.60.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 3.029/8.173/15.478/5.306 ms
root@3425af571a92:/#


seed@VM: ~/.../volumes
exit
[11/24/24]seed@VM:~/.../volumes$ docksh 34
root@3425af571a92:/# nano tun-clientT5.py
root@3425af571a92:/# chmod a+x tun-clientT5.py
root@3425af571a92:/# tun-clientT5.py
Interface Name: mehvi0
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


[SEED Labs] Capturing from br-b85d0f4bac48
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
No. Time Source Destination Protocol Length Info
1 2024-11-25 11:2... 192.168.53.99 192.168.60.5 ICMP 98 Echo (ping) request id=
2 2024-11-25 11:2... 192.168.60.5 192.168.53.99 ICMP 98 Echo (ping) reply id=
3 2024-11-25 11:2... 192.168.53.99 192.168.60.5 ICMP 98 Echo (ping) request id=
4 2024-11-25 11:2... 192.168.60.5 192.168.53.99 ICMP 98 Echo (ping) reply id=
5 2024-11-25 11:2... 192.168.53.99 192.168.60.5 ICMP 98 Echo (ping) request id=
6 2024-11-25 11:2... 192.168.60.5 192.168.53.99 ICMP 98 Echo (ping) reply id=
7 2024-11-25 11:2... 02:42:c0:a8:3c:05 02:42:c0:a8:3c:0b ARP 42 Who has 192.168.60.11? T
8 2024-11-25 11:2... 02:42:c0:a8:3c:0b 02:42:c0:a8:3c:05 ARP 42 Who has 192.168.60.5? Te
9 2024-11-25 11:2... 02:42:c0:a8:3c:0b 02:42:c0:a8:3c:05 ARP 42 192.168.60.11 is at 02:4
10 2024-11-25 11:2... 02:42:c0:a8:3c:05 02:42:c0:a8:3c:0b ARP 42 192.168.60.5 is at 02:42
Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface br-b85d0f4bac48, id 0
Ethernet II, Src: 02:42:c0:a8:3c:0b (02:42:c0:a8:3c:0b), Dst: 02:42:c0:a8:3c:05 (02:42:c0:a8:3c:05)
Internet Protocol Version 4, Src: 192.168.53.99, Dst: 192.168.60.5
Internet Control Message Protocol
0000 02 42 c0 a8 3c 05 02 42 c0 a8 3c 0b 08 00 45 00
0010 00 54 99 1d 40 00 3f 01 af d2 c0 a8 35 63 c0 a8
0020 3c 05 08 00 df 70 00 44 00 01 ed a3 44 67 00 00
0030 00 00 1f 6c 08 00 00 00 00 00 10 11 12 13 14 15
0040 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25
0050 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35
0060 36 37
br-b85d0f4bac48: <live capture in progress>
Packets: 10 · Displayed: 10 (100.0%)
Profile: Default

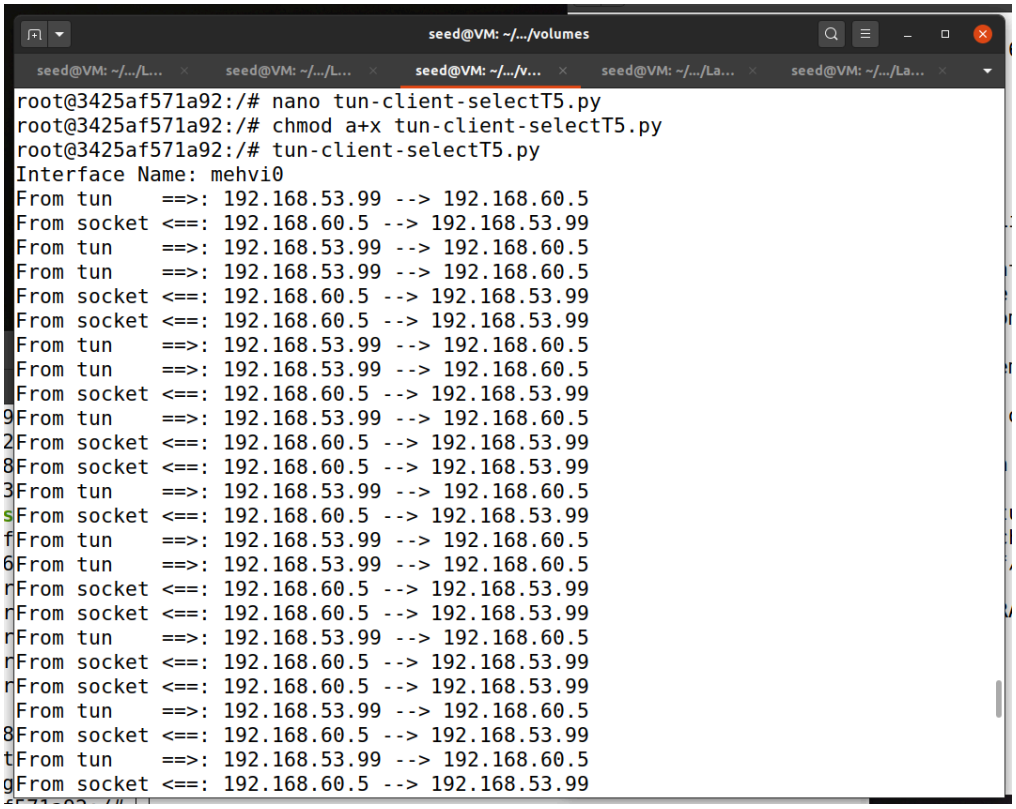


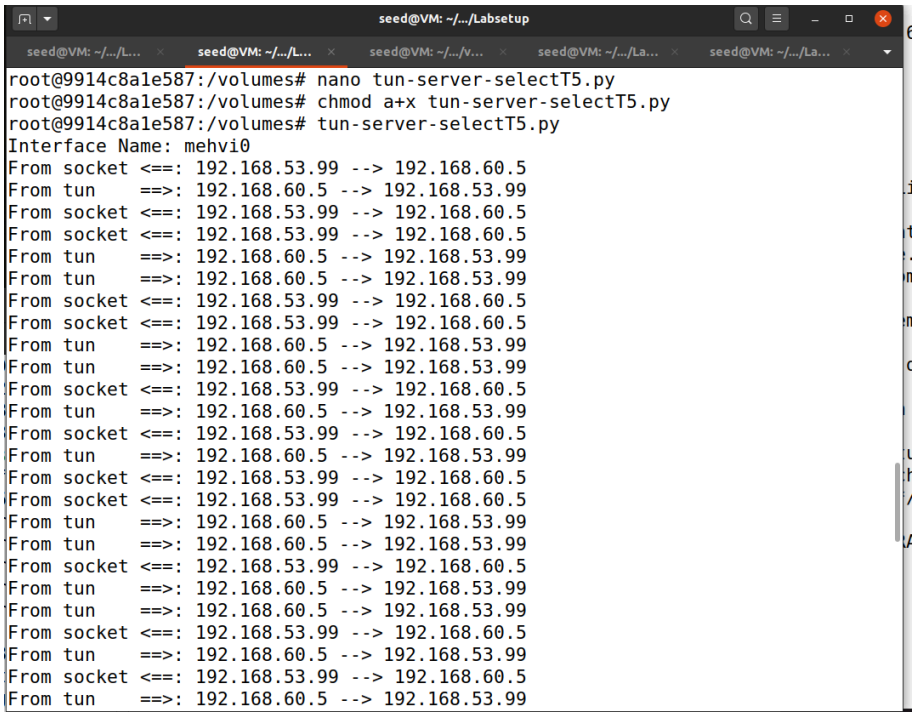

```
root@3425af571a92:/# ping 192.168.60.5 -c 3
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=6.13 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=7.28 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=2.81 ms

--- 192.168.60.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 2.814/5.406/7.281/1.893 ms
root@3425af571a92:/#
```

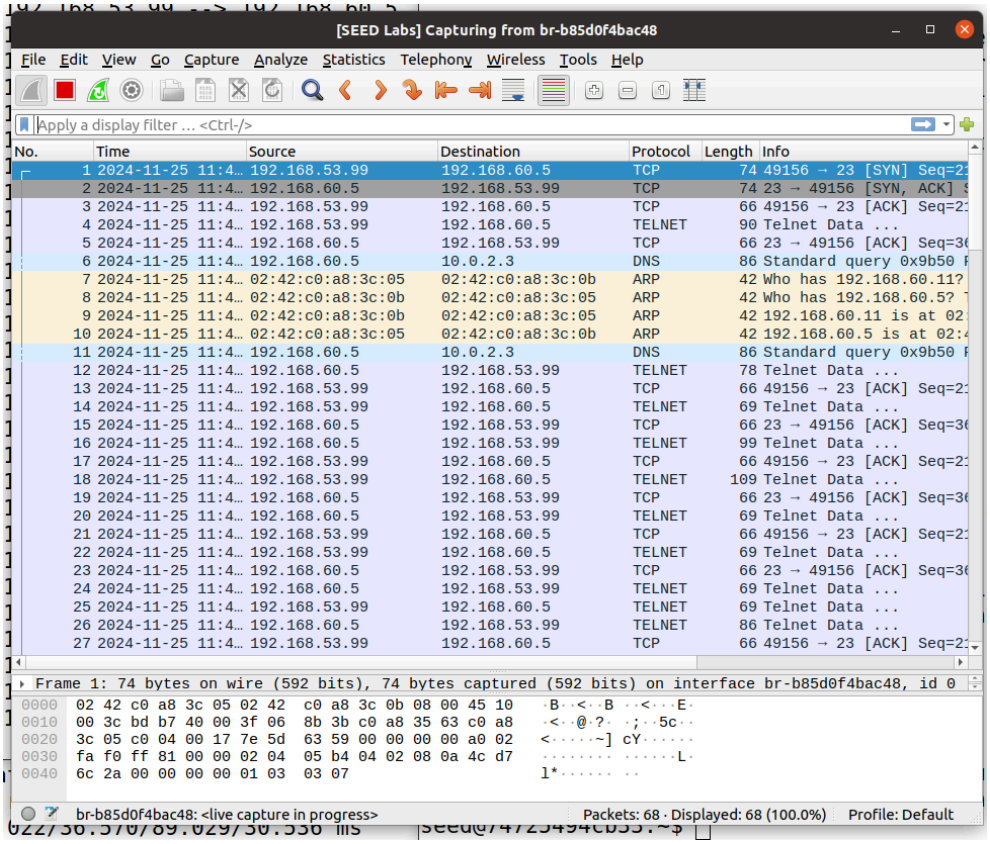
```
root@9914c8a1e587:/volumes# tun-serverT5.py
Interface Name: mehvi0
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.53.99
```

```
root@3425af571a92:/# tun-clientT5.py
Interface Name: mehvi0
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
```

```
root@3425af571a92:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
74725494cb33 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

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

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

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

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

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

seed@74725494cb33:~$ exit
logout
Connection closed by foreign host.
root@3425af571a92:/#
```


seed@VM: ~/.../volumes
seed@VM: ~/.../L...
seed@VM: ~/.../L...
seed@VM: ~/.../v...
seed@VM: ~/.../La...
seed@VM: ~/.../La...
root@3425af571a92:/# nano tun-client-selectT5.py
root@3425af571a92:/# chmod a+x tun-client-selectT5.py
root@3425af571a92:/# tun-client-selectT5.py
Interface Name: mehvi0
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 tun ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.53.99 --> 192.168.60.5
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 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 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


seed@VM: ~/.../Labsetup
seed@VM: ~/.../L...
seed@VM: ~/.../L...
seed@VM: ~/.../v...
seed@VM: ~/.../La...
seed@VM: ~/.../La...
root@9914c8a1e587:/volumes# nano tun-server-selectT5.py
root@9914c8a1e587:/volumes# chmod a+x tun-server-selectT5.py
root@9914c8a1e587:/volumes# tun-server-selectT5.py
Interface Name: mehvi0
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99



Here, initially, ICMP echo requests from Host U reached Host V via the VPN tunnel; but responses from Host V to Host U were dropped. This was due to the tunnel being configured for unidirectional traffic only.

To allow reverse traffic from HostV back to HostU, the VPN setup required handling data from two interfaces: TUN interface(handles outgoing traffic from HostU destined for HostV) and Socket Interface(receives incoming encapsulated data from the tunnel).

The Linux select() system call was used to monitor both interfaces simultaneously. Unlike constant polling, select() blocks the process until data is available on either interface, saving CPU resources while ensuring timely processing of data.

To implement:

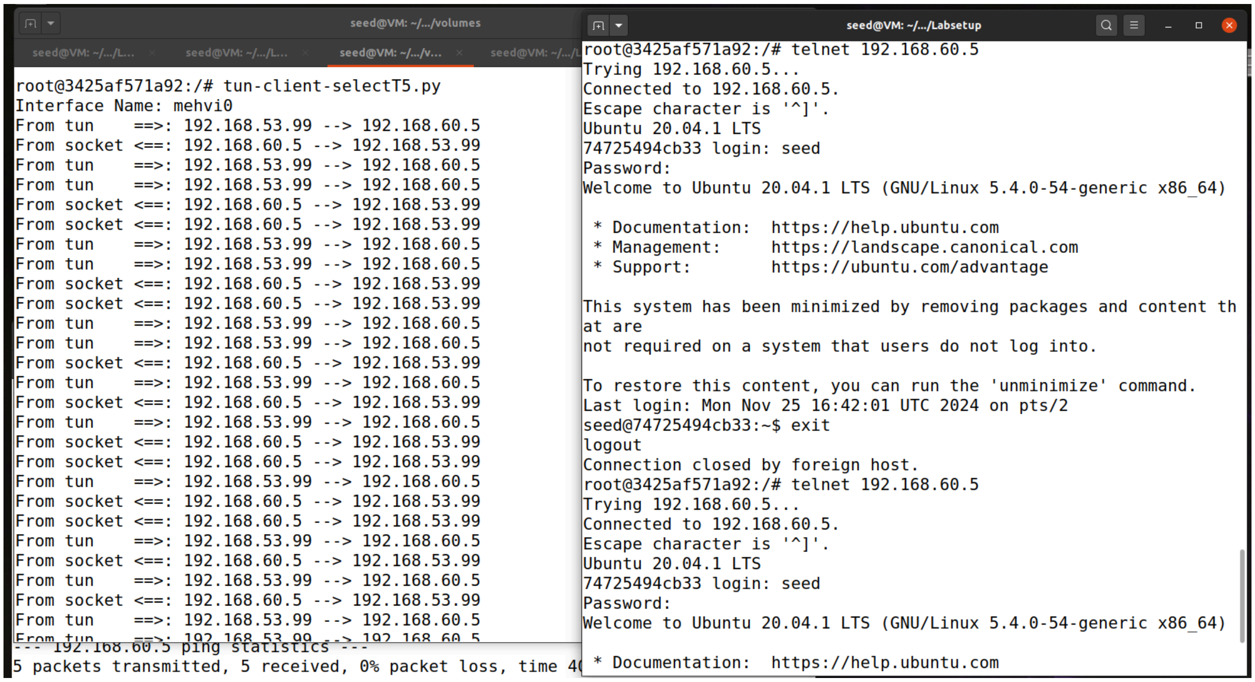
- Data from the TUN Interface: packets routed through the TUN Interface were encapsulated into UDP packets and sent via the socket interface toward the VPN server for delivery to HostV.
- Data from the Socket Interface: Encapsulated packets received from the tunnel were decapsulated and forwarded through the TUN interface to reach HostU.
- Efficient Handling: by identifying the interface with incoming data and processing it appropriately, the bidirectional communication flow was completed.

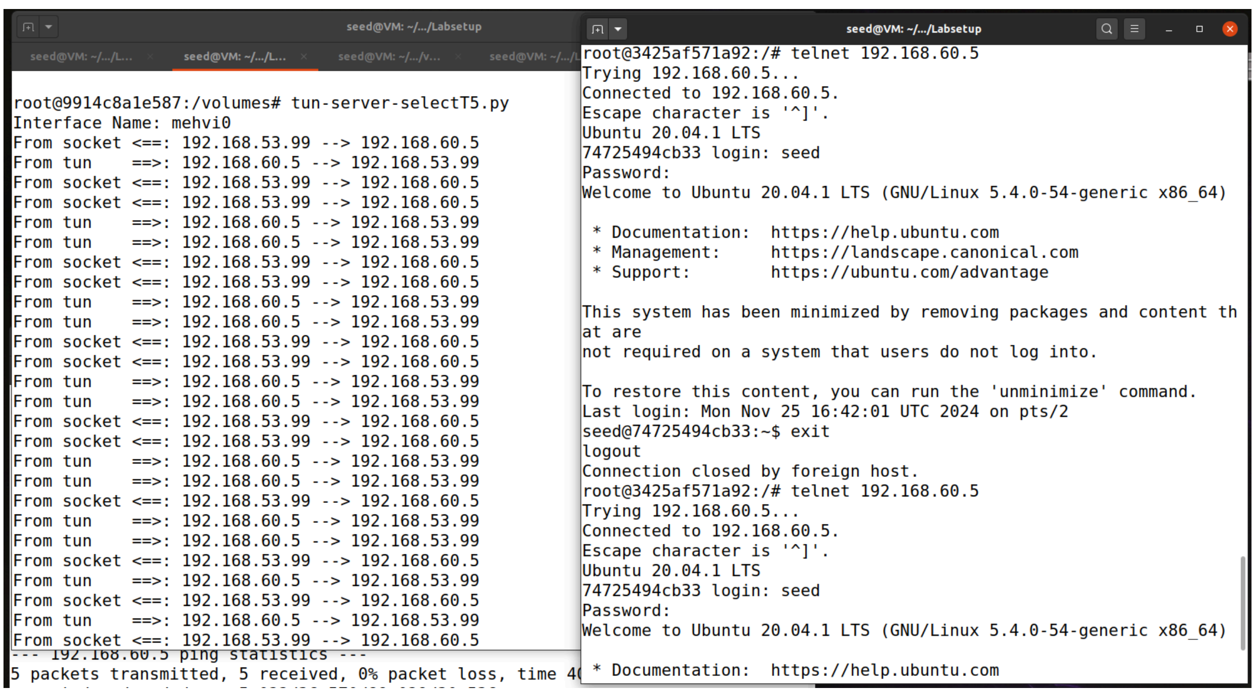
Later to test and Validate this, we firstly perform

- ping test: ICMP echo requests and replies were verified to flow correctly in both directions between HostU and HostV.
- Telnet test: TCP based communication was tested to ensure end-to-end functionality over the VPN tunnel.
- Traffic inspection: tools like Wireshark and tcpdump confirmed the correct encapsulation, decapsulation and routing of packets.

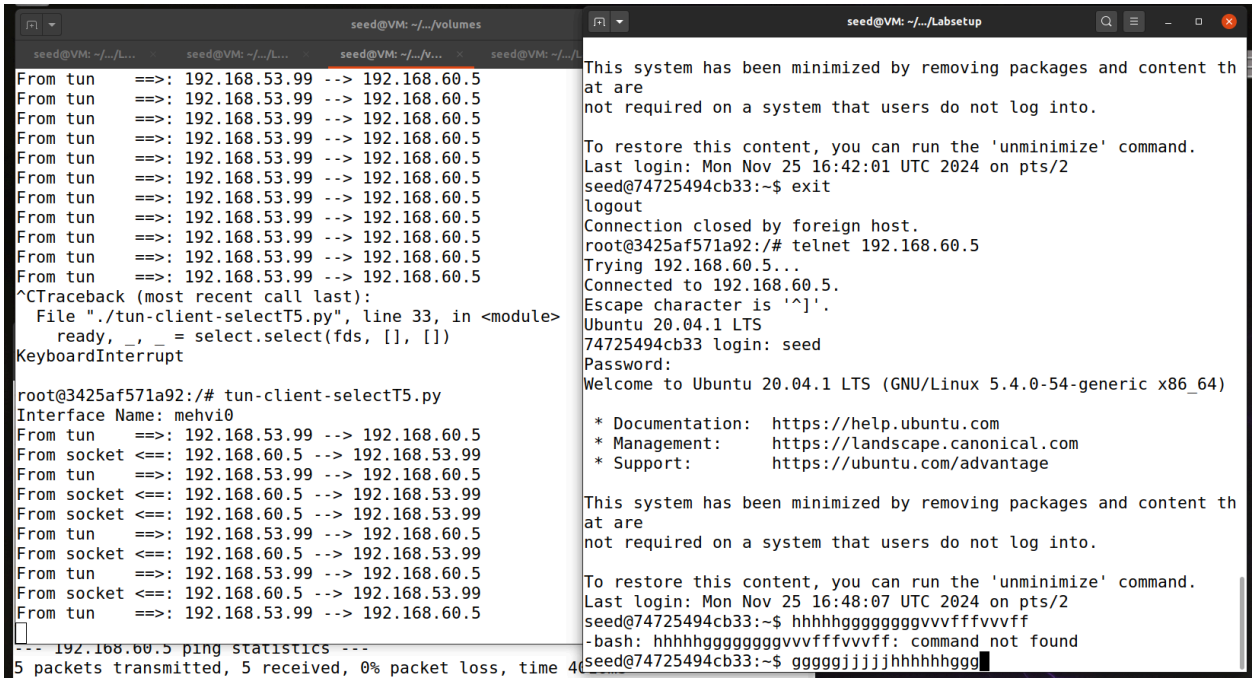
The VPN tunnel was successfully configured to handle traffic in both directions. Packets from HostU reached HostV, and responses from HostV were properly routed back to Host U, achieving full communication through the tunnel.

Task 6 Tunnel-Breaking Experiment





We found that no matter what was entered, nothing was displayed. The TCP connection is not broken. the connection can resume if the disconnection time is not too long.



While keeping the remote login online, the same situation occurs when the tunnel service of the client or server is broken, that is, no text can be entered in the remote login interface, and there will be no new output. When the service is reconnected for a short period of time, the backlogged packet buffers in the TUN file will be released one by one.

Here, the connection is not immediately broken: TCP has mechanisms like retransmissions and exponential backoff to handle temporary failures. It only closes a connection if no acknowledgement is received for a prolonged period. The connection remains in an active state, waiting for the tunnel to recover.

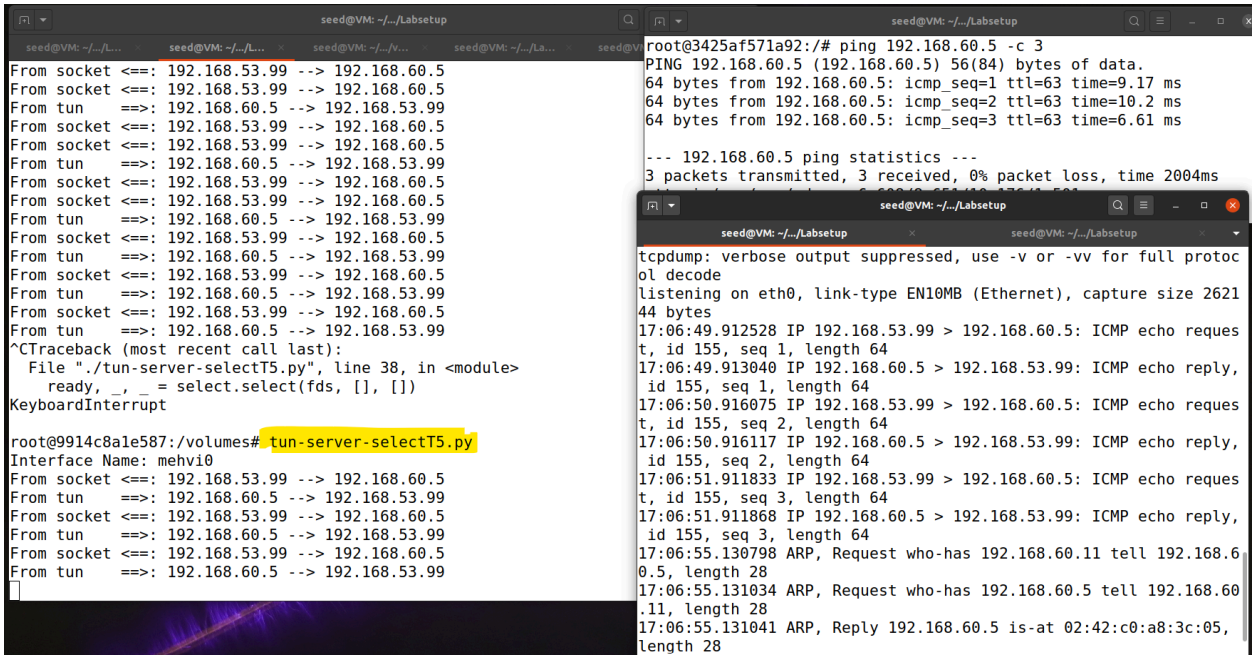
When the tunnel is re-established, the underlying network connectivity is restored. TCP resumes the session by retransmitting lost packets as it was never explicitly terminated.

Telnet depends on the TCP session since the session was not closed, no re-login is required.

The typed characters appear after reconnection because of delayed delivery. Text typed during the disconnection was held in TCP's retransmission queue. Upon reconnection, these packets are successfully delivered to the server, and the echoed responses are displayed.

Task 7 Routing Experiment on Host V

The task simulates real-world routing challenges and ensures return packets follow the correct path back to the VPN server.




```
root@3425af571a92:/# tun-client-selectT5.py
Interface Name: mehvi0
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
```

The default route is via 192.168.60.11.

Host V has a default route(via 192.168.60.11) directing all outbound packets to the VPN server. This setup ensures all traffic, including return packets from a Telnet session, uses the VPN tunnel.

To mimic a real-world scenario, we remove the default route from the Host V routing table and add a specific route for the VPN server.

Deleting the default route:

```
root@74725494cb33:/# 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@74725494cb33:/# ip route del default
root@74725494cb33:/# ip route
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5

root@74725494cb33:/#
```

Without the default route, traffic destined for networks outside the 192.168.60.0/24 range will not reach its destination. Due to this, connections relying on the default route like Telnet through the VPN will stop functioning.

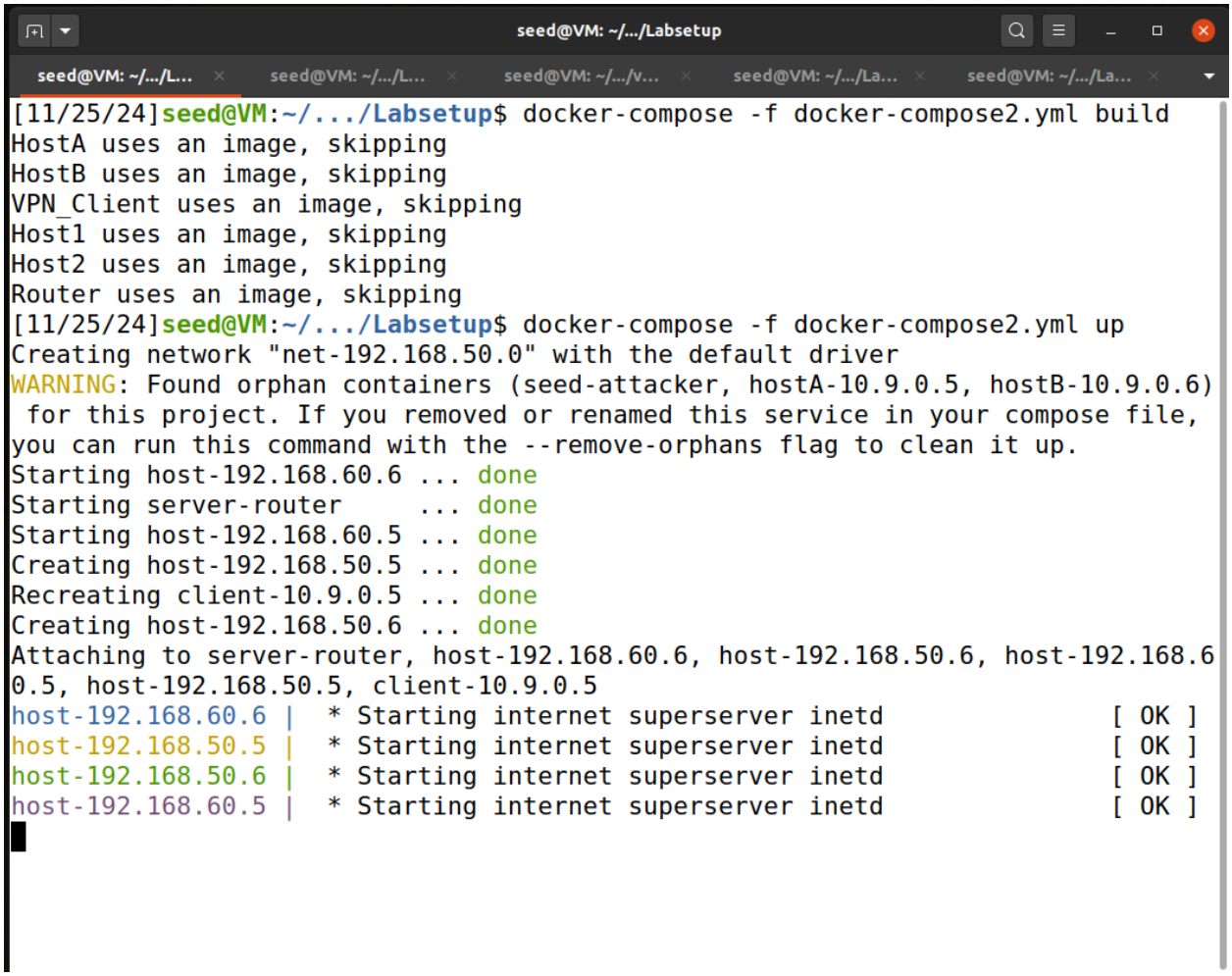
Adding a specific route:

```
root@74725494cb33:/# ip route add 192.168.53.0/24 via 192.168.60.11
root@74725494cb33:/# 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@74725494cb33:/#
```

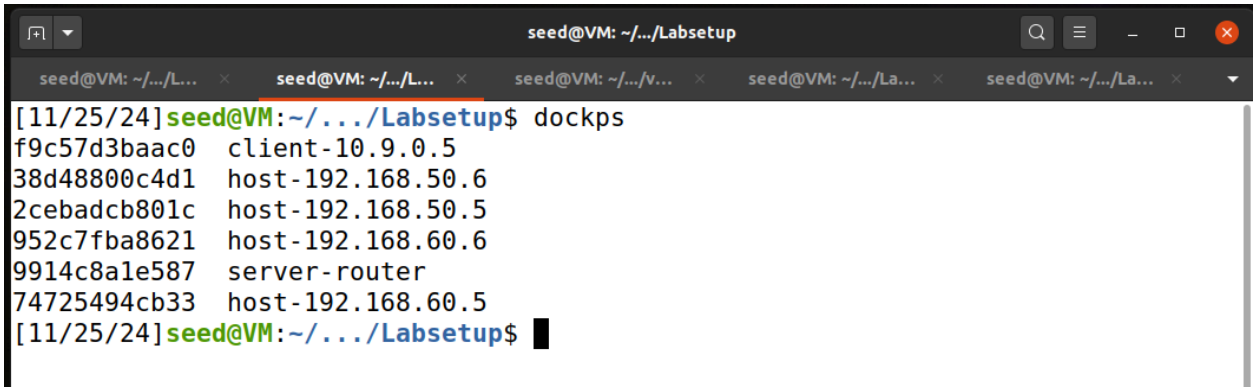
Adding this specific route ensures that any packet set from this host to an IP address in the 192.168.60.0/24 range will be routed through the gateway 192.168.60.11. Without this route, the system may not know how to reach the 192.168.60.0/24 network, leading to dropped packets.

Task 8 VPN Between Private Networks

In this task, we create a VPN tunnel to connect two private networks separated by the internet.



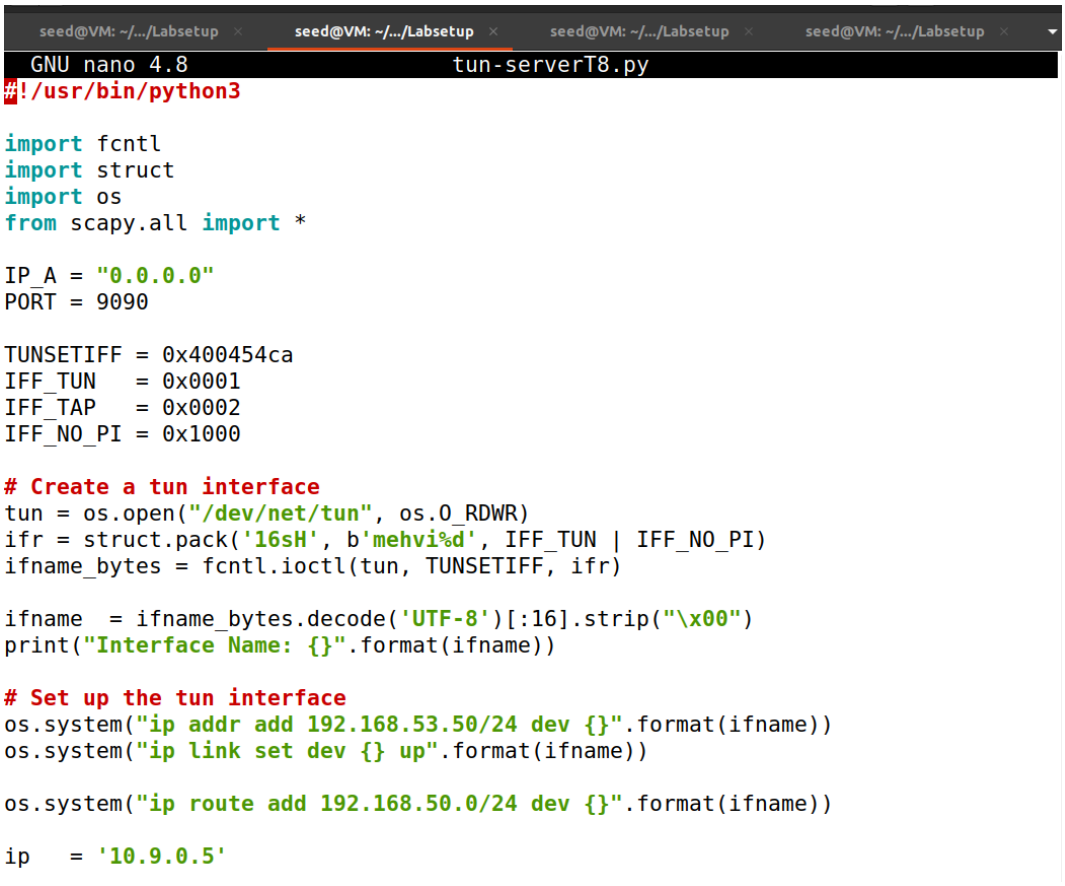
Here, build command creates the Docker images for the routers and devices, ensuring the correct environment and tools are available. The up command starts all containers, initializing the routers and network interfaces.



Here, ping to a host in a private network does not works as vpn not setup yet:

```
11/25/24]seed@VM:~/.../Labsetup$ dockps
f9c57d3baac0  client-10.9.0.5
38d48800c4d1  host-192.168.50.6
2cebadcb801c  host-192.168.50.5
952c7fba8621  host-192.168.60.6
0914c8a1e587  server-router
74725494cb33  host-192.168.60.5
11/25/24]seed@VM:~/.../Labsetup$ docksh host-192.168.50.6
root@38d48800c4d1:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
```

Code for tun-serverT8:



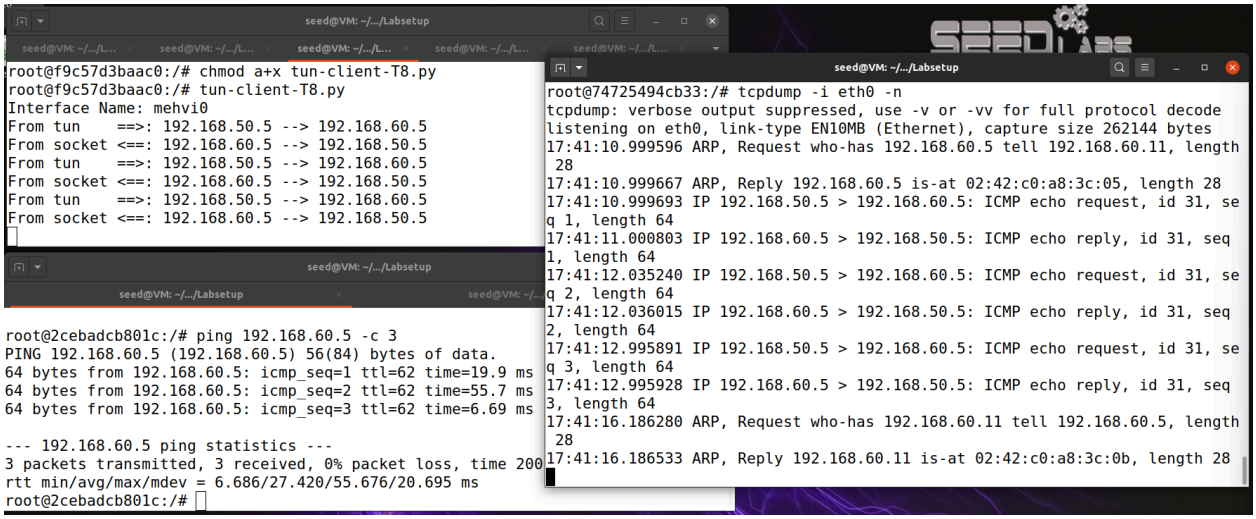
```
port = 10000

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

while True:
  # this will block until at least one socket 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, data)

    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))

^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pc
^X Exit      ^R Read File ^\ Replace   ^U Paste Text^T To Spell  ^_ Go To
```

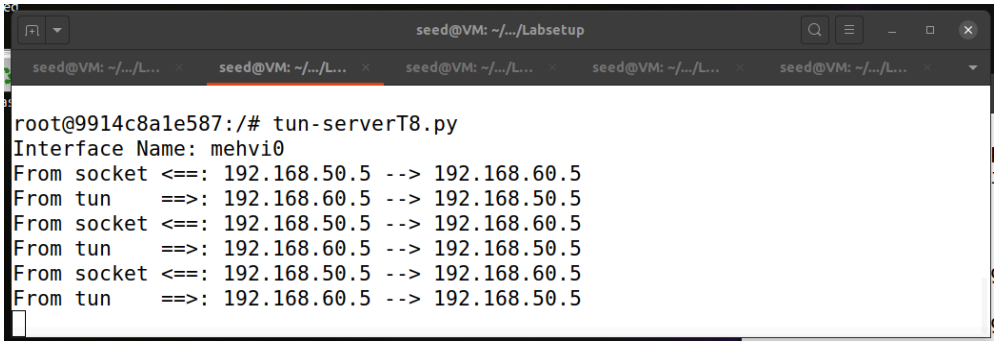
Code for tun-clientT8.py:

```
  GNU nano 4.8                    tun-clientT8.py                    Modified
#!/usr/bin/python3

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

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'mehvi%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 192.168.53.11/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))


os.system("ip route add 192.168.60.0/24 dev {}".format(ifname))
# Create UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

SERVER_IP = "10.9.0.11"
```

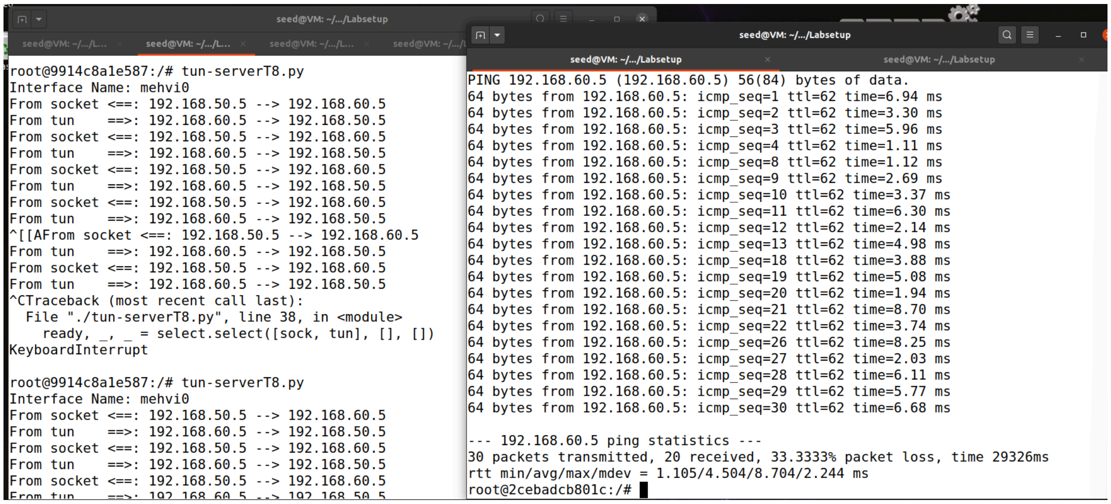
```
SERVER_PORT = 9090

pi = "10.9.0.11"
port = 1030

while True:
  # this will block until at least one socket 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, data)

    if fd is tun:
       packet = os.read(tun, 2048)
       pkt = IP(packet)
       print("From tun    ==>: {} --> {}".format(pkt.src, pkt.dst))
       sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

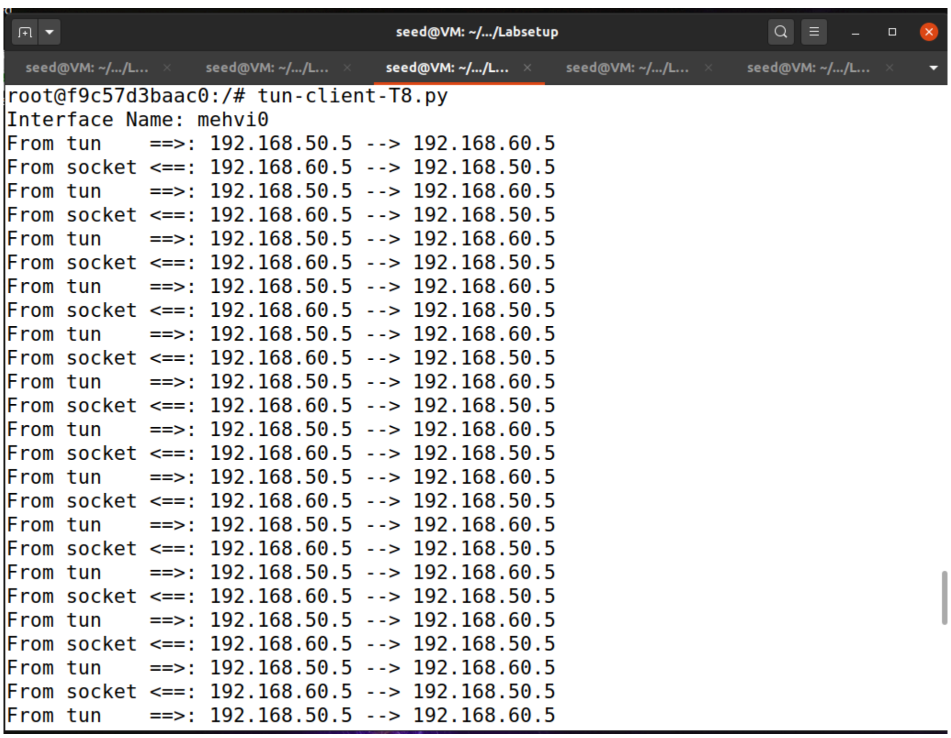
```
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Paste Text^T To Spell  ^_ Go To Line
```



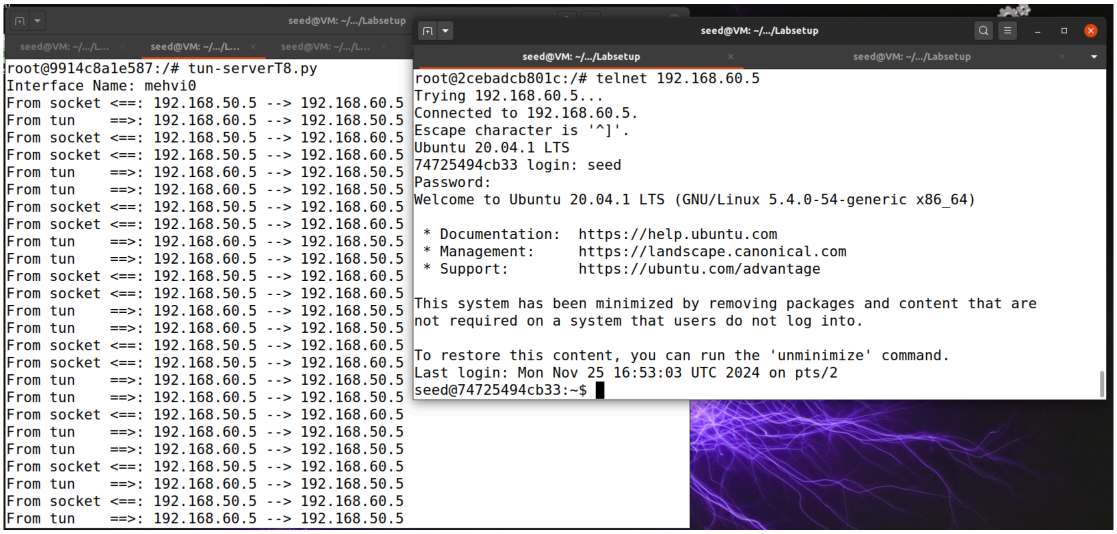


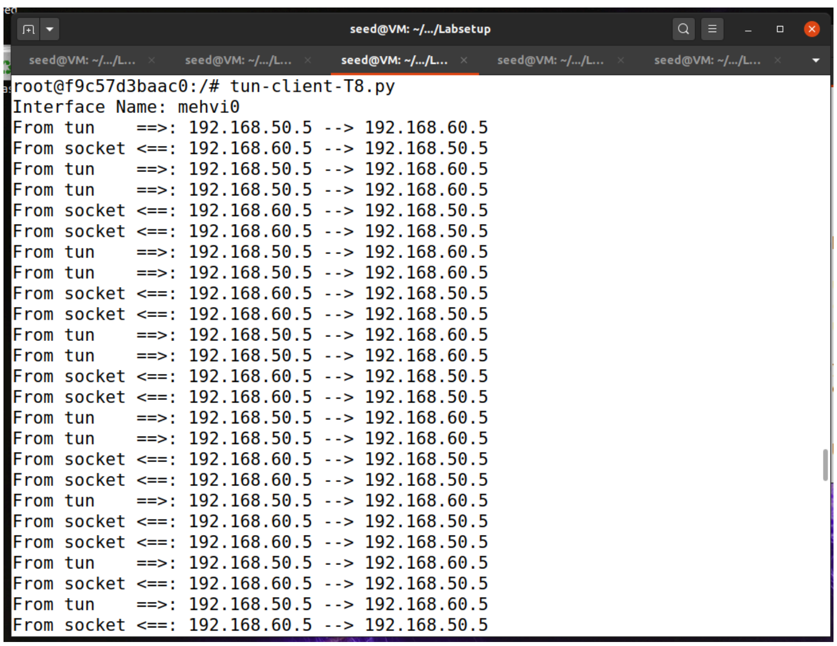
1 to check if proper connection break the connection and connect again to see that connection successfully starts again





Now we telnet from Host V to Host U to see if our VPN is configured properly - a connection being established

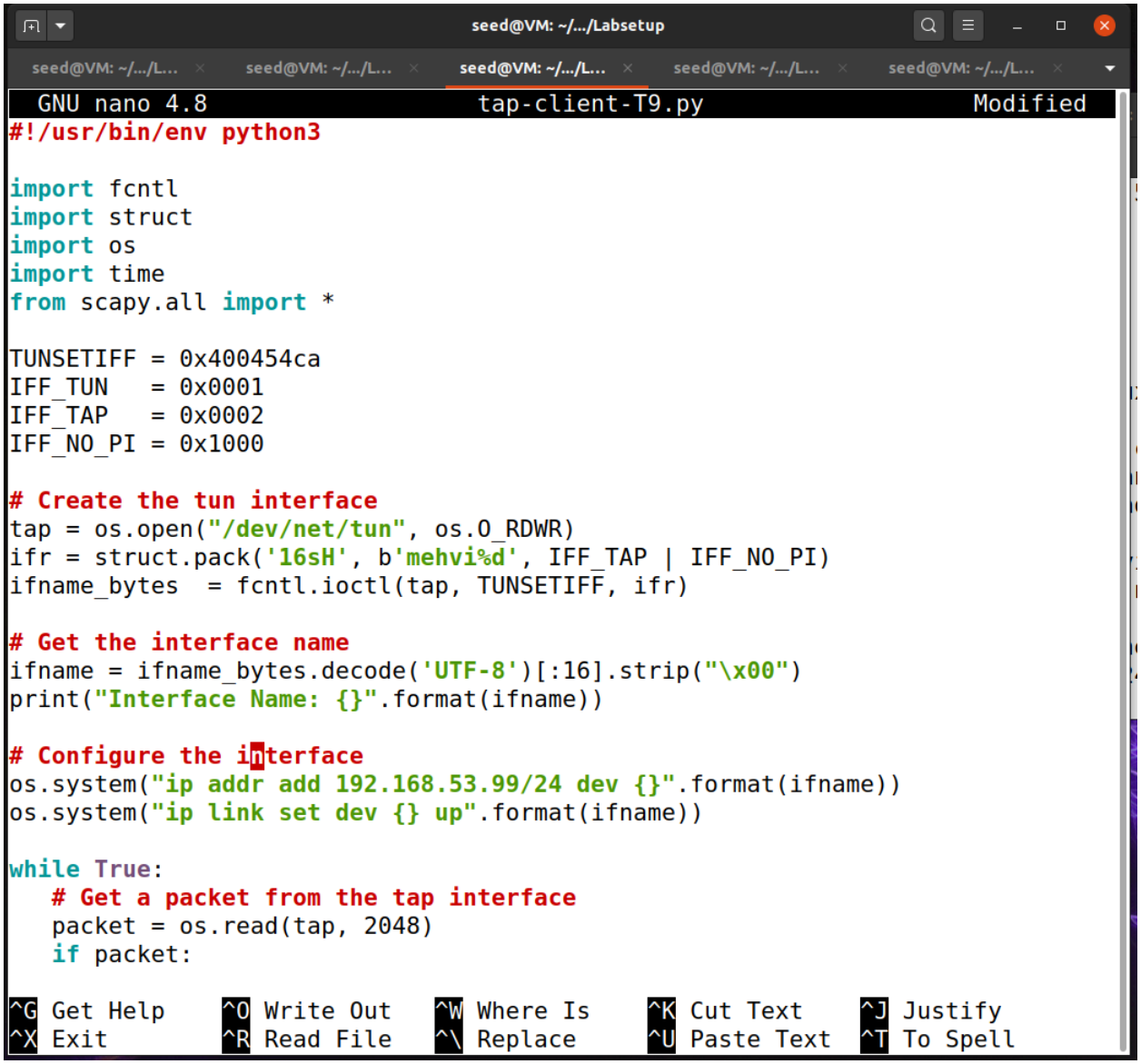




Therefore, we establish the connection for hosts on 192.168.50.0/24 can communicate with 192.168.60.0/24. These encapsulated packets confirm that traffic passes through the VPN tunnel and proper routing ensures the packets return via the correct path, avoiding network loops and drops.

Task 9: Experiment with the TAP Interface

To understand how TAP interfaces interact with the kernel and how they process and respond to Ethernet-level traffic.



When we ping 192.168.53.1, the interface directs the ARP packets to the tunnel interface. Since it doesn't know which machine has that IP address, it sends a "destination host unreachable" message

```
root@f9c57d3baac0:/volumes# nano tap-client-T9.py
root@f9c57d3baac0:/volumes# chmod a+x tap-client-T9.py
root@f9c57d3baac0:/volumes# tap-client-T9.py
Interface Name: mehvi0
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
```

```
root@f9c57d3baac0:/# ping 192.168.53.1
PING 192.168.53.1 (192.168.53.1) 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
From 192.168.53.99 icmp_seq=7 Destination Host Unreachable
^C
--- 192.168.53.1 ping statistics ---
10 packets transmitted, 0 received, +7 errors, 100% packet loss, time 9140ms
pipe 4
root@f9c57d3baac0:/#
```

Now we modify code to send spoofed ARP packets from a fake MAC address

```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'mehvi%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))

# 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))

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))
```

After running an ARP ping to the IP address 192.168.53.33, we received spoofed reply messages from the MAC address for each request sent.

```
root@f9c57d3baac0:/# arping -I mehvi0 192.168.53.33 -c 5
ARPING 192.168.53.33
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=0 time=4.012 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=1 time=4.166 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=2 time=5.006 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=3 time=4.415 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=4 time=6.348 msec

--- 192.168.53.33 statistics ---
5 packets transmitted, 5 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 4.012/4.789/6.348/0.850 ms
root@f9c57d3baac0:/#
```

```
root@f9c57d3baac0:/volumes# nano tap-client-ARP-T9.py
root@f9c57d3baac0:/volumes# chmod a+x tap-client-ARP-T9.py
root@f9c57d3baac0:/volumes# tap-client-ARP-T9.py
Interface Name: mehvi0
--------------------------------
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
```

When we attempt an ARP ping to the IP address 1.2.3.4, we receive spoofed reply messages from the MAC address for each request sent.

```
root@f9c57d3baac0:/# arping -I mehvi0 1.2.3.4 -c 5
ARPING 1.2.3.4
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=0 time=4.039 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=1 time=738.470 usec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=2 time=7.104 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=3 time=9.005 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=4 time=1.760 msec

--- 1.2.3.4 statistics ---
5 packets transmitted, 5 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 0.738/4.529/9.005/3.129 ms
root@f9c57d3baac0:/#
```

```
root@f9c57d3baac0:/volumes# tap-client-ARP-T9.py
Interface Name: mehvi0
--------------------------------
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
--------------------------------
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
```

Therefore, we explored the functionality of the TAP interface, learned how to manipulate MAC-level frames and gained a deeper understanding of network protocols like ARP along with their potential security implications.