CS4150 Computer Networks Laboratory – Assignment 10

Subhash S – 111801042
Computer Science and Engineering, IIT Palakkad

Packet sniffer on r1

File: q1.cpp.

Run the "make" command to copy the file to the VM r1 and compile it.

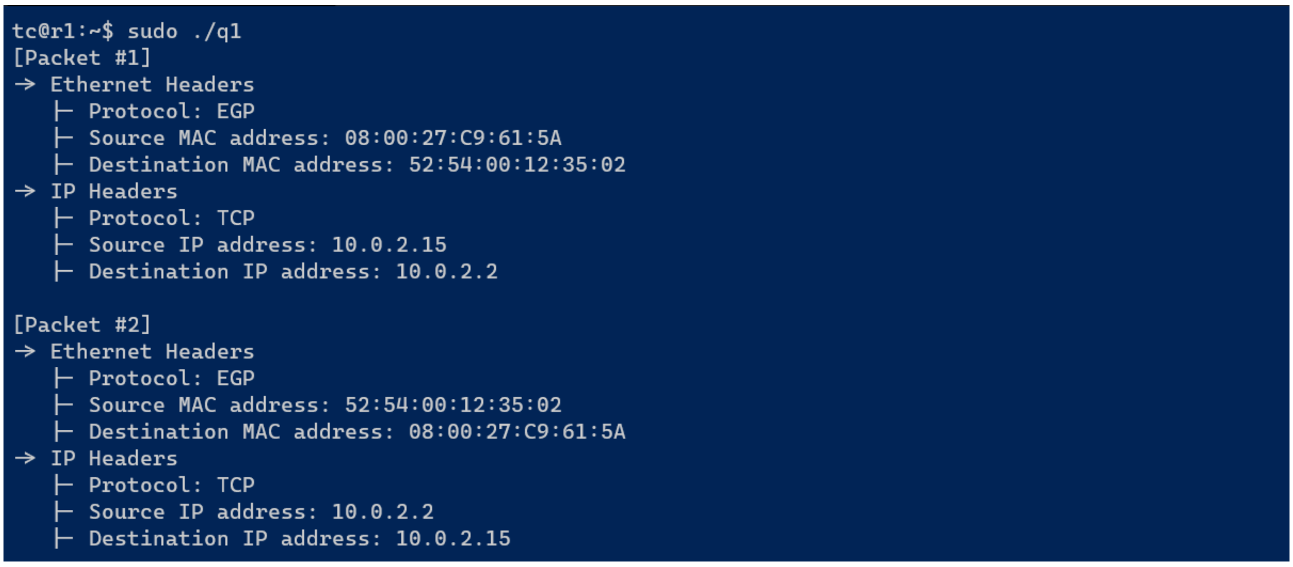
Aim

To build a packet sniffer that prints the MAC addresses (source and destination) and IP addresses (source and destination) of all IP packets sent and received by r1.

Program Flow

1. Create a socket with *AF_PACKET* domain, *SOCK_RAW* type (this tells the os to create a raw socket) and *htons(ETH_P_ALL)* as the protocol (this allows listening to both incoming and outgoing packets).
2. Forever,
    2.1. Set the buffer to zeros.
    2.2. Receive data from the socket created in 1 and store it into the buffer.
    2.3. If there was an error while receiving, print the error, close the socket and quit the program.
    2.4. Print the details in the buffer by extracting the IP headers and ethernet headers using the appropriate structs.

Figure 1: The program listens to all incoming packets and prints some necessary parts of the ethernet and IP headers



```
tc@r1:~$ sudo ./q1
[Packet #1]
→ Ethernet Headers
   ├ Protocol: EGP
   ├ Source MAC address: 08:00:27:C9:61:5A
   ├ Destination MAC address: 52:54:00:12:35:02
→ IP Headers
   ├ Protocol: TCP
   ├ Source IP address: 10.0.2.15
   ├ Destination IP address: 10.0.2.2

[Packet #2]
→ Ethernet Headers
   ├ Protocol: EGP
   ├ Source MAC address: 52:54:00:12:35:02
   ├ Destination MAC address: 08:00:27:C9:61:5A
→ IP Headers
   ├ Protocol: TCP
   ├ Source IP address: 10.0.2.2
   ├ Destination IP address: 10.0.2.15
```

Listen to incoming packets on eth1 of r1

File: q2.cpp.

Run the "make" command to copy the file to the VM r1 and compile it.

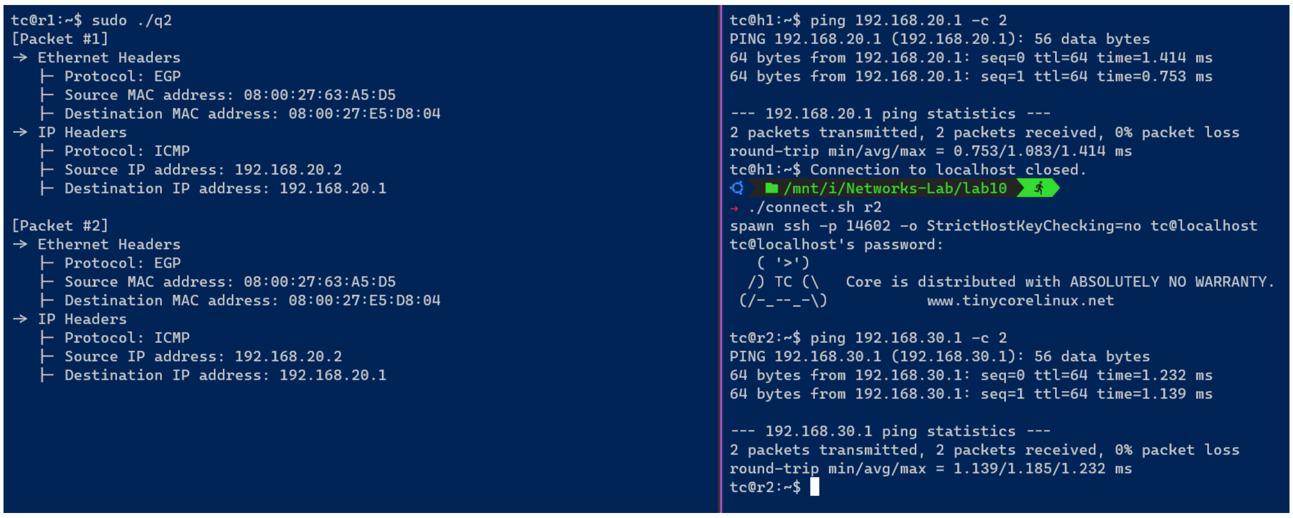
Aim

To listen for incoming packets on the interface *eth1* of *r1* and print its details.

Program Flow

1. Create a socket with *AF_PACKET* domain, *SOCK_RAW* type (this tells the os to create a raw socket) and *htons(ETH_P_IP)* as the protocol (this ensures that the socket listens to only incoming IP packets).
2. Since we need to listen only to *eth1*, I bind the socket to that interface, since this is a raw socket, I use the *sockaddr_ll* struct with the family as *AF_PACKET.* I get the interface index using *if_nametoindex* and set this index as a property of the struct. Then I specify the protocol as *htons(ETH_P_IP)* which says I need to bind and receive only incoming IP packets.
3. Forever,
    3.1. Set the buffer to zeros.
    3.2. Receive data from the socket created in 1 and store it into the buffer (This will only be incoming packets since we bound the socket to the interface in 2).
    3.3. If there was an error while receiving, print the error, close the socket and quit the program.
    3.4. Print the details in the buffer by extracting the IP headers and ethernet headers using the appropriate structs.

Figure 2: The program listens to incoming packets on eth1 and prints some parts of the ethernet and IP headers. When the ping came from h1 (on eth1) the packet details are printed and we can also see that when the ping came from r2 (on eth2), the packets weren't received or printed by the program.



```
tc@r1:~$ sudo ./q2
[Packet #1]
→ Ethernet Headers
  ├ Protocol: EGP
  ├ Source MAC address: 08:00:27:63:A5:D5
  ├ Destination MAC address: 08:00:27:E5:D8:04
→ IP Headers
  ├ Protocol: ICMP
  ├ Source IP address: 192.168.20.2
  ├ Destination IP address: 192.168.20.1

[Packet #2]
→ Ethernet Headers
  ├ Protocol: EGP
  ├ Source MAC address: 08:00:27:63:A5:D5
  ├ Destination MAC address: 08:00:27:E5:D8:04
→ IP Headers
  ├ Protocol: ICMP
  ├ Source IP address: 192.168.20.2
  ├ Destination IP address: 192.168.20.1
```

```
tc@h1:~$ ping 192.168.20.1 -c 2
PING 192.168.20.1 (192.168.20.1): 56 data bytes
64 bytes from 192.168.20.1: seq=0 ttl=64 time=1.414 ms
64 bytes from 192.168.20.1: seq=1 ttl=64 time=0.753 ms

--- 192.168.20.1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.753/1.083/1.414 ms
tc@h1:~$ Connection to localhost closed.
  /mnt/i/Networks-Lab/lab10
→ ./connect.sh r2
spawn ssh -p 14602 -o StrictHostKeyChecking=no tc@localhost
tc@localhost's password:
   ( '>')
  /) TC (\   Core is distributed with ABSOLUTELY NO WARRANTY.
 (/-_--_-\)           www.tinycorelinux.net

tc@r2:~$ ping 192.168.30.1 -c 2
PING 192.168.30.1 (192.168.30.1): 56 data bytes
64 bytes from 192.168.30.1: seq=0 ttl=64 time=1.232 ms
64 bytes from 192.168.30.1: seq=1 ttl=64 time=1.139 ms

--- 192.168.30.1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 1.139/1.185/1.232 ms
tc@r2:~$
```

Building a router that can transfer packets across the subnets

File: q3.cpp.

Run the "make" command to copy the file to the VM r1 and compile it.

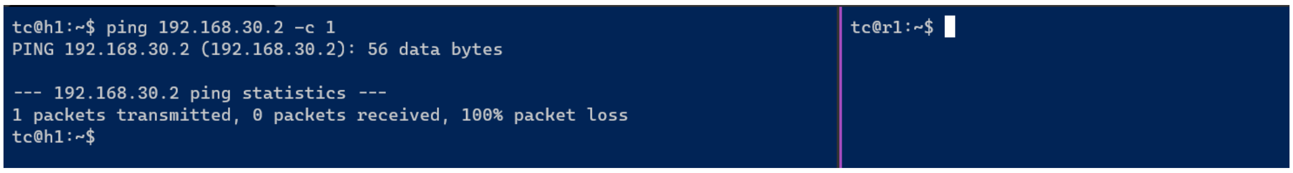
Aim

To enhance the program created in the previous exercise to transfer IP packets across the subnets by ignoring duplicate packets.

Program Flow

1. To avoid duplication ping replies from eth1 and eth2, I get the IP addresses of the interfaces using *ioctl* and compare them, later on, to decide whether to send the packet or not.
2. I create two receiving sockets (one for eth1 and one for eth2) and a sending socket to send packets to both the subnets.
3. I create the *sockaddr_ll* structs for both the receiving sockets, specifying the interfaces they need to bind to.
4. I bind both the receiving sockets to their respective interfaces using the structs created in 3 and close the sockets and exit in case of any errors while binding.
5. I use a file descriptor set to know which socket has received a packet and read from that socket. I clear this file descriptor set (indicating none of the sockets is ready to be read).
6. Forever,
   - 6.1. I clear the buffer and set the bits of the two receiving file descriptors in the *readfds* set.
   - 6.2. Then I block on the *select* call saying I need to be notified only when any of the two sockets are ready to be read (have incoming data).
   - 6.3. Now, I check which socket among the two receiving sockets have incoming data and read the data into the buffer from the appropriate socket.
   - 6.4. I create the *iphdr* structure from the buffer by offsetting it with the size of ethernet headers since IP headers start after the ethernet headers.
   - 6.5. If the packet's intended destination is either *eth1* or *eth2*, I print the received packet and continue.
   - 6.6. If the packet wasn't received on these two interfaces, I ignore sending these packets and continue.
   - *6.7.* If the packet received needs to be transferred across the subnets, I read the destination address from the *daddr* field of the IP header constructed in 6.4 and create a *sockaddr_in* structure with the destination address in the *sin_addr.s_addr* field.
   - *6.8.* Then using the *sendto* function, the sending socket created earlier and the *sockaddr_in* struct created in the previous step, I send the received buffer in 6.3 by removing the ethernet headers since this is not a packet socket, ethernet headers will be assigned automatically.
     Note: It is enough to use a single sending socket for both the interfaces since the appropriate interface to be used for sending will be decided depending on the subnet.
   - 6.9. Now I got back to 6.1 and start performing the same steps again.

Figure 3: Pinging r2 without running q3 doesn't transfer the ICMP packet across the subnets. (100% loss)

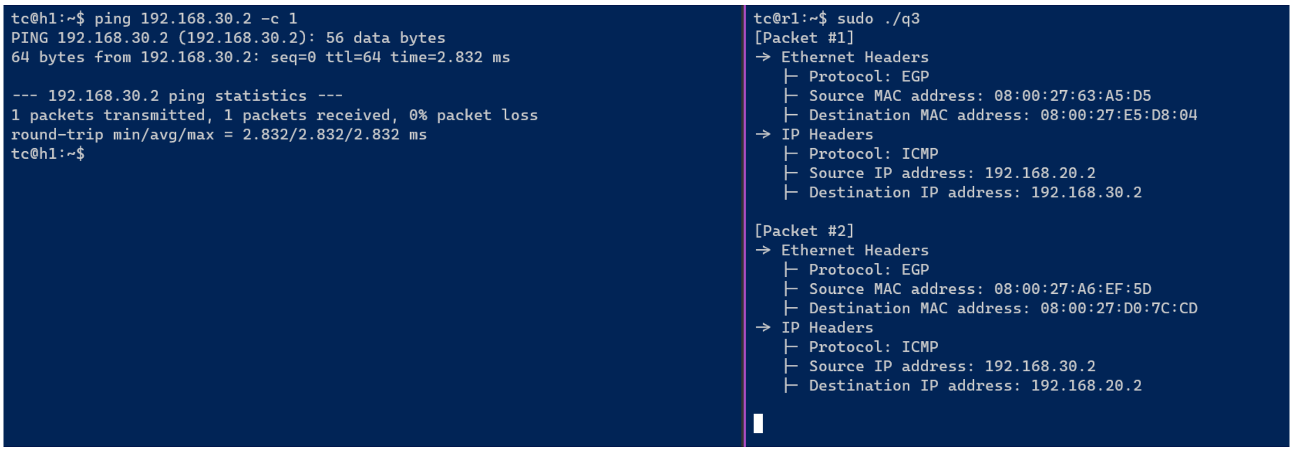


```
tc@h1:~$ ping 192.168.30.2 -c 1
PING 192.168.30.2 (192.168.30.2): 56 data bytes

--- 192.168.30.2 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
tc@h1:~$
```

```
tc@r1:~$
```

Figure 4: Pinging r2 after running q3 transfers the ICMP packet across the subnets (100% transmitted).



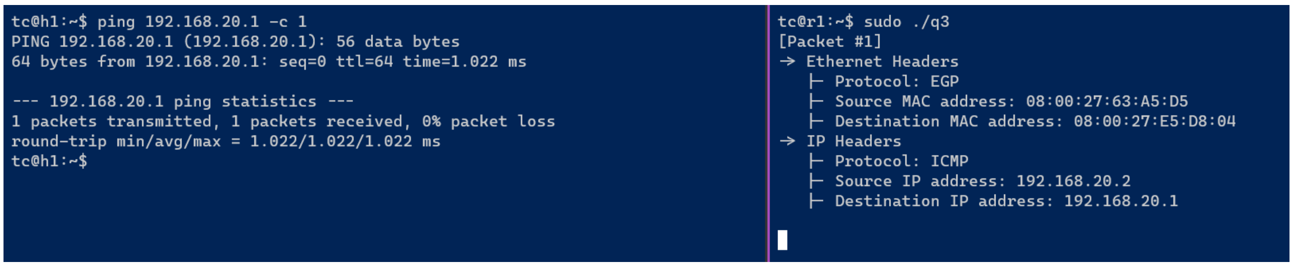
```
tc@h1:~$ ping 192.168.30.2 -c 1
PING 192.168.30.2 (192.168.30.2): 56 data bytes
64 bytes from 192.168.30.2: seq=0 ttl=64 time=2.832 ms

--- 192.168.30.2 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 2.832/2.832/2.832 ms
tc@h1:~$
```

```
tc@r1:~$ sudo ./q3
[Packet #1]
→ Ethernet Headers
   ├ Protocol: EGP
   ├ Source MAC address: 08:00:27:63:A5:D5
   ├ Destination MAC address: 08:00:27:E5:D8:04
→ IP Headers
   ├ Protocol: ICMP
   ├ Source IP address: 192.168.20.2
   ├ Destination IP address: 192.168.30.2

[Packet #2]
→ Ethernet Headers
   ├ Protocol: EGP
   ├ Source MAC address: 08:00:27:A6:EF:5D
   ├ Destination MAC address: 08:00:27:D0:7C:CD
→ IP Headers
   ├ Protocol: ICMP
   ├ Source IP address: 192.168.30.2
   ├ Destination IP address: 192.168.20.2
```

Figure 5: Pinging r1 from h1 does not send duplicate packets since q3 doesn't resend ICMP packets directed at it either through eth1 or eth2.

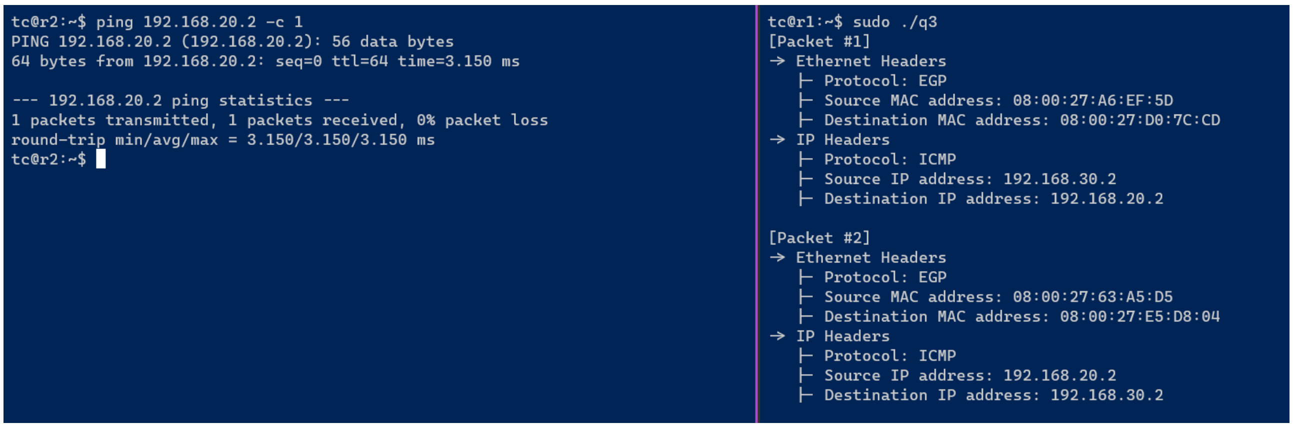


```
tc@h1:~$ ping 192.168.20.1 -c 1
PING 192.168.20.1 (192.168.20.1): 56 data bytes
64 bytes from 192.168.20.1: seq=0 ttl=64 time=1.022 ms

--- 192.168.20.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 1.022/1.022/1.022 ms
tc@h1:~$
```

```
tc@r1:~$ sudo ./q3
[Packet #1]
→ Ethernet Headers
   ├ Protocol: EGP
   ├ Source MAC address: 08:00:27:63:A5:D5
   ├ Destination MAC address: 08:00:27:E5:D8:04
→ IP Headers
   ├ Protocol: ICMP
   ├ Source IP address: 192.168.20.2
   ├ Destination IP address: 192.168.20.1
```

Figure 6: Pinging h1 from r2 also succeeds, this shows that r1 transfers packets across both the subnets.



```
tc@r2:~$ ping 192.168.20.2 -c 1
PING 192.168.20.2 (192.168.20.2): 56 data bytes
64 bytes from 192.168.20.2: seq=0 ttl=64 time=3.150 ms

--- 192.168.20.2 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 3.150/3.150/3.150 ms
tc@r2:~$
```

```
tc@r1:~$ sudo ./q3
[Packet #1]
→ Ethernet Headers
   ├ Protocol: EGP
   ├ Source MAC address: 08:00:27:A6:EF:5D
   ├ Destination MAC address: 08:00:27:D0:7C:CD
→ IP Headers
   ├ Protocol: ICMP
   ├ Source IP address: 192.168.30.2
   ├ Destination IP address: 192.168.20.2

[Packet #2]
→ Ethernet Headers
   ├ Protocol: EGP
   ├ Source MAC address: 08:00:27:63:A5:D5
   ├ Destination MAC address: 08:00:27:E5:D8:04
→ IP Headers
   ├ Protocol: ICMP
   ├ Source IP address: 192.168.20.2
   ├ Destination IP address: 192.168.30.2
```

Important points

1. Since all the files use some common methods, I have defined them in common.h so if you are copying the files manually to r1, please ensure that these files are copied along with the others. But you can just the ***make*** command and it will take care of all the copying.
2. All files must be run as the root user. Use ***sudo*** if you face any errors while creating sockets.
3. To copy the files to r1 and compile them, just run the ***make*** command. This copies all the necessary files to r1, compiles them and connects to r1 all at once.