Name: Praneet T.H

SRN: PES1UG23CS439

Section: H

Task 1:

Client:

```
client:PES1UG23CS439:Praneet:/ 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=0.347 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=2 ttl=64 time=0.053 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=3 ttl=64 time=0.075 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=4 ttl=64 time=0.128 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=5 ttl=64 time=0.128 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=6 ttl=64 time=0.163 ms
^C
--- server-router ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5082ms
rtt min/avg/max/mdev = 0.053/0.149/0.347/0.095 ms
client:PES1UG23CS439:Praneet:/ 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 ---
6 packets transmitted, 0 received, 100% packet loss, time 5116ms
```

This output from the client machine performs two key tests. It first proves it can reach the server-router (0% packet loss) but then correctly fails to reach Host V (100% packet loss) confirming that the networks are isolated.

Router:

```
router:PES1UG23CS439:Praneet:/ 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=1.64 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.162 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=64 time=0.129 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=64 time=0.057 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=64 time=0.131 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=64 time=0.130 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=64 time=0.290 ms
^C
--- 192.168.60.5 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6116ms
rtt min/avg/max/mdev = 0.057/0.362/1.635/0.523 ms
```

This terminal output from the server-router confirms it has a successful, direct connection to the host 192.168.60.5 as shown by the 0% packet loss.

Client:

```
client:PES1UG23CS439:Praneet:/ 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=0.367 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=2 ttl=64 time=0.154 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=3 ttl=64 time=0.075 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=4 ttl=64 time=0.080 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=0.155 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=7 ttl=64 time=0.157 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=8 ttl=64 time=0.147 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=9 ttl=64 time=0.160 ms
^C
--- server-router ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8169ms
rtt min/avg/max/mdev = 0.062/0.150/0.367/0.085 ms
client:PES1UG23CS439:Praneet:/
```

This output shows the client machine running a simple, successful ping against the server-router. This ping was run while the packet capture in the next image was active, purely to generate traffic to be sniffed.

Router:

```
router:PES1UG23CS439:Praneet:/ 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
13:09:06.387524 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 5, seq 1, length 64
13:09:06.387749 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 5, seq 1, length 64
13:09:07.390230 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 5, seq 2, length 64
13:09:07.390280 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 5, seq 2, length 64
13:09:08.413218 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 5, seq 3, length 64
13:09:08.413242 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 5, seq 3, length 64
13:09:09.439003 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 5, seq 4, length 64
13:09:09.439022 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 5, seq 4, length 64
13:09:10.460779 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 5, seq 5, length 64
13:09:10.460798 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 5, seq 5, length 64
13:09:11.483174 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 5, seq 6, length 64
13:09:11.483222 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 5, seq 6, length 64
13:09:11.547363 ARP, Request who-has 10.9.0.5 tell 10.9.0.11, length 28
13:09:11.547583 ARP, Request who-has 10.9.0.11 tell 10.9.0.5, length 28
13:09:11.547612 ARP, Reply 10.9.0.11 is-at 02:42:0a:09:00:0b, length 28
13:09:11.547616 ARP, Reply 10.9.0.5 is-at 02:42:0a:09:00:05, length 28
13:09:12.507418 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 5, seq 7, length 64
13:09:12.507469 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 5, seq 7, length 64
13:09:13.531759 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 5, seq 8, length 64
13:09:13.531806 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 5, seq 8, length 64
13:09:14.556162 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 5, seq 9, length 64
13:09:14.556213 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 5, seq 9, length 64
```

This tcpdump capture from the server-router's eth0 interface successfully captures the ICMP "request" and "reply" packets from the ping shown in previous snip verifying that the packet sniffing tool is operational.

Task 2.A:

Client:

```
client:PES1UG23CS439:Praneet:/chmod a+x tun.py
client:PES1UG23CS439:Praneet:/ ./tun.py &
[1] 22
client:PES1UG23CS439:Praneet:/Interface Name: tun0
 ip addr
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
8: eth0@if9: <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:PES1UG23CS439:Praneet:/
```

This terminal output shows the unmodified tun.py script being executed. It successfully creates the default tun0 virtual network interface, which is then listed as a new device by the ip addr command. This tun0 interface is now a gateway into the kernel's network stack. Any packet the kernel sends *to* tun0 will be redirected and given to your running tun.py script.

Client:

```
client:PES1UG23CS439:Praneet:/ chmod a+x tun.py
client:PES1UG23CS439:Praneet:/./tun.py &
[1] 29
client:PES1UG23CS439:Praneet:/Interface Name: CS4390
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
3: CS4390: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
8: eth0@if9: <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:PES1UG23CS439:Praneet:/
```

The output confirms that the tun.py script was successfully modified. After changing the name prefix in the code to CS439 , running the script now creates a custom virtual interface called CS4390 as verified by the ip addr output. This shows that the interface name isn't a fixed system value but a configurable parameter set by the user-space program that controls it.

Task 2.B and Task 2.C:

Client:

```
client:PES1UG23CS439:Praneet:/ip addr add 192.168.53.99/24 dev CS4390
client:PES1UG23CS439:Praneet:/ip link set dev CS4390 up
client:PES1UG23CS439:Praneet:/./tun.py &
[2] 61
client:PES1UG23CS439:Praneet:/Interface Name: CS4391
 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
^C
--- 192.168.53.5 ping statistics ---
10 packets transmitted, 0 received, 100% packet loss, time 9222ms

client:PES1UG23CS439:Praneet:/./tun.py &
[3] 66
client:PES1UG23CS439:Praneet:/Interface Name: CS4392
 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 ---
9 packets transmitted, 0 received, 100% packet loss, time 8182ms

client:PES1UG23CS439:Praneet:/
```

This screenshot perfectly demonstrates the routing behaviour of the new TUN interface. The top half shows that pinging an IP on the TUN's configured subnet (192.168.53.5) successfully routes the packets *to* the tun.py script, which prints their contents. The bottom half confirms that pinging a different network (192.168.60.5) does not route packets to the script, which prints nothing. In both cases, the ping fails with 100% loss because the script is only reading packets not forwarding or replying to them.

Task 2.D:

Client:

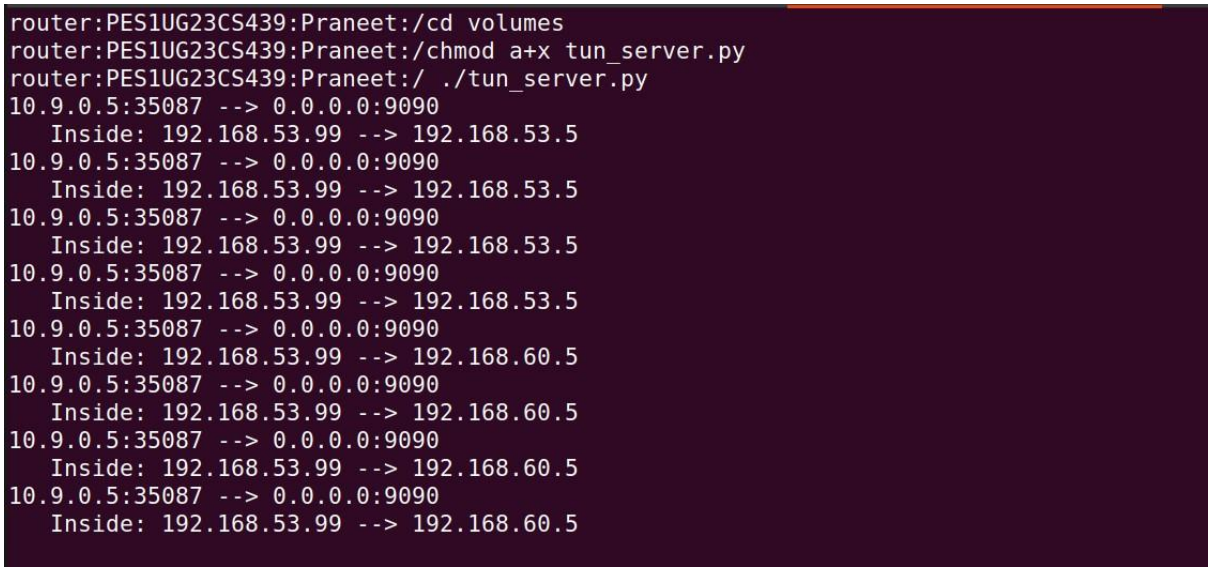
```
client:PES1UG23CS439:Praneet:/ chmod a+x tun1.py
client:PES1UG23CS439:Praneet:/ ./tun1.py &
[1] 85
client:PES1UG23CS439:Praneet:/Interface Name: CS4390
 ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
8: eth0@if9: <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
9: CS4390: <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 CS4390
       valid_lft forever preferred_lft forever
client:PES1UG23CS439:Praneet:/ ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
CS4390: 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=11.3 ms
CS4390: 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=2.82 ms
CS4390: 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=12.1 ms
CS4390: 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=5.83 ms
CS4390: 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=5.04 ms
^C
--- 192.168.53.5 ping statistics ---
```

The screenshot shows a fully functional TUN interface in action. When the script tun1.py is run, it creates the interface CS4390 with the IP address 192.168.53.99/24. A

ping to 192.168.53.5 triggers ICMP echo requests that the script intercepts, as shown by its output (CS4390: IP / ICMP 192.168.53.99 > 192.168.53.5). Instead of dropping these packets, the script crafts and writes echo replies back into the interface. The ping command then receives these replies successfully confirming smooth bidirectional communication. This demonstrates that the script is no longer passive it actively reads incoming packets, generates valid responses and integrates seamlessly with the kernel's networking stack, effectively simulating a real host.

Task 3:

Router:



On the server-router side this screenshot shows tun_server.py actively receiving UDP packets from the client. For each packet, it prints the insider details 192.168.53.99 → 192.168.53.5 and 192.168.53.99 → 192.168.60.5 showing the original inner IP packets that were encapsulated by the client. This proves that the client-side encapsulation and UDP transmission are functioning properly. The server script's role here is simple yet crucial where it's unpacking each UDP "envelope" and displaying the original IP traffic within, confirming that the tunnel is alive and correctly carrying data end-to-end. This image validates the successful transport and de-encapsulation stages of the VPN.

Client:

```
client:PES1UG23CS439:Praneet:/ chmod a+x tun_client.py
client:PES1UG23CS439:Praneet:/ ./tun_client.py &
[1] 106
client:PES1UG23CS439:Praneet:/Interface Name: CS4390
ip addr
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
8: eth0@if9: <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
10: CS4390: <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 CS4390
       valid_lft forever preferred_lft forever
client:PES1UG23CS439:Praneet:/ 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
^C
--- 192.168.53.5 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3080ms
```

This screenshot captures the client side where tun_client.py is launched, creating the virtual interface CS4390 (192.168.53.99/24). When a ping is sent to 192.168.53.5, the script successfully intercepts the ICMP echo-request packets, as shown in its output. However, the ping command reports 100% packet loss indicating that the script is no longer crafting replies like before. Instead, it's encapsulating these intercepted packets inside UDP and sending them out to the VPN server. Here the client has become the sending side of the VPN, pushing traffic into the tunnel.

Client:

```
client:PES1UG23CS439:Praneet:/ 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
^C
--- 192.168.53.5 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3080ms

client:PES1UG23CS439:Praneet:/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
^C
--- 192.168.60.5 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3070ms

client:PES1UG23CS439:Praneet:/ 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 CS4390 proto kernel scope link src 192.168.53.99
192.168.60.0/24 dev CS4390 scope link
client:PES1UG23CS439:Praneet:/
```

Here, the client pings another host, 192.168.60.5, and again, tun_client.py logs the packets being intercepted proof that the tunnel captures not just local subnet traffic but also remote private network traffic. The pings fail with 100% packet loss, which is expected since the server side isn't generating replies yet. At the bottom of the screenshot, the ip route command reveals a new routing rule: 192.168.60.0/24 dev CS4390. This is the key moment that shows how the client dynamically updates the

routing table so that all traffic meant for the remote network is rerouted into the TUN interface instead of going through the normal gateway.

Task 4:

Router:

```
router:PES1UG23CS439:Praneet:/ chmod a+x tun_server1.py
router:PES1UG23CS439:Praneet:/ ./tun_server1.py
Interface Name: CS4390
10.9.0.5:36060 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:36060 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:36060 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:36060 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:36060 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:36060 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
```

On the server-router, the tun_server1.py script is now running and functioning as the tunnel's receiver and injector. It logs the UDP packets arriving from the client and shows the extracted payload Inside: 192.168.53.99 → 192.168.60.5 which represents the original ICMP request. The script then writes this decapsulated packet into its own virtual interface (CS4390), injecting it into the server's kernel networking stack. From there the router's regular IP routing takes over where it does the forwarding of the packet to its correct destination, Host V. This confirms that encapsulation and decapsulation are both functioning perfectly on opposite ends of the tunnel.

Client:

```
client:PES1UG23CS439:Praneet:/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
^C
--- 192.168.60.5 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 5146ms
```

This screenshot shows the client (Host U) side where a ping is sent to 192.168.60.5. The background output from tun_client.py confirms it's intercepting and encapsulating these ICMP packets into UDP before sending them toward the VPN server. However, the ping itself reports 100% packet loss which is expected at this stage since the return path isn't configured yet. This shows the client acting as the tunnel's sending endpoint, successfully capturing outbound packets destined for the remote private network and forwarding them into the tunnel.

Host V:

```
hostV:PES1UG23CS439:Praneet:/ 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
14:23:11.931848 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 12, seq 1, length 64
14:23:11.932207 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 12, seq 1, length 64
14:23:12.959808 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 12, seq 2, length 64
14:23:12.959852 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 12, seq 2, length 64
14:23:13.980121 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 12, seq 3, length 64
14:23:13.980137 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 12, seq 3, length 64
14:23:15.004020 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 12, seq 4, length 64
14:23:15.004047 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 12, seq 4, length 64
14:23:16.027082 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 12, seq 5, length 64
14:23:16.027096 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 12, seq 5, length 64
14:23:16.986512 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
14:23:17.001509 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
14:23:17.001647 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
14:23:17.001667 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
14:23:17.054150 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 12, seq 6, length 64
14:23:17.054195 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 12, seq 6, length 64
```

This taken from Host V (192.168.60.5), provides proof that the 1-way VPN tunnel works. The tcpdump output clearly shows ICMP echo requests arriving from 192.168.53.99 the client's virtual IP and echo replies being sent back in response. This means that the tunnelling packets made the complete journey captured on the client sent through the tunnel decapsulated by the server and routed successfully to their target. The client's ping still shows 100% loss because the return route for the replies isn't yet established, but this screenshot marks a huge milestone the VPN tunnel is fully operational in the client-to-server direction.

Task 5:

Router:

```
router:PES1UG23CS439:Praneet:/ chmod a+x tun_server_select.py
router:PES1UG23CS439:Praneet:/ ./tun_server_select.py
Interface Name: CS4390
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
```

On the server-router, the tun_server_select.py script is doing its job. The log shows it receiving packets From socket <== (from the client) and immediately sending responses From tun ==> (back toward the client). It's the perfect mirror image of what's happening on the client side. Together, these two scripts are simultaneous of the VPN one sending,

one receiving, both constantly translating between the real network and the virtual TUN interfaces.

Client:

```
client:PES1UG23CS439:Praneet:/ chmod a+x tun_client_select.py
client:PES1UG23CS439:Praneet:/ ./tun_client_select.py
Interface Name: CS4390
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
```

This screenshot shows the of working VPN tunnel. The tun_client_select.py script is alive and multitasking listening on two “lines” at once. When it receives packets From tun ==>, it means traffic is leaving the client (like a ping or telnet request). When it logs From socket <==, it’s receiving responses coming back from the VPN server. This seamless back-and-forth flow shows that the script is now a true bi-directional bridge, perfectly transmitting packets from both directions in real time. It’s no longer just sending or just receiving it’s doing both.

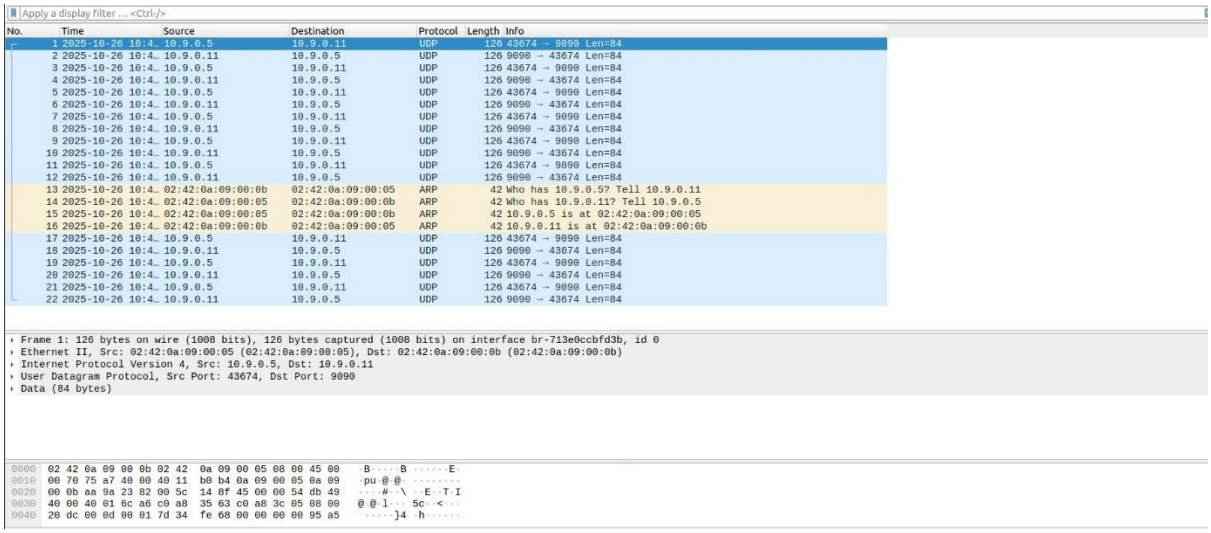
Client:

```
client:PES1UG23CS439:Praneet:/ 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=3.54 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=1.45 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=9.07 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=2.93 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=3.63 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=63 time=2.01 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=3.18 ms
64 bytes from 192.168.60.5: icmp_seq=8 ttl=63 time=4.97 ms
64 bytes from 192.168.60.5: icmp_seq=9 ttl=63 time=1.15 ms
^C
--- 192.168.60.5 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8020ms
rtt min/avg/max/mdev = 1.147/3.548/9.069/2.249 ms
client:PES1UG23CS439:Praneet:/
```

The terminal shows a ping to 192.168.60.5 running with 0% packet loss. Every request is met with a reply, confirming that the entire VPN from encapsulation to routing to decapsulation is working. The client believes it’s talking directly to the remote host but

in reality, each packet is being quietly wrapped, tunneling through UDP and unwrapped on the other side. This screenshot captures a fully functional two-way link carrying ICMP traffic end to end.

Wireshark:



Instead of seeing ICMP packets everywhere all you see are UDP packets traveling between the client (10.9.0.5) and the server (10.9.0.11) on port 9090. That's the worki og tunnelling the ping packets are still there, just hidden safely inside those UDP envelopes. From the outside world's perspective, it looks like an ordinary stream of UDP traffic, but underneath, VPN is quietly carrying full ICMP conversations. This screenshot proves the VPN's stealth and effectiveness which is real traffic, completely encapsulated.

Client:

```
client:PES1UG23CS439:Praneet:/ telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
92dd73fab3e6 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.15.0-139-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@92dd73fab3e6:~$ ls
seed@92dd73fab3e6:~$
```

The client connects to 192.168.60.5 and immediately sees a login prompt from the remote host. This isn't just a ping anymore it's a full TCP session working through the

VPN. Telnet involves complex handshakes, acknowledgments and data exchange, so seeing that login prompt means the tunnel is handling everything flawlessly. This moment confirms that the custom VPN can now transport any kind of network traffic, from pings to full application sessions.

Client:

```
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
```

You can see packets continuously flowing both ways From tun ==> and From socket <== reflecting the back-and-forth of the telnet session. It's the proof of resilience the script isn't just functional, it's stable under real network load. Each line represents another successful exchange of data through your encrypted tunnel, showing that the client can handle sustained, simultaneous traffic without missing a beat.

Wireshark:

No.	Time	Source	Destination	Protocol	Length	Info
65	2025-10-26 10:4…	10.9.0.5	10.9.0.11	UDP	94	43674 → 9090 Len=52
66	2025-10-26 10:4…	10.9.0.11	10.9.0.5	UDP	302	9090 → 43674 Len=260
67	2025-10-26 10:4…	10.9.0.5	10.9.0.11	UDP	94	43674 → 9090 Len=52
68	2025-10-26 10:4…	10.9.0.11	10.9.0.5	UDP	115	9090 → 43674 Len=73
69	2025-10-26 10:4…	10.9.0.5	10.9.0.11	UDP	94	43674 → 9090 Len=52
70	2025-10-26 10:4…	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
71	2025-10-26 10:4…	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
72	2025-10-26 10:4…	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
73	2025-10-26 10:4…	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
74	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	95	43674 → 9090 Len=53
75	2025-10-26 10:5…	10.9.0.11	10.9.0.5	UDP	95	9090 → 43674 Len=53
76	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	94	43674 → 9090 Len=52
77	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	95	43674 → 9090 Len=53
78	2025-10-26 10:5…	10.9.0.11	10.9.0.5	UDP	95	9090 → 43674 Len=53
79	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	94	43674 → 9090 Len=52
80	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	96	43674 → 9090 Len=54
81	2025-10-26 10:5…	10.9.0.11	10.9.0.5	UDP	96	9090 → 43674 Len=54
82	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	94	43674 → 9090 Len=52
83	2025-10-26 10:5…	10.9.0.11	10.9.0.5	UDP	115	9090 → 43674 Len=73
84	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	94	43674 → 9090 Len=52
85	2025-10-26 10:5…	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
86	2025-10-26 10:5…	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
87	2025-10-26 10:5…	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
88	2025-10-26 10:5…	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

```
▸ Frame 1: 107 bytes on wire (856 bits), 107 bytes captured (856 bits) on interface br-713e0ccbfd3b, id 0
▸ Ethernet II, Src: 02:42:c1:fa:9c:4a (02:42:c1:fa:9c:4a), Dst: IPv6mcast_fb (33:33:00:00:00:fb)
▸ Internet Protocol Version 6, Src: fe80::42:c1ff:fefa:9c4a, Dst: ff02::fb
▸ User Datagram Protocol, Src Port: 5353, Dst Port: 5353
▸ Multicast Domain Name System (query)
```

Captured during the telnet session, it shows a complex flow of UDP packets of various sizes some small (ACKs), some larger. Each of these represents real TCP segments being wrapped and which is being tunnled by the VPN. The different packets and their packet lengths perfectly mirrors the movement of a live TCP conversation, proving that the VPN is handling bi-directional traffic with ease.

Task 6:

Client:

```
client:PES1UG23CS439:Praneet:/telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
92dd73fab3e6 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.15.0-139-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: Sun Oct 26 14:49:53 UTC 2025 from 192.168.53.99 on pts/2
seed@92dd73fab3e6:~$ asdasdasdasd
```

This screenshot captures the telnet session right at the moment the tunnel breaks. Logging into the remote host and types a random command asdasdasdasd but nothing happens. The screen just hangs. What's really happening is that the command has been sent out through the VPN tunnel, but since the server-side script was killed, it never reaches the destination. What you see on-screen isn't the remote machine echoing your typing it's just the local terminal showing your keystrokes. Behind the scenes, the telnet client's TCP stack is still patiently waiting for an acknowledgment that will never come. The session isn't dead but just frozen and caught mid-connection while TCP keeps retrying in hope that the network will recover.

Router:

```
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From 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
^CTraceback (most recent call last):
  File "./tun_server_select.py", line 38, in <module>
    ready, _, _ = select.select(fds, [], [])
KeyboardInterrupt

router:PES1UG23CS439:Praneet:/ ./tun_server_select.py
Interface Name: CS4390
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
```

In the top half, the client's tun_client_select.py script shows a bunch of From tun ==> messages, meaning it's still sending data toward the tunnel, but there are no From socket <== replies coming back. The tunnel has gone one-way completely silent on the return path. Then comes the bottom half, where the tun_server_select.py script is restarted. Instantly, the logs light up again it receives packets From socket <== and sends responses From tun ==>. The moment the server comes back online, the backlog of re-transmitted data finally gets through and the telnet session get alive again .It's a perfect visual of a connection pausing and reviving .

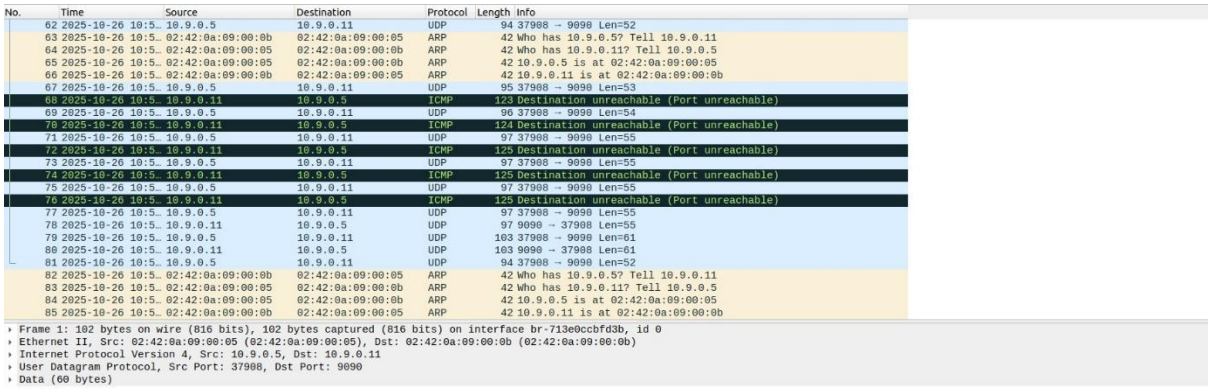
Client:

```
client:PES1UG23CS439:Praneet:/ ./tun_client_select.py
Interface Name: CS4390
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From 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 tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.53.99 --> 192.168.60.5
```

This earlier client-side log shows how things looked before the freeze a perfectly working VPN tunnel in full duplex mode. You can see the rhythmic back-and-forth of From tun ==> (keystrokes being sent out) and From socket <== (replies coming back). Each exchange represents smooth communication between the client and the remote telnet server. This snapshot sets the baseline the VPN is alive, responsive and stable

before the server script is killed. Comparing this to the “frozen” log later highlights exactly how the disruption changes the flow and how it’s restored once the server restarts.

Wireshark:



No.	Time	Source	Destination	Protocol	Length	Info
62	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	94	37908 → 9090 Len=52
63	2025-10-26 10:5…	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
64	2025-10-26 10:5…	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
65	2025-10-26 10:5…	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
66	2025-10-26 10:5…	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
67	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	95	37908 → 9090 Len=53
68	2025-10-26 10:5…	10.9.0.11	10.9.0.5	ICMP	123	Destination unreachable (Port unreachable)
69	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	96	37908 → 9090 Len=54
70	2025-10-26 10:5…	10.9.0.11	10.9.0.5	ICMP	124	Destination unreachable (Port unreachable)
71	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	97	37908 → 9090 Len=55
72	2025-10-26 10:5…	10.9.0.11	10.9.0.5	ICMP	125	Destination unreachable (Port unreachable)
73	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	97	37908 → 9090 Len=55
74	2025-10-26 10:5…	10.9.0.11	10.9.0.5	ICMP	125	Destination unreachable (Port unreachable)
75	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	97	37908 → 9090 Len=55
76	2025-10-26 10:5…	10.9.0.11	10.9.0.5	ICMP	125	Destination unreachable (Port unreachable)
77	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	97	37908 → 9090 Len=55
78	2025-10-26 10:5…	10.9.0.11	10.9.0.5	UDP	97	9090 → 37908 Len=55
79	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	103	37908 → 9090 Len=61
80	2025-10-26 10:5…	10.9.0.11	10.9.0.5	UDP	103	9090 → 37908 Len=61
81	2025-10-26 10:5…	10.9.0.5	10.9.0.11	UDP	94	37908 → 9090 Len=52
82	2025-10-26 10:5…	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
83	2025-10-26 10:5…	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
84	2025-10-26 10:5…	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
85	2025-10-26 10:5…	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

```
› Frame 1: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface br-713e0ccbfd3b, id 0
› Ethernet II, Src: 02:42:0a:09:00:05 (02:42:0a:09:00:05), Dst: 02:42:0a:09:00:0b (02:42:0a:09:00:0b)
› Internet Protocol Version 4, Src: 10.9.0.5, Dst: 10.9.0.11
› User Datagram Protocol, Src Port: 37908, Dst Port: 9090
› Data (60 bytes)
```

This Wireshark capture gives the network-level proof of what really happened during the outage. It shows the client (10.9.0.5) sending its UDP-encapsulated telnet packet, but instead of a reply, the server (10.9.0.11) immediately sends back an ICMP Destination unreachable (Port unreachable) message. This means the client’s packet made it to the server, but since the server’s VPN script was killed, there was no program listening on port 9090 to handle it. From the outside, it looks like a closed door: the client keeps knocking (sending UDP packets), but the server keeps saying a No. This is the exact moment the tunnel breaks freezing the telnet session until the tunnel is reestablished.