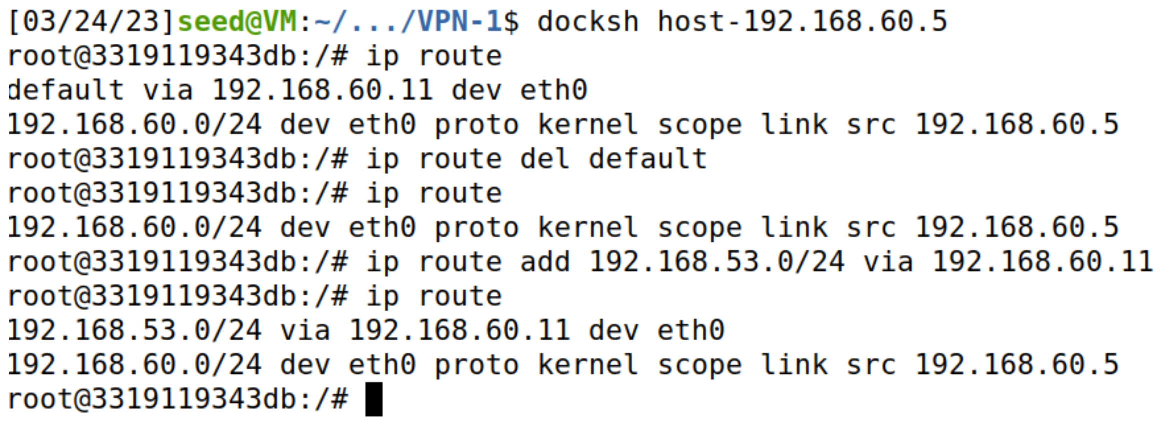
Name- Harshit Agrawal SUID: 912538842

VPN Part-2

Task 7: Routing Experiment on Host V:

In this we set up the connection on Host U and Host V side.

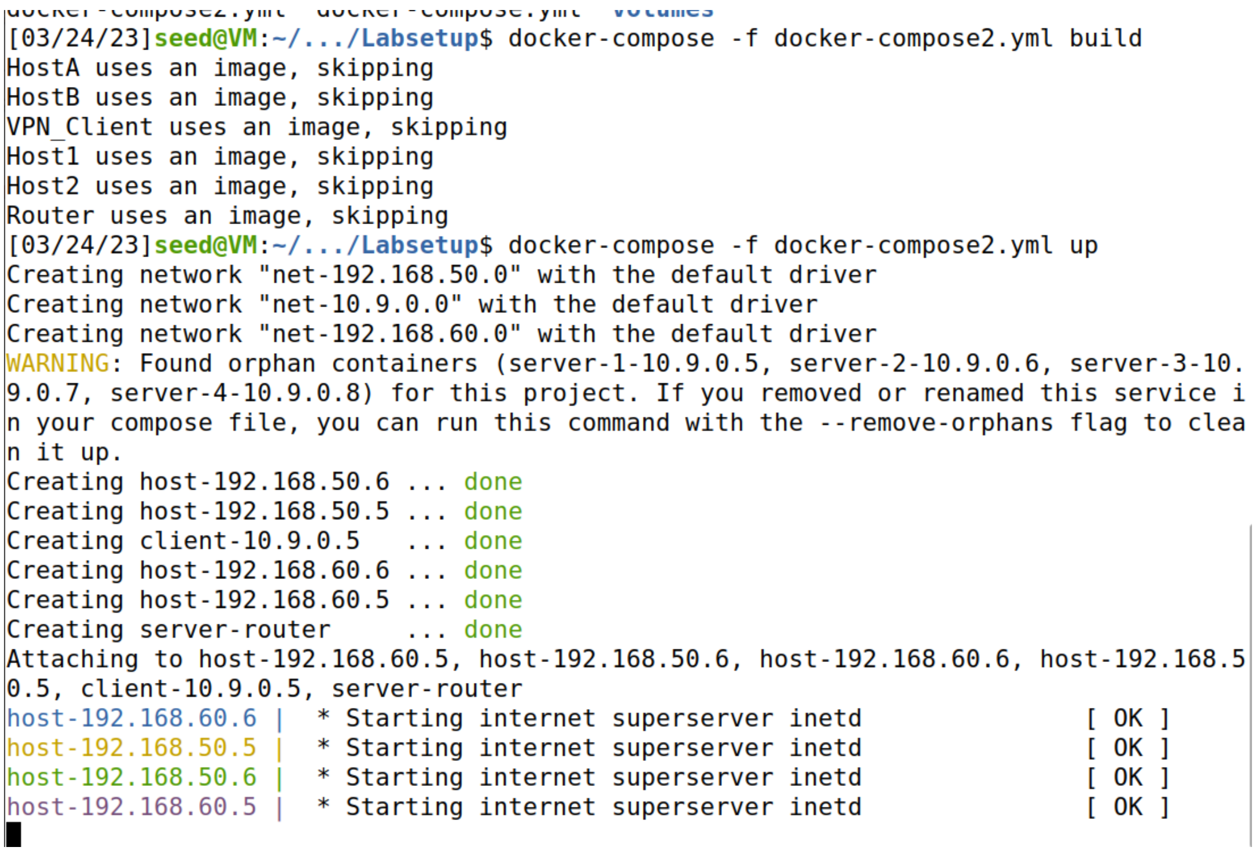


```
[03/24/23]seed@VM:~/.../VPN-1$ docksh host-192.168.60.5
root@3319119343db:/# ip route
default via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@3319119343db:/# ip route del default
root@3319119343db:/# ip route
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@3319119343db:/# ip route add 192.168.53.0/24 via 192.168.60.11
root@3319119343db:/# 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@3319119343db:/# 
```

We saw that the default route is via 192.168.60.11. Then We run the command “ip route add” to add a specific route. We run the code again to check for the route as shown above and we can see that the execution was successful.

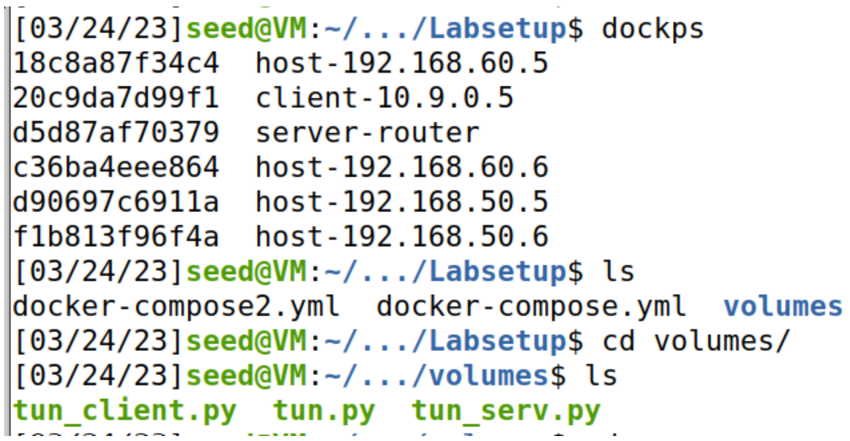
Task 8: VPN Between Private Networks:

Using the "-f docker-compose2.yml" option to ask docker-compose to use this file, instead of the default docker-compose.yml file



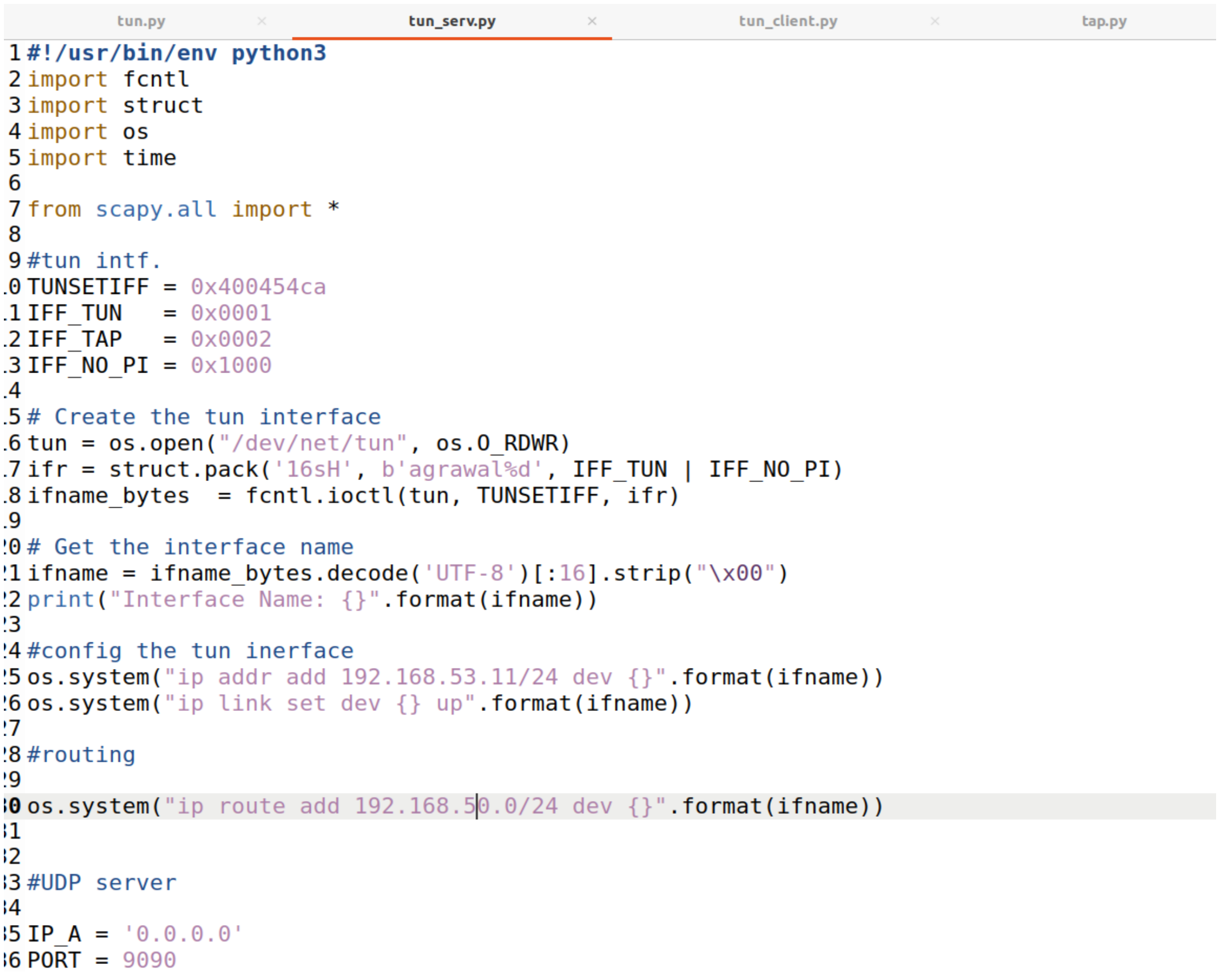
```
[03/24/23]seed@VM:~/.../Labsetup$ docker-compose -f docker-compose2.yml build
HostA uses an image, skipping
HostB uses an image, skipping
VPN_Client uses an image, skipping
Host1 uses an image, skipping
Host2 uses an image, skipping
Router uses an image, skipping
[03/24/23]seed@VM:~/.../Labsetup$ docker-compose -f docker-compose2.yml up
Creating network "net-192.168.50.0" with the default driver
Creating network "net-10.9.0.0" with the default driver
Creating network "net-192.168.60.0" with the default driver
WARNING: Found orphan containers (server-1-10.9.0.5, server-2-10.9.0.6, server-3-10.
9.0.7, server-4-10.9.0.8) for this project. If you removed or renamed this service i
n your compose file, you can run this command with the --remove-orphans flag to clea
n it up.
Creating host-192.168.50.6 ... done
Creating host-192.168.50.5 ... done
Creating client-10.9.0.5   ... done
Creating host-192.168.60.6 ... done
Creating host-192.168.60.5 ... done
Creating server-router     ... done
Attaching to host-192.168.60.5, host-192.168.50.6, host-192.168.60.6, host-192.168.5
0.5, client-10.9.0.5, server-router
host-192.168.60.6 |  * Starting internet superserver inetd               [ OK ]
host-192.168.50.5 |  * Starting internet superserver inetd               [ OK ]
host-192.168.50.6 |  * Starting internet superserver inetd               [ OK ]
host-192.168.60.5 |  * Starting internet superserver inetd               [ OK ]
```

Below are all the active containers-

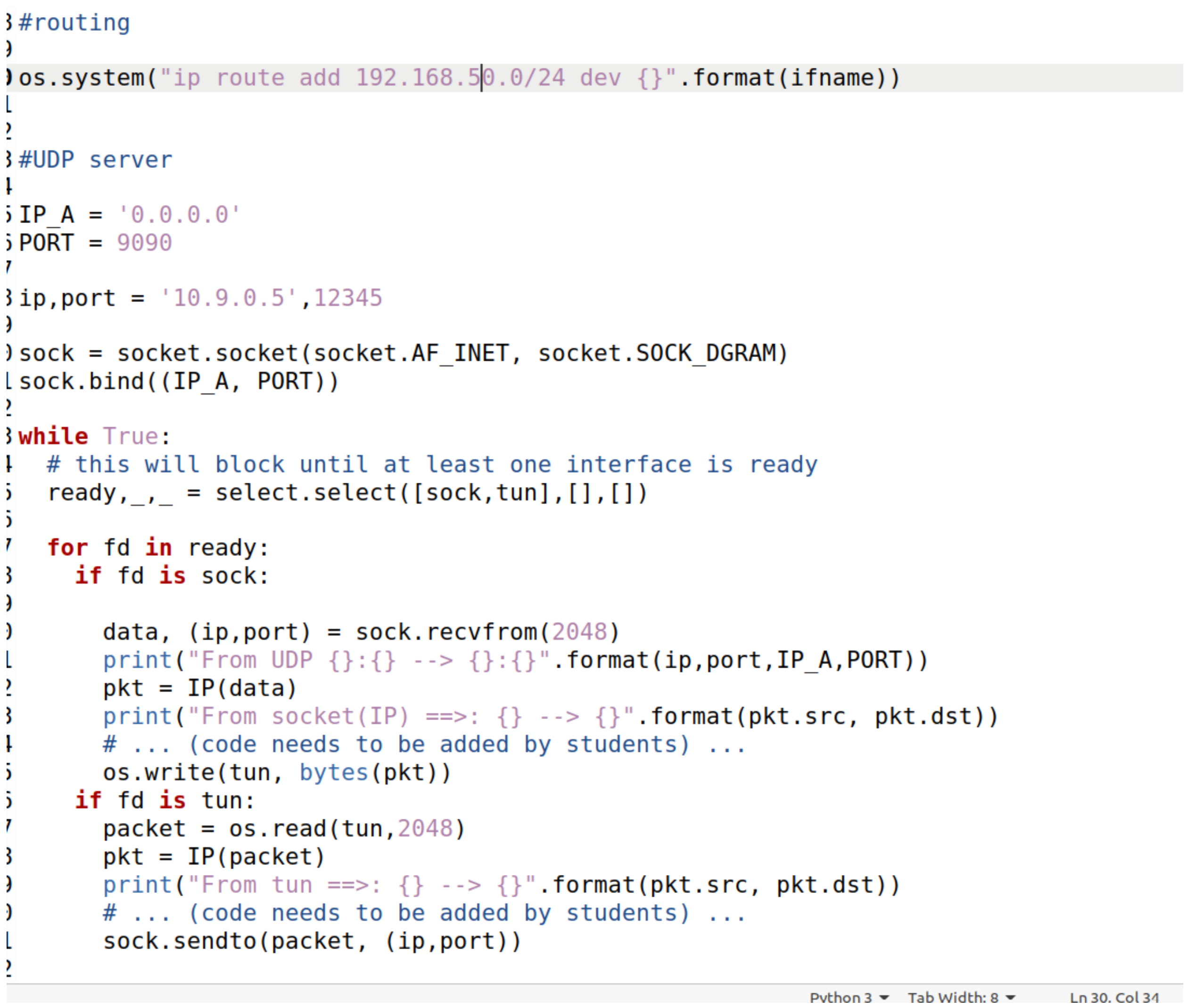


```
[03/24/23]seed@VM:~/.../Labsetup$ dockps
18c8a87f34c4  host-192.168.60.5
20c9da7d99f1  client-10.9.0.5
d5d87af70379  server-router
c36ba4eee864  host-192.168.60.6
d90697c6911a  host-192.168.50.5
f1b813f96f4a  host-192.168.50.6
[03/24/23]seed@VM:~/.../Labsetup$ ls
docker-compose2.yml  docker-compose.yml  volumes
[03/24/23]seed@VM:~/.../Labsetup$ cd volumes/
[03/24/23]seed@VM:~/.../volumes$ ls
tun_client.py  tun.py  tun_serv.py
```

The code for tun_serv.py can be seen below-



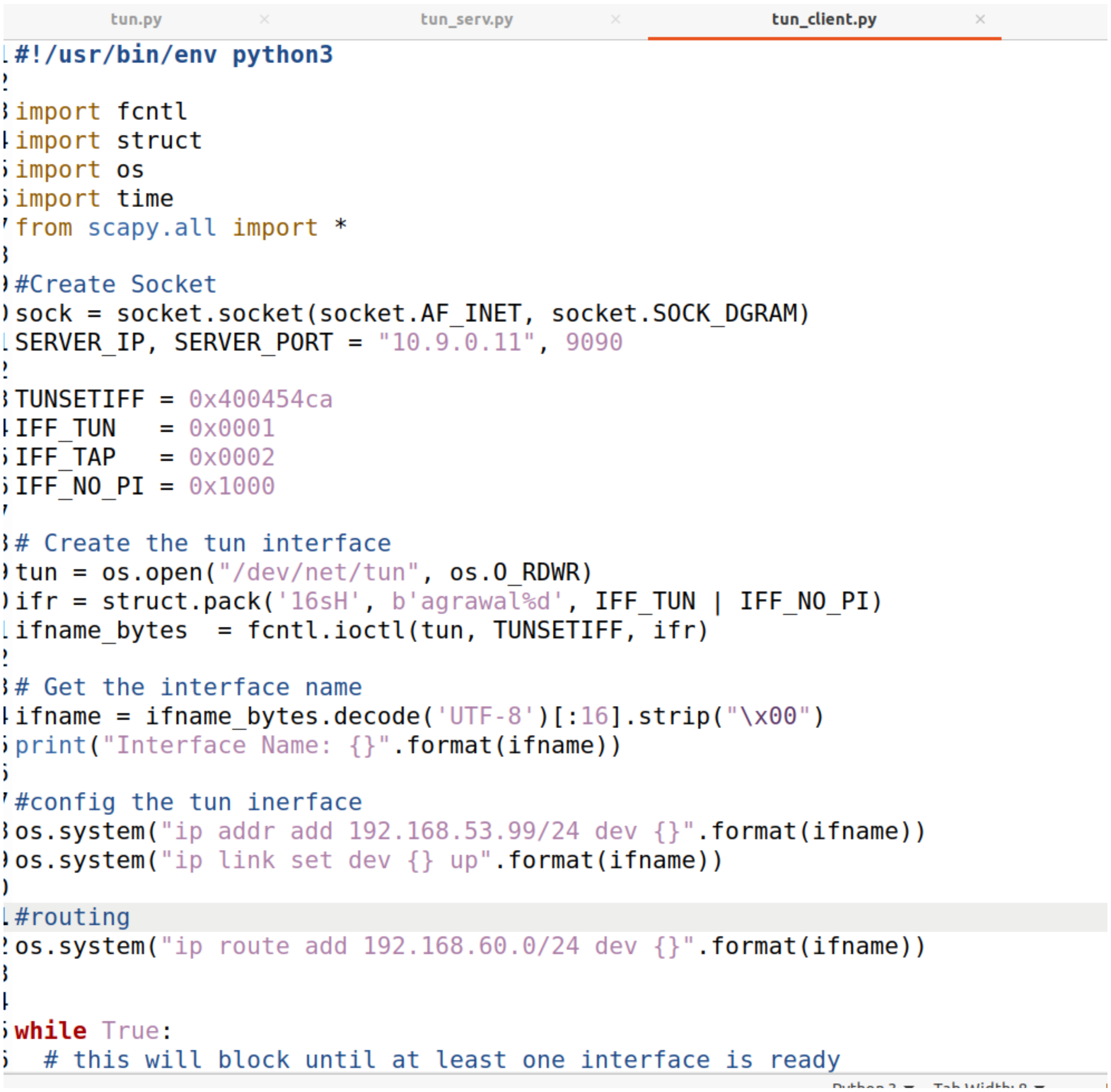
```python
#!/usr/bin/env python3
import fcntl
import struct
import os
import time

from scapy.all import *

#tun intf.
TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

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

#config the tun inerface
os.system("ip addr add 192.168.53.11/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

#routing

os.system("ip route add 192.168.50.0/24 dev {}".format(ifname))


#UDP server

IP_A = '0.0.0.0'
PORT = 9090
```



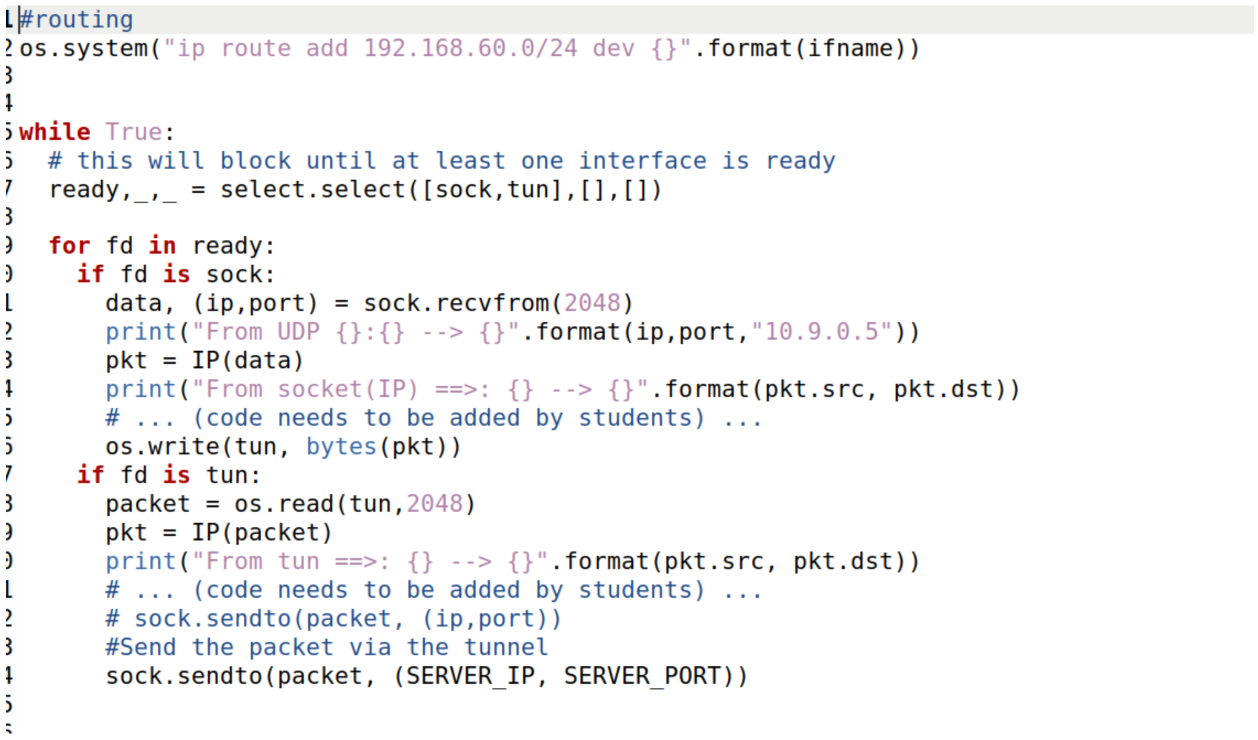
```
#routing

os.system("ip route add 192.168.50.0/24 dev {}".format(ifname))


#UDP server

IP_A = '0.0.0.0'
PORT = 9090

ip,port = '10.9.0.5',12345

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

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

  for fd in ready:
    if fd is sock:

      data, (ip,port) = sock.recvfrom(2048)
      print("From UDP {}:{} --> {}:{}".format(ip,port,IP_A,PORT))
      pkt = IP(data)
      print("From socket(IP) ==>: {} --> {}".format(pkt.src, pkt.dst))
      # ... (code needs to be added by students) ...
      os.write(tun, bytes(pkt))
    if fd is tun:
      packet = os.read(tun,2048)
      pkt = IP(packet)
      print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
      # ... (code needs to be added by students) ...
      sock.sendto(packet, (ip,port))

```

Python 3 ▾ Tab Width: 8 ▾ Ln 30, Col 34

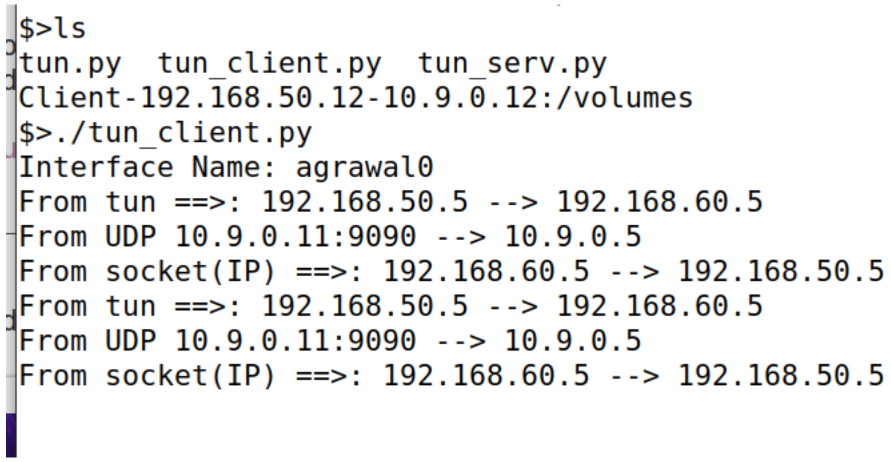
The code for tun_client.py can be seen below:

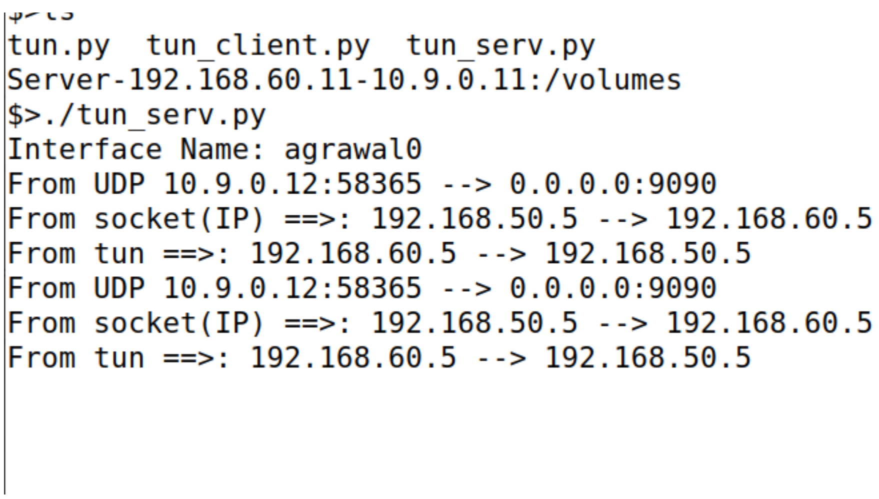


```
#!/usr/bin/env python3

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

#Create Socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
SERVER_IP, SERVER_PORT = "10.9.0.11", 9090

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

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

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

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


while True:
  # this will block until at least one interface is ready
```


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


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

  for fd in ready:
    if fd is sock:
      data, (ip,port) = sock.recvfrom(2048)
      print("From UDP {}:{} --> {}".format(ip,port,"10.9.0.5"))
      pkt = IP(data)
      print("From socket(IP) ==>: {} --> {}".format(pkt.src, pkt.dst))
      # ... (code needs to be added by students) ...
      os.write(tun, bytes(pkt))
    if fd is tun:
      packet = os.read(tun,2048)
      pkt = IP(packet)
      print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
      # ... (code needs to be added by students) ...
      # sock.sendto(packet, (ip,port))
      #Send the packet via the tunnel
      sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

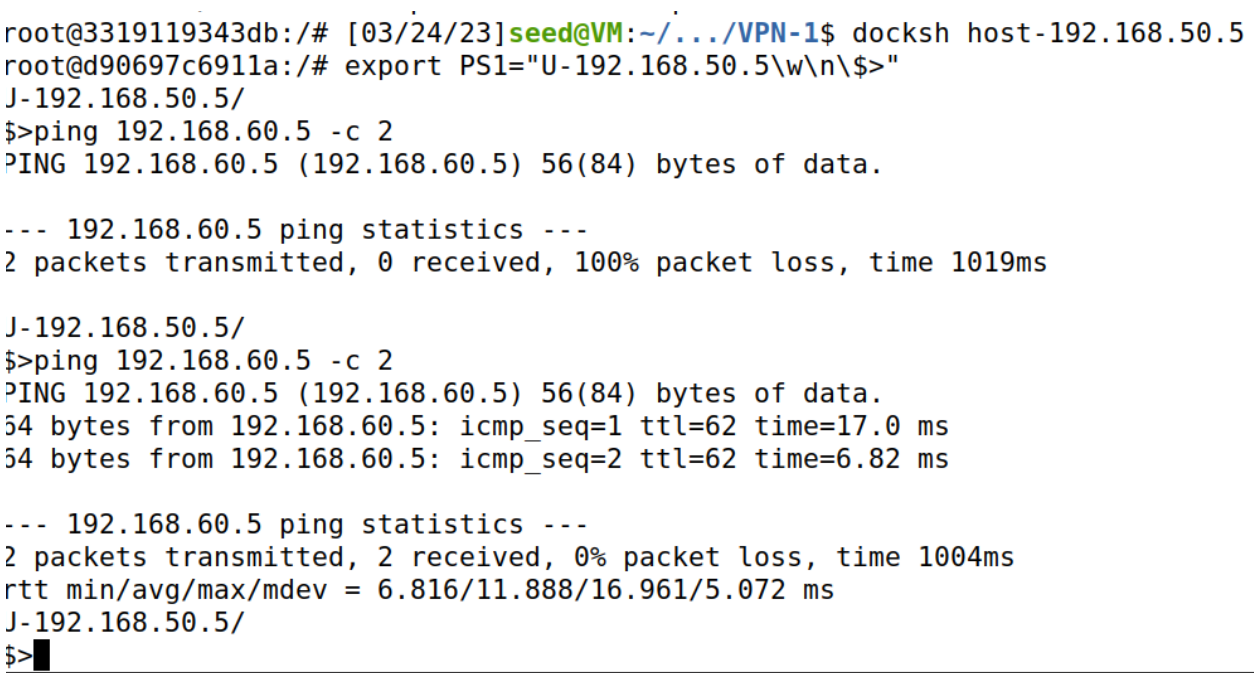
In the client code, we route the packet so that they can be routed through a private network as well such as 192.168.60.0/24.


```
$>ls
tun.py  tun_client.py  tun_serv.py
Client-192.168.50.12-10.9.0.12:/volumes
$>./tun_client.py
Interface Name: agrawal0
From tun ==>: 192.168.50.5 --> 192.168.60.5
From UDP 10.9.0.11:9090 --> 10.9.0.5
From socket(IP) ==>: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From UDP 10.9.0.11:9090 --> 10.9.0.5
From socket(IP) ==>: 192.168.60.5 --> 192.168.50.5
```

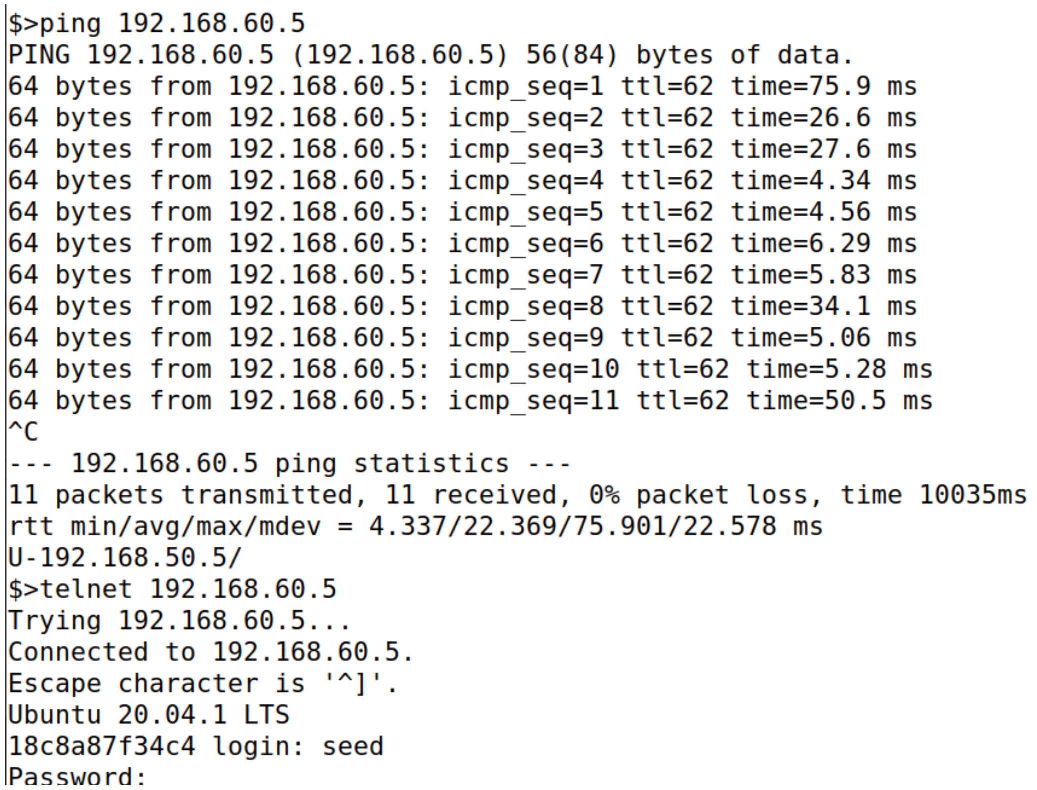


```
tun.py  tun_client.py  tun_serv.py
Server-192.168.60.11-10.9.0.11:/volumes
$>./tun_serv.py
Interface Name: agrawal0
From UDP 10.9.0.12:58365 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From UDP 10.9.0.12:58365 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
```

When we run the code, we can see that while pinging from Host U to Host V on a private network, we receive an ICMP reply back:



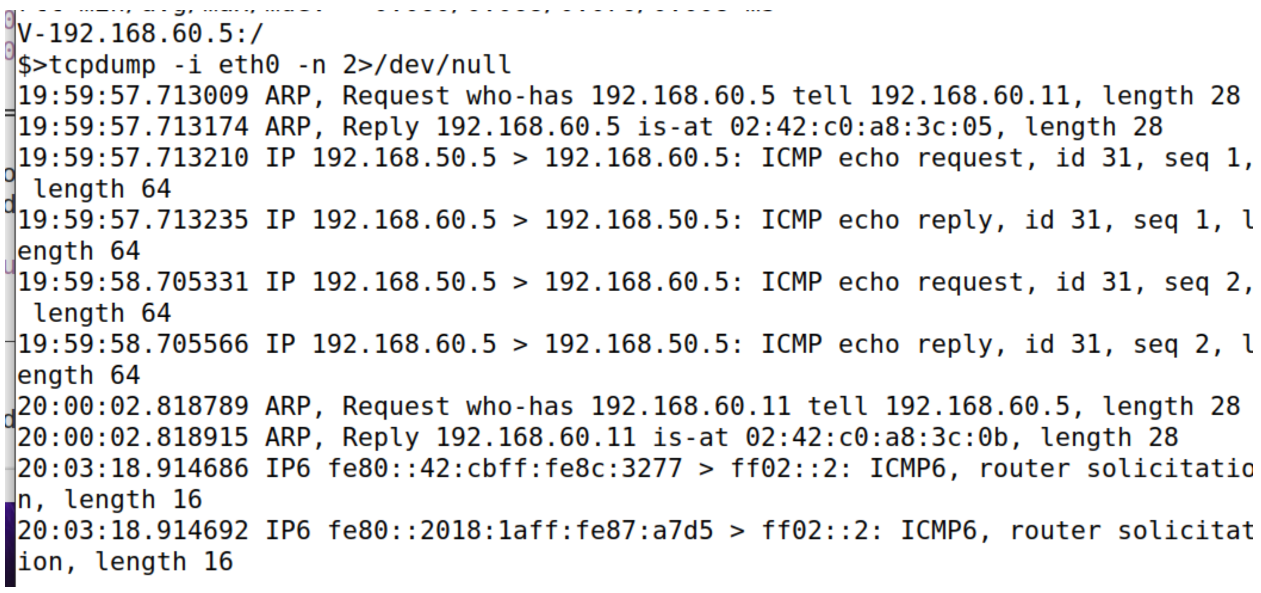
```
root@3319119343db:/# [03/24/23]seed@VM:~/.../VPN-1$ docksh host-192.168.50.5
root@d90697c6911a:/# export PS1="U-192.168.50.5\w\n\$>"
J-192.168.50.5/
$>ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1019ms

J-192.168.50.5/
$>ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
54 bytes from 192.168.60.5: icmp_seq=1 ttl=62 time=17.0 ms
54 bytes from 192.168.60.5: icmp_seq=2 ttl=62 time=6.82 ms

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1004ms
rtt min/avg/max/mdev = 6.816/11.888/16.961/5.072 ms
J-192.168.50.5/
$>
```



```
$>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=62 time=75.9 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=62 time=26.6 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=62 time=27.6 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=62 time=4.34 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=62 time=4.56 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=62 time=6.29 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=62 time=5.83 ms
64 bytes from 192.168.60.5: icmp_seq=8 ttl=62 time=34.1 ms
64 bytes from 192.168.60.5: icmp_seq=9 ttl=62 time=5.06 ms
64 bytes from 192.168.60.5: icmp_seq=10 ttl=62 time=5.28 ms
64 bytes from 192.168.60.5: icmp_seq=11 ttl=62 time=50.5 ms
^C
--- 192.168.60.5 ping statistics ---
11 packets transmitted, 11 received, 0% packet loss, time 10035ms
rtt min/avg/max/mdev = 4.337/22.369/75.901/22.578 ms
U-192.168.50.5/
$>telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
18c8a87f34c4 login: seed
Password:
```

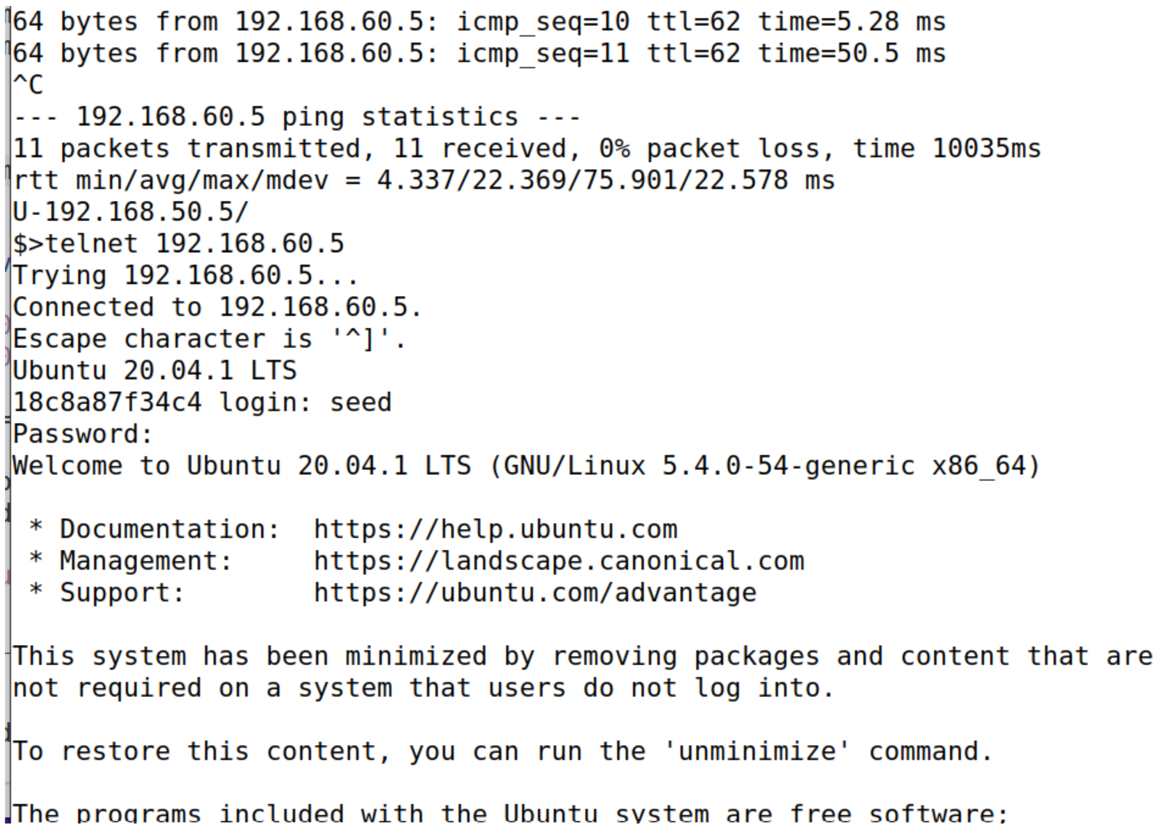
To test the connection, we break the connection as done in one of the previous tasks and connect again to see that connection successfully starts again: This can be seen using the ICMP sequence number gap during connection loss.

Here, in the above screenshot we can see that the connection was successful as we were successfully able to ping.



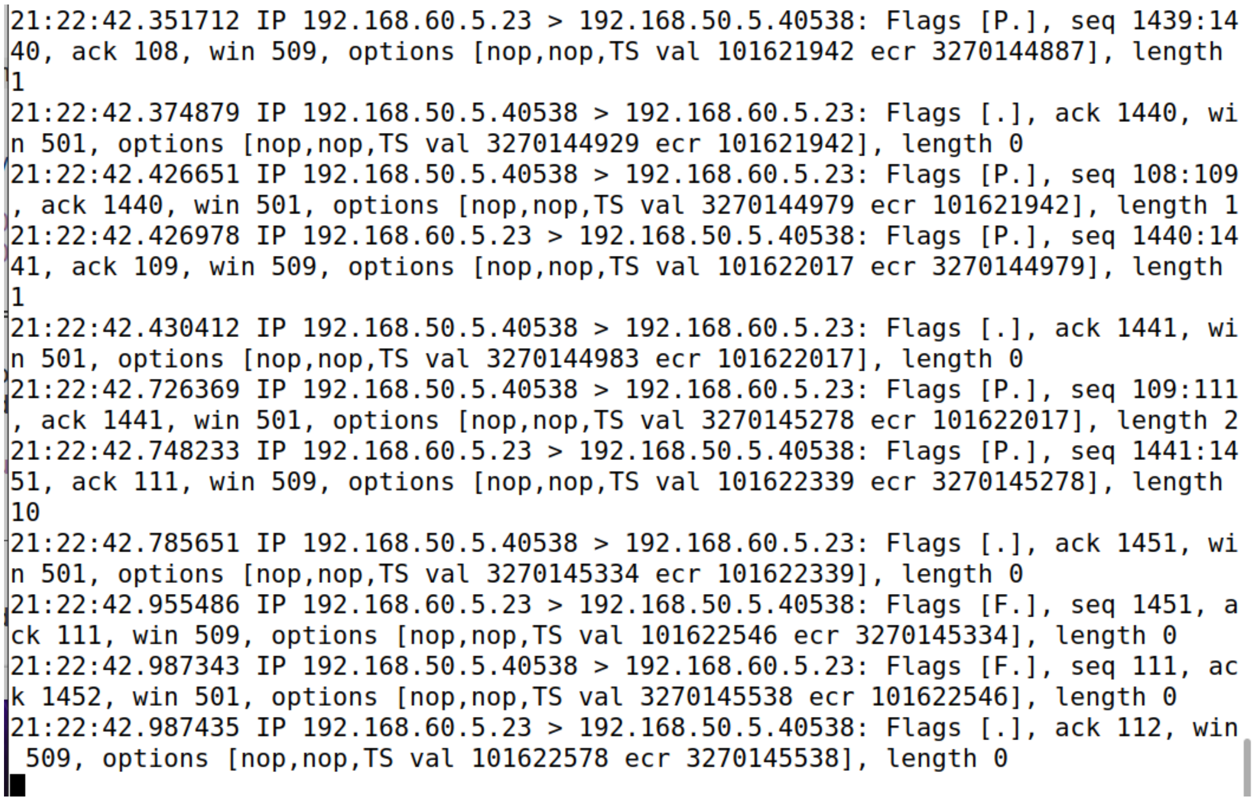
```
V-192.168.60.5:/
$>tcpdump -i eth0 -n 2>/dev/null
19:59:57.713009 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
19:59:57.713174 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
19:59:57.713210 IP 192.168.50.5 > 192.168.60.5: ICMP echo request, id 31, seq 1,
 length 64
19:59:57.713235 IP 192.168.60.5 > 192.168.50.5: ICMP echo reply, id 31, seq 1, l
ength 64
19:59:58.705331 IP 192.168.50.5 > 192.168.60.5: ICMP echo request, id 31, seq 2,
 length 64
19:59:58.705566 IP 192.168.60.5 > 192.168.50.5: ICMP echo reply, id 31, seq 2, l
ength 64
20:00:02.818789 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
20:00:02.818915 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
20:03:18.914686 IP6 fe80::42:cbff:fe8c:3277 > ff02::2: ICMP6, router solicitatio
n, length 16
20:03:18.914692 IP6 fe80::2018:1aff:fe87:a7d5 > ff02::2: ICMP6, router solicitat
ion, length 16
```

Above, we can see the TCP dump of the successful connection.



```
64 bytes from 192.168.60.5: icmp_seq=10 ttl=62 time=5.28 ms
64 bytes from 192.168.60.5: icmp_seq=11 ttl=62 time=50.5 ms
^C
--- 192.168.60.5 ping statistics ---
11 packets transmitted, 11 received, 0% packet loss, time 10035ms
rtt min/avg/max/mdev = 4.337/22.369/75.901/22.578 ms
U-192.168.50.5/
$>telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
18c8a87f34c4 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

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

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

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

The programs included with the Ubuntu system are free software;
```

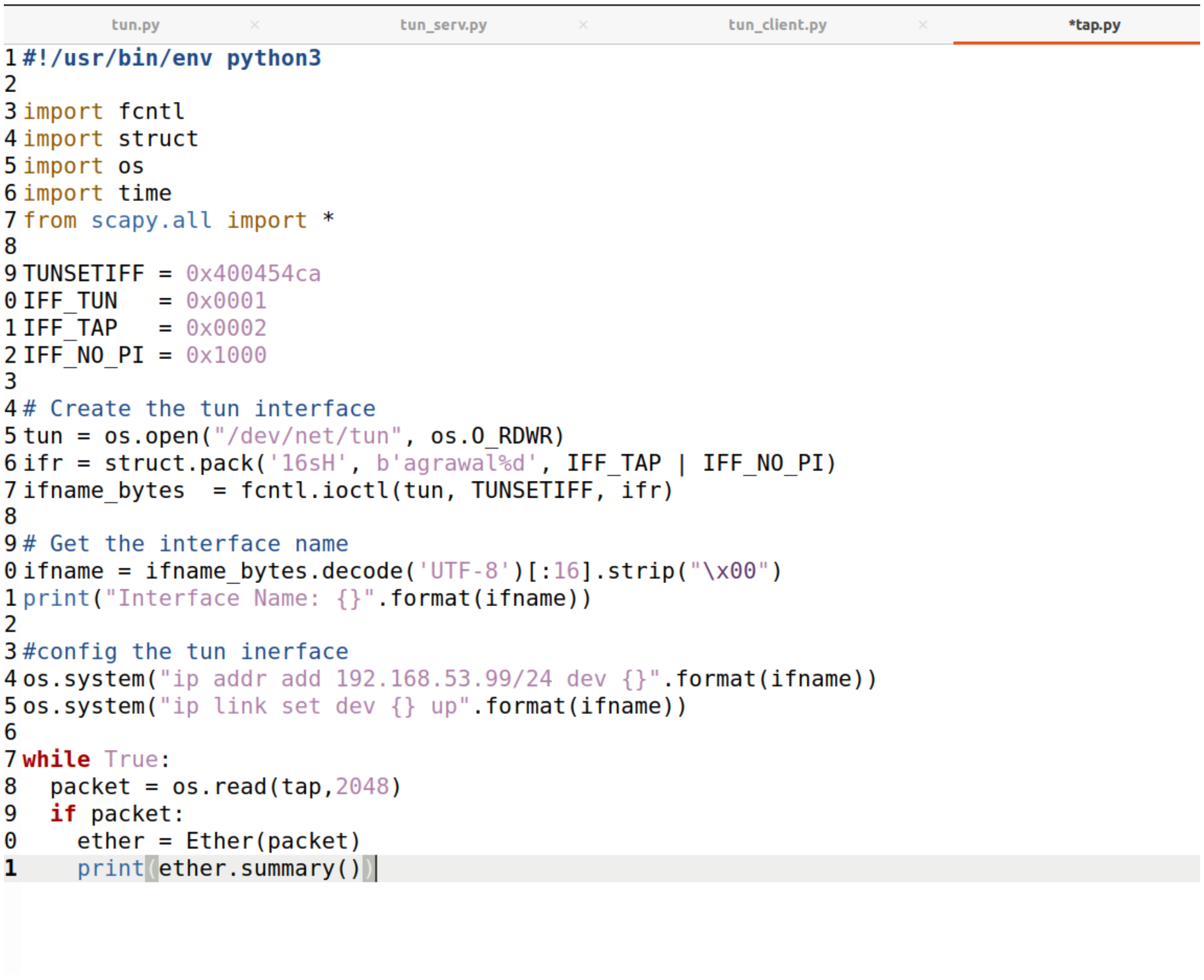
And, a successful telnet connection, above.



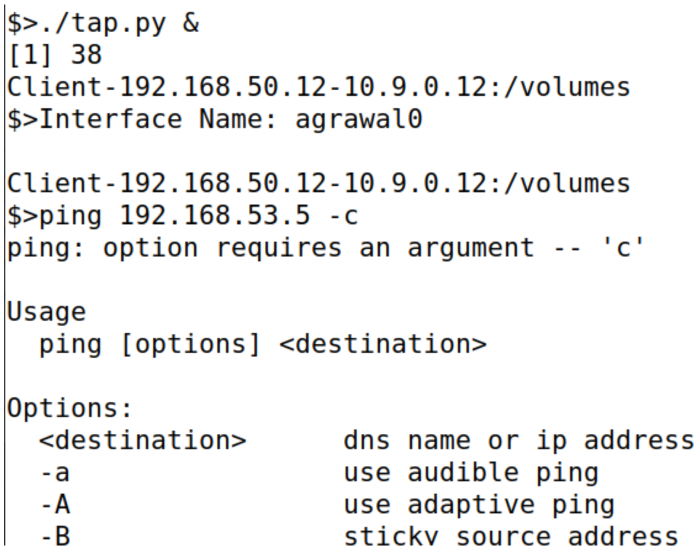
```
21:22:42.351712 IP 192.168.60.5.23 > 192.168.50.5.40538: Flags [P.], seq 1439:14
40, ack 108, win 509, options [nop,nop,TS val 101621942 ecr 3270144887], length
1
21:22:42.374879 IP 192.168.50.5.40538 > 192.168.60.5.23: Flags [.], ack 1440, wi
n 501, options [nop,nop,TS val 3270144929 ecr 101621942], length 0
21:22:42.426651 IP 192.168.50.5.40538 > 192.168.60.5.23: Flags [P.], seq 108:109
, ack 1440, win 501, options [nop,nop,TS val 3270144979 ecr 101621942], length 1
21:22:42.426978 IP 192.168.60.5.23 > 192.168.50.5.40538: Flags [P.], seq 1440:14
41, ack 109, win 509, options [nop,nop,TS val 101622017 ecr 3270144979], length
1
21:22:42.430412 IP 192.168.50.5.40538 > 192.168.60.5.23: Flags [.], ack 1441, wi
n 501, options [nop,nop,TS val 3270144983 ecr 101622017], length 0
21:22:42.726369 IP 192.168.50.5.40538 > 192.168.60.5.23: Flags [P.], seq 109:111
, ack 1441, win 501, options [nop,nop,TS val 3270145278 ecr 101622017], length 2
21:22:42.748233 IP 192.168.60.5.23 > 192.168.50.5.40538: Flags [P.], seq 1441:14
51, ack 111, win 509, options [nop,nop,TS val 101622339 ecr 3270145278], length
10
21:22:42.785651 IP 192.168.50.5.40538 > 192.168.60.5.23: Flags [.], ack 1451, wi
n 501, options [nop,nop,TS val 3270145334 ecr 101622339], length 0
21:22:42.955486 IP 192.168.60.5.23 > 192.168.50.5.40538: Flags [F.], seq 1451, a
ck 111, win 509, options [nop,nop,TS val 101622546 ecr 3270145334], length 0
21:22:42.987343 IP 192.168.50.5.40538 > 192.168.60.5.23: Flags [F.], seq 111, ac
k 1452, win 501, options [nop,nop,TS val 3270145538 ecr 101622546], length 0
21:22:42.987435 IP 192.168.60.5.23 > 192.168.50.5.40538: Flags [.], ack 112, win
 509, options [nop,nop,TS val 101622578 ecr 3270145538], length 0
```

Task 9: Experiment with the TAP Interface:

The code for Client interface will be same. The code for tap interface used is given below:

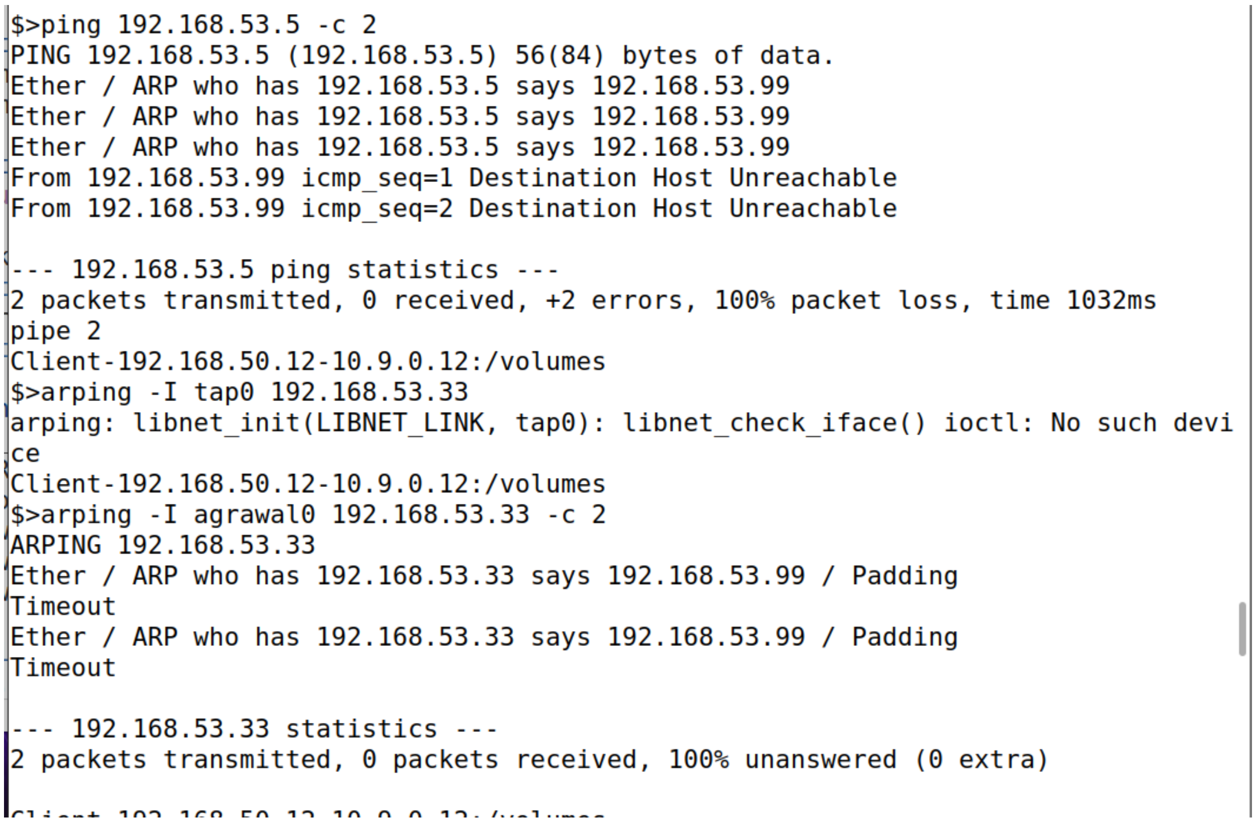


```
#!/usr/bin/env python3

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

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

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

#config the tun inerface
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())
```



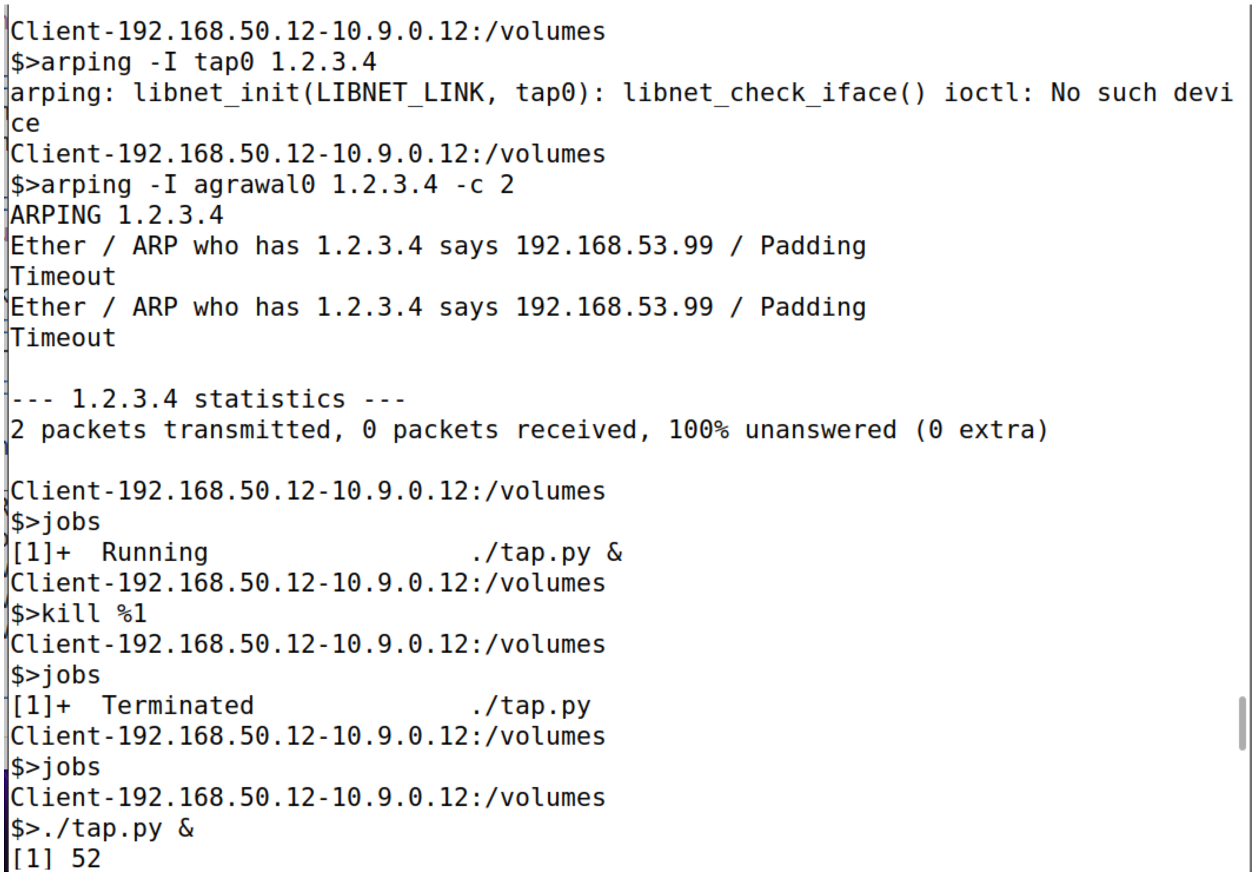
```
$>./tap.py &
[1] 38
Client-192.168.50.12-10.9.0.12:/volumes
$>Interface Name: agrawal0

Client-192.168.50.12-10.9.0.12:/volumes
$>ping 192.168.53.5 -c
ping: option requires an argument -- 'c'

Usage
  ping [options] <destination>

Options:
  <destination>       dns name or ip address
  -a                  use audible ping
  -A                  use adaptive ping
  -B                  stickv source address
```

By using the above code we are successfully able to simply read from the TAP interface. It then casts the data to a Scapy Ether object, and prints out all its fields. We tried to ping an IP address in the 192.168.53.5 network.



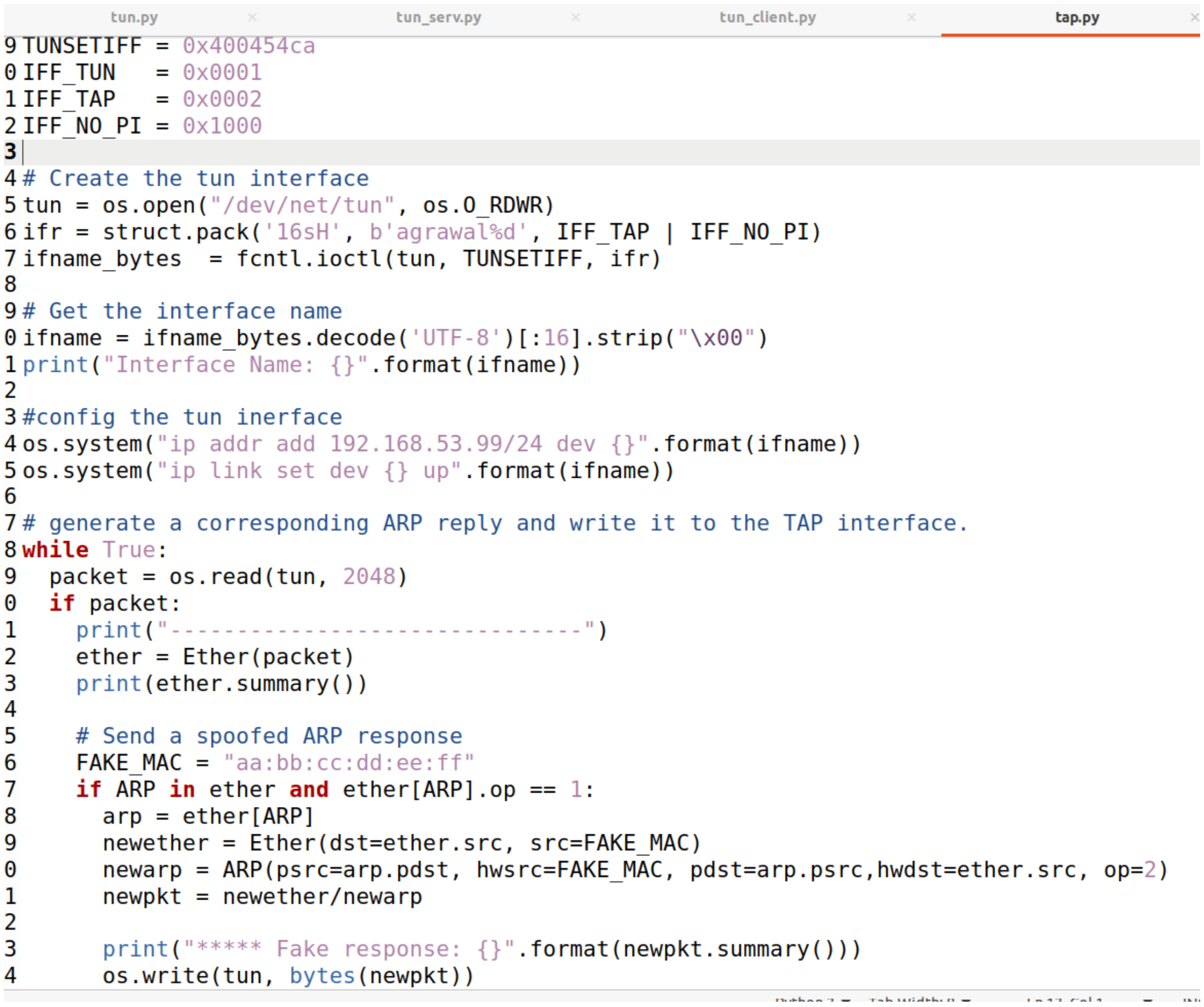
```
$>ping 192.168.53.5 -c 2
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
From 192.168.53.99 icmp_seq=1 Destination Host Unreachable
From 192.168.53.99 icmp_seq=2 Destination Host Unreachable

--- 192.168.53.5 ping statistics ---
2 packets transmitted, 0 received, +2 errors, 100% packet loss, time 1032ms
pipe 2
Client-192.168.50.12-10.9.0.12:/volumes
$>arping -I tap0 192.168.53.33
arping: libnet_init(LIBNET_LINK, tap0): libnet_check_iface() ioctl: No such devi
ce
Client-192.168.50.12-10.9.0.12:/volumes
$>arping -I agrawal0 192.168.53.33 -c 2
ARPING 192.168.53.33
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
Timeout
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
Timeout

--- 192.168.53.33 statistics ---
2 packets transmitted, 0 packets received, 100% unanswered (0 extra)
```

The interface routes the ARP packets to tunnel interface but is not sure which machine, as that IP address. Therefore, sending the destination host an unreachable message.



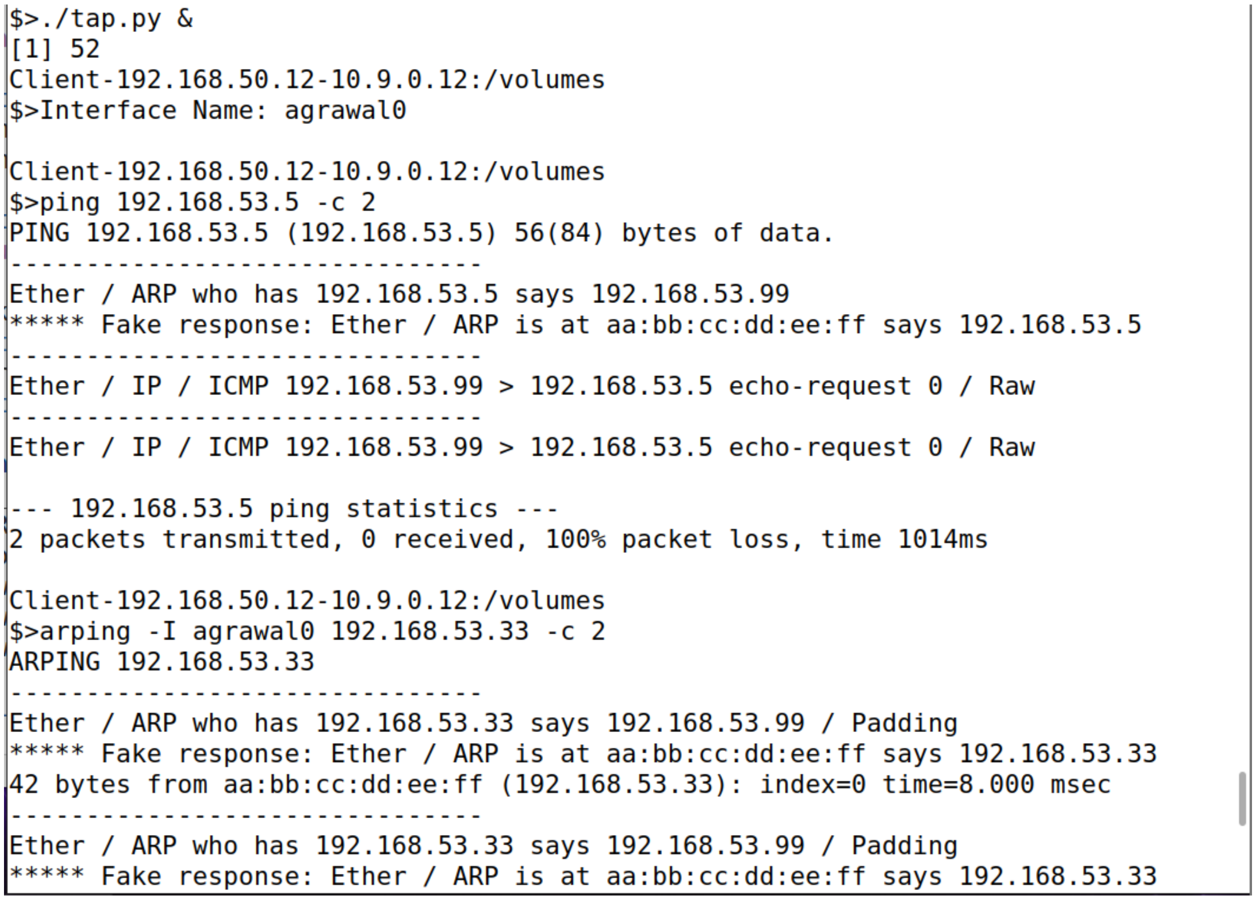
```
Client-192.168.50.12-10.9.0.12:/volumes
$>arping -I tap0 1.2.3.4
arping: libnet_init(LIBNET_LINK, tap0): libnet_check_iface() ioctl: No such devi
ce
Client-192.168.50.12-10.9.0.12:/volumes
$>arping -I agrawal0 1.2.3.4 -c 2
ARPING 1.2.3.4
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
Timeout
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
Timeout

--- 1.2.3.4 statistics ---
2 packets transmitted, 0 packets received, 100% unanswered (0 extra)

Client-192.168.50.12-10.9.0.12:/volumes
$>jobs
[1]+  Running                 ./tap.py &
Client-192.168.50.12-10.9.0.12:/volumes
$>kill %1
Client-192.168.50.12-10.9.0.12:/volumes
$>jobs
[1]+  Terminated              ./tap.py
Client-192.168.50.12-10.9.0.12:/volumes
$>jobs
Client-192.168.50.12-10.9.0.12:/volumes
$>./tap.py &
[1] 52
```

Now, we modify the code to send spoofed ARP packets from a fake MAC address:

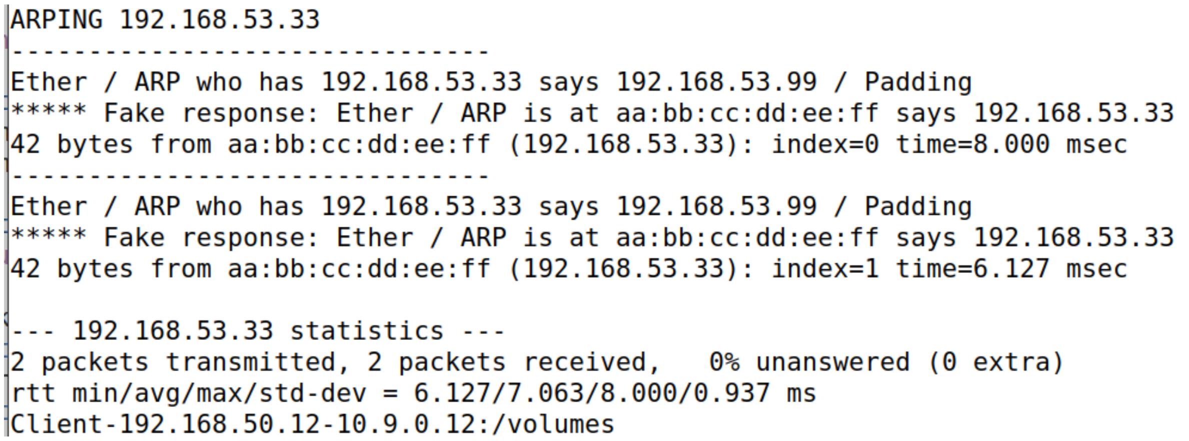


```
TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

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

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

# generate a corresponding ARP reply and write it to the TAP interface.
while True:
  packet = os.read(tun, 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(tun, bytes(newpkt))
```

Now, we try to run the code and try to arping to IP address 192.168.53.33 first. Therefore, we can see that we received a spoofed message replies from the MAC address sent for each request sent.

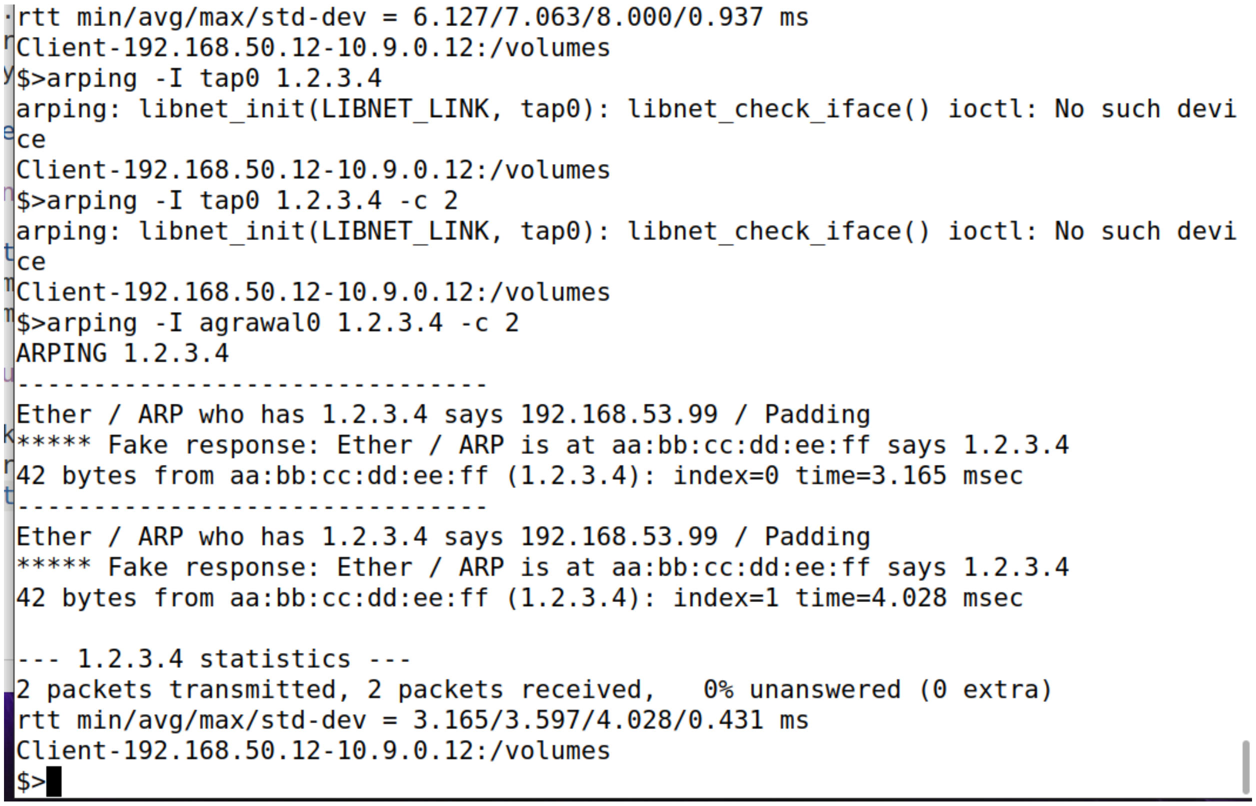


```
$>./tap.py &
[1] 52
Client-192.168.50.12-10.9.0.12:/volumes
$>Interface Name: agrawal0

Client-192.168.50.12-10.9.0.12:/volumes
$>ping 192.168.53.5 -c 2
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
-------------------------------
Ether / ARP who has 192.168.53.5 says 192.168.53.99
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.5
-------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
-------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw

--- 192.168.53.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1014ms

Client-192.168.50.12-10.9.0.12:/volumes
$>arping -I agrawal0 192.168.53.33 -c 2
ARPING 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
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=0 time=8.000 msec
-------------------------------
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
```



```
ARPING 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
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=0 time=8.000 msec
-------------------------------
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
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=1 time=6.127 msec

--- 192.168.53.33 statistics ---
2 packets transmitted, 2 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 6.127/7.063/8.000/0.937 ms
Client-192.168.50.12-10.9.0.12:/volumes
```

Now, we try arping to IP address 1.2.3.4. As, can be seen below that we received a spoofed message replies from the MAC address sent for each request sent:



```
rtt min/avg/max/std-dev = 6.127/7.063/8.000/0.937 ms
Client-192.168.50.12-10.9.0.12:/volumes
$>arping -I tap0 1.2.3.4
arping: libnet_init(LIBNET_LINK, tap0): libnet_check_iface() ioctl: No such devi
ce
Client-192.168.50.12-10.9.0.12:/volumes
$>arping -I tap0 1.2.3.4 -c 2
arping: libnet_init(LIBNET_LINK, tap0): libnet_check_iface() ioctl: No such devi
ce
Client-192.168.50.12-10.9.0.12:/volumes
$>arping -I agrawal0 1.2.3.4 -c 2
ARPING 1.2.3.4
-------------------------------
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1.2.3.4
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=0 time=3.165 msec
-------------------------------
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1.2.3.4
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=1 time=4.028 msec

--- 1.2.3.4 statistics ---
2 packets transmitted, 2 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 3.165/3.597/4.028/0.431 ms
Client-192.168.50.12-10.9.0.12:/volumes
$>
```