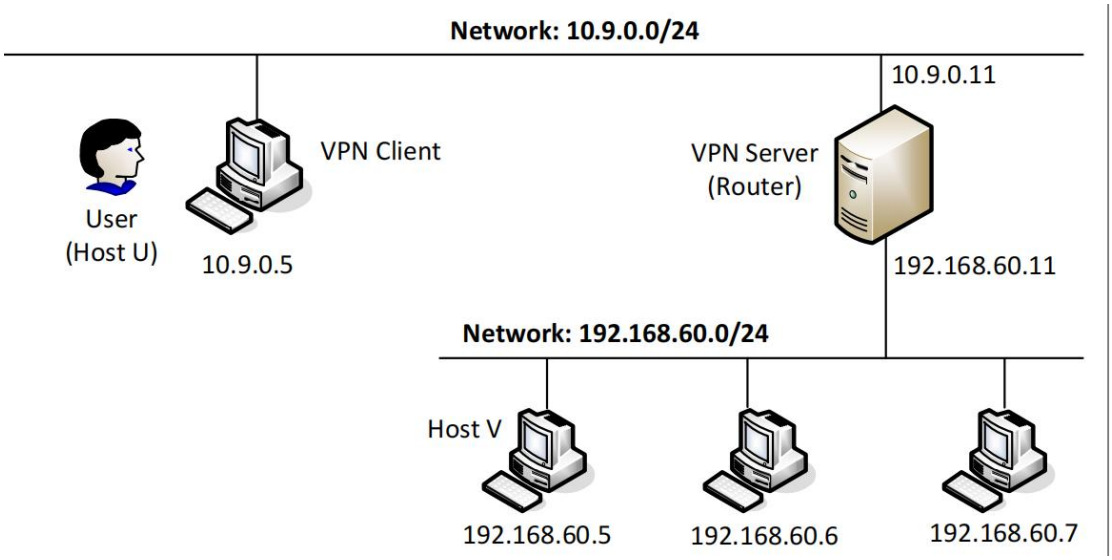
VPN_Tunnel

Wang,Xiao
Xwang99@syr.edu

Task 1: Network Setup

0. Network structure



1. Host U can communicate with VPN Server

$ ping 10.9.0.11: shows 0 loss, which means U can communicate with VPN Server

```
root@02fb7fada307:/# ping 10.9.0.11
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
64 bytes from 10.9.0.11: icmp_seq=1 ttl=64 time=0.089 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.043 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.037 ms
64 bytes from 10.9.0.11: icmp_seq=4 ttl=64 time=0.042 ms
^C
--- 10.9.0.11 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3068ms
rtt min/avg/max/mdev = 0.037/0.052/0.089/0.021 ms
root@02fb7fada307:/#
```

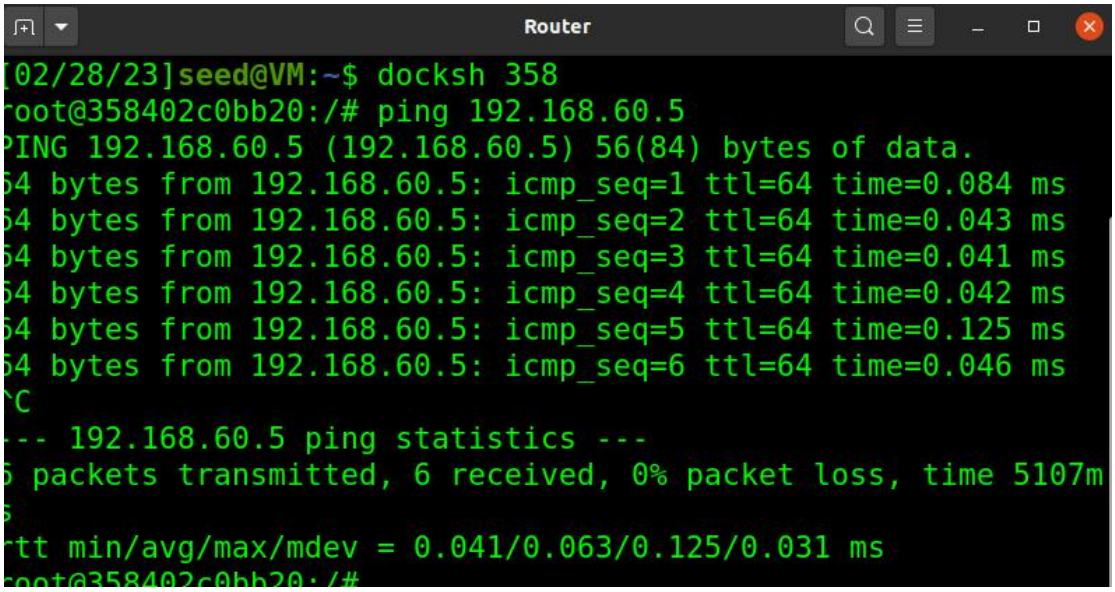
2. Host U should not be able to communicate with Host V

$ ping 192.168.60.5: shows 100% loss, which means U cannot communicate with V

```
root@02fb7fada307:/# 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 ---
5 packets transmitted, 0 received, 100% packet loss, time 4088ms

root@02fb7fada307:/#
```

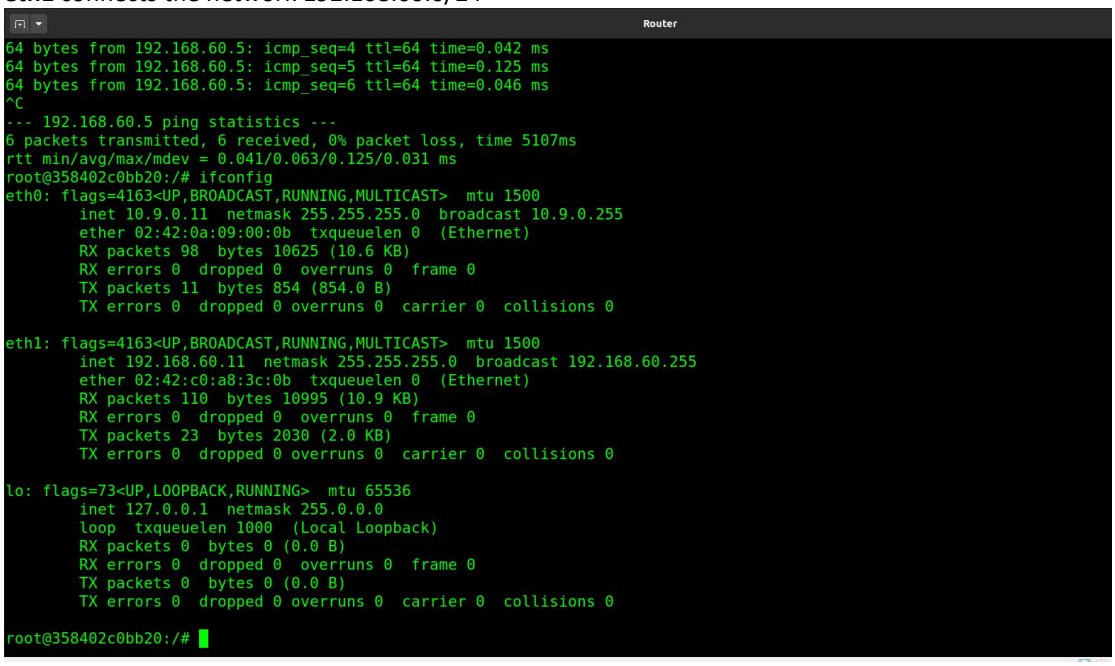
3. VPN Server can communicate with Host V

$ ping 192.168.60.5 : shows 0 loss, which means VPN Server can communicate with Host V



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

Step1 : go to the router $ ifconfig, we can find there are two interface.
eth0 connects the network 10.9.0.0/24
eth1 connects the network 192.168.60.0/24



Step2 : sinff the network 10.9.0.0/24

1. Go to Sever run **$ tcpdump -i eth0 -n**
2. Go to HostU **$ ping 10.9.0.11**

Can capture packet from **10.9.0.0/24**

```
root@358402c0bb20:/# tcpdump -i eth0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
02:58:36.452963 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 17, seq 1, length 64
02:58:36.452976 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 17, seq 1, length 64
02:58:37.480620 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 17, seq 2, length 64
02:58:37.480633 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 17, seq 2, length 64
02:58:38.495917 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 17, seq 3, length 64
02:58:38.495965 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 17, seq 3, length 64
02:58:39.520221 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 17, seq 4, length 64
02:58:39.520271 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 17, seq 4, length 64
02:58:40.547886 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 17, seq 5, length 64
02:58:40.547898 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 17, seq 5, length 64
02:58:41.507950 ARP, Request who-has 10.9.0.5 tell 10.9.0.11, length 28
02:58:41.508041 ARP, Request who-has 10.9.0.11 tell 10.9.0.5, length 28
02:58:41.508050 ARP, Reply 10.9.0.11 is-at 02:42:0a:09:00:0b, length 28
02:58:41.508051 ARP, Reply 10.9.0.5 is-at 02:42:0a:09:00:05, length 28
02:58:41.566494 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 17, seq 6, length 64
02:58:41.566528 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 17, seq 6, length 64
02:58:42.591852 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 17, seq 7, length 64
02:58:42.591866 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 17, seq 7, length 64
^C
```

Step3 : sinff the network 192.168.60.0/24

3. Go to Sever run **$ tcpdump -i eth1 -n**
4. Go to HostU **$ ping 192.168.60.11**

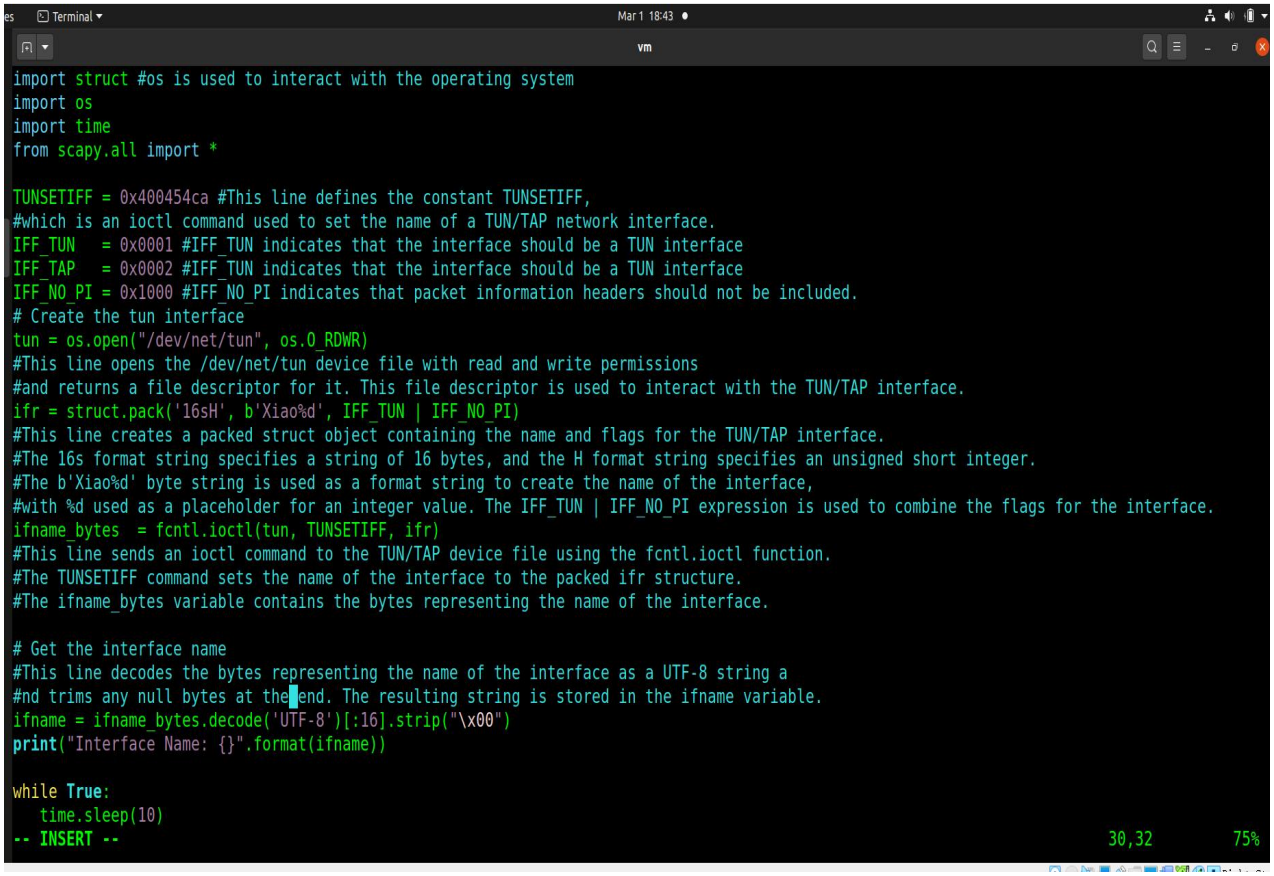
Can capture packet from 192.168.60.0/24

```
root@358402c0bb20:/# tcpdump -i eth1 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
03:04:15.991545 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 43, seq 1, length 64
03:04:15.991596 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 43, seq 1, length 64
03:04:17.023014 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 43, seq 2, length 64
03:04:17.023090 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 43, seq 2, length 64
03:04:18.047017 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 43, seq 3, length 64
03:04:18.047100 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 43, seq 3, length 64
03:04:19.071436 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 43, seq 4, length 64
03:04:19.071484 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 43, seq 4, length 64
03:04:20.094796 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 43, seq 5, length 64
03:04:20.094839 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 43, seq 5, length 64
03:04:21.120371 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 43, seq 6, length 64
03:04:21.120386 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 43, seq 6, length 64
03:04:22.142541 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 43, seq 7, length 64
03:04:22.142558 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 43, seq 7, length 64
03:04:23.167467 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 43, seq 8, length 64
03:04:23.167513 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 43, seq 8, length 64
03:04:27.359441 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
03:04:27.359549 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
```

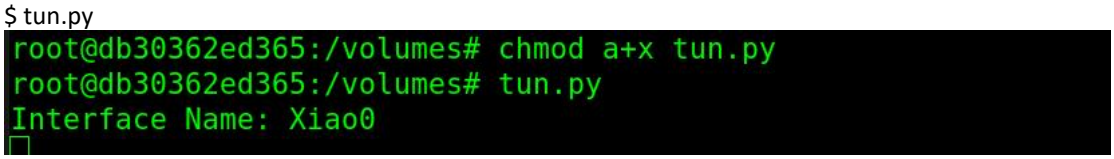
Task 2: Create and Configure TUN Interface

Task 2.a: Name of the Interface

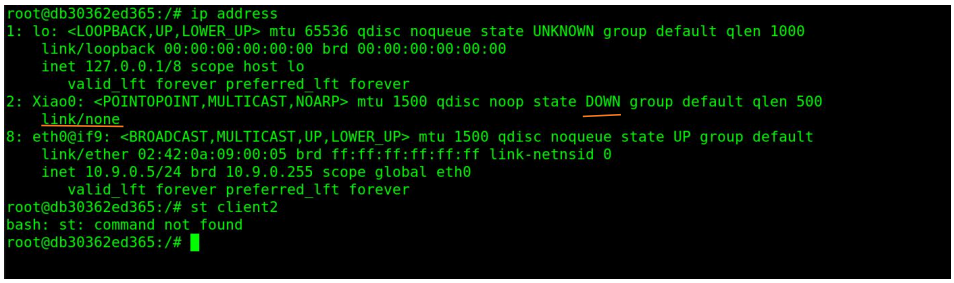
Python code of create a **interface Xiao**



Step1: Go to client(10.9.0.5)
Run: $ chmod a+x
$ tun.py



Step2: Open a new terminal for client(10.9.0.5)
$ ip address： We created interface named Xiao0, but it is still not configured.



Task 2.b: Set up the TUN Interface

I add new lines in the python code:

```
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))#First, we need to assign an IP address to it
os.system("ip link set dev {} up".format(ifname))#Second, we need to bring up the interface,
#because the interface is still in the down state
```

Using the same step as above, we can find the new interface is configured and active.
1. The interface assign a IP address 192.168.53.99/24
2. State change to UP.

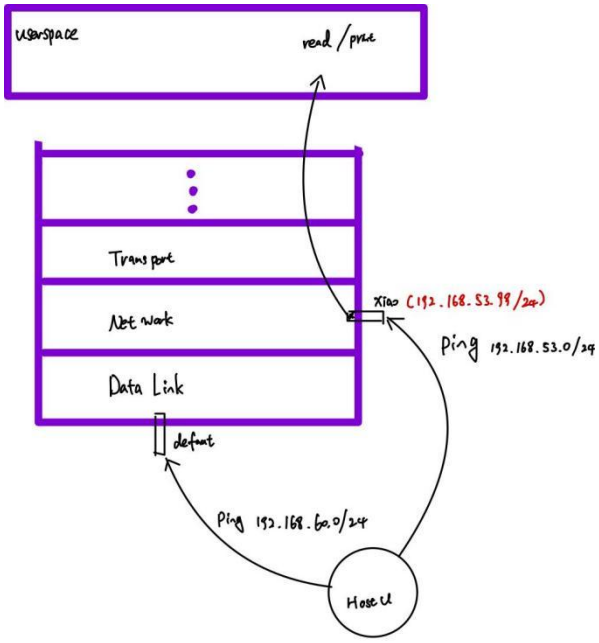
```
root@db30362ed365:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
3: Xiao0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global Xiao0
       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
root@db30362ed365:/#
```

Task 2.c: Read from the TUN Interface

Python code:

```
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))#First, we need to assign an IP address to it
os.system("ip link set dev {} up".format(ifname))#Second, we need to bring up the interface,
#because the interface is still in the down state
while True:
# Get a packet from the tun interface
    packet = os.read(tun, 2048)# buffer 2048
    if packet:
        ip = IP(packet)
        print(ip.summary())
```

Figure for the following question:



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

```
root@db30362ed365:/# ping 192.168.53.1 -c 4
PING 192.168.53.1 (192.168.53.1) 56(84) bytes of data.

--- 192.168.53.1 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3053ms
```

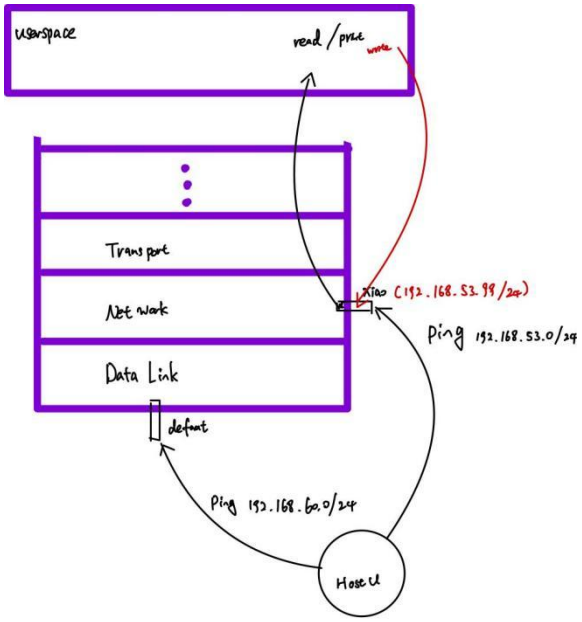
```
Interface Name: Xiao0
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
```

Based on our setting (interface : 192.168.53.99/24) when we ping 192.168.53.1, the packet will go to the interface Xiao(we capture here). Then the program read the interface and send to the user space then print to screen.

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

Print nothing.
Since **192.168.60.0/24 is not in the range 192.168.53.0/24,** the icmp packet will go to the etho rather than Xiao.

Task 2.d: Write to the TUN Interface



Python: (code from lecture)

```
while True:
# Get a packet from the tun interface
    packet = os.read(tun, 2048)# buffer 2048
    if packet:
        pkt = IP(packet)
        print(pkt.summary())
    if ICMP in pkt:
        newip = IP(src=pkt[IP].dst,dst=pkt[IP].src,ihl=pkt[IP].ihl)
        newip.ttl = 99
        newicmp = ICMP(type=0,id=pkt[ICMP].id,seq=pkt[ICMP].seq)
        if pkt.haslayer(Raw):
            data=pkt[Raw].load
            newpkt= newip/newicmp/data
        else:
            newpkt= mewip/newicmp
        os.write(tun,bytes(newpkt))
```

Code explain:

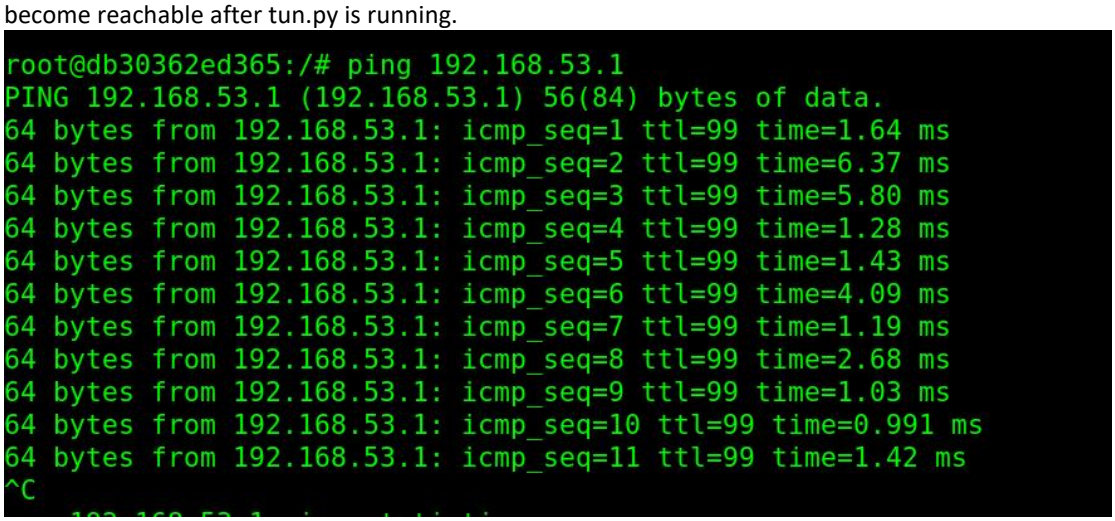
If the received packet does contain an ICMP header, the code creates a new IP header and swaps the source and destination IP addresses, sets the time-to-live (TTL) value to 99, and creates a new ICMP header with the same ID and sequence number as the original packet.

Icmp type0: ICMP type 0 refers to an ICMP Echo Reply message

If the original packet also contains a payload (i.e., a Raw layer), the payload is extracted and added to the new packet. Finally, the new packet is sent out through a network interface (tun) using the operating system's "os.write" function.

It's possible that this code is part of a network tool or program that intercepts and modifies network traffic. However, without more context, it's difficult to say for sure what the purpose of this code is.

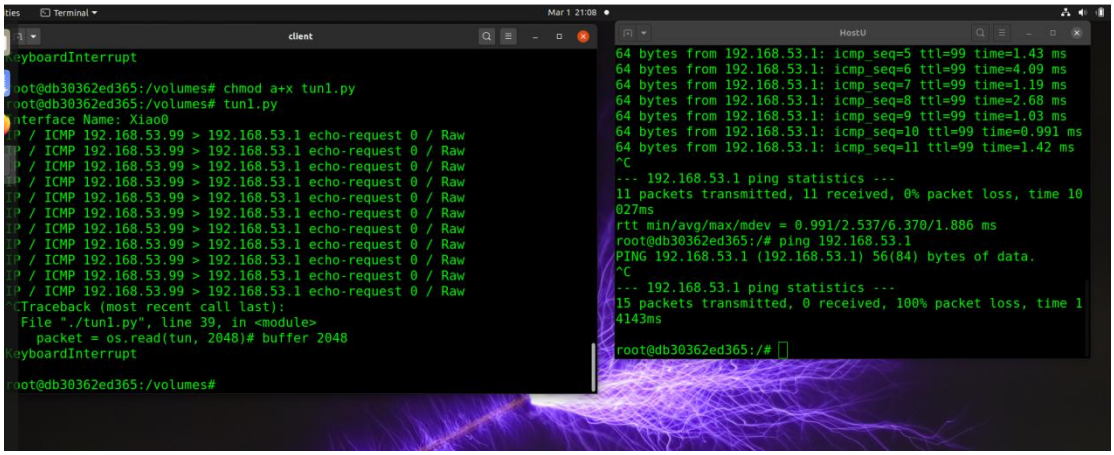
Ping from host U, this time we can get reply . which means we can write (send) packet from Interface Xiao.When I start ping 192.168.53.1, a unreachable IP address previously, 192.168.53.1 become reachable after tun.py is running.



Instead of writing an IP packet to the interface, write some arbitrary data to the interface

```
os.system('ip link set dev {} up'.format(ifname))#second, we need to bring up the interface,
#because the interface is still in the down state
while True:
# Get a packet from the tun interface
    packet = os.read(tun, 2048)# buffer 2048
    if packet:
        pkt = IP(packet)
        print(pkt.summary())
    if ICMP in pkt:
        newip = IP(src=pkt[IP].dst,dst=pkt[IP].src,ihl=pkt[IP].ihl)
        newip.ttl = 99
        newicmp = ICMP(type=0,id=pkt[ICMP].id,seq=pkt[ICMP].seq)
        if pkt.haslayer(Raw):
            data="Huge Dick is coming"
            newpkt= newip/newicmp/data
        else:
            newpkt= mewip/newicmp
        os.write(tun,bytes(newpkt))

"tun1.py" 53L, 2800C
```

Observation： From the screenshot below, we can see that the ping program think 192.168.53.1 is not reachable.



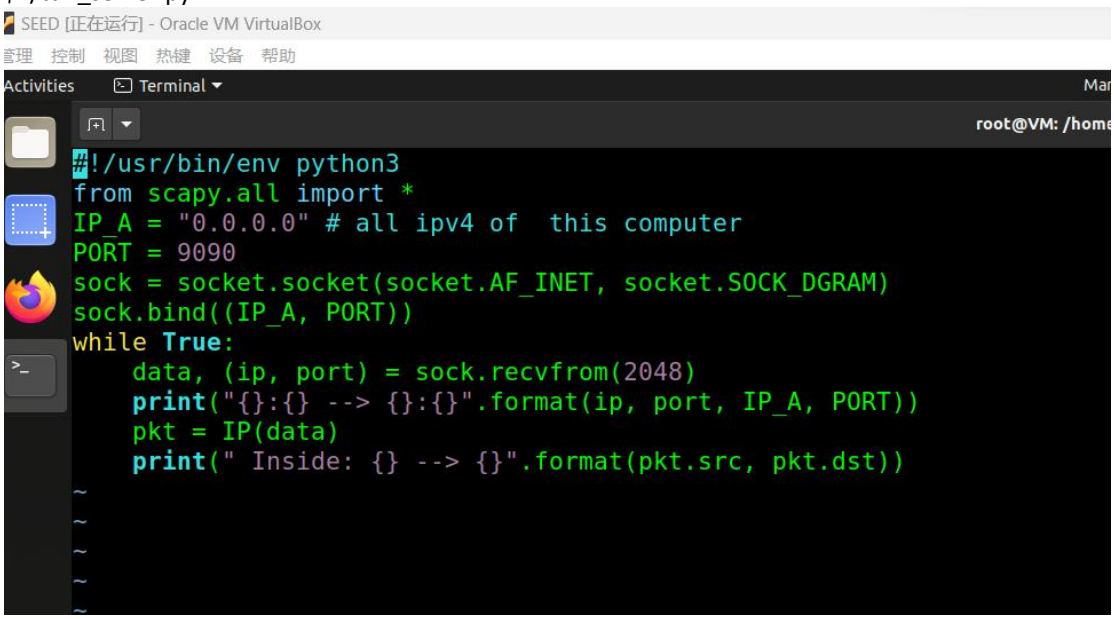
Explain:

The ping command in Linux (and other operating systems) sends ICMP Echo Request messages to a remote host, and waits for an ICMP Echo Reply message in response. The ICMP Echo Request message typically includes a payload of arbitrary data, which is used to verify that the reply packet is identical to the request packet.

When the remote host receives the ICMP Echo Request message, it generates an ICMP Echo Reply message with the same payload data, which is then sent back to the original sender. The ping command on the original sender then compares the data in the Echo Reply packet to the data in the original Echo Request packet, and if the data is the same, it considers the ping successful. If the data is different, the ping is considered a failure.

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

SeverCode:
Go to the VPN_SERVER
$ chmod a+x tun_server.py
$ ./tun_server.py



ClientCode:
Go to the VPN_client
$ chmod a+x tun_client.py
$ ./tun_client.py

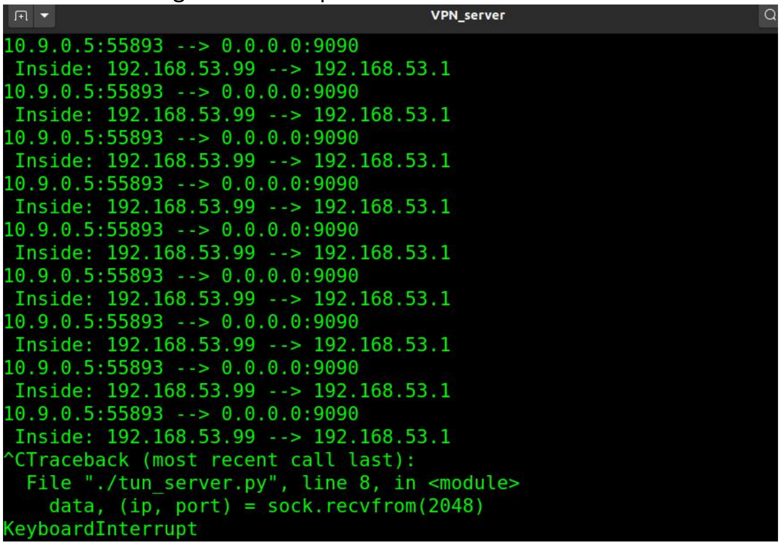
Cp a newfile of tun.py to VPN_client -->delete the while loop
Add the following code

```
# Create UDP socket
SERVER_IP = '10.9.0.11'
SERVER_PORT = 9090
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while True:
    # Get a packet from the tun interface
    packet = os.read(tun, 2048)
    if packet:
    # Send the packet via the tunnel
        sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

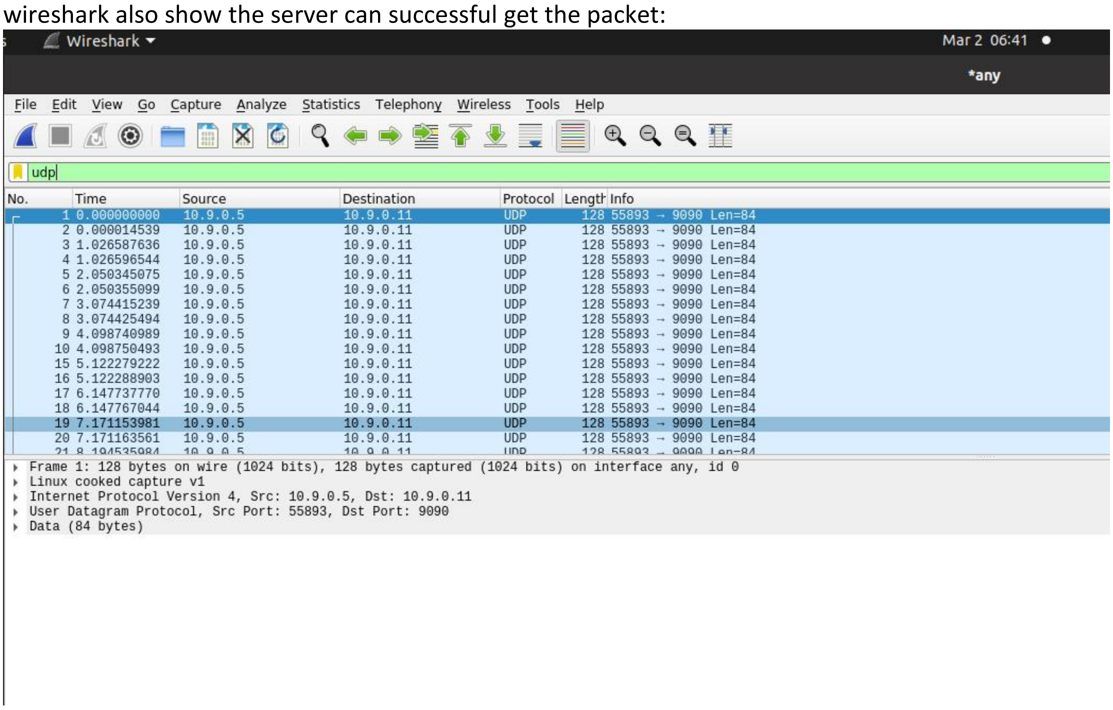
Test:

1. Open a new terminal for client. **Ping 192.168.53.1**

The server can get the ICMP packet



wireshark also show the server can successful get the packet:



2. Ping **192.168.60.5**

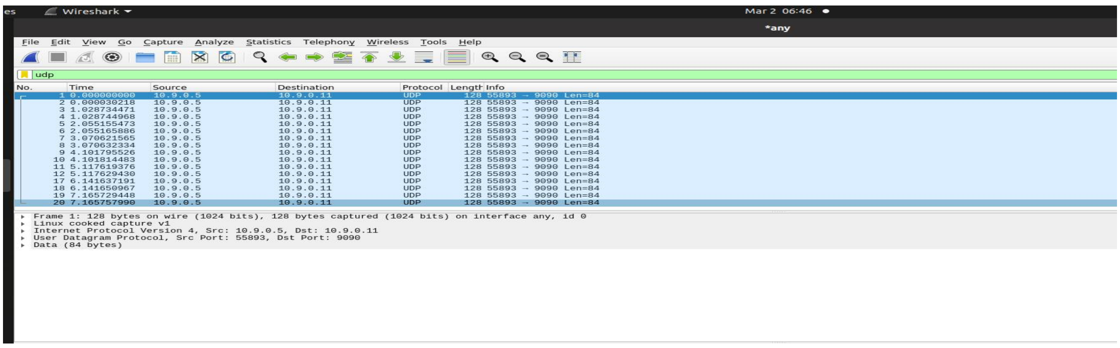
When we ping 192.168.60.5, we find that the VPN server does not print out anything. Because 192.168.60.5 is not match the network assigned to **interface Xiao** and there is no route rule for 192.168.60.5, the packet will send the packet through the default interface which can not reach to the VPN server. As the result, the VPN server get nothing and print out nothing.

The idea of the solution is to route the packet to the interface **tun0**. So I add one rule for the route table of **tun0**

```
root@f947656b9672:/# ip route add 192.168.60.0/24 dev tun0
root@f947656b9672:/# ip -br address
lo               UNKNOWN        127.0.0.1/8
tun0             UNKNOWN        192.168.53.99/24
eth0@if9         UP             10.9.0.5/24
root@f947656b9672:/# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         10.9.0.1        0.0.0.0         UG    0      0        0 eth0
10.9.0.0        0.0.0.0         255.255.255.0   U     0      0        0 eth0
192.168.53.0    0.0.0.0         255.255.255.0   U     0      0        0 tun0
192.168.60.0    0.0.0.0         255.255.255.0   U     0      0        0 tun0
root@f947656b9672:/# ping 192.168.60.5
```

Now the server can get the packet:

```
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
10.9.0.5:55893 --> 0.0.0.0:9090
 Inside: 192.168.60.0 --> 192.168.60.5
```



Task 4: Set Up the VPN Server

Python code:

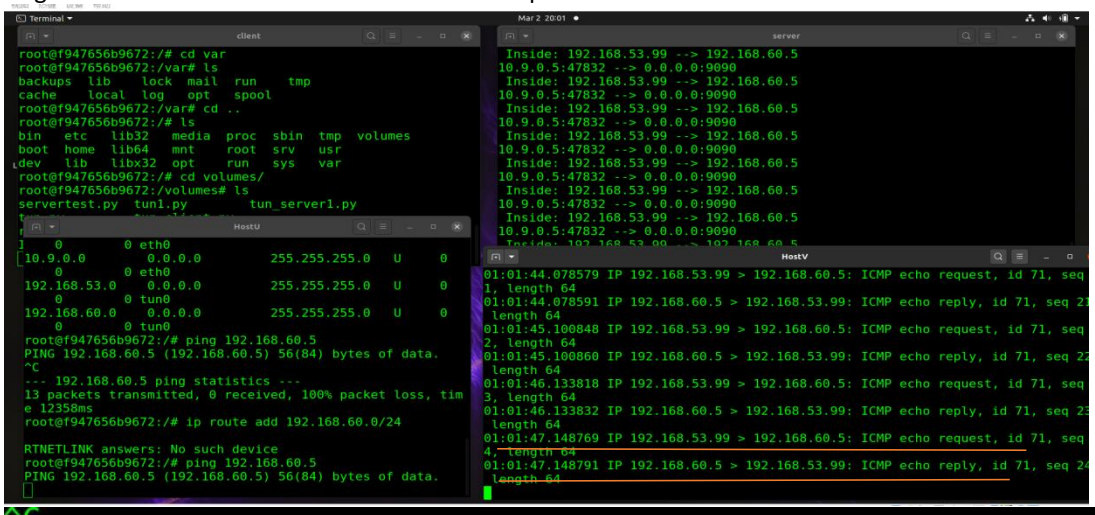
```
 GNU nano 4.8                                servertest.py
#!/usr/bin/env python3
import fcntl
import struct
import os
import time
from scapy.all import *
TUNSETIFF = 0x400454ca
IFF_TUN = 0x0001
IFF_TAP = 0x0002
IFF_NO_PI = 0x1000
SERVER_IP = '10.9.0.11'
SERVER_PORT = 9090
# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'tun%d', IFF_TUN | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))
# Assign IP to interface and turn on the interface
os.system("ip addr add 192.168.53.1/24 dev {}".format(ifname))
#os.system("ip addr add 192.168.53.1/24 dev {}".format(ifname))

os.system("ip link set dev {} up".format(ifname))
# Create the socket and listen to port 9090
IP_A = "0.0.0.0"
PORT = 9090
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_A, PORT))

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

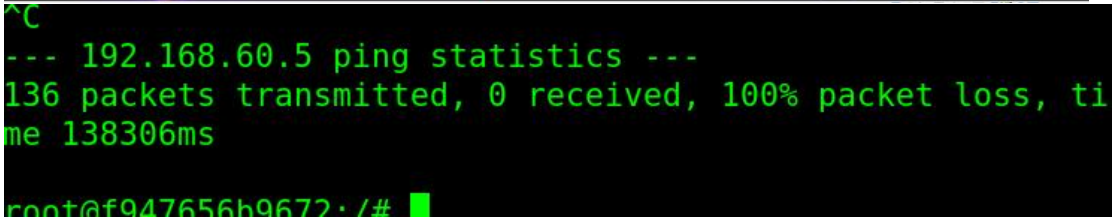
To test the VPN server, I open 4 terminals for the task.

1. Client : run the tun_client.
2. HostU(10.9.0.5): ping 192.168.60.5
3. Server: run the tun_server.
4. HostV(192.168.60.5): $ tcpdump -i eth0 -n

Observation :

From the Host V, we can see that the reply packet is sent, but Host U do not receive the reply and says that all packets are loss. It means that the request packets are arrived to Host V but the tunnel is single direction and unable to deliver the return packet.





Task 5: Handling Traffic in Both Directions

ClientCode:

```python
#!/usr/bin/env python3
import fcntl
import struct
import os
import time
from scapy.all import *
TUNSETIFF = 0x400454ca
IFF_TUN = 0x0001
IFF_TAP = 0x0002
IFF_NO_PI = 0x1000

# Create the interface tun
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'tun%d', IFF_TUN | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

# bind a IP address and up the tun
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

# build tunnel/set router table for the tun
os.system("ip route add 192.168.60.0/24 dev tun0 via 192.168.53.99")

# Create the socket and listen to port 9090
IP_A = "0.0.0.0"
PORT = 9090
recv_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
recv_socket.bind((IP_A, PORT))
```

```python
# Create UDP socket
SERVER_IP = '10.9.0.11'
SERVER_PORT = 9090
send_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

while True:
    ready, _, _ = select.select([recv_socket, tun], [], [])
    for fd in ready:
        if fd is recv_socket:
            data, (ip, port) = recv_socket.recvfrom(2048)
            pkt = IP(data)
            print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
            os.write(tun,data)
        if fd is tun:
            packet = os.read(tun, 2048)
            pkt = IP(packet)
            print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
            send_socket.sendto(packet, (SERVER_IP, SERVER_PORT))
```

Server code:

```
#!/usr/bin/env python3
import fcntl
import struct
import os
import time
from scapy.all import *
TUNSETIFF = 0x400454ca
IFF_TUN = 0x0001
IFF_TAP = 0x0002
IFF_NO_PI = 0x1000

# Create the interface tun
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'tun%d', IFF_TUN | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

# bind a IP address and up the tun
os.system("ip addr add 192.168.53.1/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))


# Create the socket and listen to port 9090
IP_A = "0.0.0.0"
PORT = 9090
recv_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
recv_socket.bind((IP_A, PORT))
```

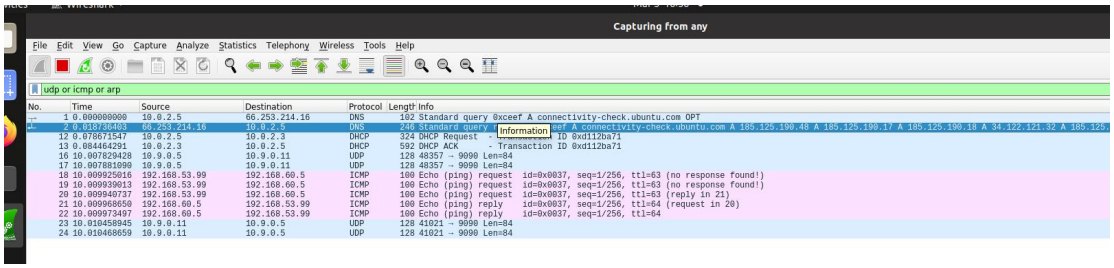
```
# Create UDP socket
CLIENT_IP = '10.9.0.5'
CLIENT_PORT = 9090
send_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

while True:
    ready, _, _ = select.select([recv_socket, tun], [], [])
    for fd in ready:
        if fd is recv_socket:
            data, (ip, port) = recv_socket.recvfrom(2048)
            pkt = IP(data)
            print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
            os.write(tun,data)
        if fd is tun:
            packet = os.read(tun, 2048)
            pkt = IP(packet)
            print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
            send_socket.sendto(packet, (CLIENT_IP, CLIENT_PORT))
```

Test:

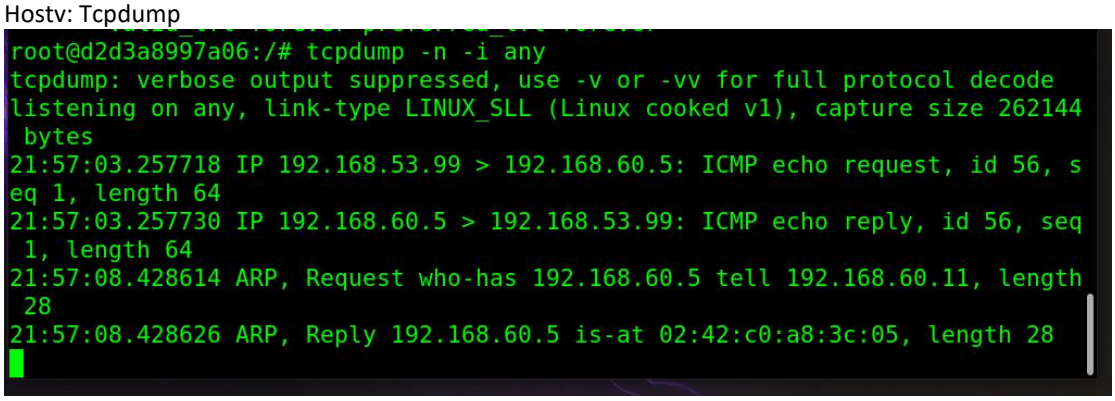
Ping:

At Host U **ping 192.168.60.5**

Wireshark:



Hostv: Tcpdump



Explain:
ICMP Request:
Client side: we can find a **icmp** packet send from 192.168.53.99(IP address we set on tun0), then send to the “Tunnel” and into a packet with **src=10.9.0.5, then** send to **10.9.0.11.**

Server side: when reach the 10.9.0.11, the packwill go to the kernel and change to 192.168.53.99->192.168.60.5 since the desination is 192.168.60.5 , the os will send it to the interface with 192.168.60.11 then broadcast a Arp request, then the packet will send to the final destination 192.168.60.5

ICMP reply：
Host V reply the ICMP request packet (192.168.53.99 -> 192.168.60.5)

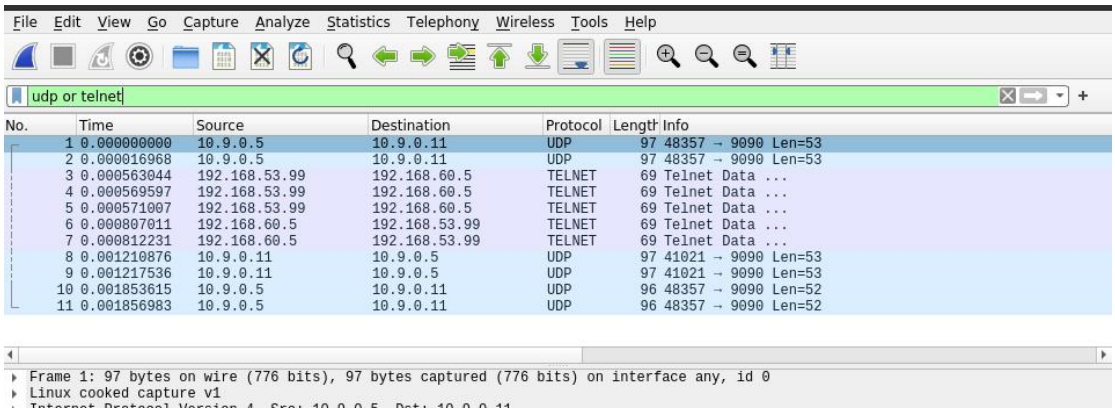
The VPN server receive the ICMP reply packet and package it into tunnel packet (10.9.0.11-> 10.9.0.5). Because User U and VPN client is a single container and the ICMP reply packet route to User inside the container, the last packet capture by Wireshark is the tunnel packet.

Telnet:

Behavior of Telnet:

In the telnet, when user type something, the client will send the content that user type to the connected remote server. Then the remote server machine receive the content and send the print out content to the client.

Wireshark:



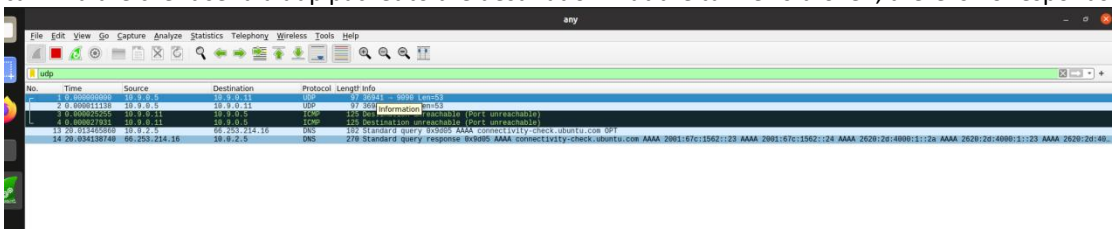
Explain:
In Host V, I type l
1. A telnet packet send (192.168.53.99->192.168.60.5)
2. (10.9.0.5->10.9.0.11) use the tunnel from client send to server
3. The sever send the packet to 192.168.60.5
4. (192.168.60.5->192.168.53.99) telnet reply
5. (10.9.0.11->10.9.0.5) use tunnel send back to client.

Because the telnet client and VPN client is single container, the VPN client receive the tunnel packet and send telnet reply packet and the telnet reply packet deliver inside the container. That is why the last packet is tunnel packet.

Task 6: Tunnel-Breaking Experiment

Broken:

I close the vpn_server,immediately , when I type a letter, there is no responds. From wireshark we can find the client send a udp packet to the destination. But the tunnel is broken, there is no responds.

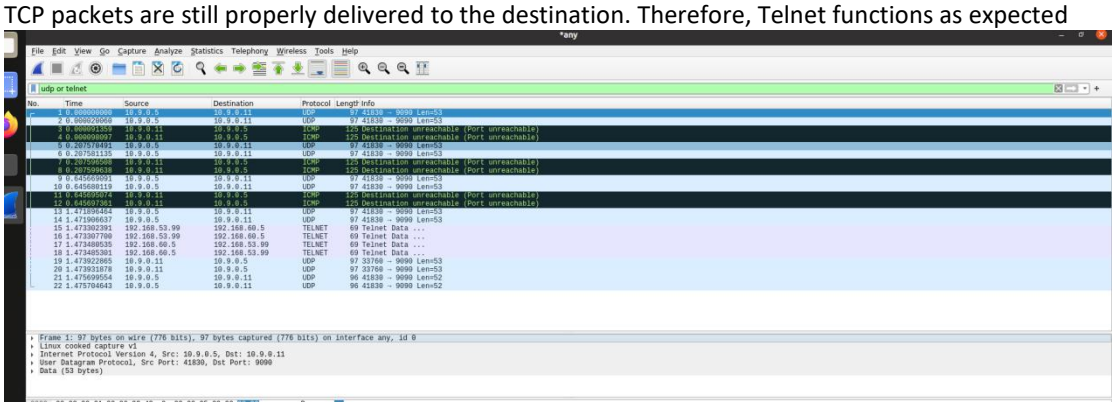


Reconnect:

I restart the server, we can find :

6. A telnet packet send (192.168.53.99->192.168.60.5)
7. (10.9.0.5->10.9.0.11) use the tunnel from client send to server
8. The sever send the packet to 192.168.60.5
9. (192.168.60.5->192.168.53.99) telnet reply
10. (10.9.0.11->10.9.0.5) use tunnel send back to client.

When we establish the tunnel again, any characters we type will be displayed on the client. This is because the TCP connection continues to reset the packets and the tunnel is quickly broken, but the TCP packets are still properly delivered to the destination. Therefore, Telnet functions as expected



Task 7: Routing Experiment on Host V

Just delete the default route

Go to Host V, $ ip route. We can find the HostV default route to 192.168.60.11. Then we delete the default route.

```
root@c28e26aabc26:/# ip route
default via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@c28e26aabc26:/# ip route del default
root@c28e26aabc26:/# ip route
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@c28e26aabc26:/#
```

Go to host U ,ping host V. We can find there is no reply.

```
root@0b80e9464b08:/# ping 192.168.60.5 -c 1
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

On host V, run tcpudp, we can find the vpn server use arp broadcast find host U send the request packet to host V. However, since there is no route for 192.158.53.99, HostV doesn't know how to send back the packet.

```
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
17:24:40.302755 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 35, seq 1, length 64
17:24:45.506790 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
17:24:45.506927 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
```

Add new route:

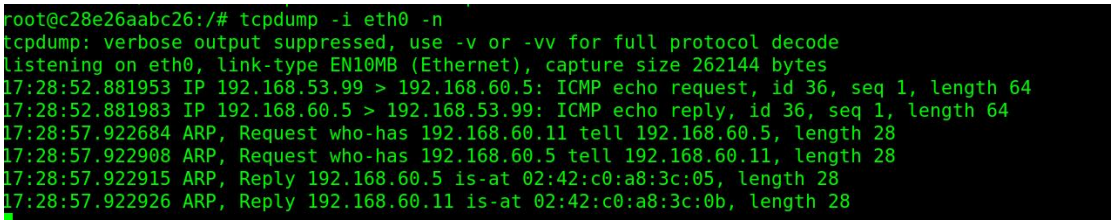
Since we want send icmp reply back to VPN_client, use
$ ip route add 192.168.53.0/24 via 192.168.60.11

```
root@c28e26aabc26:/# ip route add 192.168.53.0/24 via 192.168.60.11
root@c28e26aabc26:/# ip route
192.168.53.0/24 via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@c28e26aabc26:/#
```

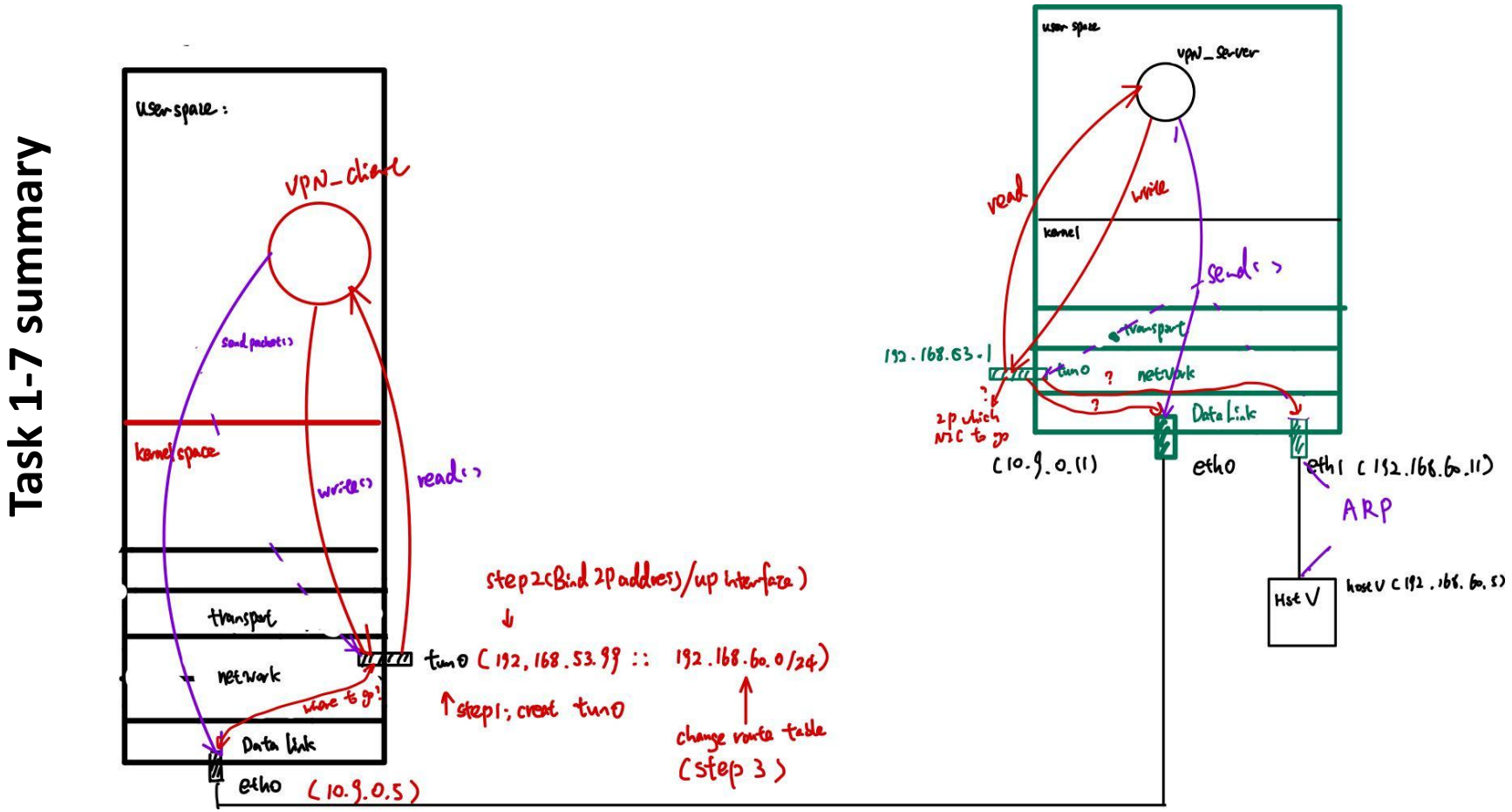
Go to host U ,ping host V. We can find we can get reply.

```
root@0b80e9464b08:/# ping 192.168.60.5 -c 1
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=3.07 ms

--- 192.168.60.5 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.071/3.071/3.071/0.000 ms
root@0b80e9464b08:/#
```

On host V, run tcpudp, we can find the vpn server use arp broadcast find host U send the request packet to host V. Now the host V know how to send back to VPN client, the HostV use ARP broadcast for 192.168.60.11 then send the packet back to VPN server.

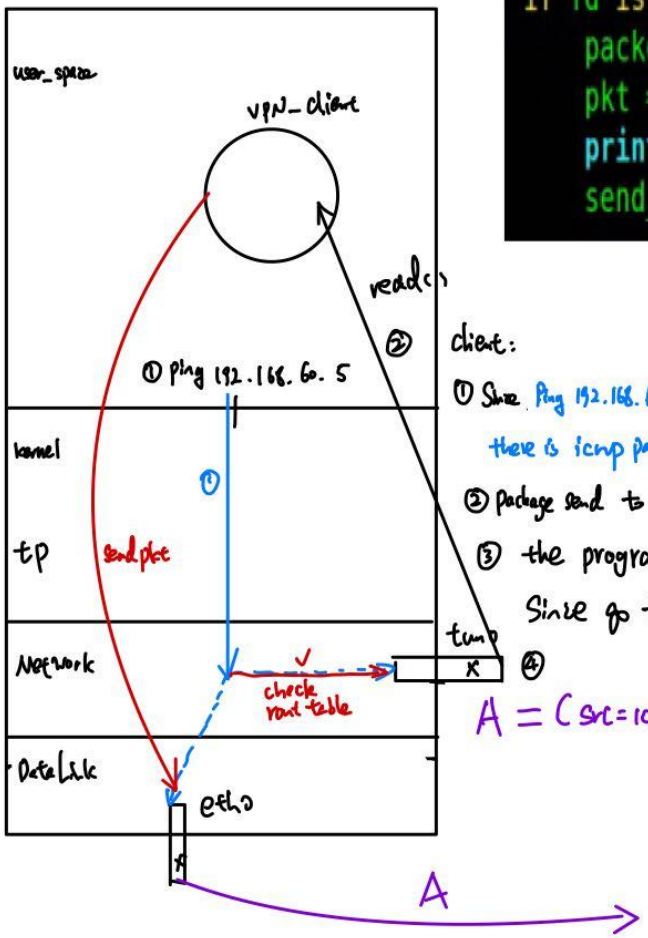


Task 1-7 summary



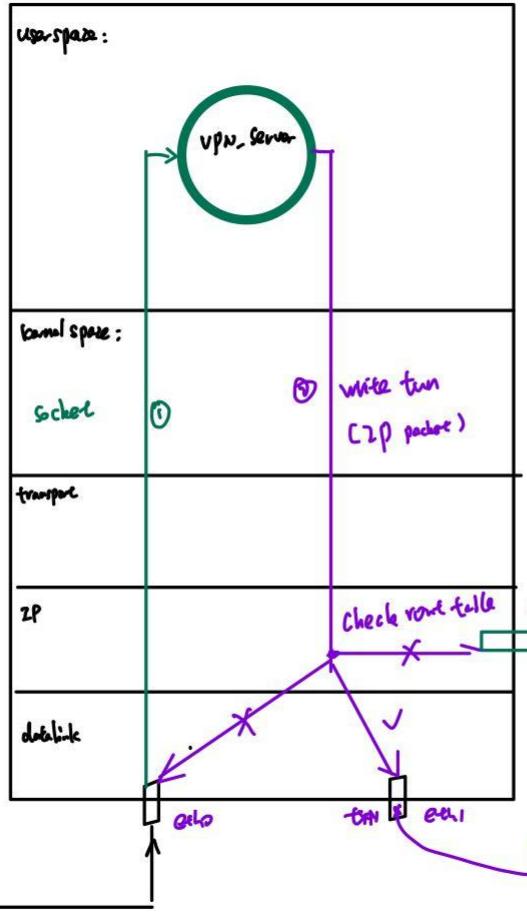
ping 192.168.60.5 (request)

client:




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

Sever



① get socket, send to user-space prog C part 9o9o)

after go to to stack, remove udp header

data = C src = 192.168.53.99, des : 192.168.60.5, ICMP)

② write to tun0

/* This is a virtual network interface, what ever is written to the interface by the application will appear in the kernal as an IP packet */

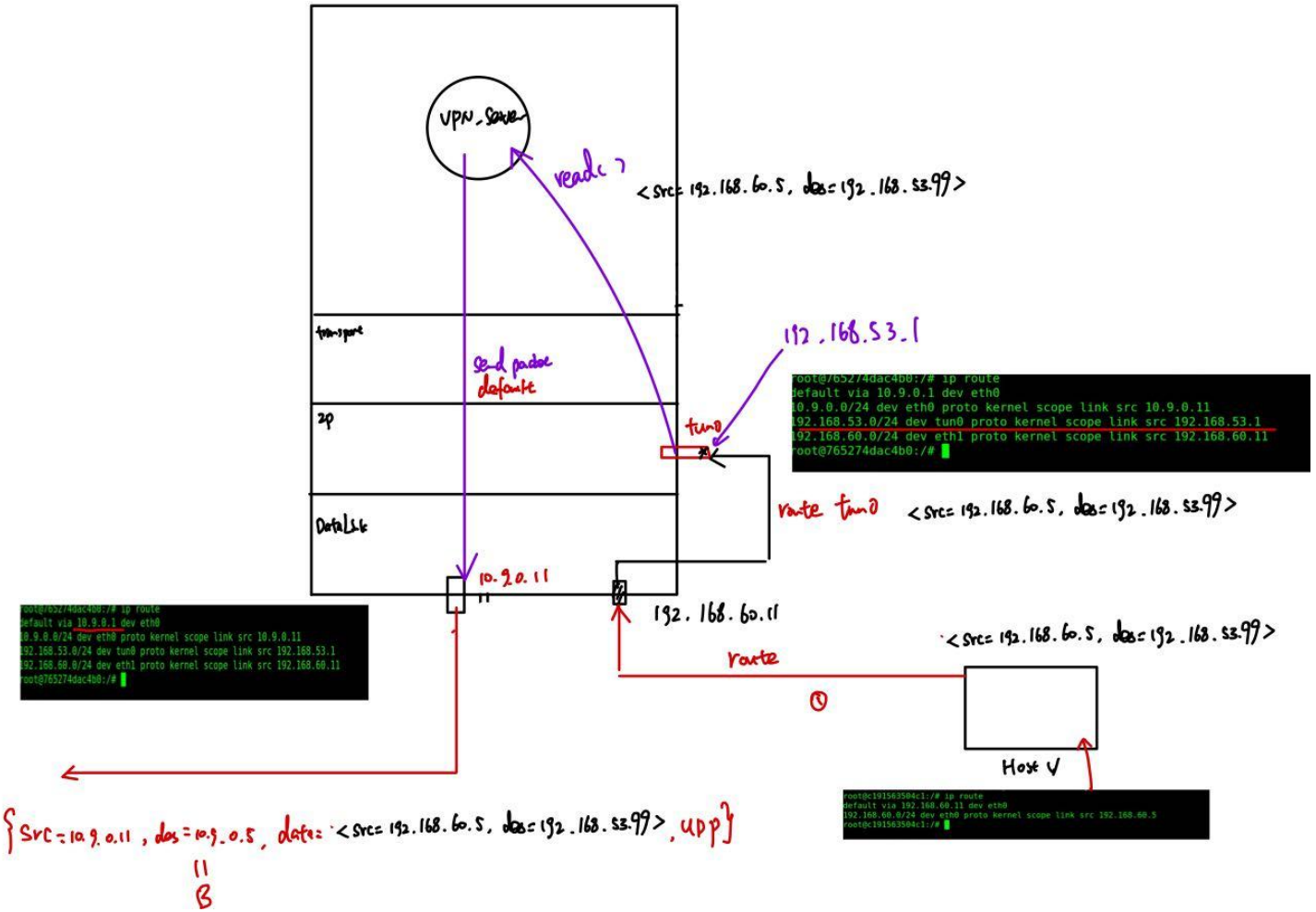
③ eth1 : ARP broad cast find ✓

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

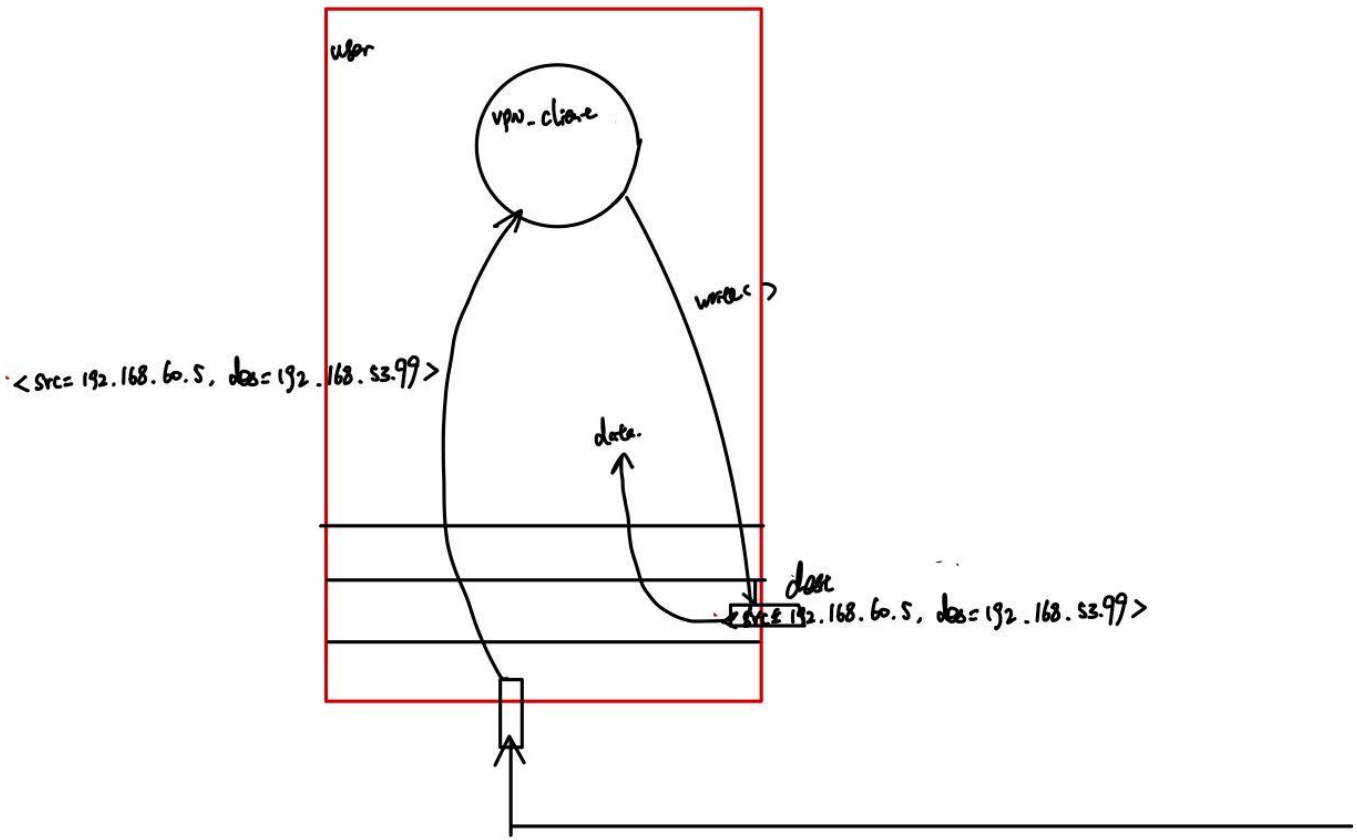
APP

A = C src = 10.9.0.5, des = 10.9.0.11, Data: C src = 192.168.53.99, des : 192.168.60.5, ICMP), type : udp)

Icmp reply :


VPN_Server
read
<Src= 192.168.60.5, des= 192.168.53.99>
transport
ip
DataLink
send packet default
192.168.53.1
tun0
route tun0
<Src= 192.168.60.5, des= 192.168.53.99>
10.9.0.11
192.168.60.11
<Src= 192.168.60.5, des= 192.168.53.99>
route
Host V
{Src=10.9.0.11, des=10.9.0.5, data=<Src= 192.168.60.5, des= 192.168.53.99>, upp}
11
B

client



{src=10.9.0.11, dest=10.9.0.5, data=<src=192.168.60.5, dest=192.168.53.99>, upp}

||

B

Task 8: VPN Between Private Networks

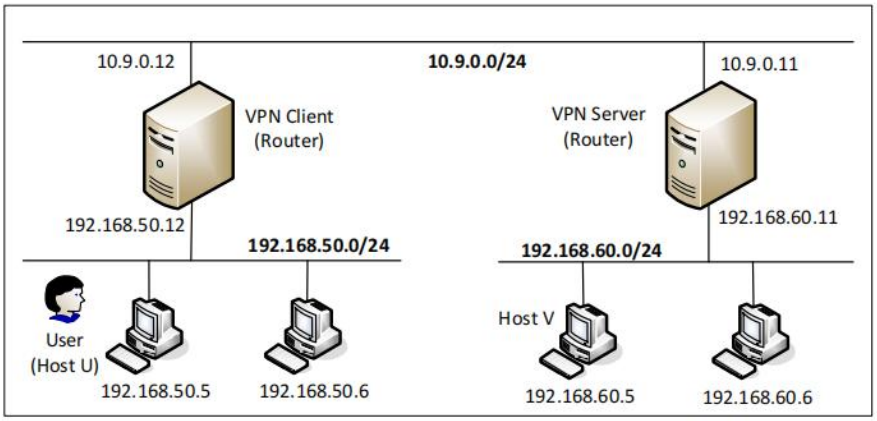


Figure 2: VPN between two private networks

Step0: rebuild the docker
$ docker-compose -f docker-compose2.yml build
$ docker-compose -f docker-compose2.yml up
$ docker-compose -f docker-compose2.yml down

Server:

```
#!/usr/bin/env python3
import fcntl
import struct
import os
import time
from scapy.all import *
TUNSETIFF = 0x400454ca
IFF_TUN = 0x0001
IFF_TAP = 0x0002
IFF_NO_PI = 0x1000

# Create the interface tun
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'tun%d', IFF_TUN | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

# bind a IP address and up the tun
os.system("ip addr add 192.168.53.1/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

# set route table:
os.system("ip route add 192.168.50.0/24 dev tun0 via 192.168.53.1")
# Create the socket and listen to port 9090
IP_A = "0.0.0.0"
PORT = 9090
recv_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
recv_socket.bind((IP_A, PORT))
```

```
# Create UDP socket
CLIENT_IP = '10.9.0.12'
CLIENT_PORT = 9090
send_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

while True:
    ready, _, _ = select.select([recv_socket, tun], [], [])
    for fd in ready:
        if fd is recv_socket:
            data, (ip, port) = recv_socket.recvfrom(2048)
            pkt = IP(data)
            print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
            os.write(tun,data)
        if fd is tun:
            packet = os.read(tun, 2048)
            pkt = IP(packet)
            print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
            send_socket.sendto(packet, (CLIENT_IP, CLIENT_PORT))
```

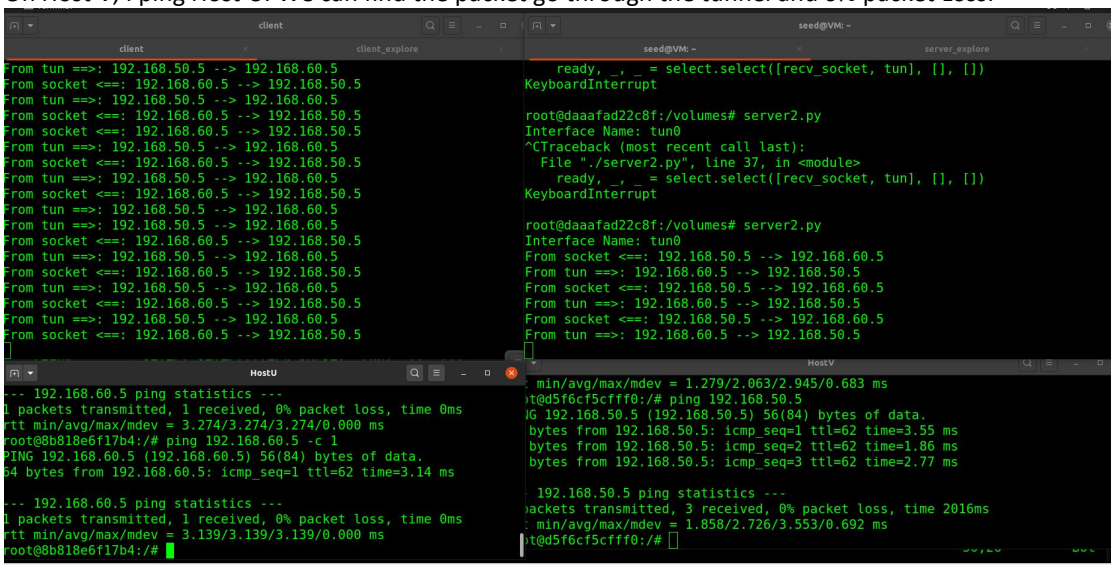
Client:

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

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

# Create the interface tun
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'tun%d', IFF_TUN | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

# bind a IP address and up the tun
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

# build tunnel/set router table for the tun
os.system("ip route add 192.168.60.0/24 dev tun0 via 192.168.53.99")

# Create the socket and listen to port 9090
IP_A = "0.0.0.0"
PORT = 9090
recv_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
recv_socket.bind((IP_A, PORT))

# Create UDP socket
SERVER_IP = '10.9.0.11'
SERVER_PORT = 9090
send_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

while True:
    ready, _, _ = select.select([recv_socket, tun], [], [])
    for fd in ready:
        if fd is recv_socket:
            data, (ip, port) = recv_socket.recvfrom(2048)
            pkt = IP(data)
            print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
            os.write(tun,data)
        if fd is tun:
            packet = os.read(tun, 2048)
            pkt = IP(packet)
            print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
            send_socket.sendto(packet, (SERVER_IP, SERVER_PORT))
```

Proofs to show that the packets between the two private networks are indeed going through a VPN tunnel.

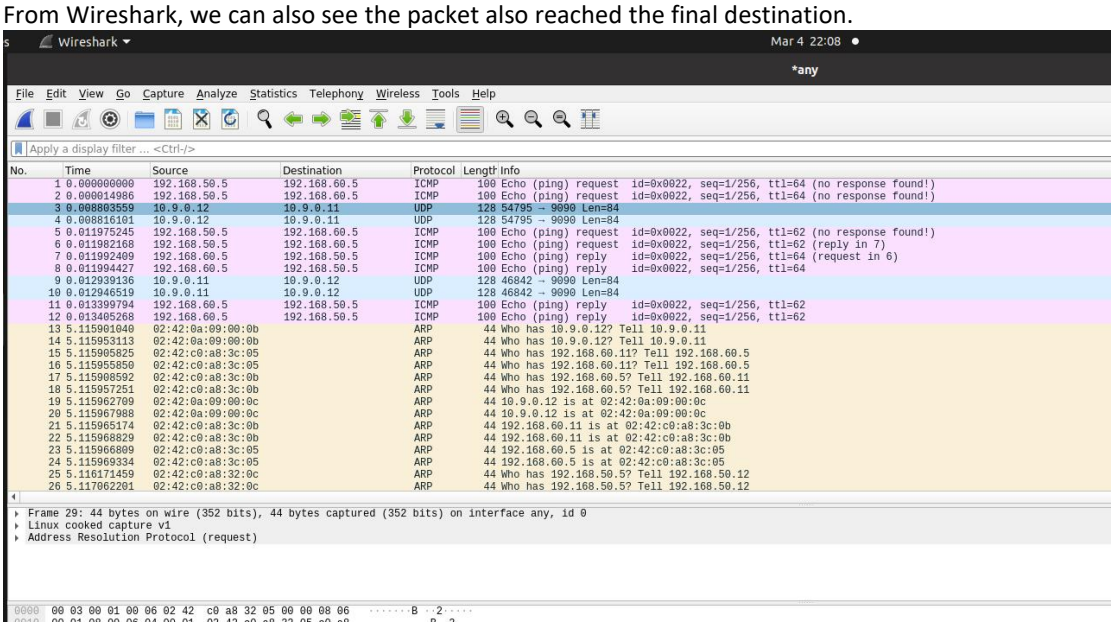
On Host U, I ping Host V.

On Host V, I ping Host U. We can find the packet go through the tunnel and 0% packet Loss.

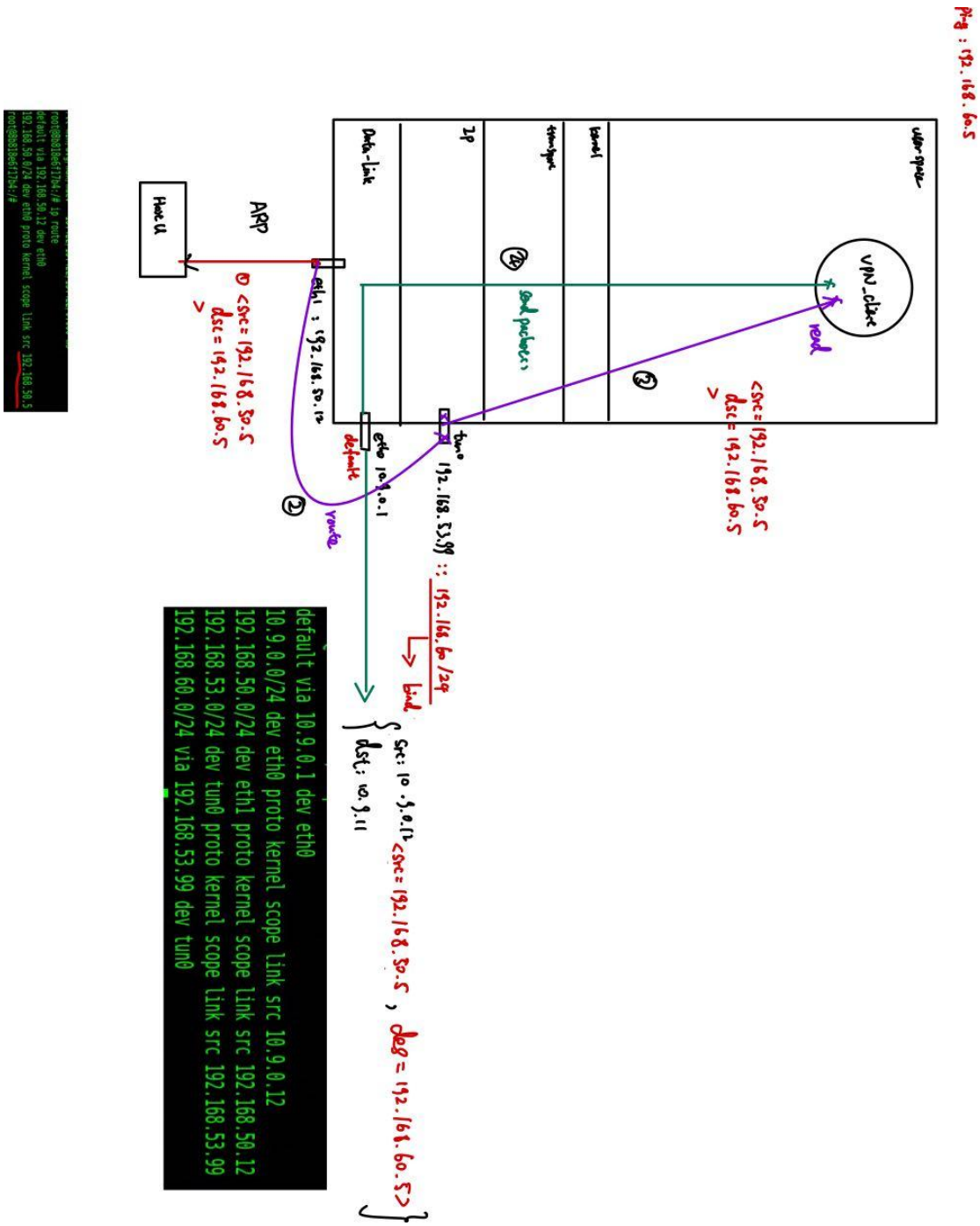


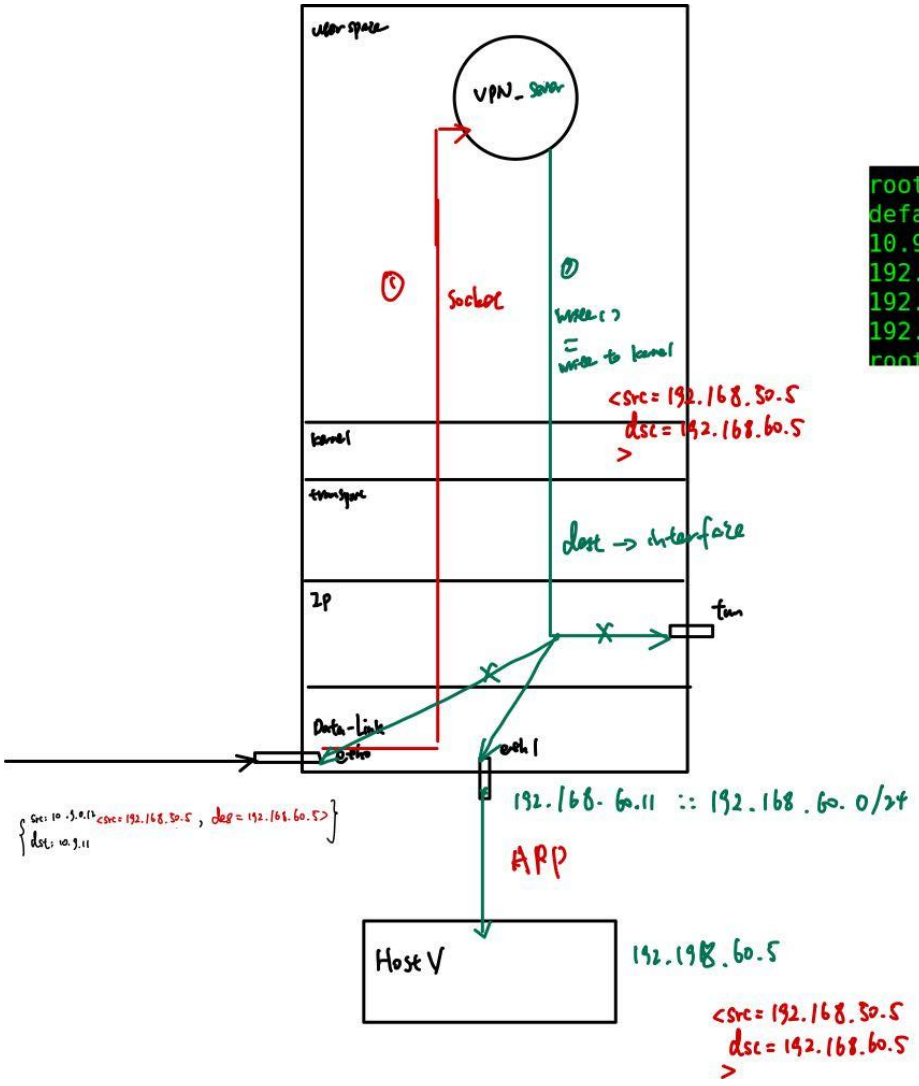
Wireshark:

From Wireshark, we can also see the packet also reached the final destination.



Graph to illustrate the procedure:






```
root@daaafad22c8f:/# ip route
default via 10.9.0.1 dev eth0
10.9.0.0/24 dev eth0 proto kernel scope link src 10.9.0.11
192.168.50.0/24 via 192.168.53.1 dev tun0
192.168.53.0/24 dev tun0 proto kernel scope link src 192.168.53.1
192.168.60.0/24 dev eth1 proto kernel scope link src 192.168.60.11
root@daaafad22c8f:/#
```

Task 9: Experiment with the TAP Interface

1: Python code:

```
es  Terminal                                                    Mar 4 22:52
                                                                      vim
#!/usr/bin/env python3
import fcntl
import struct
import os
import time
from scapy.all import *
TUNSETIFF = 0x400454ca
IFF_TUN = 0x0001
IFF_TAP = 0x0002
IFF_NO_PI = 0x1000
# Create the tan interface
tap = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'tap%d', IFF_TAP | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tap, TUNSETIFF, ifr)
# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))
while True:
    packet = os.read(tap, 2048)
    if packet:
        ether = Ether(packet)
        print(ether.summary())
~
~
~
~
~
~
```

Then I ping 192.168.54.1:

```
root@a7d94dce1e8f:/# ping 192.168.53.1
PING 192.168.53.1 (192.168.53.1) 56(84) bytes of data.
^C
```

Observation:

The program print out the ARP packet for asking the **MAC address of 192.168.53.1.**

```
root@a7d94dce1e8f:/volumes# TAP1.py
Interface Name: tap0
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
Ether / ARP who has 192.168.53.1 says 192.168.53.99
```

2:Python code:

```
vim
import struct
import os
import time
from scapy.all import *
TUNSETIFF = 0x400454ca
IFF_TUN = 0x0001
IFF_TAP = 0x0002
IFF_NO_PI = 0x1000
# Create the tan interface
tap = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'tap%d', IFF_TAP | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tap, TUNSETIFF, ifr)
# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))
while True:
    packet = os.read(tap, 2048)
    if packet:
        print("--------------------------------")
        ether = Ether(packet)
        print(ether.summary())
        # Send a spoofed ARP response
        FAKE_MAC = "aa:bb:cc:dd:ee:ff"
        if ARP in ether and ether[ARP].op == 1 :
            arp = ether[ARP]
            newether = Ether(dst=ether.src, src=FAKE_MAC)
            newarp = ARP(psrc=arp.pdst, hwsrc=FAKE_MAC, pdst=arp.psrc, hwdst=ether.src, op=2)
            newpkt = newether/newarp
            print("***** Fake response: {}".format(newpkt.summary()))
            os.write(tap, bytes(newpkt))
"TAP2.py" 34L, 1242C
```

Test:

Go to vpn client run the TAP2.py

1. Arping -I tap0 192.168.53.53, the IP address is reachable Address, we can find we capture the ARP request and reply an fake one.We can successfully see the program print out the ARP packet for asking the MAC address of 192.168.53.33 and also a fake arp packet (192.168.53.33) is revived.

```
root@a7d94dce1e8f:/# arping -I tap0 192.168.53.33
ARPING 192.168.53.33
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=0 time=
2.307 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=1 time=
3.629 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=2 time=
2.313 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=3 time=
1.270 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=4 time=
1.133 msec
^C
```

```
root@a7d94dce1e8f:/volumes# TAP2.py
Interface Name: tap0
--------------------------------
Ether / ARP who has 192.168.53.1 says 192.168.53.99
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.1
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.33
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.33
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.33
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.33
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.33
```

2. Arping -I tap0 1.2.3.4, the IP address is nonexit IP Address, we can find we capture the ARP request and reply an fake one.We can successfully see the program print out the ARP packet for asking the MAC address of 1.2.3.4 and also a fake arp packet (1.2.3.4) is revived.