

# Network Tunneling and TAP Interface Configuration

**AIM: Create a Tunnel between interface 10.0.7.0/24 and 10.0.8.0/24 and assign ipaddress 10.0.53.99 to the tunnel interface. Forward packets from client to pull at server in the two subnetworks respectively, through the tunnel.**

## INTRODUCTION

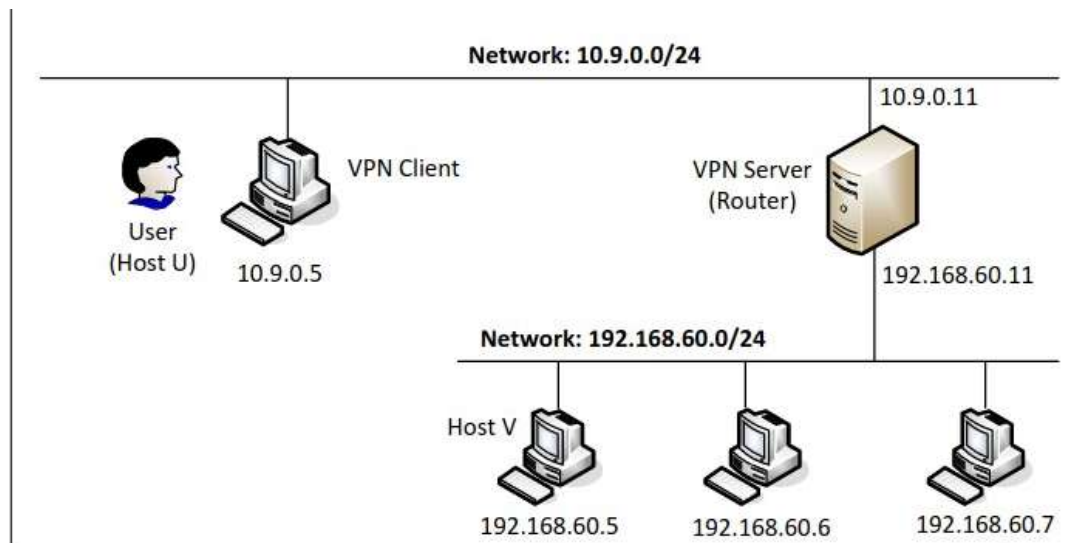
In networking, creating a tunnel between different networks allows for secure and private communication between devices located in separate subnetworks. This activity focuses on creating a tunnel between two subnetworks, namely 10.0.7.0/24 and 10.0.8.0/24, and assigning the IP address 10.0.53.99 to the tunnel interface. The objective is to establish a communication path that forwards packets from clients on one side of the tunnel to servers on the other side, ensuring seamless connectivity and data transmission. By configuring and managing the tunnel interface effectively, network administrators can optimize traffic flow and maintain network integrity across diverse subnetworks.

## STEPS

- Open docker and start the yml1

```
[04/18/24] seed@VM:~/.../vpntunnel$ dockps
577a3f5eecfa  client-10.9.0.5
31ac98ba4137  host-192.168.60.6
2ef1b534135d  host-192.168.60.5
6c79ff1f360a  server-router
```

We have to make connection between u to v by the help of vpn tunnelling.



First we have to check connection bw host u to vpn server and host server to host v

```
seed@VM: ~/... x seed@VM: ~/... x seed@VM: ~/... x seed@VM: ~/... x seed@VM: ~/... x seed@VM: ~/... x
[04/18/24]seed@VM:~/.../vpntunnel$ docksh server-router
root@6c79ff1f360a:/# 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
06:57:30.773155 ARP, Request who-has 10.9.0.11 tell 10.9.0.5, length 28
06:57:30.773172 ARP, Reply 10.9.0.11 is-at 02:42:0a:09:00:0b, length 28
06:57:30.773191 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 12, seq 1, length 64
06:57:30.773207 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 12, seq 1, length 64
06:57:31.785099 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 12, seq 2, length 64
06:57:31.785141 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 12, seq 2, length 64
06:57:32.866942 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 12, seq 3, length 64
06:57:32.866981 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 12, seq 3, length 64
06:57:33.921480 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 12, seq 4, length 64
06:57:33.921532 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 12, seq 4, length 64
06:57:34.952211 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 12, seq 5, length 64
```

### Creating tunnel after connection At 10.9.0.5

Code:

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

```
import fcntl
import struct
import os
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'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))
```

```
# Set up the tun interface
os.system("ip addr add 10.0.53.99/24 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))

# Create UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

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)
            pkt = IP(data)
            print("From socket ==>: {} --> {}".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, ("10.9.0.11", 9090))

```

### **Creating tunnel after connection At server router**

```

#!/usr/bin/python3

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

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

IP_A = "0.0.0.0"
PORT = 9090

ip,port='10.9.0.5',12345

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

#set up the tun interface
os.system("ip addr add 10.0.53.1/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

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)
            pkt = IP(data)
            print("From socket ==>: {} --> {}".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))

```

after running both the codes the tunnel is created and we can check by the command in host v  
**tcpdump -i eth0 -n 2>/dev/null**

## Results & Discussion

### At Server

```

10.9.0.5:57228 --> 0.0.0.0:9090
Inside: 10.0.53.99 --> 10.0.53.5
10.9.0.5:57228 --> 0.0.0.0:9090
Inside: 10.0.53.99 --> 10.0.53.5
10.9.0.5:57228 --> 0.0.0.0:9090
Inside: 10.0.53.99 --> 10.0.53.5

```

ping host V on host U and see if we are getting any reply.

```
U-client-10.9.0.5:/w/n/$>ping 192.168.60.5 -c2
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=3.91 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=1.77 ms

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1005ms
rtt min/avg/max/mdev = 1.765/2.837/3.910/1.072 ms
```

Now, this image shows that Host U sends the packet to Host V through tunnel and likewise receives back the packets (replies).

```
U-client-10.9.0.5:/w/n/$>python3 tun_client.py
Interface Name: tun0
From tun ==>: 10.0.53.99 --> 192.168.60.5
From tun ==>: 10.0.53.99 --> 192.168.60.5
From tun ==>: 10.0.53.99 --> 192.168.60.5
From socket ==>: 192.168.60.5 --> 10.0.53.99
From tun ==>: 10.0.53.99 --> 192.168.60.5
From socket ==>: 192.168.60.5 --> 10.0.53.99
```

Similarly, seeing from the server side it is doing the job of forwarding and receiving the packets using tunnel.

```
10.9.0.11-192.168.60.11 Router:/w/n/$>python3 tun_server.py
Interface Name: tun0
From socket ==>: 10.0.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 10.0.53.99
From socket ==>: 10.0.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 10.0.53.99
```

Talking about Host V, we can see that We have received packets from host U through tunnel and also it is able to reply to Host U using the tunnel.

---

```
V-host-192.168.60.5:/w/n/$>tcpdump -i eth0 -n 2>/dev/null
15:39:00.785374 IP 10.0.53.99 > 192.168.60.5: ICMP echo request, id 29, seq 1, length 64
15:39:00.785405 IP 192.168.60.5 > 10.0.53.99: ICMP echo reply, id 29, seq 1, length 64
15:39:01.796628 IP 10.0.53.99 > 192.168.60.5: ICMP echo request, id 29, seq 2, length 64
15:39:01.796649 IP 192.168.60.5 > 10.0.53.99: ICMP echo reply, id 29, seq 2, length 64
15:39:05.891581 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
15:39:05.891773 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
15:39:05.891793 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
15:39:05.891799 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
15:51:41.604162 IP6 fe80::a87e:aaff:fe6d:163b > ff02::2: ICMPv6, router solicitation, length 16
```

---

The successful creation of a tunnel between the 10.0.7.0/24 and 10.0.8.0/24 subnets, with the tunnel interface assigned the IP address 10.0.53.99, has enabled seamless communication and data transfer between devices in these subnetworks. By forwarding packets from clients to a server located in the respective subnets through the established tunnel, network connectivity has been effectively established across different segments of the network infrastructure. This setup not only facilitates efficient data exchange but also enhances network security by encapsulating and encrypting traffic within the tunnel, protecting it from unauthorized access or interception. The implementation of routing configurations ensures that packets are directed accurately through the tunnel, optimizing network performance and ensuring reliable connectivity between the specified subnets. Overall, this activity demonstrates the importance of robust tunneling protocols and routing mechanisms in building a resilient and interconnected network environment.

## **Learnings**

Here are some key learnings from the activity of creating a tunnel between interface 10.0.7.0/24 and 10.0.8.0/24, assigning IP address 10.0.53.99 to the tunnel interface, and forwarding packets from clients to a server in the two subnetworks through the tunnel:

- **Tunnel Creation:** Understand the process of creating a tunnel interface between two subnets. This involves configuring the tunnel interface with appropriate IP addresses and settings.
- **IP Address Assignment:** Assigning a specific IP address to the tunnel interface (in this case, 10.0.53.99) ensures proper routing and communication between the subnets.
- **Packet Forwarding:** Learn how to forward packets from clients in the 10.0.7.0/24 subnet to a server in the 10.0.8.0/24 subnet through the established tunnel. This includes setting up routing rules and configuring the tunnel interface for packet forwarding.
- **Subnet Connectivity:** Ensure that the tunnel facilitates communication between devices in different subnets, allowing seamless data transfer and connectivity across the network.
- **Routing Configuration:** Configure routing tables and protocols (such as static routing or dynamic routing protocols like OSPF or BGP) to ensure proper routing of packets through the tunnel between the specified subnets.
- **Testing and Troubleshooting:** Test the connectivity and functionality of the tunnel by sending packets from clients to the server and vice versa. Troubleshoot any connectivity issues or misconfigurations that may arise during the setup process.



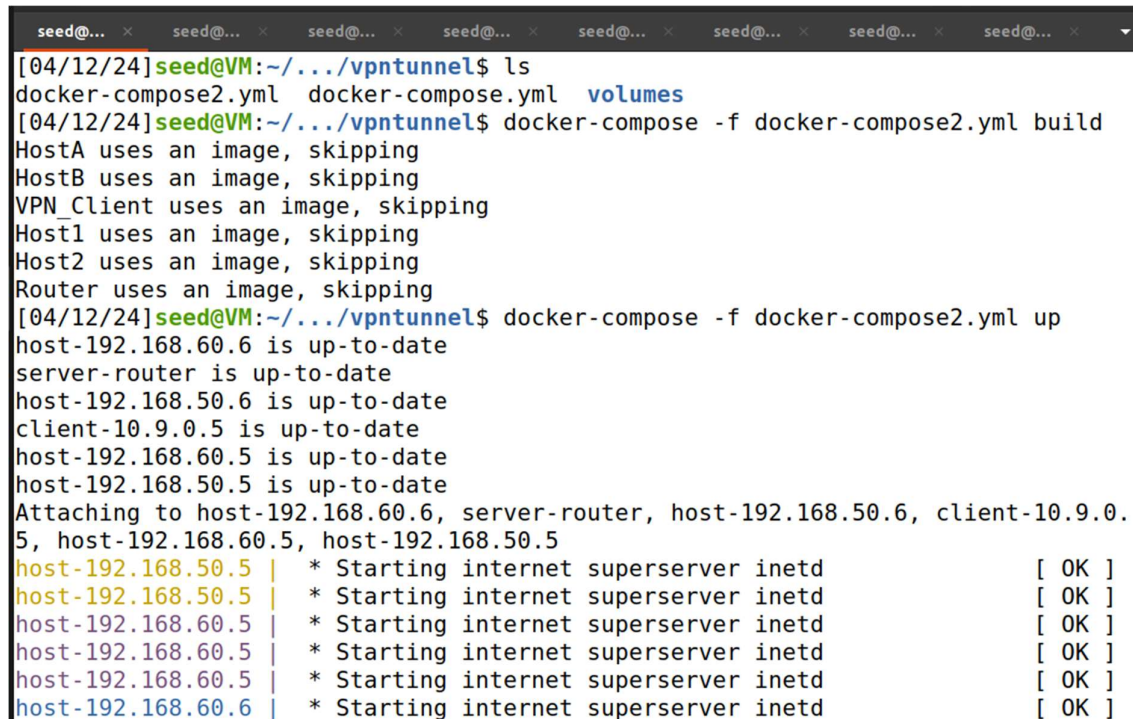
**Aim: Create a TAP interface and show how is it different from TUN.**

## Introduction

Creating a TAP (Network tap) interface is a fundamental networking concept that enables communication between virtual machines, containers, or other network entities. Unlike TUN (Network tunnel) interfaces, which operate at the network layer (Layer 3) by encapsulating packets, TAP interfaces function at the data link layer (Layer 2) and are capable of transmitting Ethernet frames.

## Steps

- Download and start the docker for VPN Tunnelling



```
seed@... x seed@... x seed@... x seed@... x seed@... x seed@... x seed@... x seed@... x
[04/12/24]seed@VM:~/.../vpntunnel$ ls
docker-compose2.yml  docker-compose.yml  volumes
[04/12/24]seed@VM:~/.../vpntunnel$ 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
[04/12/24]seed@VM:~/.../vpntunnel$ docker-compose -f docker-compose2.yml up
host-192.168.60.6 is up-to-date
server-router is up-to-date
host-192.168.50.6 is up-to-date
client-10.9.0.5 is up-to-date
host-192.168.60.5 is up-to-date
host-192.168.50.5 is up-to-date
Attaching to host-192.168.60.6, server-router, host-192.168.50.6, client-10.9.0.
5, host-192.168.60.5, host-192.168.50.5
host-192.168.50.5 | * Starting internet superserver inetd          [ OK ]
host-192.168.50.5 | * Starting internet superserver inetd          [ OK ]
host-192.168.60.5 | * Starting internet superserver inetd          [ OK ]
host-192.168.60.5 | * Starting internet superserver inetd          [ OK ]
host-192.168.60.5 | * Starting internet superserver inetd          [ OK ]
host-192.168.60.6 | * Starting internet superserver inetd          [ OK ]
```

- On all the terminal by docksh command

## TUN Interface

- Open client terminal and run tun.py

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

```
import fcntl
import struct
import os
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'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))

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

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

    # Send out a spoof packet using the tun interface
    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 = newip/newicmp

    os.write(tun, bytes(newpkt))
    • Then open another terminal for client and do ifconfig , in which we can see the tun0 is created

```



```
seed@VM: ~/.../vpntunnel
seed@VM: ~/.../v... x seed@VM: ~/.../v... x seed@VM: ~/.../v... x seed@VM: ~/.../v... x seed@VM: ~/.../v... x
[04/03/24] seed@VM:~/.../vpntunnel$ docksh client-10.9.0.5
root@e3dc7d137c29:/# 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: tun0: <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 tun0
        valid_lft forever preferred_lft forever
7: eth0@if8: <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@e3dc7d137c29:/# ping 192.168.53.10
PING 192.168.53.10 (192.168.53.10) 56(84) bytes of data.
64 bytes from 192.168.53.10: icmp_seq=1 ttl=99 time=2.56 ms
64 bytes from 192.168.53.10: icmp_seq=2 ttl=99 time=1.82 ms
64 bytes from 192.168.53.10: icmp_seq=3 ttl=99 time=1.51 ms
64 bytes from 192.168.53.10: icmp_seq=4 ttl=99 time=2.84 ms
64 bytes from 192.168.53.10: icmp_seq=5 ttl=99 time=2.03 ms
```

## Tap Interface

- Now run tap.py with same above steps

Code for tap.py

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

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

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

```
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'tap%d' % IFF_TAP | 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
```

```
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(tun, 2048)
```

```

if True:
    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("***** Sending Fake response: {}".format(newpkt.summary()))
    os.write(tun, bytes(newpkt))

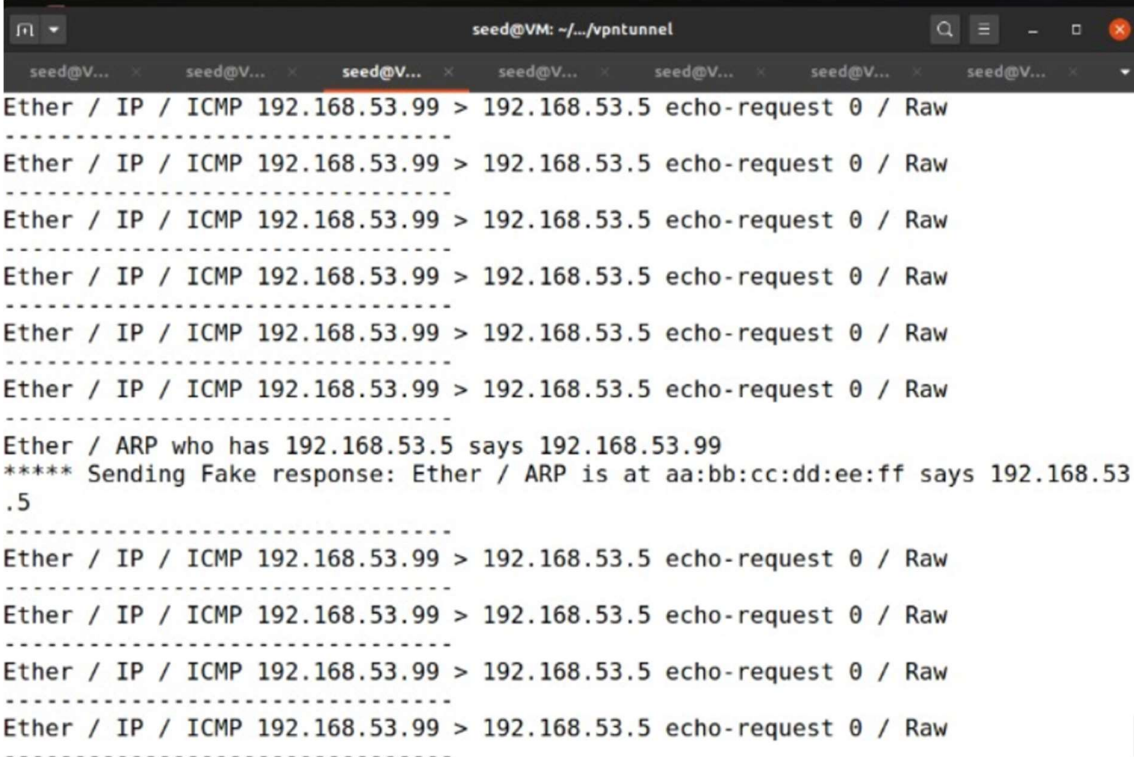
```

- Now open new terminal and first check by ifconfig that tap0 is created and do an ping

```

tunl: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST> mtu 1500
    inet 10.0.8.0 netmask 255.255.255.0 destination 10.0.8.0
    unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00 txqueuelen 500

```



```

seed@VM: ~/.../vpntunnel
seed@V... x seed@V... x seed@V... x seed@V... x seed@V... x seed@V... x seed@V... x
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
-----
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
-----
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
-----
Ether / ARP who has 192.168.53.5 says 192.168.53.99
***** Sending 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
-----
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
-----

```

## Results & Discussion

## Difference between both the interfaces

- The code for both tap and tun interfaces demonstrates that the tap interface does not contain an IP address.
- This distinction arises from the fact that TAP (network tap) and TUN (network tunnel) interfaces are virtual network interfaces used in networking applications but operate at different layers of the OSI model and serve distinct purposes.
- TAP interfaces mimic full network layer devices and operate at Layer 2 (data link layer), capable of carrying both IP and non-IP traffic.
- They are commonly used in scenarios like virtualization environments or VPNs where emulating a physical network device is necessary.
- In contrast, TUN interfaces operate at Layer 3 (network layer) and are specialized for tunneling IP packets.
- They are typically used in VPN software to create secure tunnels over untrusted networks.

## Learnings

1. Creating a TAP interface involves working at the data link layer (Layer 2) of the OSI model.
2. TAP interfaces are used for creating virtual Ethernet bridges or tunnels.
3. TAP interfaces encapsulate Ethernet frames, preserving MAC addresses for virtual network communication.