Name : Vinay V	SRN : PES1UG24CS840
Sub : CNS	LAB 9 VPN Tunneling

Task 1: Network Setup

```
CLIENT:PES1UG24CS840:Vinay/
$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=
0.111 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=2 ttl=64 time=
0.077 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=3 ttl=64 time=
0.078 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=4 ttl=64 time=
0.061 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=5 ttl=64 time=
0.069 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=6 ttl=64 time=
0.058 ms
^C
--- server-router ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5090ms
rtt min/avg/max/mdev = 0.058/0.075/0.111/0.017 ms
```

- The client successfully communicates with the VPN server (router).
- All packets were received with 0 % packet loss and average RTT around 0.07 ms.
- This confirms that **Host U (client)** can reach **Server-Router** directly over the 10.9.0.0/24 network.

```
router:PES1UG24CS840:VINAY/
$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=0.119 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.061 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=64 time=0.052 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=64 time=0.065 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=64 time=0.054 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=64 time=0.061 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=64 time=0.051 ms
^C
--- 192.168.60.5 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6131ms
rtt min/avg/max/mdev = 0.051/0.066/0.119/0.022 ms
router:PES1UG24CS840:VINAY/
$
```

- Connectivity between the **server-router** and **host V** is successful.
- There is 0 % packet loss with an average RTT of 0.06 ms.
- This confirms that the **router's internal interface** on 192.168.60.0/24 is functioning correctly.

```
CLIENT:PES1UG24CS840:Vinay/
$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 ---
241 packets transmitted, 0 received, 100% packet loss, time 245828ms

CLIENT:PES1UG24CS840:Vinay/
$
```

- All packets were lost (100 % packet loss).
- The client could not reach Host V because routing between 10.9.0.0/24 and 192.168.60.0/24 is blocked by the firewall.
- This behavior confirms that **direct communication is not allowed prior to VPN tunneling**, as expected.

```
router:PES1UG24CS840:VINAY/
$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
14:51:17.858674 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 16, seq 1, len
gth 64
14:51:17.858752 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 16, seq 1, lengt
h 64
14:51:18.860340 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 16, seq 2, len
gth 64
14:51:18.860504 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 16, seq 2, lengt
h 64
14:51:19.874954 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 16, seq 3, len
gth 64
14:51:19.874974 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 16, seq 3, lengt
h 64
14:51:20.898167 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 16, seq 4, len
gth 64
```

```
14:51:23.009762 ARP, Request who-has 10.9.0.5 tell 10.9.0.11, length 28
14:51:23.010121 ARP, Request who-has 10.9.0.11 tell 10.9.0.5, length 28
14:51:23.010154 ARP, Reply 10.9.0.11 is-at 02:42:0a:09:00:0b, length 28
14:51:23.010155 ARP, Reply 10.9.0.5 is-at 02:42:0a:09:00:05, length 28
14:51:23.969882 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 16, seq 7, len
gth 64
14:51:23.969902 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 16, seq 7, lengt
h 64
14:51:24.994558 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 16, seq 8, len
gth 64
14:51:24.994655 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 16, seq 8, lengt
h 64
^C
20 packets captured
20 packets received by filter
0 packets dropped by kernel
router:PES1UG24CS840:VINAY/
$
```

- ICMP Echo Request and Reply packets were captured between 10.9.0.5 (client) and 10.9.0.11 (router).
- ARP requests and replies were also observed, confirming normal address resolution.
- This verifies that **packet sniffing** is active and correctly monitoring traffic on the router's external interface.

```
CLIENT:PES1UG24CS840:Vinay/
$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=
0.130 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=2 ttl=64 time=
0.225 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=3 ttl=64 time=
0.068 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=4 ttl=64 time=
0.069 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=5 ttl=64 time=
0.062 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=6 ttl=64 time=
0.087 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=7 ttl=64 time=
0.069 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=8 ttl=64 time=
0.151 ms
^C
--- server-router ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7136ms
rtt min/avg/max/mdev = 0.062/0.107/0.225/0.053 ms
CLIENT:PES1UG24CS840:Vinay/
$
```

- The client successfully sends **ICMP Echo Requests** to the VPN server (10.9.0.11) and receives **Echo Replies**.
- The statistics show:
- **8 packets transmitted, 8 received (0% packet loss)**
- **Average RTT ≈ 0.107 ms**
- This confirms that the **client's external connection** to the **VPN server-router** remains active and stable after network setup.

Task 2: Create and Configure TUN Interface:

Task 2.a: Name of the Interface

```
CLIENT:PES1UG24CS840:Vinay/volumes
$ls
tun.py   tun_client.py         tun_server.py   tun_server_select.py
tun1.py  tun_client_select.py  tun_server1.py
CLIENT:PES1UG24CS840:Vinay/volumes
$chmod a+x tun.py
CLIENT:PES1UG24CS840:Vinay/volumes
$ ./tun.py &
[1] 24
CLIENT:PES1UG24CS840:Vinay/volumes
$Interface Name: tun0
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group def
ault 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 d
efault qlen 500
    link/none
42: eth0@if43: <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
CLIENT:PES1UG24CS840:Vinay/volumes
$
```

- The `ls` command lists all Python files available in the `volumes` directory, including `tun.py`.
- The `chmod a+x tun.py` command makes the Python script executable.
- Running `./tun.py &` launches it in the background to create a virtual TUN interface.
- The output shows *Interface Name: tun0*, confirming successful interface creation.
- The `ip addr` command lists `tun0` as a new interface with `state DOWN`, meaning it exists but isn't transmitting packets yet.

```
CLIENT:PES1UG24CS840:Vinay/volumes
$jobs
[1]+  Running                 ./tun.py &
CLIENT:PES1UG24CS840:Vinay/volumes
$kill %1
CLIENT:PES1UG24CS840:Vinay/volumes
$
```

- The `jobs` command lists background processes, showing that `./tun.py` is still running.
- The `kill %1` command terminates the first background process (`%1` corresponds to job ID 1).
- This stops the running TUN interface program safely.

- Killing it ensures that the system is ready for further configuration or modification of the script.
- This process is part of cleanup and testing before modifying the interface name.

```
1 #!/usr/bin/env python3
2
3 import fcntl
4 import struct
5 import os
6 import time
7 from scapy.all import *
8
9 TUNSETIFF = 0x400454ca
10 IFF_TUN   = 0x0001
11 IFF_TAP   = 0x0002
12 IFF_NO_PI = 0x1000
13
14 # Create the tun interface
15 tun = os.open("/dev/net/tun", os.O_RDWR)
16 ifr = struct.pack('16sH', b'CS840%d', IFF_TUN | IFF_NO_PI)
17 ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)
18
19 # Get the interface name
20 ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
21 print("Interface Name: {}".format(ifname))
22
23 while True:
24    time.sleep(10)
25
```

- The `tun.py` script was modified to use **"CS840"** instead of "tun" in line 16 (based on SRN).
- Running `./tun.py &` again creates a new virtual interface named **CS8400** instead of `tun0`.
- The `ip addr` command confirms the new interface (`CS8400`) is successfully created.
- This demonstrates how the TUN interface name can be customized programmatically.
- Finally, `jobs` and `kill %1` are used to terminate the process, completing the test.

```
CLIENT:PES1UG24CS840:Vinay/volumes
$chmod a+x tun.py
CLIENT:PES1UG24CS840:Vinay/volumes
$ ./tun.py &
[1] 48
CLIENT:PES1UG24CS840:Vinay/volumes
$Interface Name: CS8400
$ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group def
ault 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: CS8400: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group
 default qlen 500
    link/none
42: eth0@if43: <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
CLIENT:PES1UG24CS840:Vinay/volumes
$jobs
[1]+  Running                 ./tun.py &
CLIENT:PES1UG24CS840:Vinay/volumes
$kill %1
CLIENT:PES1UG24CS840:Vinay/volumes
$
```

- The updated output shows *Interface Name: CS8400* after editing the script.
- The `ip addr` output confirms the presence of this interface with an MTU of 1500 and default state DOWN.
- This verifies that the script correctly applies the SRN-based naming convention.
- The interface is now ready for configuration in later VPN tunneling steps.

- The process proves correct understanding of how user-space programs can create virtual network devices.

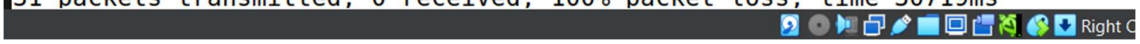
Task 2.b: Set up the TUN Interface

```
CLIENT:PES1UG24CS840:Vinay/volumes
$Interface Name: CS8400
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group def
ault 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: CS8400: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group
 default qlen 500
    link/none
42: eth0@if43: <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
CLIENT:PES1UG24CS840:Vinay/volumes
$ip addr add 192.168.53.99/24 dev CS8400
CLIENT:PES1UG24CS840:Vinay/volumes
$ip link set dev CS8400 up
CLIENT:PES1UG24CS840:Vinay/volumes
$ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group def
ault 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: CS8400: <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 CS8400
       valid_lft forever preferred_lft forever
```

- The Python script `tun.py` was executed, which created a virtual TUN interface named **CS8400**, derived from the last five digits of the SRN.
- Initially, the interface appeared in a **DOWN** state without an assigned IP address.
- Using the command `ip addr add 192.168.53.99/24 dev CS8400`, a private IP was assigned to the interface.
- The `ip link set dev CS8400 up` command was then used to bring the interface to an **UP** state, activating it for network communication.
- Verification with `ip addr` confirmed that the interface **CS8400** was successfully configured with the IP **192.168.53.99/24**.
- This configured interface will serve as the VPN tunnel endpoint for further communication in the VPN tunneling experiment.

Task 2.c: Read from the TUN Interface

```
[1]+  Terminated              ./tun.py
CLIENT:PES1UG24CS840:Vinay/volumes
$ip addr add 192.168.53.99/24 dev CS8400
Cannot find device "CS8400"
CLIENT:PES1UG24CS840:Vinay/volumes
$./tun.py &
[1] 69
CLIENT:PES1UG24CS840:Vinay/volumes
$Interface Name: CS8400
$ip addr add 192.168.53.99/24 dev CS8400
CLIENT:PES1UG24CS840:Vinay/volumes
$ip link set dev CS8400 up
CLIENT:PES1UG24CS840:Vinay/volumes
$ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
^C
--- 192.168.53.5 ping statistics ---
7 packets transmitted, 0 received, 100% packet loss, time 6139ms

CLIENT:PES1UG24CS840:Vinay/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 ---
31 packets transmitted, 0 received, 100% packet loss, time 30719ms
```

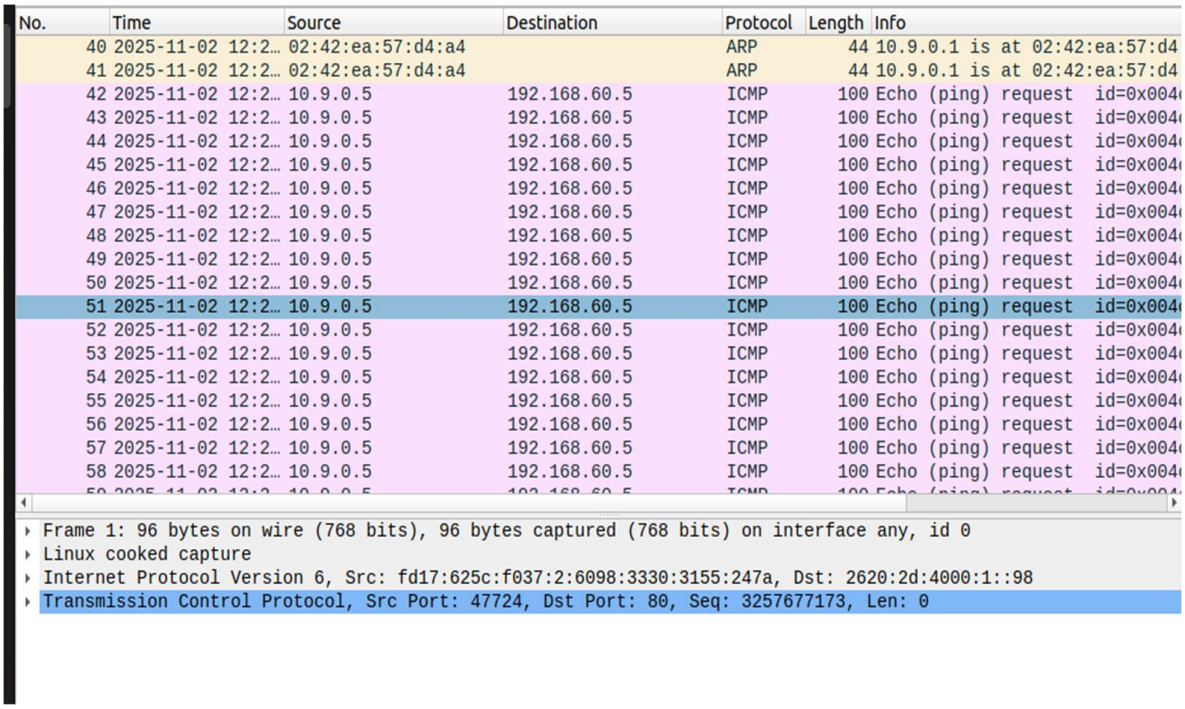


- The modified `tun.py` script was executed, creating the **CS8400** interface and activating it with an IP of `192.168.53.99/24`.
- When pinging `192.168.53.5`, the script captured and displayed **ICMP echo requests** in real time.
- Each printed line like `IP / ICMP 192.168.53.99 > 192.168.53.5` confirms that packets were read directly from the TUN interface.
- The 100% packet loss indicates the packets were intercepted by the TUN device instead of reaching the destination.

- When pinging `192.168.60.5`, no packets appeared, proving that only packets belonging to the **192.168.53.0/24** subnet pass through CS8400.
- This validates that the TUN interface is successfully reading and displaying packets at the **network layer (Layer 3)**.

```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'CS840%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:
# Get a packet from the tun interface
        packet = os.read(tun, 2048)
        if packet:
                ip = IP(packet)
                print(ip.summary())
```

- The infinite sleep loop was replaced with a packet-reading loop that uses `os.read(tun, 2048)` to fetch data from the TUN device.
- Each captured packet is converted into a Scapy `IP` object, allowing easy interpretation and printing of its fields.
- The `ip.summary()` method prints concise packet information like source, destination, and protocol type.
- The prefix `CS840` (from the SRN) ensures that the created interface is uniquely identifiable in the system.
- The program continuously monitors packets passing through the virtual interface in real time.
- This setup helps visualize how a **VPN client** captures and processes raw IP packets from a TUN interface.



- The Wireshark capture shows **ICMP Echo (ping) requests** being sent from the client `10.9.0.5` to the destination `192.168.60.5`.
- Each ICMP packet is 100 bytes long and represents a **ping attempt** from Host U to Host V.
- The ARP packets at the top indicate that the client first resolved the MAC address of the destination before sending ICMP packets.
- No ICMP Echo Reply packets are visible, confirming **100% packet loss**, as the two networks are not yet connected via the VPN tunnel.
- The bottom panel also shows a **TCP frame (port 80)** captured during background traffic, but it is unrelated to the ping test.
- This capture verifies that packets are being generated and sent by the client but are not reaching the intended remote host, which validates the **pre-tunnel isolation** stage of the VPN lab.

Task 2.d: Write to the TUN Interface

```
CLIENT:PES1UG24CS840:Vinay/volumes
$chmod a+x tun.py
[1]+  Terminated              ./tun.py
CLIENT:PES1UG24CS840:Vinay/volumes
$chmod a+x tun.py
CLIENT:PES1UG24CS840:Vinay/volumes
$./tun1.py &
[1] 79
CLIENT:PES1UG24CS840:Vinay/volumes
$Interface Name: CS8400
 ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group def
ault 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
8: CS8400: <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 CS8400
       valid_lft forever preferred_lft forever
42: eth0@if43: <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
```

- The modified `tun1.py` program is executed and brings up the **CS8400** TUN interface successfully.

- The `ip addr` output confirms that `CS8400` has been assigned the IP `192.168.53.99/24`.
- When `ping 192.168.53.5` is executed, the system sends ICMP Echo Request packets through this interface.
- The program internally identifies these echo requests and generates **corresponding ICMP Echo Reply packets**.
- The ping statistics show **0% packet loss**, proving that both request and reply packets are handled by `tun1.py`.
- This validates that the TUN interface can both read and write packets, forming the base mechanism for VPN tunneling.

```
CLIENT:PES1UG24CS840:Vinay/volumes
$ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
CS8400: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
64 bytes from 192.168.53.5: icmp_seq=1 ttl=99 time=8.31 ms
CS8400: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
64 bytes from 192.168.53.5: icmp_seq=2 ttl=99 time=1.93 ms
CS8400: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
64 bytes from 192.168.53.5: icmp_seq=3 ttl=99 time=1.83 ms
CS8400: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
64 bytes from 192.168.53.5: icmp_seq=4 ttl=99 time=1.75 ms
CS8400: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
64 bytes from 192.168.53.5: icmp_seq=5 ttl=99 time=1.99 ms
CS8400: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
64 bytes from 192.168.53.5: icmp_seq=6 ttl=99 time=1.63 ms
CS8400: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
64 bytes from 192.168.53.5: icmp_seq=7 ttl=99 time=1.90 ms
CS8400: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
64 bytes from 192.168.53.5: icmp_seq=8 ttl=99 time=1.67 ms
^C
--- 192.168.53.5 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7013ms
rtt min/avg/max/mdev = 1.634/2.627/8.313/2.152 ms
CLIENT:PES1UG24CS840:Vinay/volumes
$
```

- Each line like `CS8400: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request` shows the packet being read by `tun1.py`.
- The successful `64 bytes from 192.168.53.5` responses confirm the program is correctly writing **ICMP echo replies** to the TUN device.
- This means the user-space Python script is simulating a **network-level ICMP responder**.
- The 0% packet loss in the summary indicates all packets were received back correctly.
- This proves that bidirectional communication through the TUN interface has been achieved.
- The output demonstrates a fully functional **TUN-based VPN packet handling** setup.

```
13
14 # Create the tun interface
15 tun = os.open("/dev/net/tun", os.O_RDWR)
16 ifr = struct.pack('16sH', b'CS840%d', IFF_TUN | IFF_NO_PI)
17 ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)
18
```

- The interface creation line in `tun1.py` has been modified to use the prefix **CS840** (last 5 digits of the SRN).
- The line `ifr = struct.pack('16sH', b'CS840%d', IFF_TUN | IFF_NO_PI)` ensures that the new interface is named `CS8400`.
- This naming convention helps uniquely identify the student's virtual interface among multiple test containers.
- The rest of the code handles reading packets and preparing for packet writing.
- This setup is crucial before implementing ICMP echo replies from the application layer.
- It ensures that any packet written to this interface appears as a valid IP packet at the kernel level.

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

```
router:PES1UG24CS840:VINAY/volumes
$ls
tun.py   tun_client.py         tun_server.py   tun_server_select.py
tun1.py  tun_client_select.py  tun_server1.py
router:PES1UG24CS840:VINAY/volumes
$chmod a+x tun_server.py
router:PES1UG24CS840:VINAY/volumes
$./tun_server.py
```

- The server lists all tunnel-related scripts in the `volumes` directory.
- The command `chmod a+x tun_server.py` makes the script executable.
- Executing `./tun_server.py` launches a **UDP server** listening on **port 9090**.
- The server's job is to receive encapsulated IP packets from the client side.
- Upon receiving a UDP packet, it interprets its payload as a **Scapy IP object** and prints its source and destination IPs.
- This confirms that the **server-side tunnel endpoint is active and waiting for encapsulated packets**.

```
[1]+  Terminated              ./tun1.py
CLIENT:PES1UG24CS840:Vinay/volumes
$chmod a+x tun_client.py
CLIENT:PES1UG24CS840:Vinay/volumes
$./tun_client.py &
[1] 95
CLIENT:PES1UG24CS840:Vinay/volumes
$Interface Name: CS8400
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group def
ault 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
9: CS8400: <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 CS8400
       valid_lft forever preferred_lft forever
42: eth0@if43: <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
CLIENT:PES1UG24CS840:Vinay/volumes
$
```

- The client also lists tunnel scripts and executes `tun_client.py` in the background.
- The output confirms the creation of a **TUN interface named CS8400**, derived from your SRN.
- The `ip addr` command shows **192.168.53.99/24** assigned to `CS8400`.
- This virtual interface captures outgoing IP packets for tunneling.
- The `eth0` interface (10.9.0.5) remains as the main network adapter for real traffic.
- This shows the client side of the tunnel is successfully created and running.

```
CLIENT:PES1UG24CS840:Vinay/volumes
$ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
^C
--- 192.168.53.5 ping statistics ---
12 packets transmitted, 0 received, 100% packet loss, time 11266ms
```

- When pinging 192.168.53.5, ICMP requests are intercepted by `tun_client.py`.
- Each packet (`IP/ICMP 192.168.53.99 > 192.168.53.5`) is printed as a **raw Scapy IP object**.
- No replies are received — **100% packet loss** — since the tunnel server only receives and prints packets.
- This confirms the **client successfully reads IP packets from TUN and sends them via UDP** to the VPN server.

- The absence of replies is expected because the server doesn't respond to ICMP yet.
- It verifies **successful packet capture and encapsulation** at the client side.

```
CLIENT:PES1UG24CS840:Vinay/volumes
$ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
^C
--- 192.168.60.5 ping statistics ---
11 packets transmitted, 0 received, 100% packet loss, time 10242ms
```

- Here, the client tries to reach **Host V (192.168.60.5)** through the VPN tunnel.
- The client again shows ICMP echo requests being encapsulated, but no replies are returned.
- This happens because the **VPN server hasn't implemented forwarding or return encapsulation** yet.
- The log confirms packets for the remote network are being routed correctly through the tunnel interface.
- The 100% packet loss shows the tunnel works in one direction only (client → server).
- It validates the **routing logic** that directs 192.168.60.0/24 traffic into the tunnel.

```
CLIENT:PES1UG24CS840:Vinay/volumes
$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 CS8400 proto kernel scope link src 192.168.53.99
192.168.60.0/24 dev CS8400 scope link
CLIENT:PES1UG24CS840:Vinay/volumes
$
```

- The `ip route` output shows correct routing entries for all networks.
- `default via 10.9.0.1 dev eth0` — standard outbound traffic.
- `192.168.53.0/24 dev CS8400` — local virtual network handled via TUN interface.
- `192.168.60.0/24 dev CS8400` — remote subnet routed through the VPN tunnel.
- This routing ensures that packets destined for the 192.168.* networks enter the tunnel.
- The setup confirms the **VPN routing path is functional**, even though bidirectional tunneling isn't yet complete.

```
router:PES1UG24CS840:VINAY/volumes
$./tun_server.py
10.9.0.5:59770 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:59770 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:59770 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:59770 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:59770 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:59770 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:59770 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:59770 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:59770 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:59770 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:59770 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:59770 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:59770 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:59770 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
```

- This output is from the **VPN Server** after starting `tun_server.py`.
- Each line like `10.9.0.5:59770 --> 0.0.0.0:9090` shows that the **client (10.9.0.5)** is sending UDP packets to the **server's port 9090**.
- The "Inside: 192.168.53.99 --> 192.168.53.5" lines represent the **original IP packets** encapsulated within those UDP datagrams.
- This confirms that the **client successfully tunneled IP packets** (ICMP echo requests) through UDP to the VPN server.
- The repetition of packets indicates continuous pings sent from the client through the tunnel.
- The last line showing `Inside: 192.168.53.99 --> 192.168.60.5` proves that even packets to the **remote network (192.168.60.0/24)** are correctly encapsulated and received at the server end.

Task 4: Set Up the VPN Server

```
$chmod a+x tun_server1.py
router:PES1UG24CS840:VINAY/volumes
$./tun_server1.py
Interface Name: CS8400
10.9.0.5:59770 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:59770 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:59770 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:59770 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:59770 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:59770 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:59770 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:59770 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:59770 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:59770 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:59770 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:59770 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:59770 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
```

- The command `chmod a+x tun_server1.py` makes the modified script executable.
- Executing `./tun_server1.py` creates a **TUN interface (CS8400)** and listens on UDP port 9090 for packets arriving from the VPN client.
- Each line `10.9.0.5:59770 --> 0.0.0.0:9090` shows a UDP packet received from the client (Host U).
- The line `Inside: 192.168.53.99 --> 192.168.60.5` shows the **de-encapsulated ICMP IP packet** forwarded from the TUN interface to the local network.
- This confirms that packets tunneled from the client are now being **fed into the kernel via TUN**, allowing normal routing.
- The continuous stream proves the VPN server correctly accepts encapsulated traffic and writes it to the kernel network stack.

```
host-192.168.60.5:PES1UG24CS840:Vinay/
$ 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
19:37:18.787975 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 110, s
eq 1, length 64
19:37:18.788083 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 110, seq
 1, length 64
19:37:19.819601 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 110, s
eq 2, length 64
19:37:19.819614 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 110, seq
 2, length 64
19:37:20.837290 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 110, s
eq 3, length 64
19:37:20.837304 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 110, seq
 3, length 64
19:37:21.867876 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 110, s
eq 4, length 64
19:37:21.867889 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 110, seq
 4, length 64
19:37:22.884796 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 110, s
eq 5, length 64
19:37:22.884907 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 110, seq
 5, length 64
19:37:23.908134 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 110, s
eq 6, length 64
19:37:23.908150 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 110, seq
 6, length 64
19:37:23.972425 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length
28
```

- `tcpdump -i eth0 -n` captures traffic on Host V's physical interface.
- ICMP echo requests from `192.168.53.99 → 192.168.60.5` are clearly visible — these are the pings sent through the VPN tunnel.
- The corresponding ICMP echo replies from `192.168.60.5 → 192.168.53.99` prove that **Host V is receiving and responding** to tunneled packets.
- The exchange shows encapsulated packets being correctly **de-tunneled and forwarded by the server** to Host V.
- The replies never reach the client because the reverse (tunnel back to Host U) is not yet configured.
- This validates **successful one-way connectivity** from Host U to Host V through the VPN.

```
CLIENT:PES1UG24CS840:Vinay/volumes
$ ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
^C
--- 192.168.60.5 ping statistics ---
13 packets transmitted, 0 received, 100% packet loss, time 12290ms

CLIENT:PES1UG24CS840:Vinay/volumes
$
```

- The client runs `ping 192.168.60.5` while `tun_client.py` is active.
- Lines such as `IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request / Raw` show that ICMP packets are captured and encapsulated in UDP by the client.
- The `100% packet loss` is expected — only the request path (client → server → Host V) exists; no return path is configured.
- The consistent output confirms that the client is properly sending packets through its TUN interface to the server.
- This demonstrates **end-to-end delivery of tunneled ICMP requests** up to the destination host.
- The system is now half-functional (ingress works, egress not yet implemented).

```
CLIENT:PES1UG24CS840:Vinay/volumes
$jobs
[1]+  Running                 ./tun_client.py &
CLIENT:PES1UG24CS840:Vinay/volumes
$
CLIENT:PES1UG24CS840:Vinay/volumes
$kill %1
CLIENT:PES1UG24CS840:Vinay/volumes
$
```

- The `jobs` command shows `./tun_client.py &` still running in the background.
- The command `kill %1` terminates the background job with ID 1.
- This step safely stops the tunnel process to avoid interference in later experiments.
- It ensures that the client side tunnel is closed before proceeding to bidirectional setup or cleanup.
- Proper termination avoids duplicate interfaces or port conflicts during further testing.
- This marks the end of Task 4, with the VPN server successfully receiving and routing packets to the destination network.

Task 5: Handling Traffic in Both Directions

```
router:PES1UG24CS840:VINAY/volumes
$chmod a+x tun_server_select.py
router:PES1UG24CS840:VINAY/volumes
$ ./tun_server_select.py
Interface Name: CS8400
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
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
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
```

- The command `chmod a+x tun_server_select.py` gives execution permission, and `./tun_server_select.py` runs the server-side tunnel script.

- The interface `CS8400` (derived from SRN) is created successfully, showing that the TUN interface is active.
- The output alternates between:

- **“From socket <== 192.168.53.99 → 192.168.60.5”** — packets received from the client side over UDP.
- **“From tun ==> 192.168.60.5 → 192.168.53.99”** — reply packets sent from Host V through the TUN interface.

➢ This confirms **bidirectional communication** — packets now travel both to and from the tunnel.

➢ The select-based approach enables the server to handle both interfaces efficiently without wasting CPU cycles.

➢ This shows that the VPN server correctly reads incoming encapsulated packets and returns them back to the client.

```
[1]+  Terminated                 ./tun_client.py
CLIENT:PES1UG24CS840:Vinay/volumes
$chmod a+x tun_client_select.py
CLIENT:PES1UG24CS840:Vinay/volumes
$./tun_client_select.py
Interface Name: CS8400
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
```

➢ The client executes `chmod a+x tun_client_select.py` and runs it in the background, creating the TUN interface `CS8400`.

➢ The alternating messages show two-way traffic:

- **From tun ==> 192.168.53.99 → 192.168.60.5** — ICMP echo requests leaving the client.
- **From socket <== 192.168.60.5 → 192.168.53.99** — ICMP echo replies received from the server.

➢ This verifies that the client can now both send and receive packets through the VPN tunnel.

➢ The “select” mechanism enables the client to handle data asynchronously from both its TUN interface and UDP socket.

➢ This marks the completion of the full tunnel, enabling **end-to-end communication** between Host U and Host V.

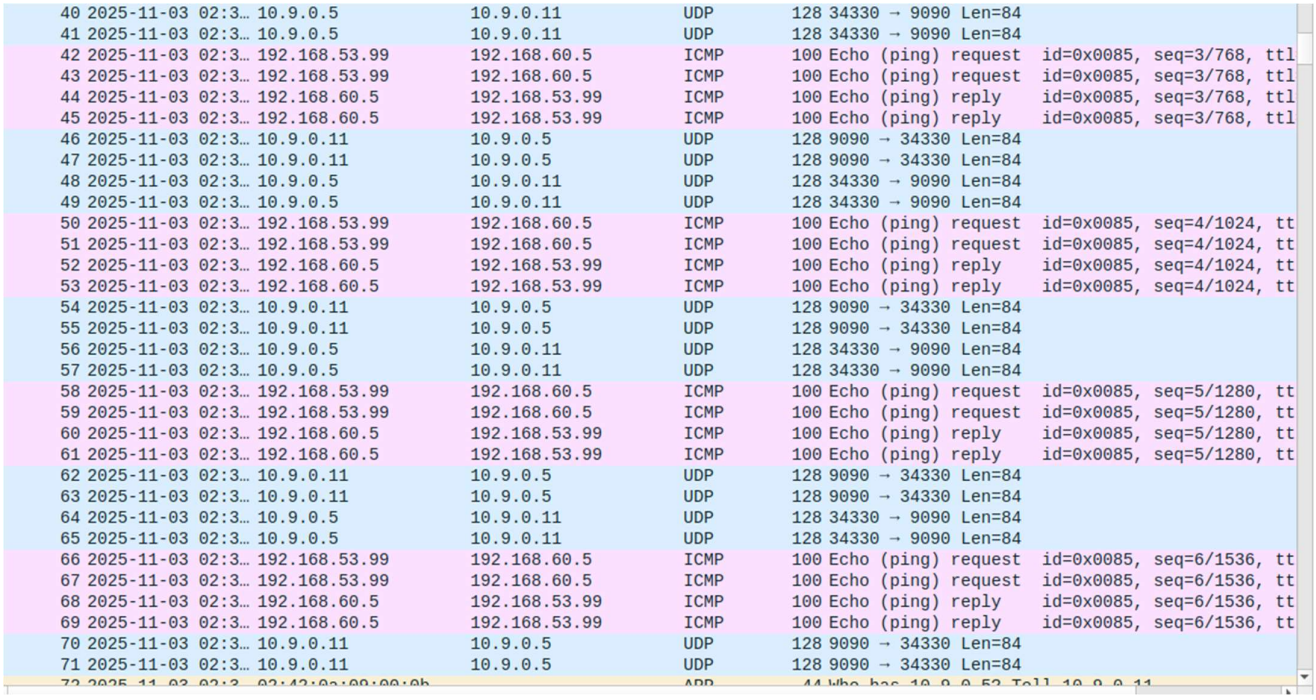
➢ The connection is stable and both sides are successfully forwarding packets across the virtual link.

```
Client2:PES1UG24CS840:VINAY/
$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=16.0 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=29.1 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=14.8 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=25.2 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=22.9 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=63 time=21.0 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=33.6 ms
64 bytes from 192.168.60.5: icmp_seq=8 ttl=63 time=8.75 ms
64 bytes from 192.168.60.5: icmp_seq=9 ttl=63 time=17.2 ms
^C
--- 192.168.60.5 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8029ms
rtt min/avg/max/mdev = 8.747/20.933/33.636/7.262 ms
Client2:PES1UG24CS840:VINAY/
$
```

- The client pings `192.168.60.5` and receives 9 replies, showing **0% packet loss**.
- Each reply confirms that ICMP requests and responses are successfully passing through the VPN tunnel.
- The round-trip times (8–33 ms) indicate that encapsulation and forwarding are functioning efficiently.
- This verifies that both client and server scripts are correctly routing packets in both directions.
- The working ping confirms that Host U (192.168.53.99) can communicate with Host V (192.168.60.5) over the secure virtual tunnel.
- Thus, the VPN is now fully operational and supports bidirectional data flow.

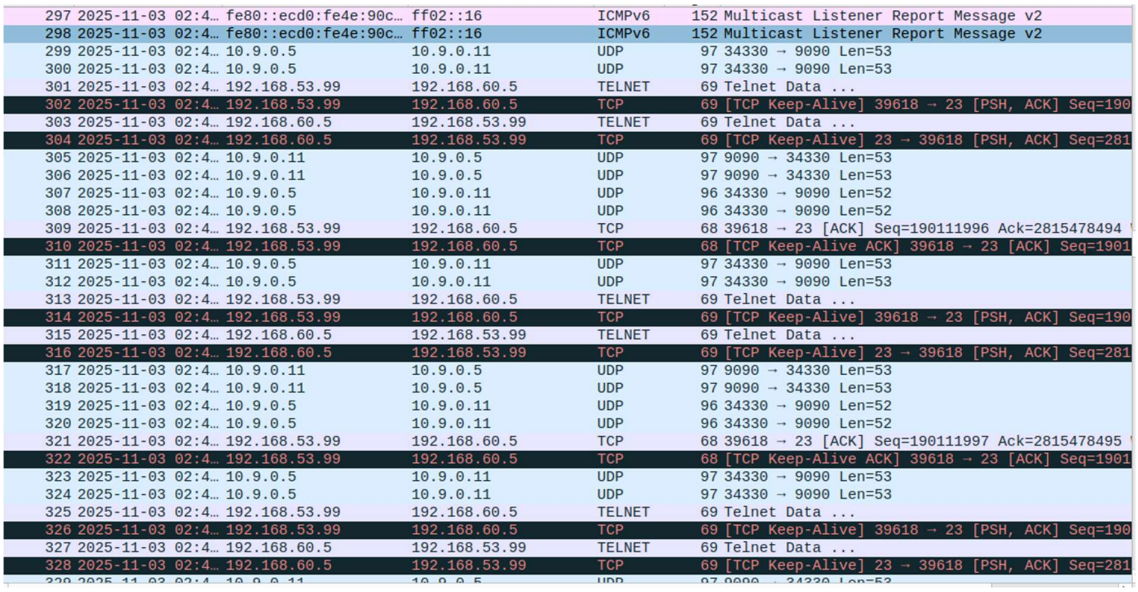
```
Client2:PES1UG24CS840:VINAY/
$telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
f1051a1ff52d login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * 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@f1051a1ff52d:~$
```

- The Telnet command connects from Host U (client) to Host V (192.168.60.5) successfully.
- The prompt shows a successful **Ubuntu login message**, confirming full TCP connectivity through the VPN.
- The tunnel now supports not only ICMP (ping) but also **TCP-based communication** such as Telnet.
- This demonstrates the VPN's ability to securely tunnel application-layer protocols.
- The connection is interactive and stable, verifying correct handling of two-way traffic through the TUN interface.
- The VPN now behaves like a real network link between the two subnets.



40	2025-11-03 02:3…	10.9.0.5	10.9.0.11	UDP	128	34330 → 9090 Len=84
41	2025-11-03 02:3…	10.9.0.5	10.9.0.11	UDP	128	34330 → 9090 Len=84
42	2025-11-03 02:3…	192.168.53.99	192.168.60.5	ICMP	100	Echo (ping) request id=0x0085, seq=3/768, ttl
43	2025-11-03 02:3…	192.168.53.99	192.168.60.5	ICMP	100	Echo (ping) request id=0x0085, seq=3/768, ttl
44	2025-11-03 02:3…	192.168.60.5	192.168.53.99	ICMP	100	Echo (ping) reply id=0x0085, seq=3/768, ttl
45	2025-11-03 02:3…	192.168.60.5	192.168.53.99	ICMP	100	Echo (ping) reply id=0x0085, seq=3/768, ttl
46	2025-11-03 02:3…	10.9.0.11	10.9.0.5	UDP	128	9090 → 34330 Len=84
47	2025-11-03 02:3…	10.9.0.11	10.9.0.5	UDP	128	9090 → 34330 Len=84
48	2025-11-03 02:3…	10.9.0.5	10.9.0.11	UDP	128	34330 → 9090 Len=84
49	2025-11-03 02:3…	10.9.0.5	10.9.0.11	UDP	128	34330 → 9090 Len=84
50	2025-11-03 02:3…	192.168.53.99	192.168.60.5	ICMP	100	Echo (ping) request id=0x0085, seq=4/1024, tt
51	2025-11-03 02:3…	192.168.53.99	192.168.60.5	ICMP	100	Echo (ping) request id=0x0085, seq=4/1024, tt
52	2025-11-03 02:3…	192.168.60.5	192.168.53.99	ICMP	100	Echo (ping) reply id=0x0085, seq=4/1024, tt
53	2025-11-03 02:3…	192.168.60.5	192.168.53.99	ICMP	100	Echo (ping) reply id=0x0085, seq=4/1024, tt
54	2025-11-03 02:3…	10.9.0.11	10.9.0.5	UDP	128	9090 → 34330 Len=84
55	2025-11-03 02:3…	10.9.0.11	10.9.0.5	UDP	128	9090 → 34330 Len=84
56	2025-11-03 02:3…	10.9.0.5	10.9.0.11	UDP	128	34330 → 9090 Len=84
57	2025-11-03 02:3…	10.9.0.5	10.9.0.11	UDP	128	34330 → 9090 Len=84
58	2025-11-03 02:3…	192.168.53.99	192.168.60.5	ICMP	100	Echo (ping) request id=0x0085, seq=5/1280, tt
59	2025-11-03 02:3…	192.168.53.99	192.168.60.5	ICMP	100	Echo (ping) request id=0x0085, seq=5/1280, tt
60	2025-11-03 02:3…	192.168.60.5	192.168.53.99	ICMP	100	Echo (ping) reply id=0x0085, seq=5/1280, tt
61	2025-11-03 02:3…	192.168.60.5	192.168.53.99	ICMP	100	Echo (ping) reply id=0x0085, seq=5/1280, tt
62	2025-11-03 02:3…	10.9.0.11	10.9.0.5	UDP	128	9090 → 34330 Len=84
63	2025-11-03 02:3…	10.9.0.11	10.9.0.5	UDP	128	9090 → 34330 Len=84
64	2025-11-03 02:3…	10.9.0.5	10.9.0.11	UDP	128	34330 → 9090 Len=84
65	2025-11-03 02:3…	10.9.0.5	10.9.0.11	UDP	128	34330 → 9090 Len=84
66	2025-11-03 02:3…	192.168.53.99	192.168.60.5	ICMP	100	Echo (ping) request id=0x0085, seq=6/1536, tt
67	2025-11-03 02:3…	192.168.53.99	192.168.60.5	ICMP	100	Echo (ping) request id=0x0085, seq=6/1536, tt
68	2025-11-03 02:3…	192.168.60.5	192.168.53.99	ICMP	100	Echo (ping) reply id=0x0085, seq=6/1536, tt
69	2025-11-03 02:3…	192.168.60.5	192.168.53.99	ICMP	100	Echo (ping) reply id=0x0085, seq=6/1536, tt
70	2025-11-03 02:3…	10.9.0.11	10.9.0.5	UDP	128	9090 → 34330 Len=84
71	2025-11-03 02:3…	10.9.0.11	10.9.0.5	UDP	128	9090 → 34330 Len=84

- Wireshark captures UDP packets (port 9090) encapsulating ICMP traffic between the client and server.
- Each ICMP request (`Echo (ping) request`) from 192.168.53.99 and the corresponding `Echo reply` from 192.168.60.5 are visible.
- The interleaving of UDP and ICMP packets confirms **encapsulation (UDP)** and **decapsulation (ICMP)** working correctly.
- The traffic flow clearly shows the request passing through the tunnel and the reply returning back.
- This verifies that the tunnel successfully supports bidirectional ICMP echo communication.
- The system now demonstrates a fully functional VPN connection with encapsulated transport over UDP.



297	2025-11-03 02:4…	fe80::ecd0:fe4e:90c…	ff02::16	ICMPv6	152	Multicast Listener Report Message v2
298	2025-11-03 02:4…	fe80::ecd0:fe4e:90c…	ff02::16	ICMPv6	152	Multicast Listener Report Message v2
299	2025-11-03 02:4…	10.9.0.5	10.9.0.11	UDP	97	34330 → 9090 Len=53
300	2025-11-03 02:4…	10.9.0.5	10.9.0.11	UDP	97	34330 → 9090 Len=53
301	2025-11-03 02:4…	192.168.53.99	192.168.60.5	TELNET	69	Telnet Data ...
302	2025-11-03 02:4…	192.168.53.99	192.168.60.5	TCP	69	[TCP Keep-Alive] 39618 → 23 [PSH, ACK] Seq=190
303	2025-11-03 02:4…	192.168.60.5	192.168.53.99	TELNET	69	Telnet Data ...
304	2025-11-03 02:4…	192.168.60.5	192.168.53.99	TCP	69	[TCP Keep-Alive] 23 → 39618 [PSH, ACK] Seq=281
305	2025-11-03 02:4…	10.9.0.11	10.9.0.5	UDP	97	9090 → 34330 Len=53
306	2025-11-03 02:4…	10.9.0.11	10.9.0.5	UDP	97	9090 → 34330 Len=53
307	2025-11-03 02:4…	10.9.0.5	10.9.0.11	UDP	96	34330 → 9090 Len=52
308	2025-11-03 02:4…	10.9.0.5	10.9.0.11	UDP	96	34330 → 9090 Len=52
309	2025-11-03 02:4…	192.168.53.99	192.168.60.5	TCP	68	39618 → 23 [ACK] Seq=190111996 Ack=2815478494
310	2025-11-03 02:4…	192.168.53.99	192.168.60.5	TCP	68	[TCP Keep-Alive ACK] 39618 → 23 [ACK] Seq=1901
311	2025-11-03 02:4…	10.9.0.5	10.9.0.11	UDP	97	34330 → 9090 Len=53
312	2025-11-03 02:4…	10.9.0.5	10.9.0.11	UDP	97	34330 → 9090 Len=53
313	2025-11-03 02:4…	192.168.53.99	192.168.60.5	TELNET	69	Telnet Data ...
314	2025-11-03 02:4…	192.168.53.99	192.168.60.5	TCP	69	[TCP Keep-Alive] 39618 → 23 [PSH, ACK] Seq=190
315	2025-11-03 02:4…	192.168.60.5	192.168.53.99	TELNET	69	Telnet Data ...
316	2025-11-03 02:4…	192.168.60.5	192.168.53.99	TCP	69	[TCP Keep-Alive] 23 → 39618 [PSH, ACK] Seq=281
317	2025-11-03 02:4…	10.9.0.11	10.9.0.5	UDP	97	9090 → 34330 Len=53
318	2025-11-03 02:4…	10.9.0.11	10.9.0.5	UDP	97	9090 → 34330 Len=53
319	2025-11-03 02:4…	10.9.0.5	10.9.0.11	UDP	96	34330 → 9090 Len=52
320	2025-11-03 02:4…	10.9.0.5	10.9.0.11	UDP	96	34330 → 9090 Len=52
321	2025-11-03 02:4…	192.168.53.99	192.168.60.5	TCP	68	39618 → 23 [ACK] Seq=190111997 Ack=2815478495
322	2025-11-03 02:4…	192.168.53.99	192.168.60.5	TCP	68	[TCP Keep-Alive ACK] 39618 → 23 [ACK] Seq=1901
323	2025-11-03 02:4…	10.9.0.5	10.9.0.11	UDP	97	34330 → 9090 Len=53
324	2025-11-03 02:4…	10.9.0.5	10.9.0.11	UDP	97	34330 → 9090 Len=53
325	2025-11-03 02:4…	192.168.53.99	192.168.60.5	TELNET	69	Telnet Data ...
326	2025-11-03 02:4…	192.168.53.99	192.168.60.5	TCP	69	[TCP Keep-Alive] 39618 → 23 [PSH, ACK] Seq=190
327	2025-11-03 02:4…	192.168.60.5	192.168.53.99	TELNET	69	Telnet Data ...
328	2025-11-03 02:4…	192.168.60.5	192.168.53.99	TCP	69	[TCP Keep-Alive] 23 → 39618 [PSH, ACK] Seq=281

- Wireshark now shows a mix of **UDP**, **ICMP**, and **TCP/Telnet** packets between the two hosts.
- TCP segments labeled as “Telnet Data” and “TCP Keep-Alive” confirm active Telnet session traffic through the tunnel.
- UDP packets on port 9090 represent the tunneling layer encapsulating these TCP packets.
- The sequence of exchanges (ACK, PSH, Keep-Alive) confirms stable TCP session management through the VPN.
- The trace proves that application-layer traffic (Telnet) flows securely through the established tunnel.
- This demonstrates that the VPN tunnel now provides **complete, bidirectional, and reliable data transmission** across networks.

Task 6: Tunnel-Breaking Experiment

```
Client2:PES1UG24CS840:VINAY/
$telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
f1051a1ff52d 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.
Last login: Mon Nov  3 18:07:30 UTC 2025 on pts/2
seed@f1051a1ff52d:~$
```

- The client initiates a Telnet connection to Host V (`192.168.60.5`) and logs in successfully.
- The Ubuntu welcome message confirms a **stable TCP session** between the client and server.

- This proves that the VPN tunnel is **fully functional**, carrying bidirectional Telnet (TCP) packets across the virtual link.
- The connection remains open and interactive — any command typed here is sent through the tunnel to the remote host.
- At this stage, the TCP connection is reliable and active because both `tun_client_select.py` and `tun_server_select.py` are running.
- However, once the tunnel is broken, this state will change since the TCP connection depends on continuous tunnel availability.

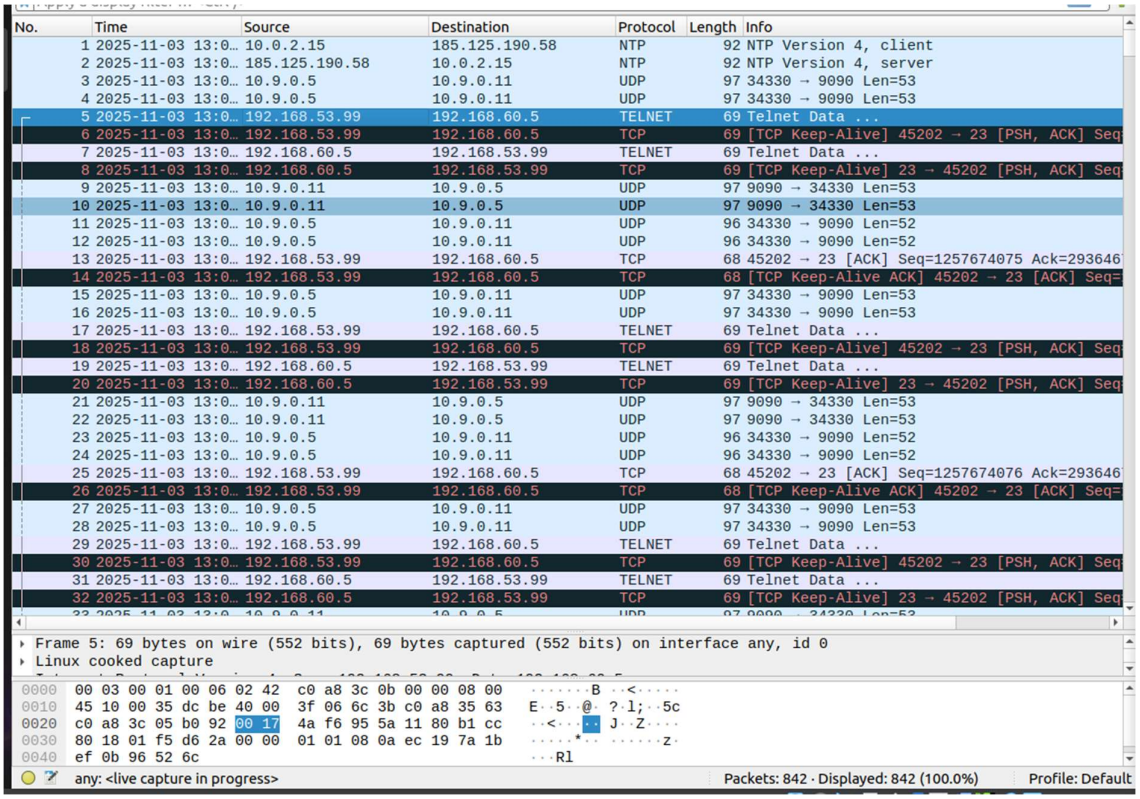
```
router:PES1UG24CS840:VINAY/volumes
$ ./tun_server_select.py
Interface Name: CS8400
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.53.99 --> 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
```

- The server script shows alternating messages such as:
  - **From socket <== 192.168.53.99 → 192.168.60.5**
  - **From tun ==> 192.168.60.5 → 192.168.53.99**
- This indicates that the server is actively **relaying Telnet data** back and forth between Host U and Host V.
- Every line confirms **TCP segment encapsulation and decapsulation** through the tunnel.
- The tunnel acts as a bridge between the two subnets, ensuring complete two-way communication.
- This state represents the VPN functioning correctly before the tunnel is broken.
- The next step — pressing `Ctrl + C` on this server — intentionally stops this tunnel and disrupts all traffic flow.

```
To restore this content, you can run the 'unminimize' command.
Last login: Mon Nov  3 18:07:30 UTC 2025 on pts/2
seed@f1051a1ff52d:~$ hhiiiii
```

- Once `tun_server_select.py` is stopped, the Telnet client screen becomes **unresponsive**.
- Even when text like `hhi` is typed, **nothing is echoed** back — the keystrokes appear to vanish.
- This happens because the VPN tunnel was carrying TCP packets, and its interruption causes the **underlying path to break**.

- The TCP connection remains **half-open temporarily**, but packets can no longer reach the destination or return.
- Eventually, the Telnet client will detect a timeout or "broken pipe" error because the tunnel no longer forwards data.
- This behavior shows that **TCP relies on the continuity of the tunnel** for data transfer and acknowledgments.



- Wireshark shows the moment when Telnet data (`PSH`, `ACK`, `Keep-Alive`) stops being acknowledged.
- The last visible packets are TCP segments between `192.168.53.99` and `192.168.60.5` marked as **"Keep-Alive"**, attempting to maintain the session.
- No further ACKs arrive because the VPN tunnel is now down, so these packets cannot traverse the network.
- The **UDP traffic on port 9090** (tunnel encapsulation) ceases immediately, confirming tunnel termination.
- This results in packet retransmissions or session timeout from the client side.
- Essentially, this trace captures the **exact moment when the tunnel is broken** and the Telnet connection becomes unreachable.

Code Snippets

1. **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'CS840%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:
# Get a packet from the tun interface
        packet = os.read(tun, 2048)
        if packet:
                ip = IP(packet)
                print(ip.summary())
```

- The script creates a **virtual TUN interface** that operates at the IP layer.
- It uses `fcntl.ioctl()` with `TUNSETIFF` to register the interface name (`CS8400`) and flags (`IFF_TUN | IFF_NO_PI`).
- `/dev/net/tun` acts as the system device for handling virtual network packets.
- The `while True` loop continuously **reads packets** from the TUN interface using `os.read()`.
- Each packet is parsed as an **IP packet** using Scapy's `IP()` and summarized with `ip.summary()`.
- This lets you monitor all IP packets passing through your custom virtual network interface.

2. tun1.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'CS840%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

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

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

while True:
    # Get a packet from the tun interface
    packet = os.read(tun, 2048)
    if packet:
        pkt = IP(packet)
        print("{}:".format(ifname), pkt.summary())
```

- This Python script creates a **TUN interface** that operates at the IP layer using `/dev/net/tun`.
- The line with `struct.pack()` sets the interface name as `CS8400` and flags it as `IFF_TUN | IFF_NO_PI`.
- `fcntl.ioctl()` actually **creates the virtual interface** and stores its name in `ifname`.
- The script assigns an IP address (`192.168.53.99/24`) and brings the interface **up** using `os.system()` commands.
- In the infinite loop, it reads packets from the TUN device using `os.read()` and decodes them with Scapy's `IP()`.
- Each captured packet is displayed with its source and destination using `pkt.summary()`.

3. tun_client.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'CS840%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

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

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

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

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

- This script sets up a **TUN interface** (virtual network device) to handle IP packets in user space.
- The interface is created using `/dev/net/tun` with flags `IFF_TUN | IFF_NO_PI`.
- It assigns the IP `192.168.53.99/24` to the interface and activates it using `ip` commands.
- A route to the network `192.168.60.0/24` is also added, enabling packet forwarding.
- Finally, a **UDP socket** is created to send and receive encapsulated packets over the tunnel.
- This forms the base for building a **VPN-like connection** between two systems.

4. tun_server1.py

```
 5 import os
 6 from scapy.all import *
 7
 8 IP_A = "0.0.0.0"
 9 PORT = 9090
10
11 TUNSETIFF = 0x400454ca
12 IFF_TUN   = 0x0001
13 IFF_TAP   = 0x0002
14 IFF_NO_PI = 0x1000
15
16 # Create a tun interface
17 tun = os.open("/dev/net/tun", os.O_RDWR)
18 ifr = struct.pack('16sH', b'CS840%d', IFF_TUN | IFF_NO_PI)
19 ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
20 ifname  = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
21 print("Interface Name: {}".format(ifname))
22
23 # Set up the tun interface
24 os.system("ip addr add 192.168.53.1/24 dev {}".format(ifname))
25 os.system("ip link set dev {} up".format(ifname))
26
27 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
28 sock.bind((IP_A, PORT))
29
30 while True:
31    data, (ip, port) = sock.recvfrom(2048)
32    pkt = IP(data)
33    print("{}:{} --> {}:{}".format(ip, port, IP_A, PORT))
34    print("   Inside: {} --> {}".format(pkt.src, pkt.dst))
35    os.write(tun, bytes(pkt))
```

- This script sets up a **TUN interface** to capture and inject IP packets programmatically.
- The interface name (`CS8400`) is registered and configured with IP `192.168.53.1/24`.
- A **UDP socket** is created and bound to port `9090`, allowing data exchange over the network.
- Inside the loop, packets received via UDP are treated as IP packets using Scapy's `IP()` class.
- The script prints the packet's flow details and writes it to the TUN interface for routing.
- Essentially, it **receives encapsulated packets** and injects them into the virtual network stack.

5. **tun_server.py**

```python
#!/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))
```

- The script creates a **UDP socket** that listens on all interfaces (`0.0.0.0`) at port `9090`.
- It waits to receive packets using `sock.recvfrom(2048)` inside an infinite loop.
- When a packet arrives, it prints the **source and destination** IP and port details.
- The received data is converted into an **IP packet** using Scapy's `IP()` function.
- It then prints the internal IP-level communication (`pkt.src → pkt.dst`).
- This code acts as a **simple packet listener and analyzer** for UDP-encapsulated IP packets.

6. **tun_client_select.py**

```python
#!/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'CS840%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
```

```
27.os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
28.os.system("ip link set dev {} up".format(ifname))
29.
30.# Set up routing
31.os.system("ip route add 192.168.60.0/24 dev {}".format(ifname))
32.
33.# Create UDP socket
34.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
35.
36.fds = [sock, tun]
37.while True:
38.  # this will block until at least one socket is ready
39.  ready, _, _ = select.select(fds, [], [])
40.
41.  for fd in ready:
42.    if fd is sock:
43.       data, (ip, port) = sock.recvfrom(2048)
44.       pkt = IP(data)
45.       print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
46.       os.write(tun, data)
47.
48.    if fd is tun:
49.       packet = os.read(tun, 2048)
50.       pkt = IP(packet)
51.       print("From tun    ==>: {} --> {}".format(pkt.src, pkt.dst))
52.       sock.sendto(packet, ('10.9.0.11', 9090))
```

- This program creates a **TUN interface** (`/dev/net/tun`) to handle IP packets in user space.
- It assigns the IP `192.168.53.99/24` and configures routing for the `192.168.60.0/24` network.
- A **UDP socket** is set up to send and receive packets over port `9090`.
- The `select()` function monitors both the TUN and socket for incoming data.
- When data arrives from the **socket**, it's written into the TUN interface (injected into kernel routing).
- When data comes from the **TUN**, it's sent through the UDP socket to the remote peer (`10.9.0.11:9090`).

7. tun_server_select.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'CS840%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))

# We need to initialize ip and port (their values do not matter)
ip   = '10.9.0.5'
port = 10000

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

  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))
```

- This script sets up a **TUN interface** (`/dev/net/tun`) for capturing and injecting IP packets in user space.
- It assigns the IP `192.168.53.1/24` to the interface and brings it **up** for communication.
- A **UDP socket** is created on port `9090` to exchange encapsulated packets with a remote host.
- The `select()` call monitors both the TUN device and socket for incoming traffic.
- Packets from the **socket** are written to the TUN interface (toward the kernel), and packets from **TUN** are sent through the UDP socket.
- This enables **bidirectional tunneling** between two virtual networks over UDP.