



# VPN Tunneling Lab

**Submitted By: Mahika Gupta**

**SRN: PES1UG20CS243**

**Date: 15/11/2022**

## Contents

**LAB SETUP 1**

**LAB OVERVIEW 2**

**TASK 1: NETWORK SETUP 3**

**TASK 2: CREATE AND CONFIGURE TUN INTERFACE 5**

**TASK 3: SEND THE IP PACKET TO THE VPN SERVER THROUGH A TUNNEL 10**

**TASK 4: SET UP THE VPN SERVER 12**

**TASK 5: HANDLING TRAFFIC IN BOTH DIRECTIONS 13**

**TASK 6: TUNNEL-BREAKING EXPERIMENT 15**

## Lab Setup

Please download the Labsetup.zip file from the below link to your VM, unzip it, enter the Labsetup folder, and use the docker-compose.yml file to set up the lab environment.

[https://seedsecuritylabs.org/Labs\\_20.04/Files/VPN\\_Tunnel/Labsetup.zip](https://seedsecuritylabs.org/Labs_20.04/Files/VPN_Tunnel/Labsetup.zip)

We will create a VPN tunnel between a computer (client) and a gateway, allowing the computer to securely access a private network via the gateway.

We need at least three machines: VPN client (also serving as Host U), VPN server (the router/gateway), and a host in the private network (Host V).

In practice, the VPN client and VPN server are connected via the Internet. For the sake of simplicity, we directly connect these two machines to the same LAN in this lab, i.e., this LAN simulates the Internet. The third machine, Host V, is a computer inside the private network. Users on Host U (outside of the private network) want to communicate with Host V via the VPN tunnel. To simulate this setup, we connect Host V to VPN Server (also serving as a gateway). In such a setup, Host V is not directly accessible from the Internet; nor is it directly accessible from Host U

```
PES1UG20CS243:Mahika~/.../Labsetup$>dockps
1283c87a1ac6  server-router
534316c6eb2a  host-192.168.60.5
a91a4a9d8949  client-10.9.0.5
3cee1426dcf2  host-192.168.60.6
PES1UG20CS243:Mahika~/.../Labsetup$>
```

## Lab Overview

A Virtual Private Network (VPN) is a private network built on top of a public network, usually the Internet. Computers inside a VPN can communicate securely, just like if they were on a real private network that is physically isolated from outside, even though their traffic may go through a public network. VPN enables employees to securely access a company's intranet while traveling; it also allows companies to expand their private networks to places across the country and around the world.

The objective of this lab is to help students understand how VPN works. We focus on a specific type of VPN (the most common type), which is built on top of the transport layer. We will build a very simple VPN from the scratch, and use the process to illustrate how each piece of the VPN technology works. A real VPN program has two essential pieces, tunneling and encryption. This lab only focuses on the tunneling part, helping students understand the tunneling technology, so the tunnel in this lab is not encrypted. There is another more comprehensive VPN lab that includes the encryption part.

This lab covers the following topics:

- Virtual Private Network
- The TUN/TAP virtual interface
- IP tunneling
- Routing

## Task 1: Network Setup

**We are only using docker-compose.yml, please delete docker-compose2.yml** and open all the docker container terminals using the given Labsetup folder.

**Shared Folder** - In this lab, we need to write our own code and run it inside containers. Code editing is more convenient inside the VM than in containers, because we can use our favorite editors. In order for the VM and container to share files, we have created a shared folder between the VM and the container using the Docker volumes. If you look at the Docker Compose file, you will find out that we have added the following entry to some of the containers. It indicates mounting the ./volumes folder on the host machine (i.e., the VM) to the /volumes folder inside the container. We will write our code in the ./volumes folder (on the VM), so they can be used inside the containers

**Packet sniffing** - Being able to sniff packets is very important in this lab, because if things do not go as expected, being able to look at where packets go can help us identify the problems. - Running tcpdump on containers. We have already installed tcpdump on each container. To sniff the packets going through a particular interface, we just need to find out the interface name, and then do the following (assume that the interface name is eth0):

### Command:

```
# tcpdump -i eth0 -n
```

```
PES1UG20CS243:Mahika~/.../Labsetup$>docksh a9
client-10.9.0.5:PES1UG20CS243:Mahika:/
#>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
█
```

## Testing

Please conduct the following testings to ensure that the lab environment is set up correctly:

- Host U - 10.9.0.5 can communicate with VPN Server (server-router)

On Client-10.9.0.5

**Command:**

**# ping server-router**

```
client-10.9.0.5:PES1UG20CS243:Mahika:/
#>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=0.144 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=2 ttl=64 time=0.063 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=3 ttl=64 time=0.130 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=4 ttl=64 time=0.278 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=5 ttl=64 time=0.145 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=6 ttl=64 time=0.155 ms
^C
--- server-router ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5120ms
rtt min/avg/max/mdev = 0.063/0.152/0.278/0.063 ms
client-10.9.0.5:PES1UG20CS243:Mahika:/
#>■
```

**Ping is successful and the Client (host U) can communicate with the VPN server.**

- VPN Server (server-router) can communicate with Host V (host-192.168.60.5)

On server-router

**Command :**

**# ping 192.168.60.5**

```
server-router:PES1UG20CS243:Mahika:/
#>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=0.148 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.151 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=64 time=0.234 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=64 time=0.126 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=64 time=0.090 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=64 time=0.047 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=64 time=0.149 ms
64 bytes from 192.168.60.5: icmp_seq=8 ttl=64 time=0.148 ms
64 bytes from 192.168.60.5: icmp_seq=9 ttl=64 time=0.152 ms
64 bytes from 192.168.60.5: icmp_seq=10 ttl=64 time=0.140 ms
64 bytes from 192.168.60.5: icmp_seq=11 ttl=64 time=0.150 ms
64 bytes from 192.168.60.5: icmp_seq=12 ttl=64 time=0.149 ms
64 bytes from 192.168.60.5: icmp_seq=13 ttl=64 time=0.150 ms
64 bytes from 192.168.60.5: icmp_seq=14 ttl=64 time=0.101 ms
64 bytes from 192.168.60.5: icmp_seq=15 ttl=64 time=0.151 ms
^C
--- 192.168.60.5 ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 14325ms
rtt min/avg/max/mdev = 0.047/0.139/0.234/0.038 ms
server-router:PES1UG20CS243:Mahika:/
```

**Ping is successful, hence we conclude that VPN server can communicate with with host V in private network**

- Host U (Client - 10.9.0.5) should not be able to communicate with Host V (host 192.168.60.5) On Client 10.9.0.5

**Command:**

```
# ping 192.168.60.5  
client-10.9.0.5:PES1UG20CS243:Mahika:/  
#>ping 192.168.60.5  
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.  
■
```

**The Ping is not successful hence we see that the client can not communicate with host V. This is because the VPN tunnel has not yet been set up.**

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

On Client - 10.9.0.5

**Command:**

```
# ping server-router
```



```

client-10.9.0.5:PES1UG20CS243:Mahika:/
#>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=0.143 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=2 ttl=64 time=0.205 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=3 ttl=64 time=0.314 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=4 ttl=64 time=0.189 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=5 ttl=64 time=0.125 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=6 ttl=64 time=0.075 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=7 ttl=64 time=0.125 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=8 ttl=64 time=0.073 ms

```

The router is sent ping requests.

On server-router run -

**Command:**

**# tcpdump -i eth0 -n**

```

server-router:PES1UG20CS243:Mahika:/
#>tcpdump -i eth0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
06:49:26.624515 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 31, seq 1, length 64
06:49:26.624591 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 31, seq 1, length 64
06:49:27.647241 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 31, seq 2, length 64
06:49:27.647314 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 31, seq 2, length 64
06:49:28.670632 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 31, seq 3, length 64
06:49:28.670715 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 31, seq 3, length 64
06:49:29.694294 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 31, seq 4, length 64
06:49:29.694351 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 31, seq 4, length 64
06:49:30.718493 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 31, seq 5, length 64
06:49:30.718524 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 31, seq 5, length 64

```

On using tcpdump, we can capture the icmp request packets sent from client(10.9.0.5) to VPN server(10.9.0.11) and the icmp replies from server to client.



## Task 2: Create and Configure TUN Interface

The VPN tunnel that we are going to build is based on the TUN/TAP technologies. TUN and TAP are virtual network kernel drivers; they implement network devices that are supported entirely in software. TUN (as in network TUNnel) simulates a network layer device and it operates with layer-3 packets such as IP packets. With TUN/TAP, we can create virtual network interfaces.

A user-space program is usually attached to the TUN virtual network interface. Packets sent by an operating system via a TUN network interface are delivered to the user-space program. On the other hand, packets sent by the program via a TUN network interface are injected into the operating system network stack. To the operating system, it appears that the packets come from an external source through the virtual network interface

The objective of this task is to get familiar with the TUN technology. We will conduct several experiments to learn the technical details of the TUN interface. We will use the following Python program as the basis for the experiments, and we will modify this base code throughout this lab. The code is already included in the volumes folder in the zip file.

## Task 2.a: Name of the Interface

We will run the tun.py program on Host U (Client-10.9.0.5). Make the above tun.py program executable, and run it using the root privilege.

On Client - 10.9.0.5 (change directory to ./volumes)

**Command:**

```
# chmod a+x tun.py
```

```
# ./tun.py &
```

```
# ip addr
```

Take screenshots and explain your understanding.

You should be able to find an interface called **tun0**.

---

```
client-10.9.0.5:PES1UG20CS243:Mahika:/
#>cd volumes
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>chmod a+x tun.py
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>./tun.py &
[1] 34
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>Interface Name: tun0
^C
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>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
6: eth0@if7: <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-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>█
```

**On running the program tun.py, we have set up the tun interface on the system. On using ip addr command, we can see the tun0 interface. It has not yet been assigned an IP address.**

Kill the earlier Tunnel Process -

On Client - 10.9.0.5

**Command:**

**# kill %1**

**(In case you get an error, run**

**# jobs**

**Note down the number for ./tun.py, then run**

**# kill %[the number] )**

```
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>kill %1
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
```

Your job in this task is to **change the tun.py program**, so instead of using tun as the prefix of the interface name, use the last 5 digits of your SRN as the prefix. **Please show your results with appropriate screenshots.**

In **tun.py** replace “tun” with the last 5 characters of your SRN in line 16.

Line 16 - `ifr = struct.pack('16sH', b'tun%d', IFF_TUN | IFF_NO_PI)`

```
13
14 # Create the tun interface
15 tun = os.open("/dev/net/tun", os.O_RDWR)
16 ifr = struct.pack('16sH', b'CS243%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
```

Now run the following commands again and see the new tunnel interface, it should be “SRN0”

On Client - 10.9.0.5

**Command:**

```
# chmod a+x tun.py
```

```
# ./tun.py &
```

```
# ip addr
```

```
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>chmod a+x tun.py
[1]+  Terminated                  ./tun.py
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>./tun.py &
[1] 41
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>Interface Name: CS2430
#>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: CS2430: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
6: eth0@if7: <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-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>█
```

**We can see the tun interface with the name CS2430 being set up.**

## Task 2.b: Set up the TUN Interface

At this point, the TUN interface is not usable, because it has not been configured yet. There are two things that we need to do before the interface can be used. First, we need to assign an IP address to it. Second, we need to bring up the interface, because the interface is still in the down state. We can use the following two commands for the configuration:

On Client - 10.9.0.5

**Command:**

```
# ip addr add 192.168.53.99/24 dev <SRN>0
```

```
# ip link set dev <SRN>0 up
```

**Replace <SRN> with what you have used to change link 16 in tun.py**

```
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>ip addr add 192.168.53.99/24 dev CS2430
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>ip link set dev CS2430 up
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>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: CS2430: <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 CS2430
        valid_lft forever preferred_lft forever
6: eth0@if7: <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-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#> █
```

**IP address of 192.168.53.99 is assigned to the tun interface, which is visible on using ip addr command.**



## Task 2.c: Read from the TUN Interface

In this task, we will read from the TUN interface. Whatever coming out from the TUN interface is an IP packet. We can cast the data received from the interface into a Scapy IP object, so we can print out each field of the IP packet.

Please use the following while loop to replace the **one in tun.py**:

Replace the following in tun.py -

while True:

time.sleep(10)

**With -**

while True:

# Get a packet from the tun interface

packet = os.read(tun, 2048)

if packet:

ip = IP(packet)

print(ip.summary())

# 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())

**This is done so that the fields of the IP packet sent is displayed, so that we know which interface the packet is constructed and sent out with, and which IP address it is sent to.**

Kill the earlier Tunnel Process -

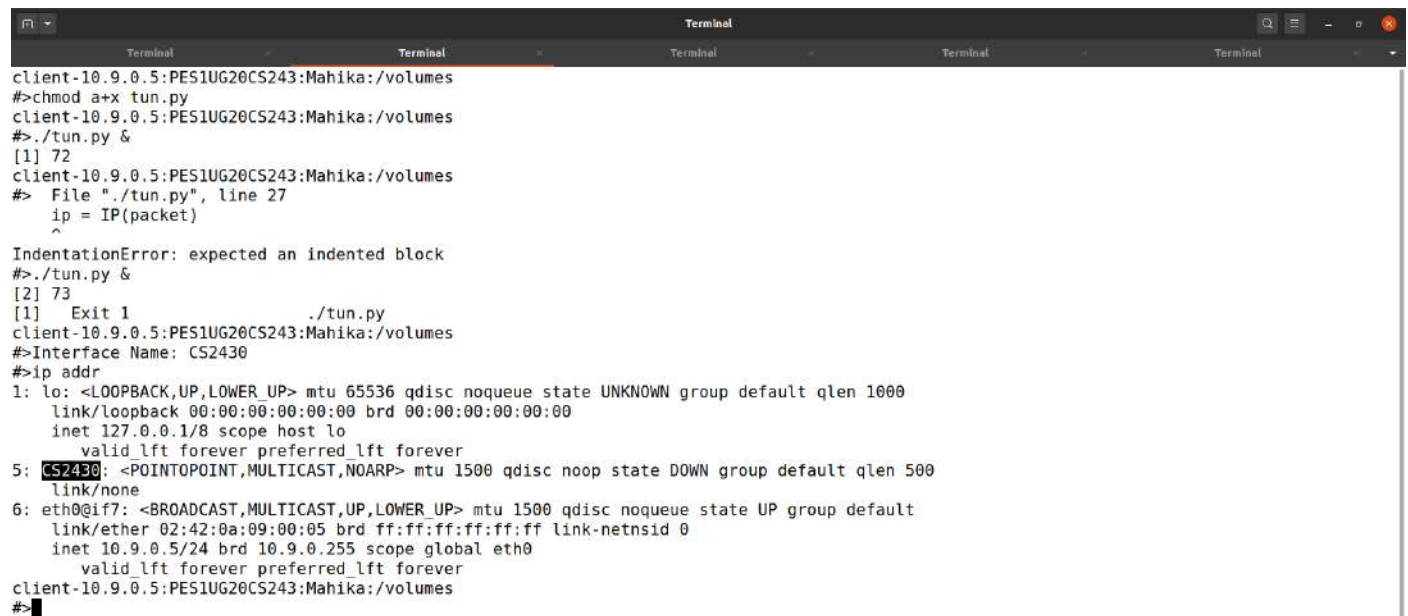
On Client - 10.9.0.5

**Command:**

**# kill %1**

```
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>kill %1
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
```

Set up the TUN Interface -



```
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>chmod a+x tun.py
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>./tun.py &
[1] 72
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#> File "./tun.py", line 27
    ip = IP(packet)
    ^
IndentationError: expected an indented block
#>./tun.py &
[2] 73
[1] Exit 1
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>Interface Name: CS2430
#>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
5: CS2430: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
6: eth0@if7: <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-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>
```

The interface is set up again, after it was killed.

On Client - 10.9.0.5

**Command:**

```
# ip addr add 192.168.53.99/24 dev <SRN>0
```

```
# ip link set dev <SRN>0 up
```

**Replace <SRN> with what you have used to change link 16 in tun.py**

```
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>ip addr add 192.168.53.99/24 dev CS2430
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>ip link set dev CS2430 up
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>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
4: CS2430: <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 CS2430
        valid_lft forever preferred_lft forever
6: eth0@if7: <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
```

**IP address is assigned to the tun interface.**

Run the revised tun.py program -

On Client - 10.9.0.5

**Command:**

```
# ./tun.py &
```

```
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>chmod a+x tun.py
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>./tun.py &
[1] 56
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>Interface Name: CS2430
```

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

On Client - 10.9.0.5

**Command:**

```
# ping 192.168.53.5
```

```

client-10.9.0.5:PES1UG20CS243:Mahika:/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
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 ---
14 packets transmitted, 0 received, 100% packet loss, time 13308ms

client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>

```

A host on the 192.168.53.0/24 network is pinged. As the host is in the same LAN, it is reachable and hence the packets get sent. However, the interface is not set up to receive replies yet, and hence no reply packets are received.

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

On Client - 10.9.0.5

**Command:**

**# ping 192.168.60.5**

Provide appropriate screenshots along with your explanations.

```

client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

```

On ping to a host on the private network 192.168.60.0/24, the ping does not go through as the tun interface has not been configured yet to send packets to a host outside the network. It is unreachable.

Kill the earlier Tunnel Process -

On Client - 10.9.0.5

**Command:**

# kill %1

```
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>kill %1
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
```

## Task 2.d: Write to the TUN Interface

In this task, we will write to the TUN interface. Since this is a virtual network interface, whatever is written to the interface by the application will appear in the kernel as an IP packet.

We will modify the tun.py program, so after getting a packet from the TUN interface, we construct a new packet based on the received packet. We then write the new packet to the TUN interface.

Your job in this task is to **change the tun1.py program**, so instead of using tun as the prefix of the interface name, use the last 5 digits of your SRN as the prefix. **Please show your results with appropriate screenshots.**

In **tun1.py** replace “tun” with the last 5 characters of your SRN in line 16.

Line 16 - `ifr = struct.pack('16sH', b'tun%d', IFF_TUN | IFF_NO_PI)`

3

4 # Create the tun interface

5 `tun = os.open("/dev/net/tun", os.O_RDWR)`

6 `ifr = struct.pack('16sH', b'CS243%d', IFF_TUN | IFF_NO_PI)`

7 `ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)`

8



On Client - 10.9.0.5

**Command:**

**# chmod a+x tun.py**

**# ./tun1.py &**

**# ip addr**

```
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>chmod a+x tun1.py
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>./tun1.py &
[1] 120
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>Interface Name: CS2430
#>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
6: eth0@if7: <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: CS2430: <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 CS2430
        valid_lft forever preferred_lft forever
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>█
```

**The tun interface is set up and now we will be able to write to the tun interface.**

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

On Client - 10.9.0.5

**Command:**

**# ping 192.168.53.5**

Take appropriate screenshots of each step with the required observations.

```
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
CS2430: 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=2.97 ms
CS2430: 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=4.73 ms
CS2430: 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=5.92 ms
CS2430: 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=5.65 ms
CS2430: 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=4.84 ms
CS2430: 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=5.41 ms
CS2430: 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=5.63 ms
CS2430: 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=5.80 ms
CS2430: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
64 bytes from 192.168.53.5: icmp_seq=9 ttl=99 time=5.22 ms
CS2430: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
64 bytes from 192.168.53.5: icmp_seq=10 ttl=99 time=3.16 ms
CS2430: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
64 bytes from 192.168.53.5: icmp_seq=11 ttl=99 time=5.75 ms
^C
--- 192.168.53.5 ping statistics ---
11 packets transmitted, 11 received, 0% packet loss, time 10324ms
rtt min/avg/max/mdev = 2.969/5.007/5.924/0.986 ms
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#~
```

**Here we can see that the ping is successful. ICMP requests are sent and also received (which means they are constructed by the tun interface and written to the tun interface).**

Kill the earlier Tunnel Process -

On Client - 10.9.0.5

**Command:**

**# kill %1**

```
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>kill %1
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
```

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

In this task, we will put the IP packet received from the TUN interface into the UDP payload field of a new IP packet, and send it to another computer. Namely, we place the original packet inside a new packet. This is called IP tunneling. The tunnel implementation is just standard client/server programming. It can be built on top of TCP or UDP. In this task, we will use UDP. Namely, we put an IP packet inside the payload field of a UDP packet.

**The server program `tun_server.py`** - We will run the `tun_server.py` program on VPN Server. This program is just a standard UDP server program. It listens to port 9090 and prints out whatever is received. The program assumes that the data in the UDP payload field is an IP packet, so it casts the payload to a Scapy IP object, and prints out the source and destination IP address of the enclosed IP packet.

On server-router run (**change directory to `./volumes`**)

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

    # Send the packet via the tunnel
    sock.sendto(packet, ("10.9.0.11", 9090))
```

We see that the destination IP address (of the VPN server) is 10.9.0.11 and it is listening on port 9090

**Command:**

```
# chmod a+x tun_server.py
```

```
# ./tun_server.py
```

```
#>chmod a+x tun_server.py
server-router:PES1UG20CS243:Mahika:/volumes
#>./tun_server.py
```

**Implement the client program tun\_client.py** - First, we need to modify the TUN program tun.py. Let's rename it, and call it tun\_client.py. Sending data to another computer using UDP can be done using the standard socket programming. Replace the while loop in the program with the following: The SERVER IP and SERVER PORT should be replaced with the actual IP address and port number of the server program running on VPN Server

Note - In **tun\_client.py** replace "tun" with the last 5 characters of your SRN in line 16.

```
# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'CS243%d', IFF_TUN | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
```

Run tun\_client.py on Client - 10.9.0.5

**Command:**

```
# chmod a+x tun_client.py
```

```
# ./tun_client.py &
```

```
# ip addr
```



```

client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>chmod a+x tun_client.py
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>./tun_client.py &
[1] 135
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>Interface Name: CS2430
#>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
6: eth0@if7: <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
11: CS2430: <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 CS2430
        valid_lft forever preferred_lft forever
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
.. █

```

**The interface is added and IP address is assigned. This program will construct and send the real packet as payload to the outer UDP packet.**

To test whether the tunnel works or not, ping any IP address belonging to the 192.168.53.0/24 network. What is printed out on VPN Server? Why?

On Client - 10.9.0.5

**Command:**

**# ping 192.168.53.5**

```

client-10.9.0.5:PES1UG20CS243:Mahika:/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 ---
6 packets transmitted, 0 received, 100% packet loss, time 5115ms

client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>█

```

**Ping requests are sent successfully but replies are not received. This is because the interface has not been configured yet on the receiving end.**



Let us ping Host V, and see whether the ICMP packet is sent to VPN Server through the tunnel.

On Client - 10.9.0.5

**Command:**

**# ping 192.168.60.5**

```
client-10.9.0.5:PES1UG20CS243:Mahika:/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
^C
--- 192.168.60.5 ping statistics ---
7 packets transmitted, 0 received, 100% packet loss, time 6144ms

client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
```

Similarly, icmp requests are sent, but no replies are received for the same reason.





This is done through routing, i.e., packets going to the 192.168.60.0/24 network should be routed to the TUN interface and be given to the tun\_client.py program. The routing has already been added to the tun\_client.py program.

To check the routing run the following command on Client - 10.9.0.5

**Command:**

```
# ip route  
client-10.9.0.5:PES1UG20CS243:Mahika:/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 CS2430 proto kernel scope link src 192.168.53.99  
192.168.60.0/24 dev CS2430 scope link  
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes  
#> █
```

**Through this command we can see that the default packet is routed to eth0 interface. However, when the destination is in 192.168.53.0/24 or 192.168.60.0/24 network, the packet should be routed through tun interface with ip address 192.168.53.99.**

## Task 4: Set Up the VPN Server

After `tun_server.py` gets a packet from the tunnel, it needs to feed the packet to the kernel, so the kernel can route the packet towards its final destination. This needs to be done through a TUN interface, just like what we did in Task 2. We have modified `tun_server.py`, so it can do the following:

- Create a TUN interface and configure it.
- Get the data from the socket interface; treat the received data as an IP packet.
- Write the packet to the TUN interface.

Note - In `tun_server1.py` replace “tun” with the last 5 characters of your SRN in line 18.

```
# Create a tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'CS243%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))
```

On server-router run

**Command:**

```
# chmod a+x tun_server1.py
```

```
# ./tun_server1.py
```

```
server-router:PES1UG20CS243:Mahika:/volumes
#>chmod a+x tun_server1.py
server-router:PES1UG20CS243:Mahika:/volumes
#>./tun_server1.py
Interface Name: CS2430
10.9.0.5:37382 --> 0.0.0.0:9090
    Inside: 192.168.53.99 --> 192.168.60.5
```

**This program sets up the tun interface on the server end.**



On host-192.168.60.5 run -

**Command:**

**# tcpdump -i eth0 -n**

```
host-192.168.60.5:PES1UG20CS243:Mahika:/  
#>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
```

**Host V is listening on eth0 interface.**

On Client - 10.9.0.5 run -

**Command:**

**# ping 192.168.60.5**

```
client-10.9.0.5:PES1UG20CS243:Mahika:/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  
^C  
--- 192.168.60.5 ping statistics ---  
11 packets transmitted, 0 received, 100% packet loss, time 10238ms  
  
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes  
#>█
```



**Ping request sent to host V from client (host U). It does not receive a reply as the receiver end has not been configured to send replies yet.**

Note - If you haven't set up the tun\_client (client side tunnel) then please do the same, by following the previous task. (Running tun\_client.py)

If everything is set up properly, we can ping Host V (192.168.60.5) from Host U (10.9.0.5). The ICMP echo request packets should eventually arrive at Host V through the tunnel. Please show your proof. It should be noted that although Host V will respond to the ICMP packets, the reply will not get back to Host U, because we have not set up everything yet. Therefore, for this task, it is sufficient to show (tcpdump) that the ICMP packets have arrived at Host V.

Take screenshots of all the terminals and explain your observation.

```
host-192.168.60.5:PE51UG20CS243:Mahika:/
#>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
13:16:44.928976 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
13:16:44.929115 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
13:17:01.285377 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 211, seq 1, length 64
13:17:01.285413 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 211, seq 1, length 64
13:17:02.305003 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 211, seq 2, length 64
13:17:02.305026 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 211, seq 2, length 64
13:17:03.332461 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 211, seq 3, length 64
13:17:03.332581 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 211, seq 3, length 64
13:17:04.366148 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 211, seq 4, length 64
13:17:04.366203 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 211, seq 4, length 64
13:17:05.384327 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 211, seq 5, length 64
13:17:05.384344 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 211, seq 5, length 64
13:17:06.401136 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 211, seq 6, length 64
13:17:06.401153 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 211, seq 6, length 64
13:17:06.432152 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
13:17:06.432182 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
13:17:07.428712 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 211, seq 7, length 64
13:17:07.428871 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 211, seq 7, length 64
13:17:07.968536 IP 192.168.60.5.23 > 192.168.53.99.51058: Flags [P.U], seq 42341123:42341150, ack 332099
p,TS val 2167870489 ecr 2886087253], length 27 [telnet DMARK]
13:17:08.452944 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 211, seq 8, length 64
13:17:08.453065 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 211, seq 8, length 64
13:17:09.476843 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 211, seq 9, length 64
13:17:09.476908 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 211, seq 9, length 64
13:17:10.499696 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 211, seq 10, length 64
13:17:10.499852 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 211, seq 10, length 64
13:17:11.525000 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 211, seq 11, length 64
```

**On the tcpdump output we can see that ICMP packets are being received by host V from 192.168.53.99 ( which is the src ip address of tun interface on host U). ICMP reply is also sent from host V to the src address of the icmp request packet. From this we can see that the tunnel is successful.**



## Task 5: Handling Traffic in Both Directions

After getting to this point, one direction of your tunnel is complete, i.e., we can send packets from Host U to Host V via the tunnel. If we look at the Wireshark trace on Host V, we can see that Host V has sent out the response, but the packet gets dropped somewhere. This is because our tunnel is only one directional; we need to set up its other direction, so returning traffic can be tunneled back to Host U.

To achieve that, our TUN client and server programs need to read data from two interfaces, the TUN interface and the socket interface. All these interfaces are represented by file descriptors, so we need to monitor them to see whether there is data coming from them. One way to do that is to keep polling them, and see whether there is data on each of the interfaces. The performance of this approach is undesirable, because the process has to keep running in an idle loop when there is no data. Another way is to read from an interface. By default, read is blocking, i.e. The process will be suspended if there is no data. When data becomes available, the process will be unblocked, and its execution will continue. This way, it does not waste CPU time when there is no data.

We use two new programs **tun\_client\_select.py** and **tun\_server\_select.py**. In **both the mentioned programs** replace “tun” with the last 5 characters of your SRN in line 15 (client) and line 19 (server).

The line is - “ifr = struct.pack('16sH', b'tun%d', IFF\_TUN | IFF\_NO\_PI)”

```
# Create a tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'CS243%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
```

Open **two terminals of the Client - 10.9.0.5** Machine, this improves comprehensibility for what we are about to execute.



You will **need wireshark** for this task, capturing the packets on the client interface.

On the server-router run -

**Command:**

```
# chmod a+x tun_server_select.py
# ./tun_server_select.py
```

---

```
server-router:PES1UG20CS243:Mahika:/volumes
#>chmod a+x tun_server_select.py
server-router:PES1UG20CS243:Mahika:/volumes
#>./tun_server_select.py
Interface Name: CS2430
```

On one terminal of Client - 10.9.0.5 run -

**Command:**

```
# chmod a+x tun_client_select.py
# ./tun_client_select.py
```

---

```
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>chmod a+x tun_client_select.py
client-10.9.0.5:PES1UG20CS243:Mahika:/volumes
#>./tun_client_select.py
Interface Name: CS2430
```

On the other terminal of Client - 10.9.0.5 run -

**Command:**

```
# ping 192.168.60.5
```

```
client-10.9.0.5:PES1UG20CS243:Mahika:/
#>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=6.55 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=2.69 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=3.20 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=4.05 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=1.62 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=63 time=3.29 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=2.15 ms
64 bytes from 192.168.60.5: icmp_seq=8 ttl=63 time=4.87 ms
64 bytes from 192.168.60.5: icmp_seq=9 ttl=63 time=2.35 ms
^C
--- 192.168.60.5 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8014ms
rtt min/avg/max/mdev = 1.622/3.419/6.551/1.449 ms
client-10.9.0.5:PES1UG20CS243:Mahika:/
#>■
```

**We can see that the ping is successful. This is because now both ends of the connection have been set up, and the tunnel is working.**



1

```

server-router:PE51UG20CS243:Mahika:/volumes
#> ./tun_server_select.py
Interface Name: CS2430
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 terminal, we see the path of the packets sent. First the icmp request packet is received from socket with src ip as 192.18.53.99 and destination as 192.18.60.5. Then the reply packet with src as host V and dest as 192.18.53.99 (tun interface on client) is sent through the tun interface

22	2022-11-14 11:0...	10.9.0.5	10.9.0.11	UDP	128 47247 → 9090 Len=84	
23	2022-11-14 11:0...	10.9.0.5	10.9.0.11	UDP	128 47247 → 9090 Len=84	
30	2022-11-14 11:0...	192.168.53.99	192.168.60.5	ICMP	100 Echo (ping) request	id=0x0026, seq=1/256, ttl=63 (no respons...
31	2022-11-14 11:0...	192.168.53.99	192.168.60.5	ICMP	100 Echo (ping) request	id=0x0026, seq=1/256, ttl=63 (reply in 3...
32	2022-11-14 11:0...	192.168.60.5	192.168.53.99	ICMP	100 Echo (ping) reply	id=0x0026, seq=1/256, ttl=64 (request in...
33	2022-11-14 11:0...	192.168.60.5	192.168.53.99	ICMP	100 Echo (ping) reply	id=0x0026, seq=1/256, ttl=64
34	2022-11-14 11:0...	10.9.0.11	10.9.0.5	UDP	128 9090 → 47247 Len=84	
35	2022-11-14 11:0...	10.9.0.11	10.9.0.5	UDP	128 9090 → 47247 Len=84	
36	2022-11-14 11:0...	10.9.0.5	10.9.0.11	UDP	128 47247 → 9090 Len=84	
37	2022-11-14 11:0...	10.9.0.5	10.9.0.11	UDP	128 47247 → 9090 Len=84	
38	2022-11-14 11:0...	192.168.53.99	192.168.60.5	ICMP	100 Echo (ping) request	id=0x0026, seq=2/512, ttl=63 (no respons...
39	2022-11-14 11:0...	192.168.53.99	192.168.60.5	ICMP	100 Echo (ping) request	id=0x0026, seq=2/512, ttl=63 (reply in 4...
40	2022-11-14 11:0...	192.168.60.5	192.168.53.99	ICMP	100 Echo (ping) reply	id=0x0026, seq=2/512, ttl=64 (request in...
41	2022-11-14 11:0...	192.168.60.5	192.168.53.99	ICMP	100 Echo (ping) reply	id=0x0026, seq=2/512, ttl=64
42	2022-11-14 11:0...	10.9.0.11	10.9.0.5	UDP	128 9090 → 47247 Len=84	
43	2022-11-14 11:0...	10.9.0.11	10.9.0.5	UDP	128 9090 → 47247 Len=84	
44	2022-11-14 11:0...	10.9.0.5	10.9.0.11	UDP	128 47247 → 9090 Len=84	
45	2022-11-14 11:0...	10.9.0.5	10.9.0.11	UDP	128 47247 → 9090 Len=84	
46	2022-11-14 11:0...	192.168.53.99	192.168.60.5	ICMP	100 Echo (ping) request	id=0x0026, seq=3/768, ttl=63 (no respons...
47	2022-11-14 11:0...	192.168.53.99	192.168.60.5	ICMP	100 Echo (ping) request	id=0x0026, seq=3/768, ttl=63 (reply in 4...
48	2022-11-14 11:0...	192.168.60.5	192.168.53.99	ICMP	100 Echo (ping) reply	id=0x0026, seq=3/768, ttl=64 (request in...
49	2022-11-14 11:0...	192.168.60.5	192.168.53.99	ICMP	100 Echo (ping) reply	id=0x0026, seq=3/768, ttl=64
50	2022-11-14 11:0...	10.9.0.11	10.9.0.5	UDP	128 9090 → 47247 Len=84	
51	2022-11-14 11:0...	10.9.0.11	10.9.0.5	UDP	128 9090 → 47247 Len=84	
52	2022-11-14 11:0...	10.9.0.5	10.9.0.11	UDP	128 47247 → 9090 Len=84	
53	2022-11-14 11:0...	10.9.0.5	10.9.0.11	UDP	128 47247 → 9090 Len=84	
54	2022-11-14 11:0...	192.168.53.99	192.168.60.5	ICMP	100 Echo (ping) request	id=0x0026, seq=4/1024, ttl=63 (no respon...
55	2022-11-14 11:0...	192.168.53.99	192.168.60.5	ICMP	100 Echo (ping) request	id=0x0026, seq=4/1024, ttl=63 (reply in ...
56	2022-11-14 11:0...	192.168.60.5	192.168.53.99	ICMP	100 Echo (ping) reply	id=0x0026, seq=4/1024, ttl=64 (request i...
57	2022-11-14 11:0...	192.168.60.5	192.168.53.99	ICMP	100 Echo (ping) reply	id=0x0026, seq=4/1024, ttl=64
58	2022-11-14 11:0...	10.9.0.11	10.9.0.5	UDP	128 9090 → 47247 Len=84	
59	2022-11-14 11:0...	10.9.0.11	10.9.0.5	UDP	128 9090 → 47247 Len=84	
60	2022-11-14 11:0...	10.9.0.5	10.9.0.11	UDP	128 47247 → 9090 Len=84	

The Wireshark output shows the outer UDP packet sent from host U to VPN server, then the inner ICMP request packet from tun interface of client to host V, then the inner ICMP reply packet from Host V to client's tun interface, and the outer UDP packet from the VPN server to host U.

The source IP of the ICMP request packet is the IP address of the tun interface as the packet is constructed at the tun interface.

Now we Telnet into 192.168.60.5 - (same terminal as the one who've pinged from)

**Command:**

**# telnet 192.168.60.5**

```
client-10.9.0.5:PES1UG20CS243:Mahika:/
#>telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
534316c6eb2a 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 14 12:45:21 UTC 2022 on pts/3
seed@534316c6eb2a:~$ █
```

**Telnet connection is also successful.**



```
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 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
```

On the client end we see that from tun interface, we sent a tcp packet with src ip as tun interface ip and dest as host V. From socket we receive the tcp packet with src as host V and dest as the clients tun interface.

```

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

```

On the server terminal, we see the path of the packets sent. First, the tcp packet is received from socket with src ip as 192.18.53.99 and destination as 192.18.60.5. Then the tcp packet with src as host V and dest as 192.18.53.99 (tun interface on client) is sent through the tun interface.

No.	Time	Source	Destination	Protocol	Length	Info
1	2022-11-14 11:00:00.000000	10.9.0.5	10.9.0.11	UDP	104	47247 → 9090 Len=60
2	2022-11-14 11:00:00.000000	10.9.0.5	10.9.0.11	UDP	104	47247 → 9090 Len=60
3	2022-11-14 11:00:00.000000	192.168.53.99	192.168.60.5	TCP	76	41738 → 23 [SYN] Seq=3474863734 Win=64240 Len=0 MSS=1460 SACK_
4	2022-11-14 11:00:00.000000	192.168.53.99	192.168.60.5	TCP	76	[TCP Out-Of-Order] 41738 → 23 [SYN] Seq=3474863734 Win=64240
5	2022-11-14 11:00:00.000000	192.168.60.5	192.168.53.99	TCP	76	23 → 41738 [SYN, ACK] Seq=205107056 Ack=3474863735 Win=65160
6	2022-11-14 11:00:00.000000	192.168.60.5	192.168.53.99	TCP	76	[TCP Out-Of-Order] 23 → 41738 [SYN, ACK] Seq=205107056 Ack=34
7	2022-11-14 11:00:00.000000	10.9.0.11	10.9.0.5	UDP	104	9090 → 47247 Len=60
8	2022-11-14 11:00:00.000000	10.9.0.11	10.9.0.5	UDP	104	9090 → 47247 Len=60
9	2022-11-14 11:00:00.000000	10.9.0.5	10.9.0.11	UDP	96	47247 → 9090 Len=52
10	2022-11-14 11:00:00.000000	10.9.0.5	10.9.0.11	UDP	96	47247 → 9090 Len=52
11	2022-11-14 11:00:00.000000	192.168.53.99	192.168.60.5	TCP	68	41738 → 23 [ACK] Seq=3474863735 Ack=205107057 Win=64256 Len=0
12	2022-11-14 11:00:00.000000	192.168.53.99	192.168.60.5	TCP	68	[TCP Dup ACK 11#1] 41738 → 23 [ACK] Seq=3474863735 Ack=205107
13	2022-11-14 11:00:00.000000	10.9.0.5	10.9.0.11	UDP	120	47247 → 9090 Len=76
14	2022-11-14 11:00:00.000000	10.9.0.5	10.9.0.11	UDP	120	47247 → 9090 Len=76
15	2022-11-14 11:00:00.000000	192.168.53.99	192.168.60.5	TELNET	92	Telnet Data ...
16	2022-11-14 11:00:00.000000	192.168.53.99	192.168.60.5	TCP	92	[TCP Retransmission] 41738 → 23 [PSH, ACK] Seq=3474863735 Ack
17	2022-11-14 11:00:00.000000	192.168.60.5	192.168.53.99	TCP	68	23 → 41738 [ACK] Seq=205107057 Ack=3474863759 Win=65152 Len=0
18	2022-11-14 11:00:00.000000	192.168.60.5	192.168.53.99	TCP	68	[TCP Dup ACK 17#1] 23 → 41738 [ACK] Seq=205107057 Ack=3474863
19	2022-11-14 11:00:00.000000	10.9.0.5	10.9.0.11	UDP	96	9090 → 47247 Len=52
20	2022-11-14 11:00:00.000000	10.9.0.11	10.9.0.5	UDP	96	9090 → 47247 Len=52
21	2022-11-14 11:00:00.000000	192.168.60.5	192.168.0.1	DNS	88	Standard query 0xd2ec PTR 99.53.168.192.in-addr.arpa
22	2022-11-14 11:00:00.000000	192.168.60.5	192.168.0.1	DNS	88	Standard query 0xd2ec PTR 99.53.168.192.in-addr.arpa
23	2022-11-14 11:00:00.000000	02:42:0a:99:00:0b	02:42:0a:99:00:0b	ARP	44	Who has 10.9.0.1? Tell 10.9.0.11
24	2022-11-14 11:00:00.000000	02:42:0a:99:00:0b	02:42:0a:99:00:0b	ARP	44	Who has 10.9.0.1? Tell 10.9.0.11
25	2022-11-14 11:00:00.000000	02:42:0a:99:00:0b	02:42:0a:99:00:0b	ARP	44	Who has 10.9.0.1? Tell 10.9.0.11
26	2022-11-14 11:00:00.000000	02:42:0a:99:00:0b	02:42:0a:99:00:0b	ARP	44	Who has 10.9.0.1? Tell 10.9.0.11

\* Frame 1: 104 bytes on wire (832 bits), 104 bytes captured (832 bits) on interface any, id 0  
 \* Linux cooked capture  
 \* Internet Protocol Version 4, Src: 10.9.0.5, Dst: 10.9.0.11  
 \* User Datagram Protocol, Src Port: 47247, Dst Port: 9090  
 \* Data (60 bytes)



On Wireshark we see, that the outer UDP packet is sent from host U to VPN server, then the inner telnet/TCP from tun interface of client to host V, then the telnet/TCP packet from Host V to client's tun interface, and the outer UDP packet from the VPN server to host U.

## Task 6: Tunnel-Breaking Experiment

On Host U (10.9.0.5), telnet to Host V (192.168.60.5).

(Perform Task 5 again, in case you have closed both the Client and Server Tunnel Connections)

On Client - 10.9.0.5

**Command:**

**# telnet 192.168.60.5**

```
client-10.9.0.5:~$ telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
534316c6eb2a 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 14 16:06:27 UTC 2022 on pts/2
```

**Telnet connection is successful.**

**On client end:**

```
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
```

On server end:

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

- Log on and while keeping the telnet connection alive, we break the VPN tunnel by stopping the tun\_server\_select.py program - **Ctrl + C in server-router**

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

The connection is interrupted from the server end.

We then type something in the telnet window.

Do you see what you type? What happens to the TCP connection? Is the connection broken? Explain.

**On trying to type, no text is visible on the telnet window. This means that the connection has been lost on one end.**

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

**Data was sent, but not received as connection is broken on receiving end.**

Let us now reconnect the VPN tunnel (do not wait for too long).

**On reconnection:**

On the server-router run -

**Command:**

```
# ./tun_server_select.py
```



```

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

server-router:PES1UG20CS243:Mahika:/volumes
#>./tun_server_select.py
Interface Name: CS2430
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

```

We can see that the connection is re-established and data is being sent and received successfully.

Once the tunnel is re-established, what is going to happen to the telnet connection? Please describe and explain your observations.

**On client end: the connection is back to normal, data is being sent and received.**

```

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

```

**On the telnet window, we are able to see the text that we typed earlier which was not visible.**

```
client-10.9.0.5:PES1UG20CS243:Mahika:/
#>telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
534316c6eb2a 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 14 16:06:27 UTC 2022 on pts/2
seed@534316c6eb2a:~$ trying to type somethnghi im trying to type█
```

The text which wasnt visible earlier is visible once the connection is re-established.