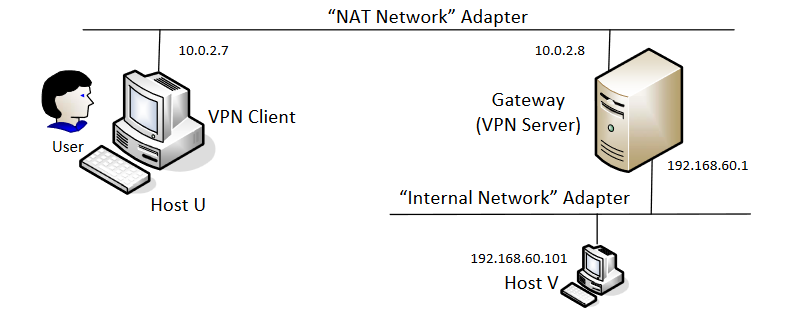
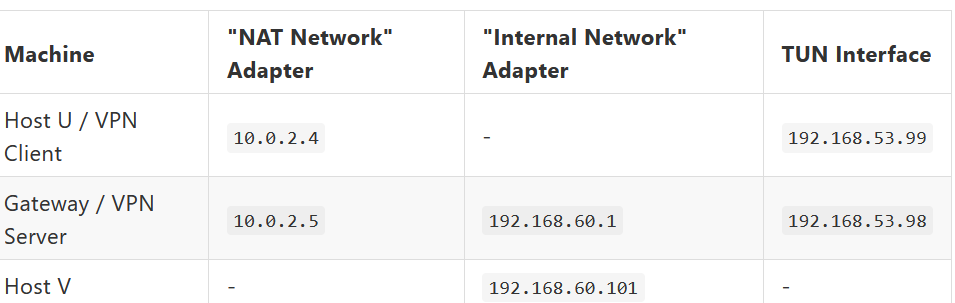
# VPN Tunneling Lab



Below are the machine-to-IP mappings used for this lab:

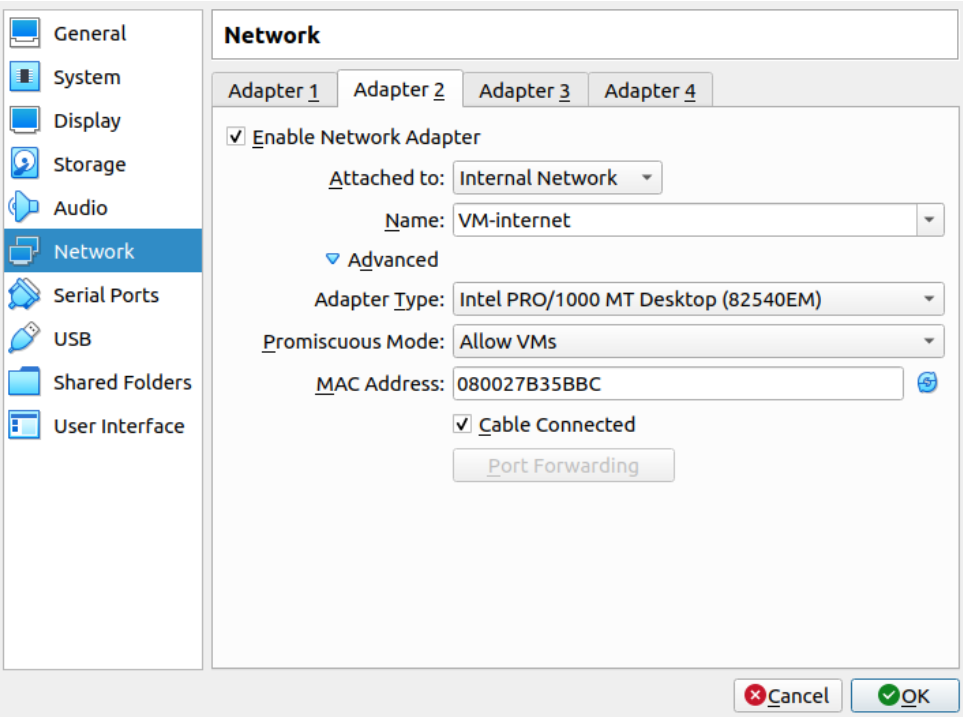


# Task 1: Network Setup

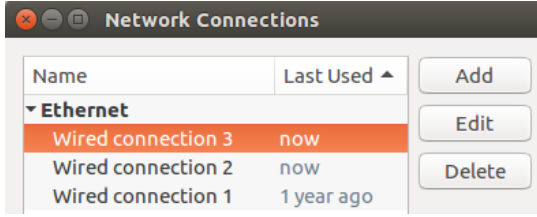
We will first set up the network for the gateway, followed by Host V then Host U. After which, we will test that the network behavior is in line with what is suggested for the lab.

For the Gateway, set up a additional network adapter. Before booting up the VM, navigate to the Network panel by selecting the right machine, followed by Settings and then Network. Navigate to Adapter 2, check the box for "Enable Network Adapter", and fill in the following fields as according to the image below:

1. Attached to
2. Name
3. Promiscuous Mode

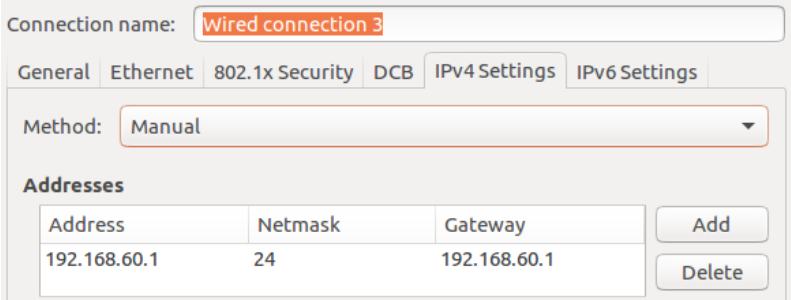


Boot up the Gateway VM. Click on the network icon on the top right of the screen. After clicking "Edit Connections", the pop-up for the Network Connections should have at least 2 networks as seen below.



Although my Gateway machine has 3 Wired Connections, only Wired connection 2 and Wired connection 3 are used in this lab. Wired connection 2 will be the "NAT Network" Adapter while Wired connection 3 will be the "Internal Network" Adapter. As the "NAT Network" Adapter has already been set up for previous labs, only the setup for the "Internal Network" Adapter will be explained below.

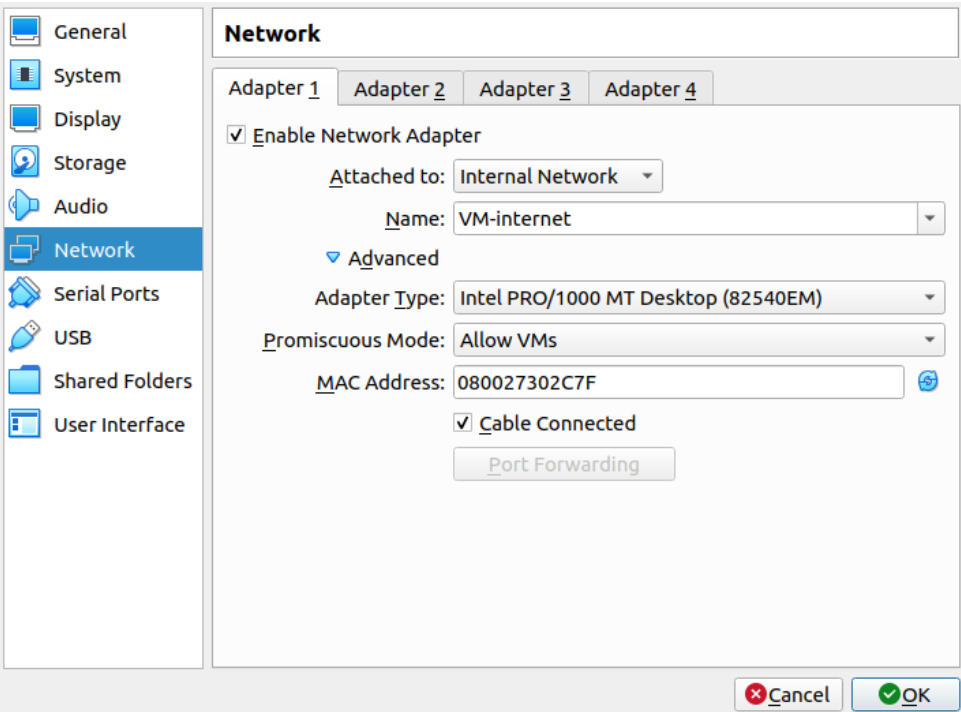
Double click "Wired connection 3" and select the "IPv4 Settings" tab. Change the "Method" field to the "Manual" option and add a new address as shown below.



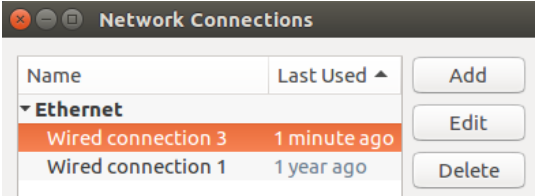
Next, we will configure Host V.

Before booting up the machine, adapter should first be configured. Navigate to the Network panel by selecting the right machine, followed by Settings and then Network. Change Adapter 1 to the Internel Network Adapter by filling in the following fields as according to the image below:

1. Attached to
2. Name
3. Promiscuous Mode



Ensure that the MAC address is different from that of the Gateway VM. After booting up the machine and clicking the network icon followed by "Edit Connections", the following window should appear. We will only be using Wired connection 3 for this lab.



Similar to when configuring the Gateway machine, double click the Wired connection 3, switch to the IPv4 tab and fill in the Method and Addresses as shown below.



|  |
| --- |
| Testing. Please conduct the following testings to ensure that the network setup is performed correctly: |

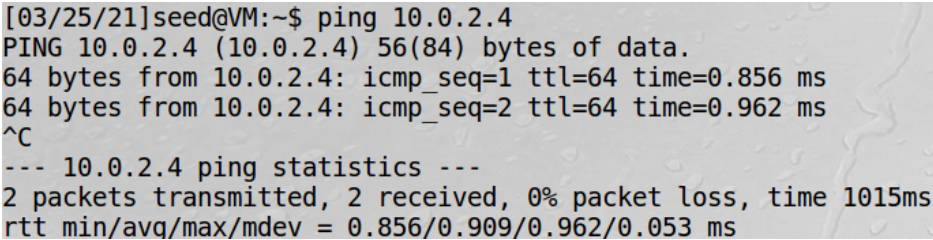
The network setup will be tested with ping, and results are determined by whether machines are able to receive ICMP packets from one another.

|  |
| --- |
| Host U can communicate with VPN Server. |

Host U is able to ping VPN Server as as shown below.



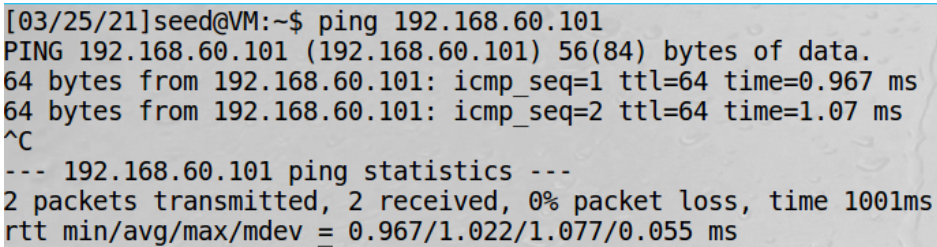
VPN Server is able to ping Host U as shown below.



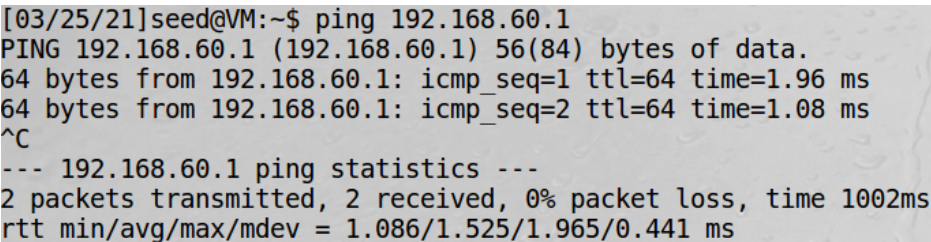
Hence, Host U and VPN Server are able to communicate with each other.

|  |
| --- |
| VPN Server can communicate with Host V. |

VPN Server is able to ping Host V as shown below.



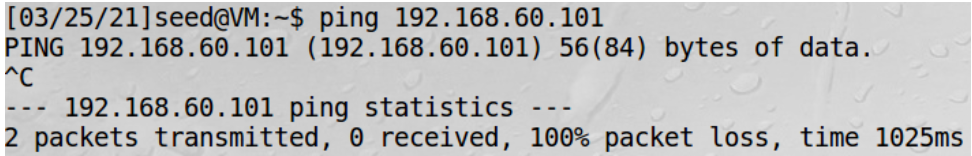
Host V is able to ping VPN Server as shown below.



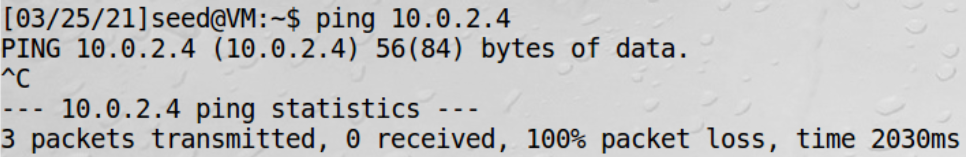
Hence, VPN Server and Host V are able to communicate with one another.

|  |
| --- |
| Host U should not be able to communicate with Host V. |

Host U is unable to ping Host V as shown below.



Host V is unable to ping Host U as shown below.



Hence, Host U and V are unable to communicate with one another.

# Task 2: Create and Configure TUN Interface

## Task 2.a: Name of the Interface

The tun.py code for this task is saved as host\_u/task2a.py

|  |
| --- |
| Your job in this task is to change the tun.py program, so instead of using tun as the prefix of the interface name, use your last name as the prefix. For example, if your last name is smith, you should use smith as the prefix. If your last name is long, you can use the first five characters. Please show how your results. |

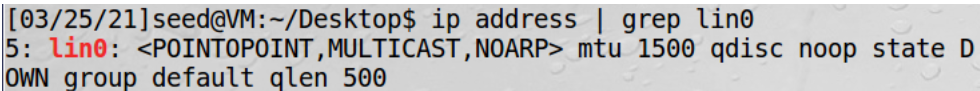
To change the prefix of the interface name to my last name "lin", the ifr variable was defined in tun.py as follows.

|  |
| --- |
| ifr = struct.pack('16sH', b'lin%d', IFF\_TUN | IFF\_NO\_PI) |

After saving the file, make the python an executable with chmod +x tun.py and run it with root privilege with sudo ./tun.py. After which, open a new terminal and run the following command to check that the tun interface is successfully named accordingly:

|  |
| --- |
| ip address | grep lin0 |

The output seen should be as follows:



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

Note: The tun.py code for this task is saved as host\_u/task2b.py.

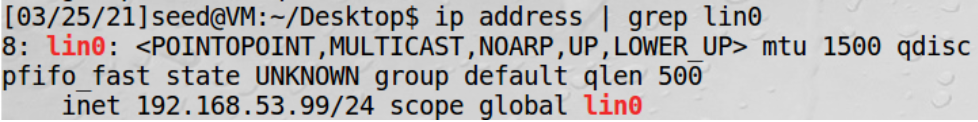
There are two things that we need to do before the interface can be used. First, we need to assign an IP address to it. Second, we need to bring up the interface, because the interface is still in the down state.

The following code is added to tun.py:

|  |
| --- |
| # imports, define variables, create 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 loop |

|  |
| --- |
| After running the two commands above, run the "ip address" command again, and report your observation. How is it different from that before running the configuration commands? |

Run the code again with sudo ./tun.py. Then, open a new terminal and run ip address | grep lin0 again. The output is seen below.



In the first line, UP, LOWER\_UP indicates that the lin0 tunnel interface is enabled, and the device is connected to the network.

The last line indicates that the IP address subnet 192.168.53.99/24 was assigned to the tun interface.

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

Note: The tun.py code for this task is saved as host\_u/task2c.py.

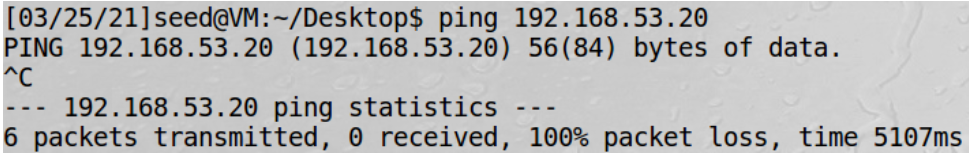
Please run the revised tun.py program on Host U, configure the TUN interface accordingly, and then conduct the following experiments. Please describe your observations:

On Host U, ping a host in the 192.168.53.0/24 network.

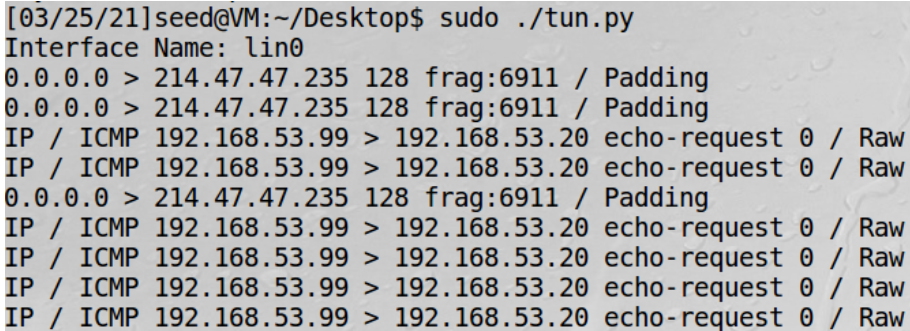
The host 192.168.53.20 is pinged with the following command:

|  |
| --- |
| ping 192.168.53.20 |

ICMP packets are sent as a result, as seen below.



At the same time, tun.py prints IP / ICMP 192.168.53.99 > 192.168.53.20 echo-request 0 / Raw as seen below.



This means that the tun interface (192.168.53.99) is sending ICMP packets to the host queried (192.168.53.20). This happens because when the host 192.168.53.20 is pinged, the ICMP packet would be sent through the tun interface as it is in the same subnet.

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

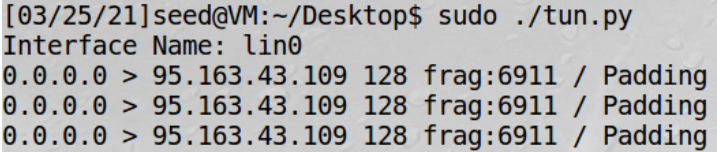
The host 192.168.60.20 is pinged with the following command:

|  |
| --- |
| ping 192.168.60.20 |

Although such a host does not exist, it can be seen that the ICMP packets are sent out as follows:



However, tun.py does not print anything. This is because the host would be within the internal network and thus the packets sent would not be passed to the tun interface. Therefore, as tun.py only prints when packets are received, tun.py would not print out anything.



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

|  |
| --- |
| Please modify the tun.py code according to the following requirements:  After getting a packet from the TUN interface, if this packet is an ICMP echo request packet, construct a corresponding echo reply packet and write it to the TUN interface. Please provide evidence to show that the code works as expected. |

Note: The tun.py code for this part of the task is saved as host\_u/task2d\_1.py.

To check for the echo request packet, the type of packet is checked with the following function:

|  |
| --- |
| def check\_icmp\_req(bytes\_in):  pkt\_in = IP(bytes\_in)  if ICMP in pkt\_in: # checks for ICMP packet  if pkt\_in[ICMP].type == 8: # checks for echo-request type  return True  return False |

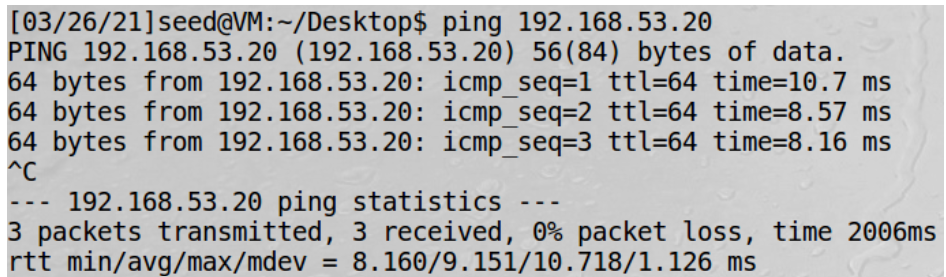
The function create\_icmp\_reply is then defined to create an ICMP echo reply packet based on any ICMP echo request packet received. This is done as seen below.

|  |
| --- |
| def create\_icmp\_reply(bytes\_in):  pkt\_in = IP(bytes\_in)  ip\_out = IP(src=pkt\_in.dst, dst=pkt\_in.src)  pkt\_out = ip\_out / pkt\_in.payload  pkt\_out[ICMP].type = 0 # set ICMP packet type as echo-reply  return bytes(pkt\_out) |

In the while True loop, when a packet is read from the tun interface, it is first checked by the check\_icmp\_req function to check if the packet is an ICMP echo request packet. If it is, the create\_icmp\_reply function is called and the created reply is written to the tun interface. This is done as seen below.

|  |
| --- |
| while True:  # Get a packet from the tun interface  packet = os.read(tun, 2048)  if check\_icmp\_req(packet):  reply\_bytes = create\_icmp\_reply(packet)  os.write(tun, reply\_bytes) |

After saving the code, run the code with sudo ./tun.py. In a new terminal, run ping 192.168.53.20. The pings are replied as seen below.



As the ICMP request packets are replied, the code works as expected.

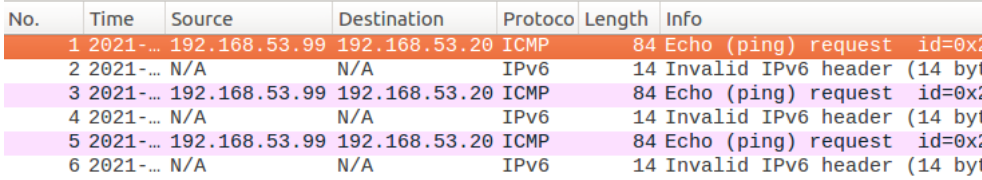
|  |
| --- |
| Instead of writing an IP packet to the interface, write some arbitrary data to the interface, and report your observation. |

Note: The tun.py code for this part of the task is saved as host\_u/task2d\_2.py.

The code can be modified as seen below to sent arbitrary data instead.

|  |
| --- |
| while True:  # Get a packet from the tun interface  packet = os.read(tun, 2048)  if check\_icmp\_req(packet):  os.write(tun, bytes("arbitrary data", encoding="utf-8")) |

When sudo ./tun.py is run, WireShark can be used to capture the packets in lin0 interface. After ping 192.168.53.20 is executed in a new terminal, WireShark logs the following packets:



The ping does not receive any responses as well as seen below.



Based on the above, it can be observed that because arbitrary data instead of IP packets were written to the interface, the bytes written to the tunnel were not valid and the request pings do not receive responses.

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