Name: Dhanashree Reddy Srinivasa Reddy

SUID: 393473169

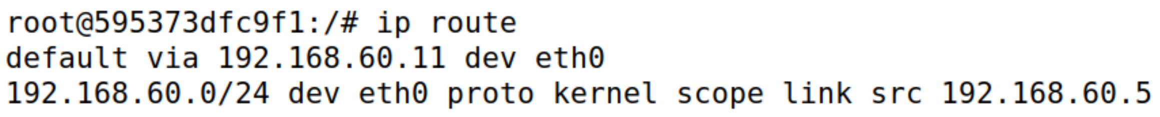
NetId: dsriniva@syr.edu

Course: Internet Security

Task 7: Routing experiment on Host V

First, we will check if the return packets are routed back to the vpn server. We will remove the default entry from Host V and will add an entry to the routing table.

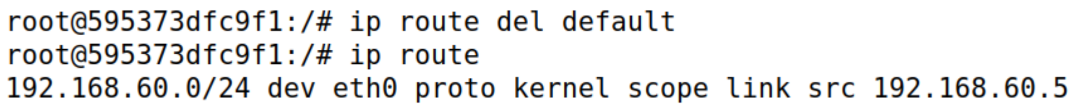
Command: ip route



```
root@595373dfc9f1:/# 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
```

Command: ip route del default

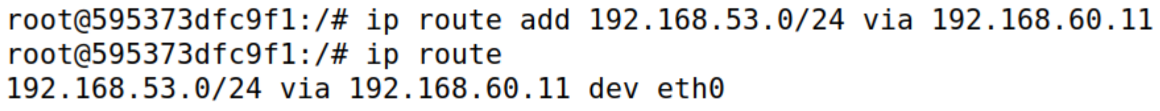
Next delete the default entry from the routing table of the machine Host V and then try to ping the Host V machine from another machine host U 10.9.0.5.



```
root@595373dfc9f1:/# ip route del default
root@595373dfc9f1:/# ip route
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
```

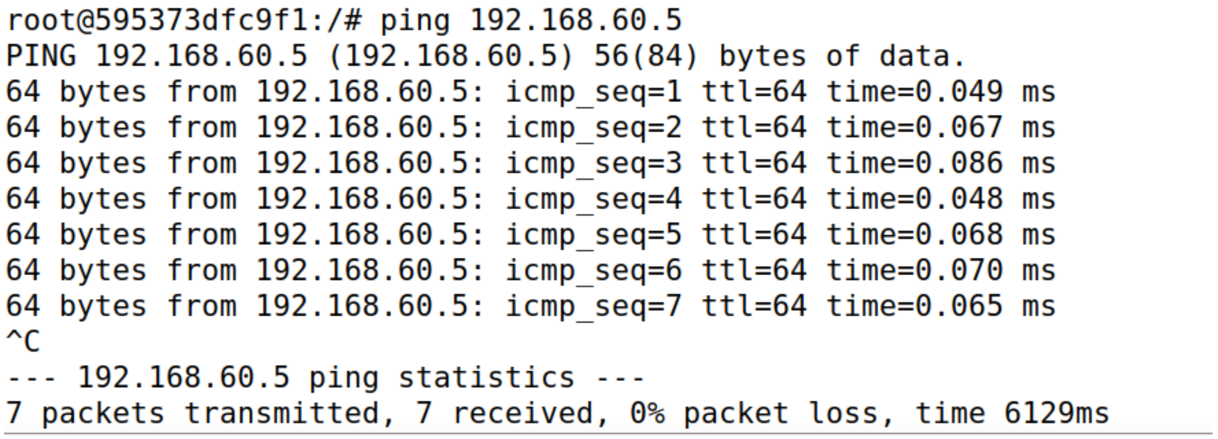
To add an entry to the routing table.

Command: ip route add



```
root@595373dfc9f1:/# ip route add 192.168.53.0/24 via 192.168.60.11
root@595373dfc9f1:/# ip route
192.168.53.0/24 via 192.168.60.11 dev eth0
```

Then try to ping the Host-V machine from another machine Host-U 10.9,.0.5. But as you can see that packets were going in the single direction, not from the Host-V to Host-U because of the Routing table. Then I changed the routing table on the Machine Host-V and then try again. It worked as you can see in the pictures.


```
root@595373dfc9f1:/# 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=0.049 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.067 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=64 time=0.086 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=64 time=0.048 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=64 time=0.068 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=64 time=0.070 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=64 time=0.065 ms
^C
--- 192.168.60.5 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6129ms
```

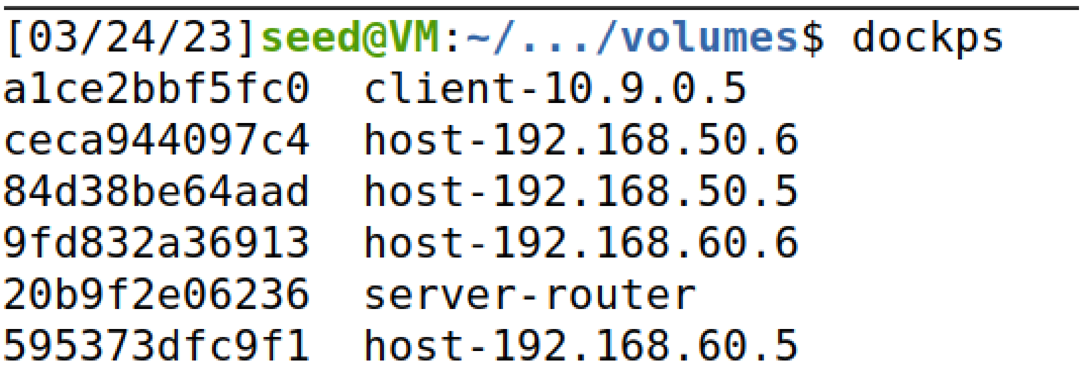
The above output shows the routing table of the Host V machine. Initially removed the default entry from the routing table and then add an entry for the ip address belonging to the network “192.168.53.0/24” which is the network id of the TUN interface. After this ping command successfully worked on the machine “client-10.9.0.5 / Host-U”. Now after adding a new entry in the routing table, the packets that are sent to the address 192.168.53.0/24 particular address will all be routed towards the gateway 192.168.60.11

Task 8: VPN between Private Networks

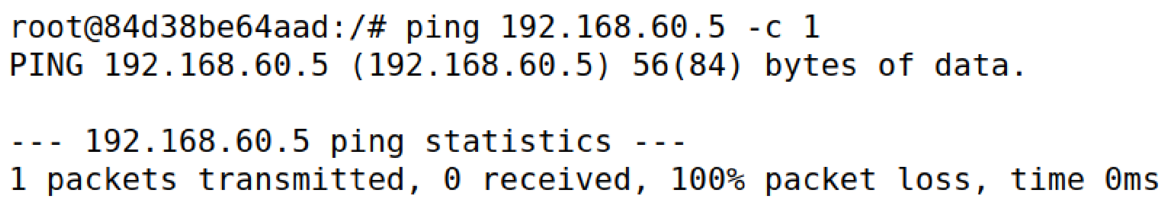
Build the docker environment using the docker-composer2.yml file

Command to build: docker-compose –f docker-compose2.yml build

Below are the containers:


```
[03/24/23]seed@VM:~/.../volumes$ dockps
a1ce2bbf5fc0  client-10.9.0.5
ceca944097c4  host-192.168.50.6
84d38be64aad  host-192.168.50.5
9fd832a36913  host-192.168.60.6
20b9f2e06236  server-router
595373dfc9f1  host-192.168.60.5
```

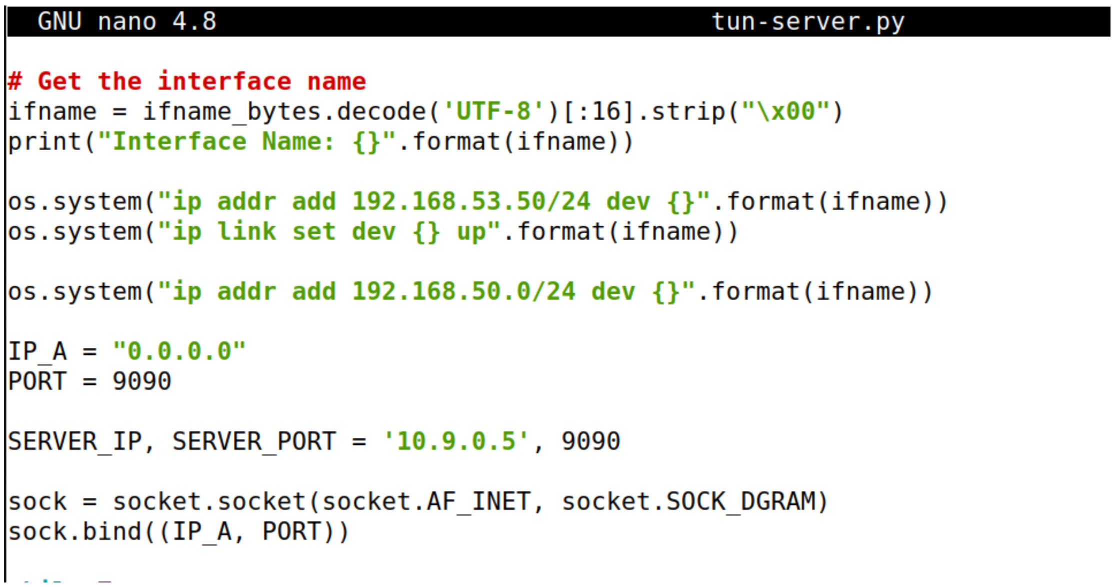
Host container: ping 192.168.60.5



```
root@84d38be64aad:/# 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
```

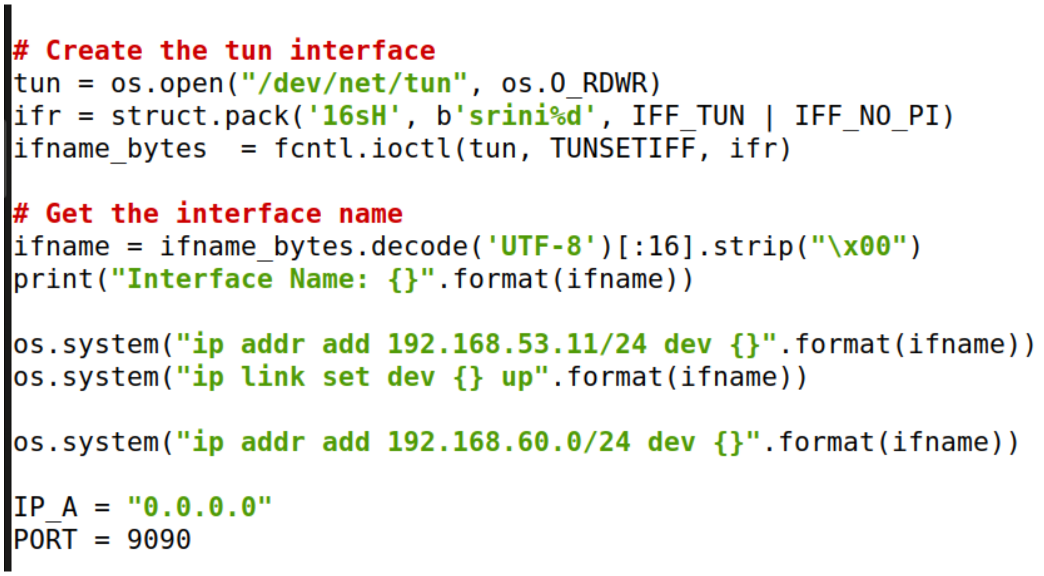
From the Host machine “Host U” send packets to machine “Host-V”. As the VPN is not set up a ping to the host does not work.

tun-server.py

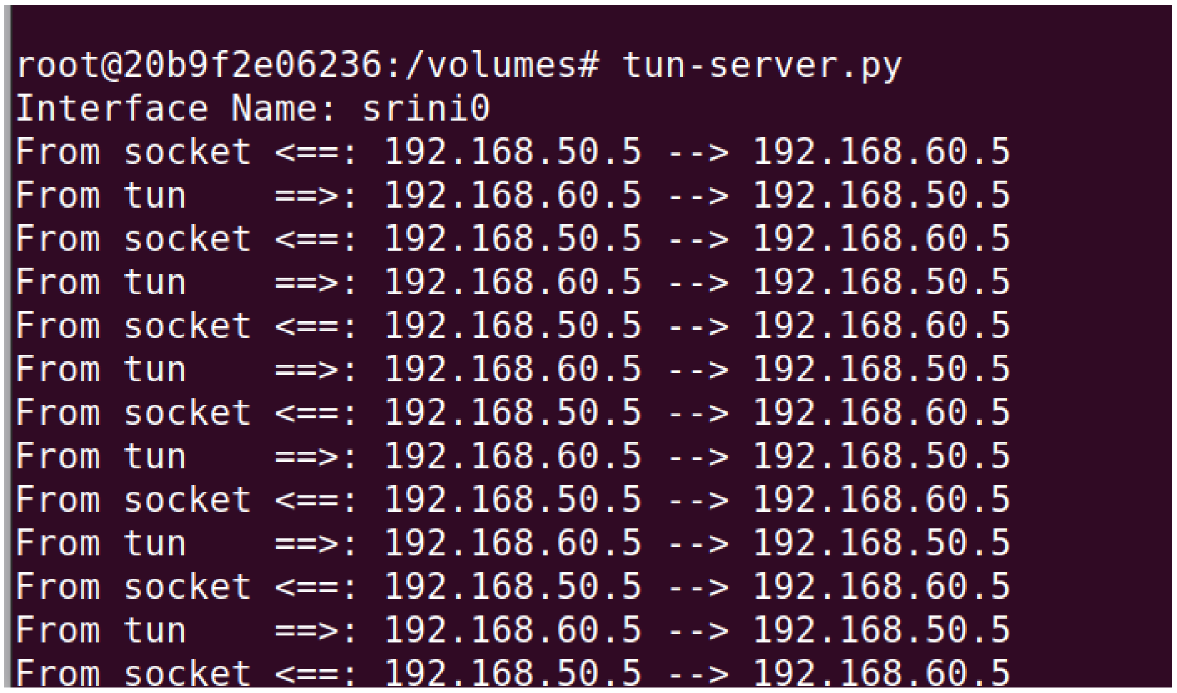


```
  GNU nano 4.8                                   tun-server.py

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

os.system("ip addr add 192.168.53.50/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

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

IP_A = "0.0.0.0"
PORT = 9090

SERVER_IP, SERVER_PORT = '10.9.0.5', 9090

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

tun-client.py

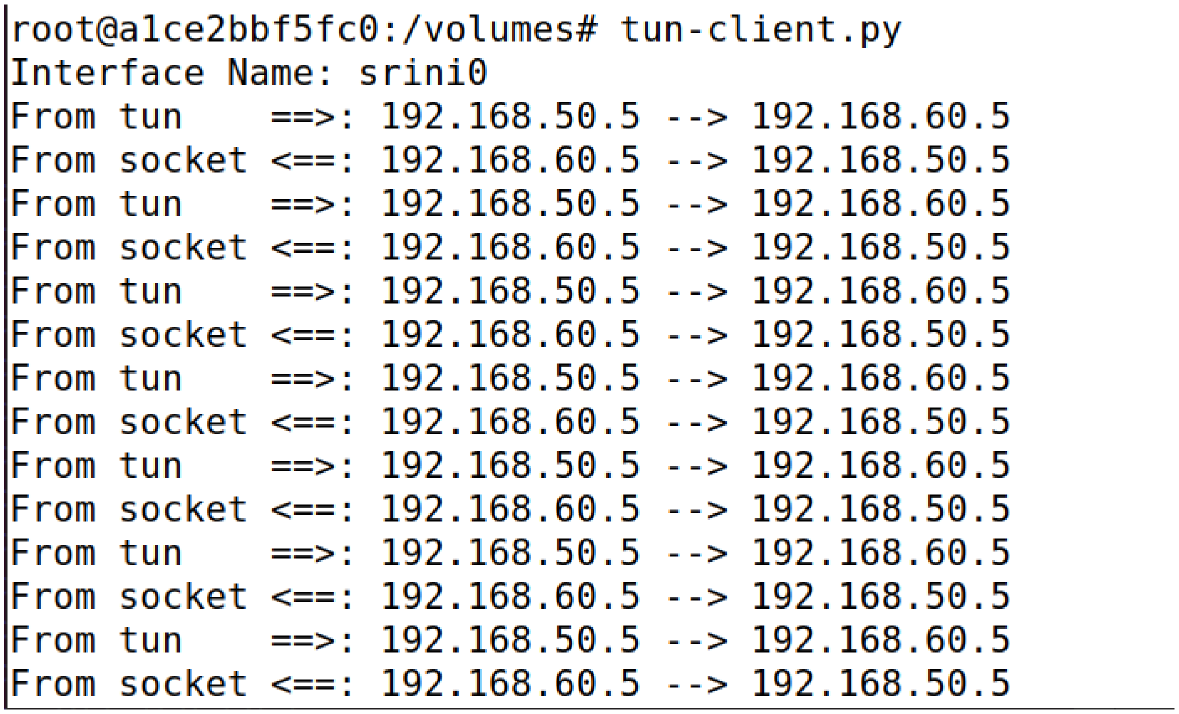


```
# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'srini%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

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

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

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

IP_A = "0.0.0.0"
PORT = 9090
```



```
root@20b9f2e06236:/volumes# tun-server.py
Interface Name: srini0
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
```

Run the tun.server.py program on the Host U machine (192.168.50.5)

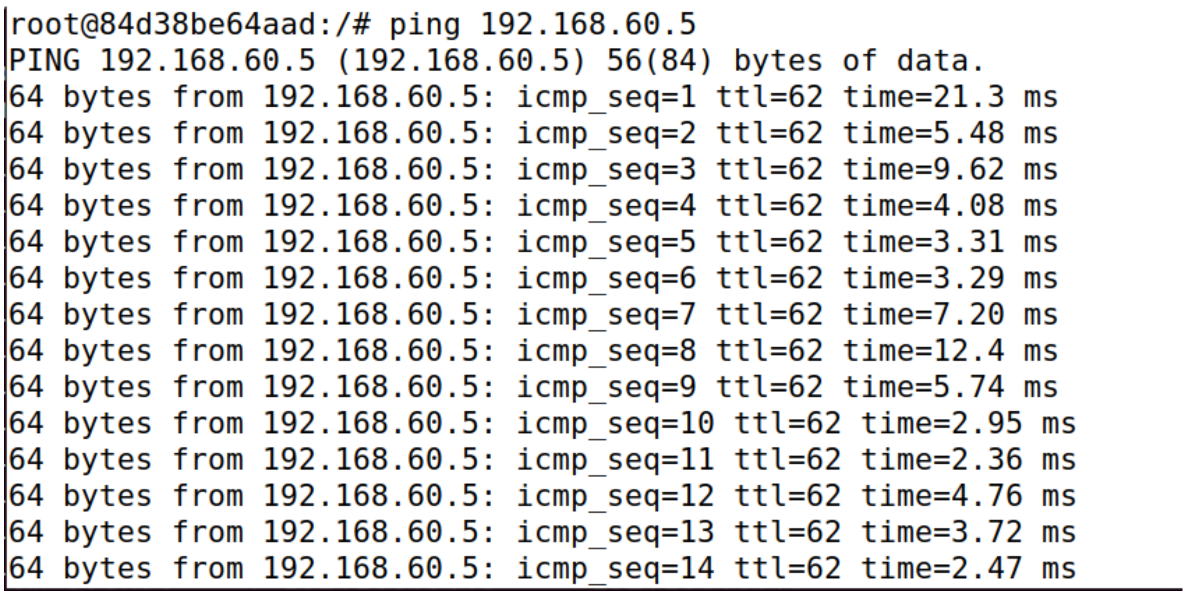


```
root@a1ce2bbf5fc0:/volumes# tun-client.py
Interface Name: srini0
From tun    ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun    ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun    ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun    ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun    ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun    ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun    ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
```

On the client machine (10.9.0.5) run the tun-client.py program

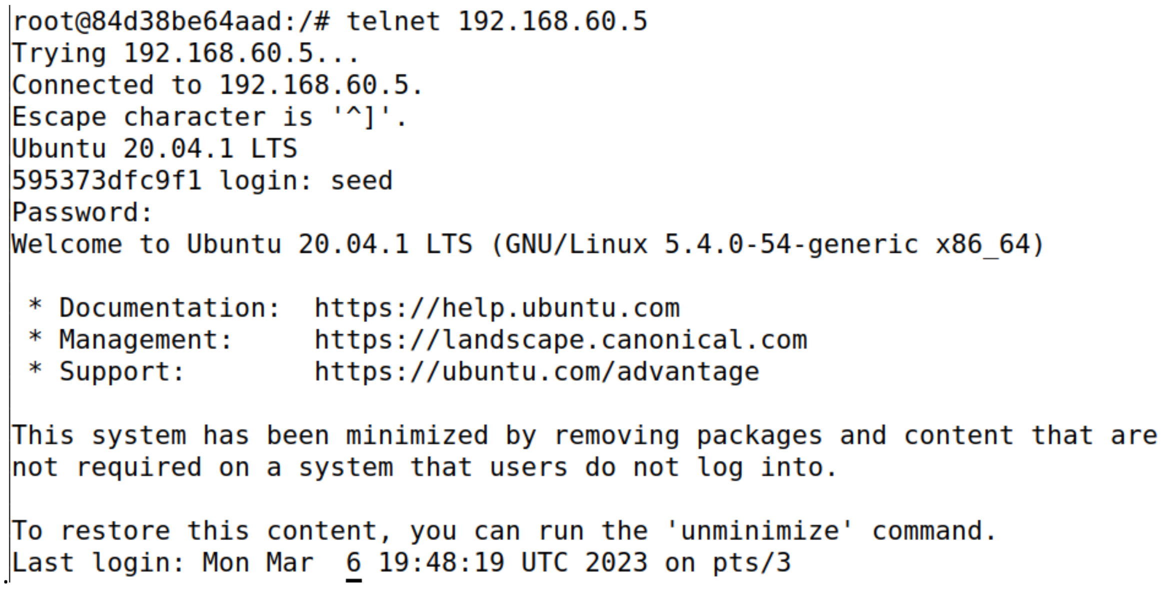
As we can see in the above screenshot, we tried to create a connection between two private networks i.e between 192.168.50.5 & 192.168.60.5 which are two private networks. After executing the tun-server.py and tun-client.py program the packets have been transmitted between both the networks on the client and server side. We then try to ping on Host V

192.168.60.5 as shown below to keep track of all the packets that are sent and also the VPN server also received those packets from the VPN-tunnel.



```
root@84d38be64aad:/# 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=21.3 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=62 time=5.48 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=62 time=9.62 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=62 time=4.08 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=62 time=3.31 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=62 time=3.29 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=62 time=7.20 ms
64 bytes from 192.168.60.5: icmp_seq=8 ttl=62 time=12.4 ms
64 bytes from 192.168.60.5: icmp_seq=9 ttl=62 time=5.74 ms
64 bytes from 192.168.60.5: icmp_seq=10 ttl=62 time=2.95 ms
64 bytes from 192.168.60.5: icmp_seq=11 ttl=62 time=2.36 ms
64 bytes from 192.168.60.5: icmp_seq=12 ttl=62 time=4.76 ms
64 bytes from 192.168.60.5: icmp_seq=13 ttl=62 time=3.72 ms
64 bytes from 192.168.60.5: icmp_seq=14 ttl=62 time=2.47 ms
```

Steps that are followed are: We first created a route on the client side so that packets from 192.168.60.0/24 packets are routed towards the VPN client. Then perform the same thing on server side 192.168.60.0/24 packets to the TUN packets towards TUN server. Then between the VPN client and server machine we routed all the packets towards the TUN interface. Now the VPN client will act as a router.



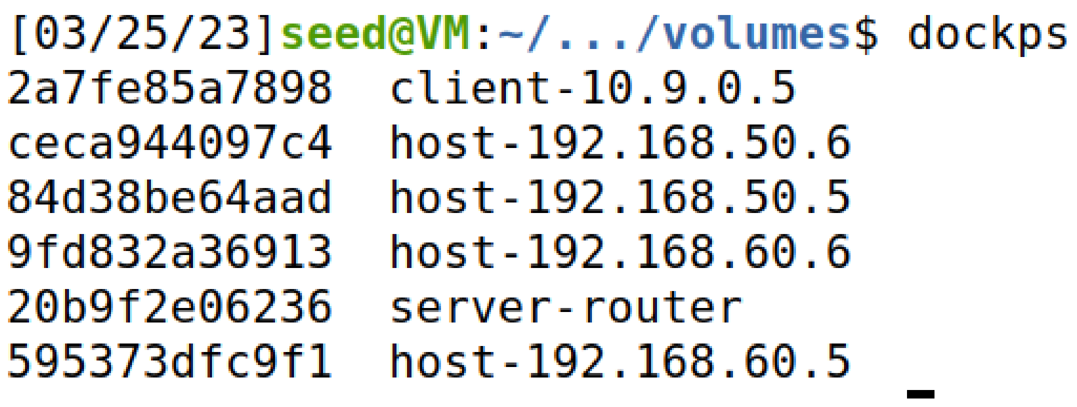
```
root@84d38be64aad:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
595373dfc9f1 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * 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 Mar  6 19:48:19 UTC 2023 on pts/3
```

As our connection is working and up, we telnet from Host V to Host U to see a connection being established which shows our VPN was configured properly.

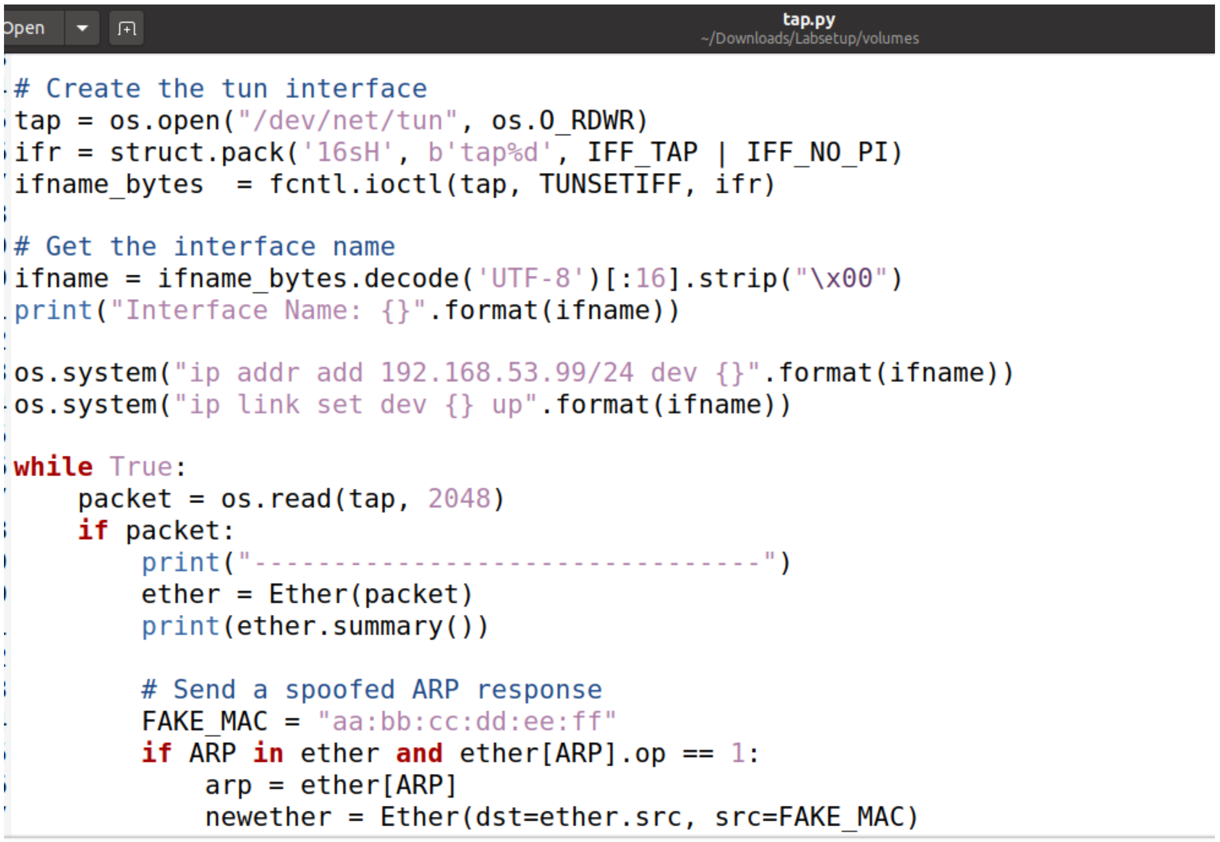
Task 9: Experiment with the TAP interface

We shall create a TAP interface instead of the TUN Interface for this task. The TUN interface is with the IP layer and the TAP interface is with the MAC layer.

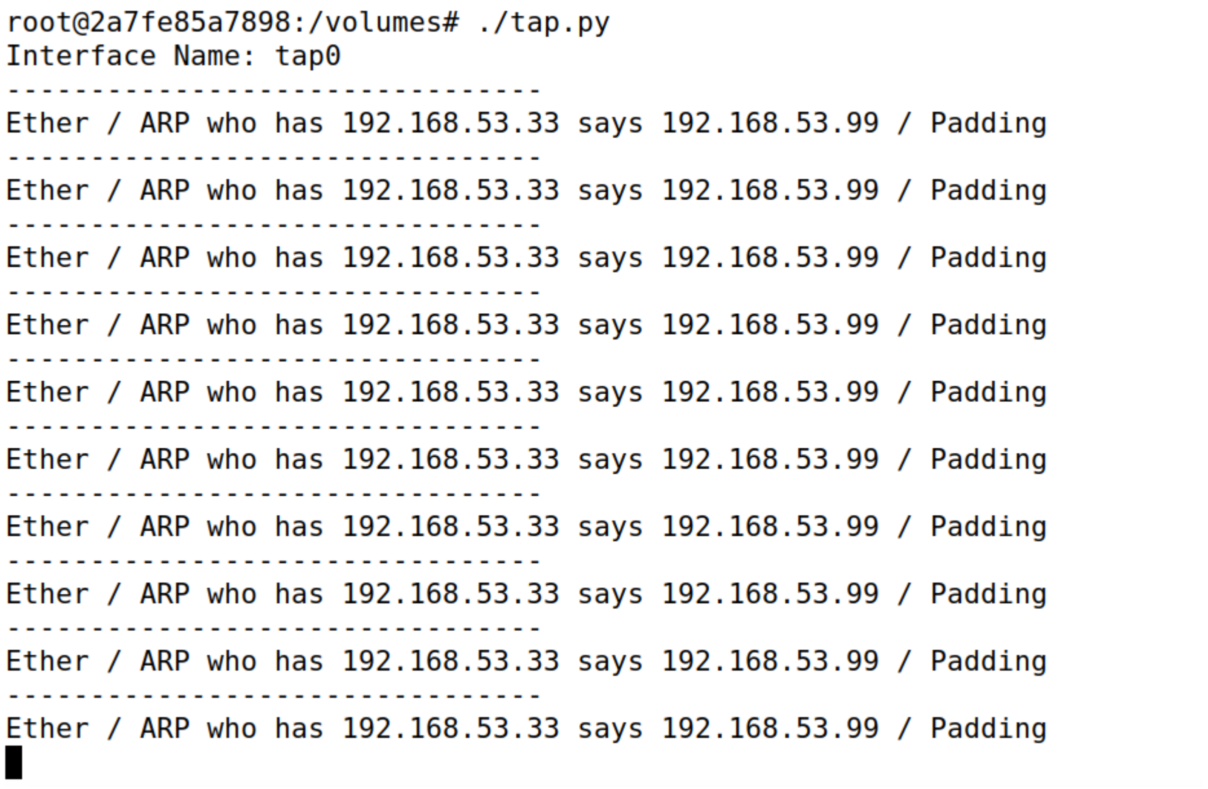
Containers:



```
[03/25/23]seed@VM:~/.../volumes$ dockps
2a7fe85a7898  client-10.9.0.5
ceca944097c4  host-192.168.50.6
84d38be64aad  host-192.168.50.5
9fd832a36913  host-192.168.60.6
20b9f2e06236  server-router
595373dfc9f1  host-192.168.60.5
```

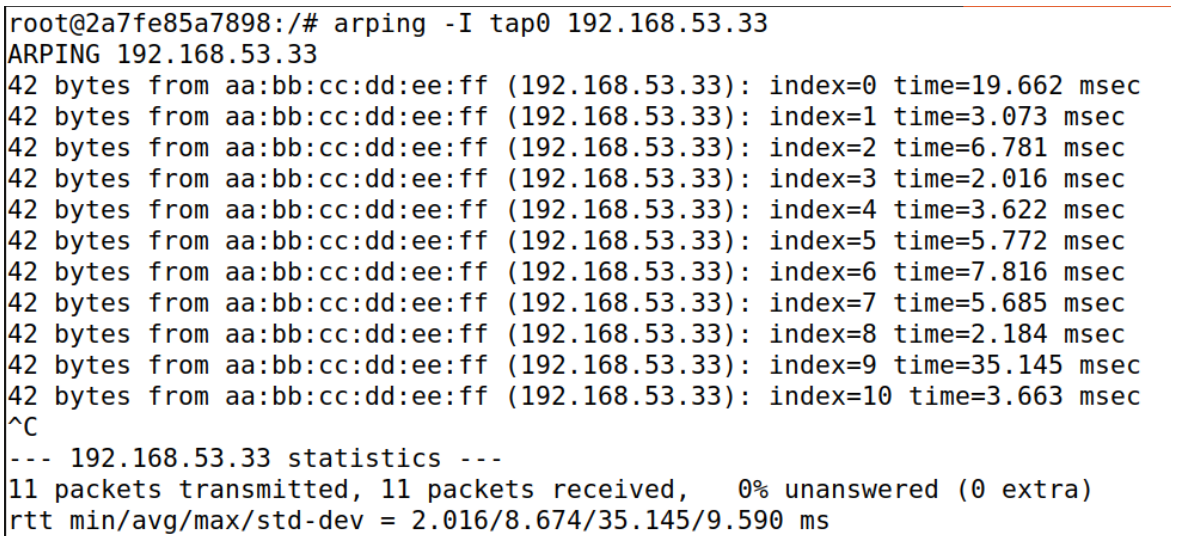


```
# Create the tun interface
tap = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'tap%d', IFF_TAP | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tap, TUNSETIFF, ifr)

# 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)
```



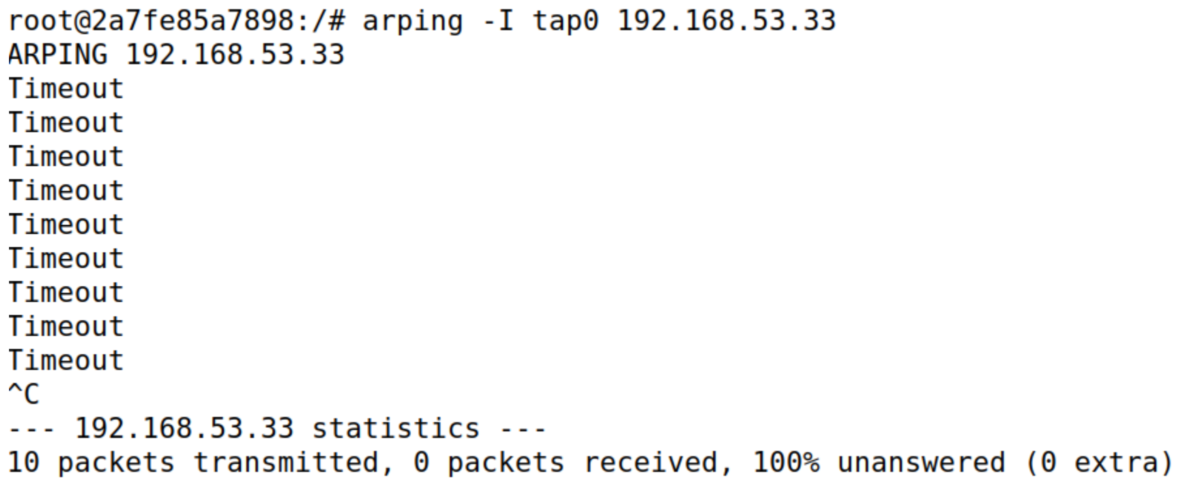
```
root@2a7fe85a7898:/volumes# ./tap.py
Interface Name: tap0
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
```

Run the tap.py program on the client machine. We can see that it is casting the scapy ether object which shows which Ip has what address.



```
root@2a7fe85a7898:/# 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=19.662 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=1 time=3.073 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=2 time=6.781 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=3 time=2.016 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=4 time=3.622 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=5 time=5.772 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=6 time=7.816 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=7 time=5.685 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=8 time=2.184 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=9 time=35.145 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=10 time=3.663 msec
^C
--- 192.168.53.33 statistics ---
11 packets transmitted, 11 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 2.016/8.674/35.145/9.590 ms
```

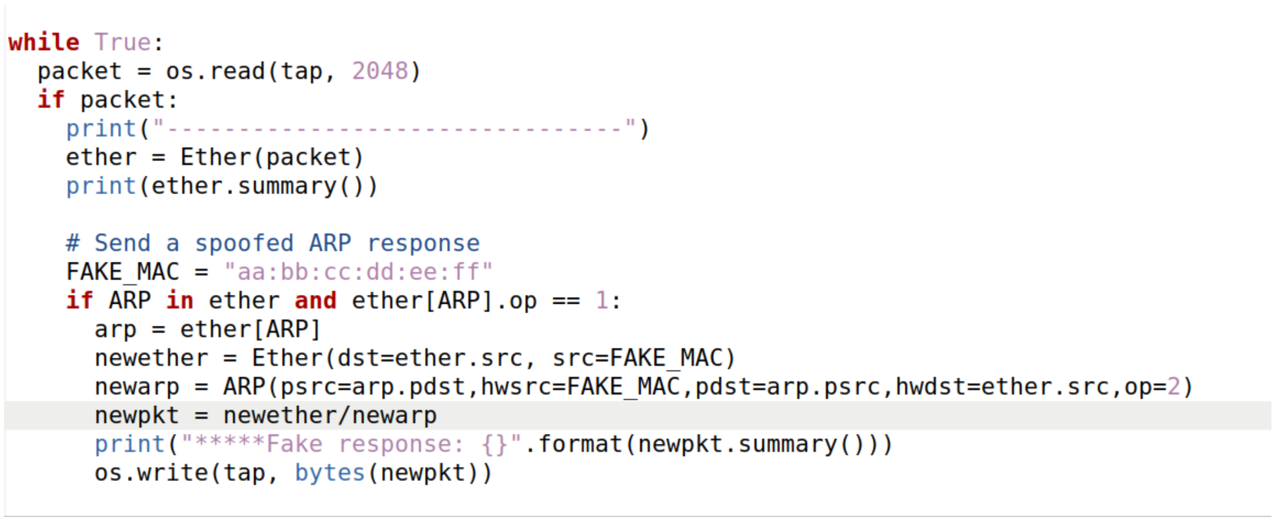
arping -I tap0 192.168.53.33



```
root@2a7fe85a7898:/# arping -I tap0 192.168.53.33
ARPING 192.168.53.33
Timeout
Timeout
Timeout
Timeout
Timeout
Timeout
Timeout
Timeout
Timeout
^C
--- 192.168.53.33 statistics ---
10 packets transmitted, 0 packets received, 100% unanswered (0 extra)
```

Now next we must check if Ethernet frame that we got from the Tap is the ARP request or not. We can use the ARP command to check. If we verify that the request is an ARP request, then we can generate a corresponding ARP reply.

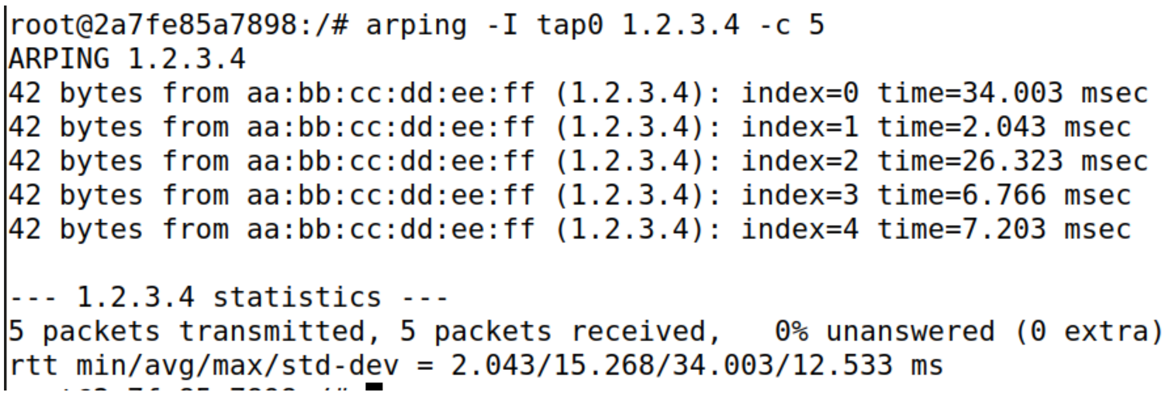
Modify the tap.py code below with slight changes.



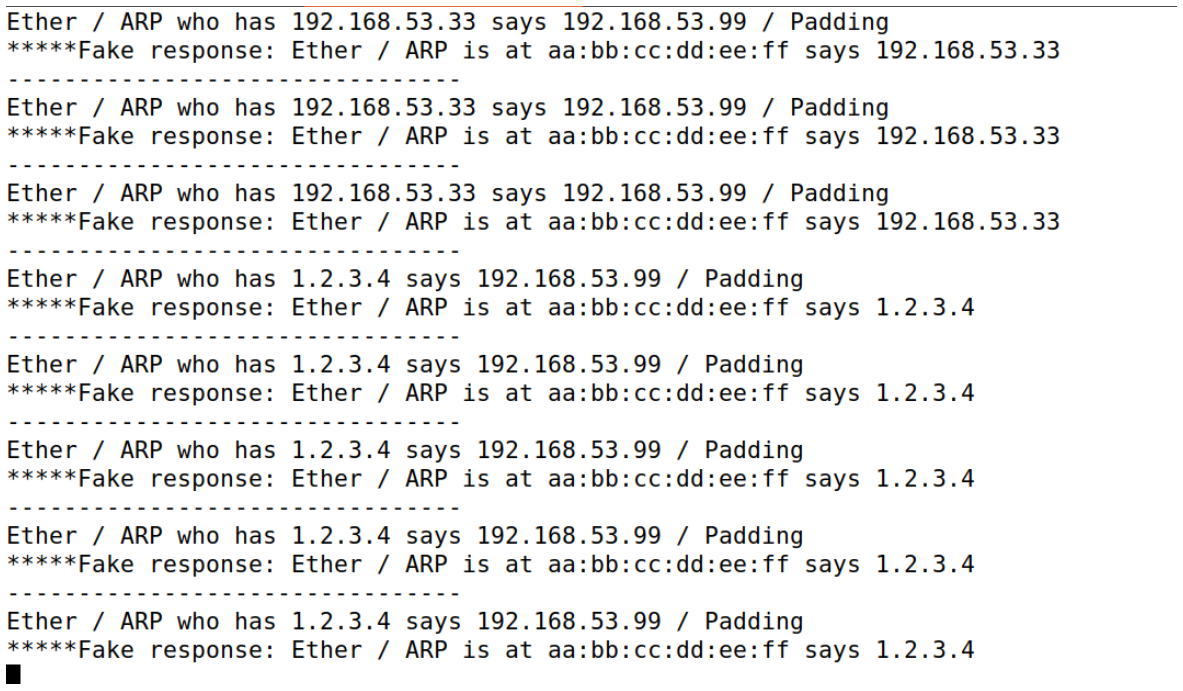
```
while True:
  packet = os.read(tap, 2048)
  if packet:
    print("--------------------------------")
    ether = Ether(packet)
    print(ether.summary())

    # 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))
```

```
root@2a7fe85a7898:/volumes# ./tap.py
Interface Name: tap0
--------------------------------
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
--------------------------------
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
```

Command: arping -I tap0 1.2.3.4



```
root@2a7fe85a7898:/# arping -I tap0 1.2.3.4 -c 5
ARPING 1.2.3.4
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=0 time=34.003 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=1 time=2.043 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=2 time=26.323 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=3 time=6.766 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=4 time=7.203 msec

--- 1.2.3.4 statistics ---
5 packets transmitted, 5 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 2.043/15.268/34.003/12.533 ms
```


```
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 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
--------------------------------
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
--------------------------------
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
--------------------------------
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
--------------------------------
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
```

From the above observation we get to know that the above frame is an ARP request. An ARP reply is generated for each ARP request.