```
[11/22/25]seed@1006859:~/.../Labsetup$ dockps
62bdbdd96608  host-192.168.60.5
b23925a2639c  client-10.9.0.5
c1d162444e27  server-router
7df5ed2f38f0  host-192.168.60.6
```

The above will be the details of the container setup for this lab.

Task 1: Network Setup

1. Host U can communicate with VPN Server.

```
root@b23925a2639c:/# ping 10.9.0.11
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
64 bytes from 10.9.0.11: icmp_seq=1 ttl=64 time=0.218 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.228 ms

--- 10.9.0.11 ping statistics ---
^C2 packets transmitted, 2 received, 0% packet loss, time 1008ms
rtt min/avg/max/mdev = 0.218/0.223/0.228/0.005 ms
root@b23925a2639c:/#
```

   From Host U, we can see that Host U can reach the VPN server.

2. VPN Server can communicate with Host V.

```
root@c1d162444e27:/# 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=0.306 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.094 ms
^C
--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1022ms
rtt min/avg/max/mdev = 0.094/0.200/0.306/0.106 ms
root@c1d162444e27:/#
```

   From the VPN Server, we can see that we can reach Host V.

3. Host U should not be able to communicate with Host V.

```
root@b23925a2639c:/# 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 ---
3 packets transmitted, 0 received, 100% packet loss, time 2039ms

root@b23925a2639c:/#
```

   From Host U, we are unable to reach Host V, so we can't communicate with it.

4. Run tcpdump on the router, and sniff the traffic on each of the network. Show that you can capture packets.
   First, I ran tcpdump on the eth0 interface of the router, and proceeded to ping the router from Host U. From the image below, we can see that we are able to capture the ping requests and replies between Host U and the router.

```
root@c1d162444e27:/# 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
06:18:31.714420 IP6 fe80::42:91ff:fe70:abec > ff02::2: ICMP6, router solicitation, length 16
06:19:05.115201 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 33, seq 1, length 64
06:19:05.115628 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 33, seq 1, length 64
06:19:06.116262 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 33, seq 2, length 64
06:19:06.116318 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 33, seq 2, length 64
06:19:10.369827 ARP, Request who-has 10.9.0.5 tell 10.9.0.11, length 28
06:19:10.369883 ARP, Request who-has 10.9.0.11 tell 10.9.0.5, length 28
06:19:10.369906 ARP, Reply 10.9.0.11 is-at 02:42:0a:09:00:0b, length 28
06:19:10.369926 ARP, Reply 10.9.0.5 is-at 02:42:0a:09:00:05, length 28
```

Then, I also ran tcpdump on the eth1 interface of the router, and proceeded to ping the router from Host V. From the image below, we can see that we are able to capture the ping requests and replies between Host V and the router.

```
root@c1d162444e27:/# 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
06:22:52.565890 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 42, seq 1, length 64
06:22:52.566047 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 42, seq 1, length 64
06:22:53.595749 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 42, seq 2, length 64
06:22:53.595777 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 42, seq 2, length 64
06:22:57.691588 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
06:22:57.691890 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
06:22:57.691917 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
06:22:57.691941 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
```

Task 2: Create and Configure TUN Interface

Task 2.a: Name of the Interface

Executing tun.py on Host U, we know the interface name that we created is tun0.

```
root@b23925a2639c:/volumes# tun.py
Interface Name: tun0
```

From “ip address”, we can see that apart from the original eth0 interface, we now have a tun0 interface, but note that it is “DOWN”.

```
root@b23925a2639c:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
167: eth0@if168: <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@b23925a2639c:/#
```

To change the prefix of the interface name, we can change the highlighted portion as shown in the image below.

```
root@b23925a2639c:/volumes# cat tun.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'pea%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

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

Running tun.py showed that the interface name has been changed successfully.

```
root@b23925a2639c:/volumes# tun.py
Interface Name: pea0
```

Task 2.b: Set up the TUN Interface

```
#!/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'tun%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))

# Assign IP address to the interface
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))

# Bring up the interface
os.system("ip link set dev {} up".format(ifname))

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

After running the above tun.py to set up the TUN interface, we obtain the results below when running “ip address”.

```
root@b23925a2639c:/volumes# 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
5: 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
167: eth0@if168: <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
```

Previously, tun0 was stated as “DOWN” and was not assigned to any IP address. But after running the 2 commands above, we can see that it is now “UNKNOWN” instead of “DOWN”, and it is assigned to the IP address 192.168.53.99/24.

Task 2.c: Read from the TUN Interface

First, we update the while loop as shown below in tun.py.

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

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

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

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

# Assign IP address to the interface
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))

# Bring up the interface
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())
```

1. On Host U, ping a host in the 192.168.53.0/24 network. What are printed out by the tun.py program? What has happened? Why?

   First, I pinged a host in the network, 192.168.53.10 from Host U.

   ```
   root@b23925a2639c:/# ping 192.168.53.10
   PING 192.168.53.10 (192.168.53.10) 56(84) bytes of data.
   ^C
   --- 192.168.53.10 ping statistics ---
   3 packets transmitted, 0 received, 100% packet loss, time 2036ms
   ```

From the image below, we can see that tun.py captured ICMP echo requests only.

```
root@b23925a2639c:/volumes# tun.py
Interface Name: tun0
IP / ICMP 192.168.53.99 > 192.168.53.10 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.10 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.10 echo-request 0 / Raw
```

This occurred because there are no other hosts on the 192.168.53.0/24 network apart from Host U since the interface was created by Host U and hosts have not been added to the network yet.

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

   First, I pinged 192.168.60.5 from Host U.

```
root@b23925a2639c:/# 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 ---
2 packets transmitted, 0 received, 100% packet loss, time 1011ms
```

   From the image below, we can see that tun.py did not print anything new, which means no packets were captured.

```
root@b23925a2639c:/volumes# tun.py
Interface Name: tun0
IP / ICMP 192.168.53.99 > 192.168.53.10 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.10 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.10 echo-request 0 / Raw
```

   This occurred as tun.py is only capturing packets that travel through the tun0 interface. Since tun0 is only responsible for the 192.168.53.0/24 network, it will not capture packets travelling outside the network as 192.168.60.5 is not in the 192.168.53.0/24 network.
   We would only be able to ping 192.168.53.99 as it is the IP address of the local adapter of Host U.

Task 2.d: Write to the TUN Interface

1. After getting a packet from the TUN interface, if this packet is an ICMP echo request packet, construct a corresponding echo reply packet and write it to the TUN interface. Please provide evidence to show that the code works as expected.

   Below, we have changed tun.py, such that it checks whether the packet is an ICMP echo request, then it flips the source and destination, change it to an ICMP echo reply packet, and write the spoofed packet back to the TUN interface.

```
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'tun%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))

# Assign IP address to the interface
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))

# Bring up the interface
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())
      if ip.haslayer(ICMP):
         icmp = ip.getlayer(ICMP)
         if icmp.type == 8:
            print("changing packet")
            newip = IP(src=ip.dst, dst=ip.src)
            newicmp = ICMP(type=0, id=icmp.id, seq=icmp.seq)
            newpkt = newip/newicmp/icmp.payload
            os.write(tun, bytes(newpkt))
```

Previously, we were not able to ping 192.168.53.10. After running tun.py, we were able to successfully ping 192.168.53.10, as shown below.

```
root@b23925a2639c:/# ping 192.168.53.10
PING 192.168.53.10 (192.168.53.10) 56(84) bytes of data.
64 bytes from 192.168.53.10: icmp_seq=1 ttl=64 time=10.6 ms
64 bytes from 192.168.53.10: icmp_seq=2 ttl=64 time=3.56 ms
^C
--- 192.168.53.10 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1003ms
rtt min/avg/max/mdev = 3.562/7.070/10.579/3.508 ms
root@b23925a2639c:/#
```

2. Instead of writing an IP packet to the interface, write some arbitrary data to the interface, and report your observation.

   I have added a few lines to write “i am writing data” to the interface as shown below.

```
while True:
   # Get a packet from the tun interface
   packet = os.read(tun, 2048)
   if packet:
      ip = IP(packet)
      print(ip.summary())
      if ip.haslayer(ICMP):
         icmp = ip.getlayer(ICMP)
         if icmp.type == 8:
            print("changing packet")
            newip = IP(src=ip.dst, dst=ip.src)
            newicmp = ICMP(type=0, id=icmp.id, seq=icmp.seq)
            newpkt = newip/newicmp/icmp.payload
            os.write(tun, bytes(newpkt))
   print(bytes("i am writing data", encoding="ascii"))
   os.write(tun, bytes("i am writing data", encoding="ascii"))
```

   However, we are not able to read the data on the interface as shown by the output of tun.py as shown below.

```
root@b23925a2639c:/volumes# tun.py
Interface Name: tun0
IP / ICMP 192.168.53.99 > 192.168.53.10 echo-request 0 / Raw
changing packet
b'i am writing data'
```

This means that the packet could possibly be discarded, hence it was not captured by tun.py.

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

For tun_client.py:

```
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'tun%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))

# Assign IP address to the interface
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))

# Bring up the interface
os.system("ip link set dev {} up".format(ifname))

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

while True:
   # Get a packet from the tun interface
   packet = os.read(tun,2048)
   if packet:
      # Send the packet via the tunnel
      sock.sendto(packet, ("10.9.0.11", 9090))
```

For tun_server.py:

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

from scapy.all import *

IP_A = "0.0.0.0"
PORT = 9090

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

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

Running tun_server.py on the VPN server and tun_client.py on Host U, we now ping 192.168.53.10, which belongs to the 192.168.53.0/24 network as shown below.

```
root@b23925a2639c:/# ping 192.168.53.10
PING 192.168.53.10 (192.168.53.10) 56(84) bytes of data.
^C
--- 192.168.53.10 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 1ms

root@b23925a2639c:/#
```

On the VPN server, we can see the following output due to the ping request as shown below.

```
root@c1d162444e27:/volumes# tun_server.py
10.9.0.5:58852 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.53.10
```

We can see that a UDP packet is being sent from Host U port 58852 to the VPN server's port 9090. However, there is an IP packet hidden inside the UDP packet, which is sent within the 192.168.53.0/24 network. This is because after initialising the UDP socket, "sock.sendto" will wrap our IP packet in a UDP packet. Hence, we can see the output as shown above.

After pinging Host V, we realise that the ICMP packet has not been sent to the VPN server through the tunnel as shown below (no new output).

```
root@b23925a2639c:/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 ---
2 packets transmitted, 0 received, 100% packet loss, time 1015ms
```

```
root@c1d162444e27:/volumes# tun_server.py
10.9.0.5:58852 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.53.10
```

They are not sent to the VPN server through the tunnel because we have not resolved the route for packets going to the 192.168.60.0/24 network. The tunnel is only responsible for packets in the 192.168.53.0/24 network. Hence, we need to add a route to the 192.168.60.0/24 network through the tunnel interface.

So, we can do "ip route add 192.168.60.0/24 dev tun0 via 192.168.53.99".
Now, when we ping Host V:

```
root@b23925a2639c:/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 ---
2 packets transmitted, 0 received, 100% packet loss, time 1022ms
```

We can see that the ICMP packets are received through the tunnel as shown below, where the ICMP packets are going from 192.168.53.99 to 192.168.60.5.

```
root@c1d162444e27:/volumes# tun_server.py
10.9.0.5:58852 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.53.10
10.9.0.5:34431 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:34431 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
```

Task 4: Set Up the VPN Server

First, we modify tun_server.py so that it creates a TUN interface and configure it. Then, we will treat the received data as an IP packet, and write it to the TUN interface.

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

import fcntl
import struct
import os
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'tun%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))

# Assign IP address to the interface
os.system("ip addr add 192.168.53.0/24 dev {}".format(ifname))

# Bring up the interface
os.system("ip link set dev {} up".format(ifname))

IP_A = "0.0.0.0"
PORT = 9090

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

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

Since IP forwarding is enabled, the packets meant for the 192.168.60.0/24 network will be routed to that network automatically.

After pinging Host V from the VPN client:

```
root@c1d162444e27:/volumes# tun_server.py
Interface Name: tun0
10.9.0.5:34431 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
Writing packet to the TUN interface
```

We can see that Host V received the ICMP echo request packets successfully with tcpdump.

```
root@62bdbdd96608:/# 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
07:35:22.371960 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 381, seq 1, length 64
07:35:22.372079 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 381, seq 1, length 64
07:35:27.571128 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
07:35:27.571358 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
07:35:27.571400 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
07:35:27.571445 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
```

Task 5: Handling Traffic in Both Directions

For tun_client.py:

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

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

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

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

# Assign IP address to the interface
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))

# Bring up the interface
os.system("ip link set dev {} up".format(ifname))

# Add route
os.system("ip route add 192.168.60.0/24 dev {} via 192.168.53.99".format(ifname))

# Create UDP Socket
IP_A = "0.0.0.0"
PORT = 9090

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

```python
while True:
   # this will block until at least one interface is ready
   ready, _, _ = select.select([sock,tun], [], [])

   for fd in ready:
      if fd is sock:
         data, (ip, port) = sock.recvfrom(2048)
         pkt = IP(data)
         print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
         os.write(tun, bytes(pkt))

      if fd is tun:
         packet = os.read(tun, 2048)
         pkt = IP(packet)
         print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
         sock.sendto(packet, ("10.9.0.11", 9090))
```

We binded the UDP socket to port 9090 so that data received on the tun interface on the VPN server can be sent to our socket.

If the packet came from the socket, it must be from the VPN server, so we assume it's a UDP packet that encapsulates an IP packet, so we extract the IP packet.

If the packet came from the tun interface, it needs to be encapsulated inside a UDP packet before sending it to the VPN server, hence it will go through the socket.

For tun_server.py:

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

import fcntl
import struct
import os
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'tun%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))

# Assign IP address to the interface
os.system("ip addr add 192.168.53.0/24 dev {}".format(ifname))

# Bring up the interface
os.system("ip link set dev {} up".format(ifname))

IP_A = "0.0.0.0"
PORT = 9090

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

```python
while True:
    # this will block until at least one interface is ready
    ready, _, _ = select.select([sock, tun], [], [])

    for fd in ready:
        if fd is sock:
            data, (ip, port) = sock.recvfrom(2048)
            pkt = IP(data)
            print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
            os.write(tun, bytes(pkt))

        if fd is tun:
            packet = os.read(tun, 2048)
            pkt = IP(packet)
            print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
            sock.sendto(packet, ("10.9.0.5", 9090))
```

When the VPN server receives a packet from the socket, it's assumed to contain an IP packet inside a UDP packet, so the IP packet is extracted.
When the VPN server receives a packet from the tun interface, it needs to be encapsulated in a UDP packet before sending it to the VPN client, so it will go through the UDP socket.

Ping

Running both tun_client.py and tun_server.py, I proceeded to ping Host V from Host U as shown below.

```
root@b23925a2639c:/volumes# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=4.15 ms
^C
--- 192.168.60.5 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 4.149/4.149/4.149/0.000 ms
```

```
root@b23925a2639c:/volumes# tun_client.py
Interface Name: tun0
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
```

```
root@c1d162444e27:/volumes# tun_server.py
Interface Name: tun0
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
```

We successfully obtained the ping reply from Host V on Host U, hence there is traffic flowing in both directions.

We also captured the ping request through Wireshark.

No.	Time	Source	Destination	Protocol	Length	Info
1	2025-11-23 04:1…	10.9.0.5	10.9.0.11	UDP	126	9090 → 9090 Len=84
2	2025-11-23 04:1…	10.9.0.11	10.9.0.5	UDP	126	9090 → 9090 Len=84
3	2025-11-23 04:1…	02:42:0a:09:00:05	02:42:0a:09:00:0b	ARP	42	Who has 10.9.0.11? Tell 10.9.0.5
4	2025-11-23 04:1…	02:42:0a:09:00:0b	02:42:0a:09:00:05	ARP	42	10.9.0.11 is at 02:42:0a:09:00:0b
5	2025-11-23 04:1…	02:42:0a:09:00:0b	02:42:0a:09:00:05	ARP	42	Who has 10.9.0.5? Tell 10.9.0.11
6	2025-11-23 04:1…	02:42:0a:09:00:05	02:42:0a:09:00:0b	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05

The image above shows the UDP packet that is being sent between the VPN client and server.

No.	Time	Source	Destination	Protocol	Length	Info
1	2025-11-23 04:1…	192.168.53.99	192.168.60.5	ICMP	98	Echo (ping) request id=0x01c5, seq=1/256, ttl=63 (reply in 2)
2	2025-11-23 04:1…	192.168.60.5	192.168.53.99	ICMP	98	Echo (ping) reply id=0x01c5, seq=1/256, ttl=64 (request in…
3	2025-11-23 04:1…	02:42:c0:a8:3c:05	02:42:c0:a8:3c:0b	ARP	42	Who has 192.168.60.11? Tell 192.168.60.5
4	2025-11-23 04:1…	02:42:c0:a8:3c:0b	02:42:c0:a8:3c:05	ARP	42	Who has 192.168.60.5? Tell 192.168.60.11
5	2025-11-23 04:1…	02:42:c0:a8:3c:0b	02:42:c0:a8:3c:05	ARP	42	192.168.60.11 is at 02:42:c0:a8:3c:0b
6	2025-11-23 04:1…	02:42:c0:a8:3c:05	02:42:c0:a8:3c:0b	ARP	42	192.168.60.5 is at 02:42:c0:a8:3c:05

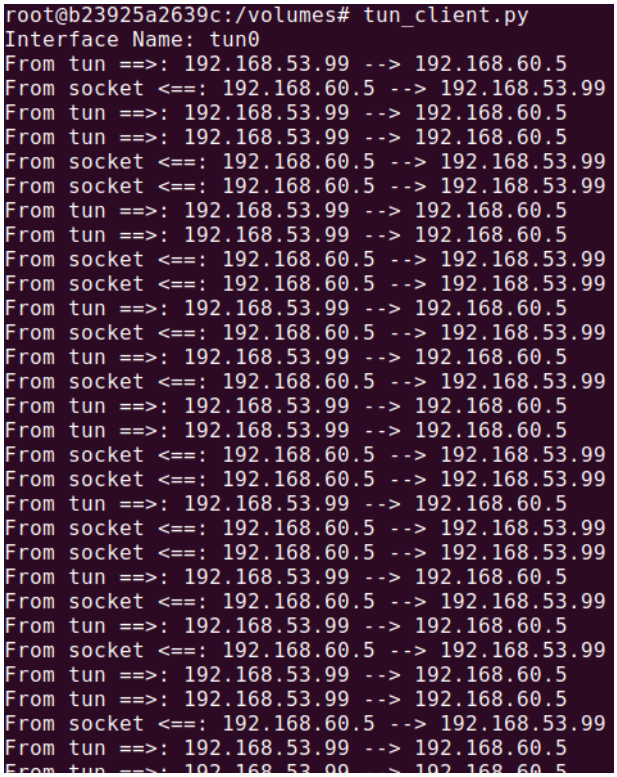
The image above shows the ICMP echo request and reply packets that were sent to and from Host V.

Telnet

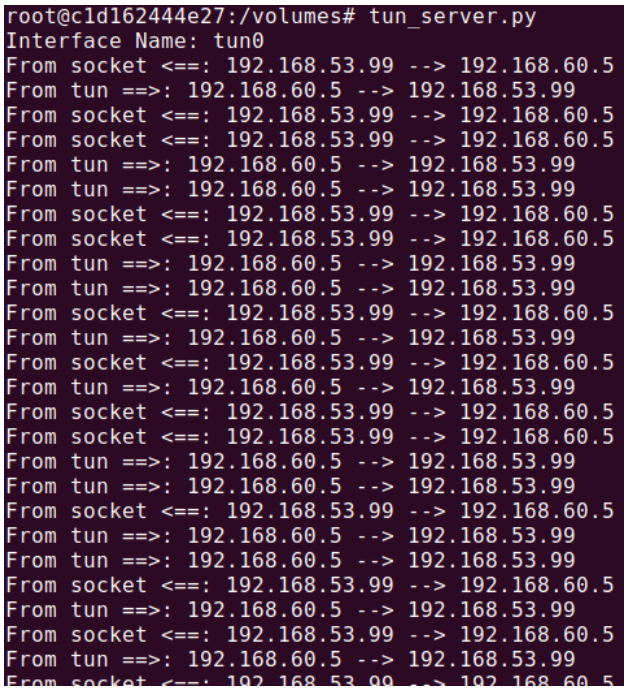
Running both tun_client.py and tun_server.py, I proceeded to telnet to Host V from Host U as shown below.

```
root@b23925a2639c:/volumes# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
62bdbdd96608 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

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

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

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

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

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

seed@62bdbdd96608:~$
```

From tun_client.py:

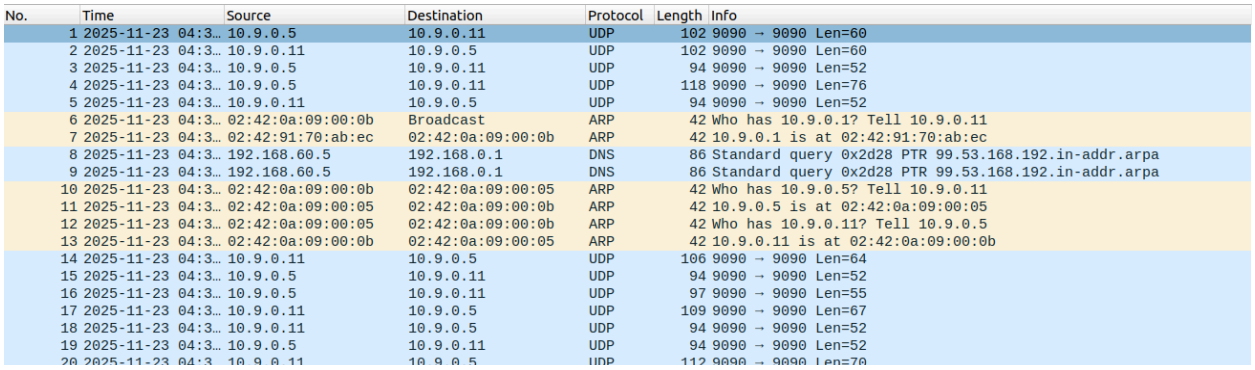


From tun_server.py:



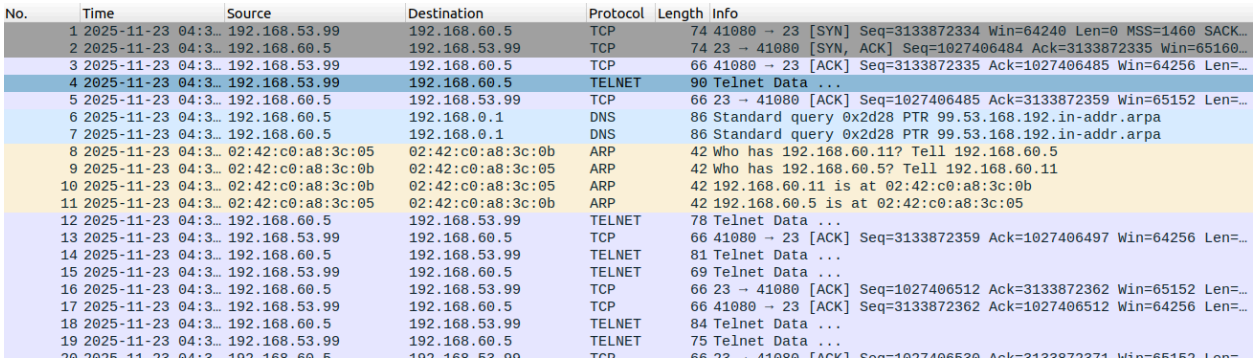
From the first image, we can see that we successfully Telnet into Host V from Host U, hence there is traffic flowing in both directions.

We also captured the packets through Wireshark.



No.	Time	Source	Destination	Protocol	Length	Info
1	2025-11-23 04:3…	10.9.0.5	10.9.0.11	UDP	102	9090 → 9090 Len=60
2	2025-11-23 04:3…	10.9.0.11	10.9.0.5	UDP	102	9090 → 9090 Len=60
3	2025-11-23 04:3…	10.9.0.5	10.9.0.11	UDP	94	9090 → 9090 Len=52
4	2025-11-23 04:3…	10.9.0.5	10.9.0.11	UDP	118	9090 → 9090 Len=76
5	2025-11-23 04:3…	10.9.0.11	10.9.0.5	UDP	94	9090 → 9090 Len=52
6	2025-11-23 04:3…	02:42:0a:09:00:0b	Broadcast	ARP	42	Who has 10.9.0.1? Tell 10.9.0.11
7	2025-11-23 04:3…	02:42:91:70:ab:ec	02:42:0a:09:00:0b	ARP	42	10.9.0.1 is at 02:42:91:70:ab:ec
8	2025-11-23 04:3…	192.168.60.5	192.168.0.1	DNS	86	Standard query 0x2d28 PTR 99.53.168.192.in-addr.arpa
9	2025-11-23 04:3…	192.168.60.5	192.168.0.1	DNS	86	Standard query 0x2d28 PTR 99.53.168.192.in-addr.arpa
10	2025-11-23 04:3…	02:42:0a:09:00:0b	02:42:0a:09:00:05	ARP	42	Who has 10.9.0.5? Tell 10.9.0.11
11	2025-11-23 04:3…	02:42:0a:09:00:05	02:42:0a:09:00:0b	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05
12	2025-11-23 04:3…	02:42:0a:09:00:05	02:42:0a:09:00:0b	ARP	42	Who has 10.9.0.11? Tell 10.9.0.5
13	2025-11-23 04:3…	02:42:0a:09:00:0b	02:42:0a:09:00:05	ARP	42	10.9.0.11 is at 02:42:0a:09:00:0b
14	2025-11-23 04:3…	10.9.0.11	10.9.0.5	UDP	106	9090 → 9090 Len=64
15	2025-11-23 04:3…	10.9.0.5	10.9.0.11	UDP	94	9090 → 9090 Len=52
16	2025-11-23 04:3…	10.9.0.5	10.9.0.11	UDP	97	9090 → 9090 Len=55
17	2025-11-23 04:3…	10.9.0.11	10.9.0.5	UDP	109	9090 → 9090 Len=67
18	2025-11-23 04:3…	10.9.0.11	10.9.0.5	UDP	94	9090 → 9090 Len=52
19	2025-11-23 04:3…	10.9.0.5	10.9.0.11	UDP	94	9090 → 9090 Len=52
20	2025-11-23 04:3…	10.9.0.11	10.9.0.5	UDP	112	9090 → 9090 Len=70

From the image above, we can see that only UDP packets were sent between the VPN client and server.



No.	Time	Source	Destination	Protocol	Length	Info
1	2025-11-23 04:3…	192.168.53.99	192.168.60.5	TCP	74	41080 → 23 [SYN] Seq=3133872334 Win=64240 Len=0 MSS=1460 SACK…
2	2025-11-23 04:3…	192.168.60.5	192.168.53.99	TCP	74	23 → 41080 [SYN, ACK] Seq=1027406484 Ack=3133872335 Win=65160…
3	2025-11-23 04:3…	192.168.53.99	192.168.60.5	TCP	66	41080 → 23 [ACK] Seq=3133872335 Ack=1027406485 Win=64256 Len=…
4	2025-11-23 04:3…	192.168.53.99	192.168.60.5	TELNET	90	Telnet Data ...
5	2025-11-23 04:3…	192.168.60.5	192.168.53.99	TCP	66	23 → 41080 [ACK] Seq=1027406485 Ack=3133872359 Win=65152 Len=…
6	2025-11-23 04:3…	192.168.60.5	192.168.0.1	DNS	86	Standard query 0x2d28 PTR 99.53.168.192.in-addr.arpa
7	2025-11-23 04:3…	192.168.60.5	192.168.0.1	DNS	86	Standard query 0x2d28 PTR 99.53.168.192.in-addr.arpa
8	2025-11-23 04:3…	02:42:c0:a8:3c:05	02:42:c0:a8:3c:0b	ARP	42	Who has 192.168.60.11? Tell 192.168.60.5
9	2025-11-23 04:3…	02:42:c0:a8:3c:0b	02:42:c0:a8:3c:05	ARP	42	Who has 192.168.60.5? Tell 192.168.60.11
10	2025-11-23 04:3…	02:42:c0:a8:3c:0b	02:42:c0:a8:3c:05	ARP	42	192.168.60.11 is at 02:42:c0:a8:3c:0b
11	2025-11-23 04:3…	02:42:c0:a8:3c:05	02:42:c0:a8:3c:0b	ARP	42	192.168.60.5 is at 02:42:c0:a8:3c:05
12	2025-11-23 04:3…	192.168.60.5	192.168.53.99	TELNET	78	Telnet Data ...
13	2025-11-23 04:3…	192.168.53.99	192.168.60.5	TCP	66	41080 → 23 [ACK] Seq=3133872359 Ack=1027406497 Win=64256 Len=…
14	2025-11-23 04:3…	192.168.60.5	192.168.53.99	TELNET	81	Telnet Data ...
15	2025-11-23 04:3…	192.168.53.99	192.168.60.5	TELNET	69	Telnet Data ...
16	2025-11-23 04:3…	192.168.60.5	192.168.53.99	TCP	66	23 → 41080 [ACK] Seq=1027406512 Ack=3133872362 Win=65152 Len=…
17	2025-11-23 04:3…	192.168.53.99	192.168.60.5	TCP	66	41080 → 23 [ACK] Seq=3133872362 Ack=1027406512 Win=64256 Len=…
18	2025-11-23 04:3…	192.168.60.5	192.168.53.99	TELNET	84	Telnet Data ...
19	2025-11-23 04:3…	192.168.53.99	192.168.60.5	TELNET	75	Telnet Data ...
20	2025-11-23 04:3…	192.168.60.5	192.168.53.99	TCP	66	23 → 41080 [ACK] Seq=1027406520 Ack=3133872371 Win=65152 Len=…

From the image above, we can see the actual packets that were being encapsulated by the UDP packet to and from Host V.

For both ping and telnet, our packets will begin from Machine U, and if the target is in 192.168.60.0/24 network, it will be sent through the tun0 interface. It will be read and encapsulated as a UDP datagram before being sent to the VPN server. After receiving the UDP datagram, the server extracts the inner IP packet, then writing to the server's tun0 interface and IP forwarding will route the packet to machine V inside the 192.168.60.0/24 network. Machine V will then send replies back to the VPN server, and the server will send the replies back over the UDP tunnel back to Machine U.

Task 6: Tunnel-Breaking Experiment

After breaking the VPN tunnel by stopping tun_server.py, we are not able to see what we typed as the TCP packets are unable to reach Host V since the tunnel is broken. However, the TCP connection is not broken immediately and the TCP packets will be retransmitted to Host V for a period of time. Hence, the TCP connection is "paused".
After running tun_server.py again, the tunnel is re-established and the telnet connection is restored. Then, we can see everything that we have typed before when the VPN tunnel was broken. This is because the TCP packets that were continuously retransmitted (containing our keystrokes) are finally able to reach the destination through the re-established tunnel.