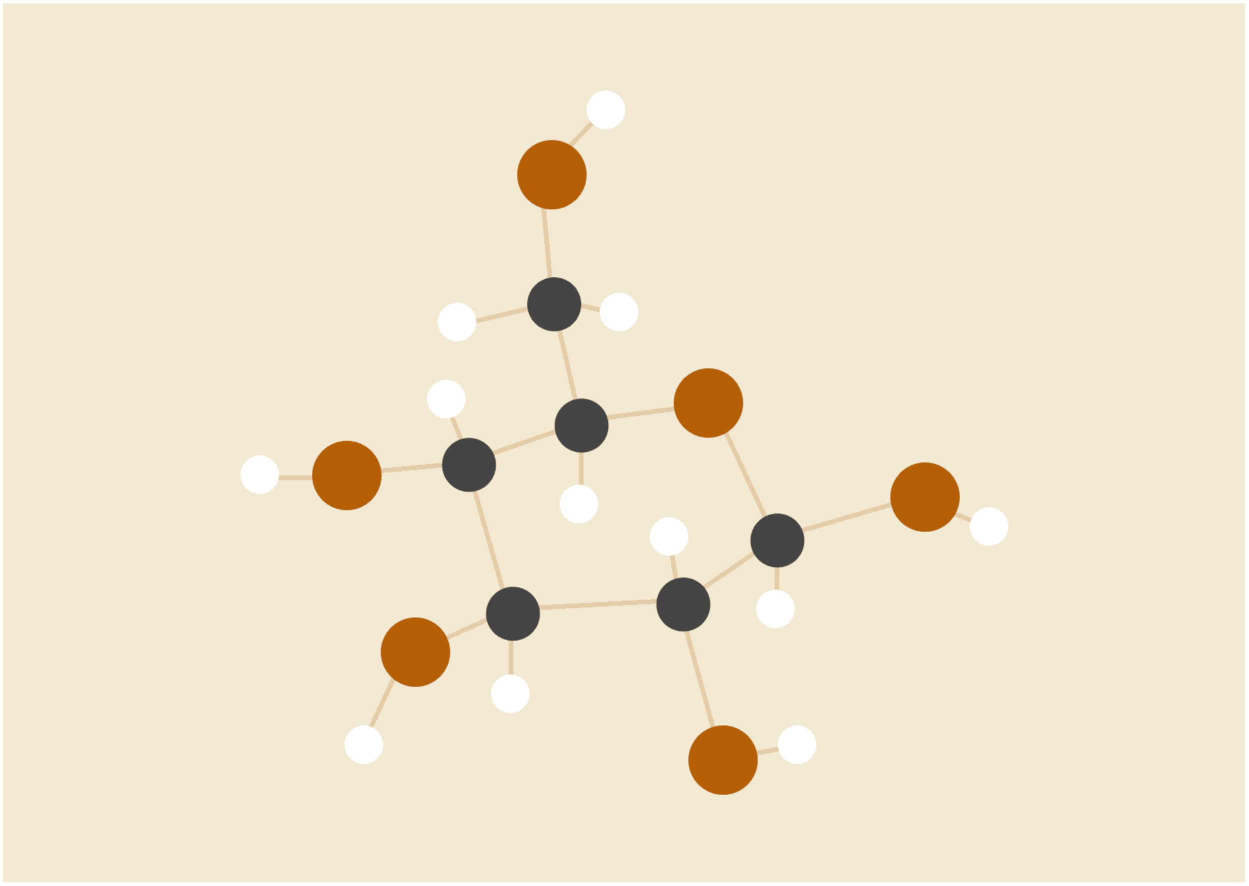
Computer Networks Lab 7



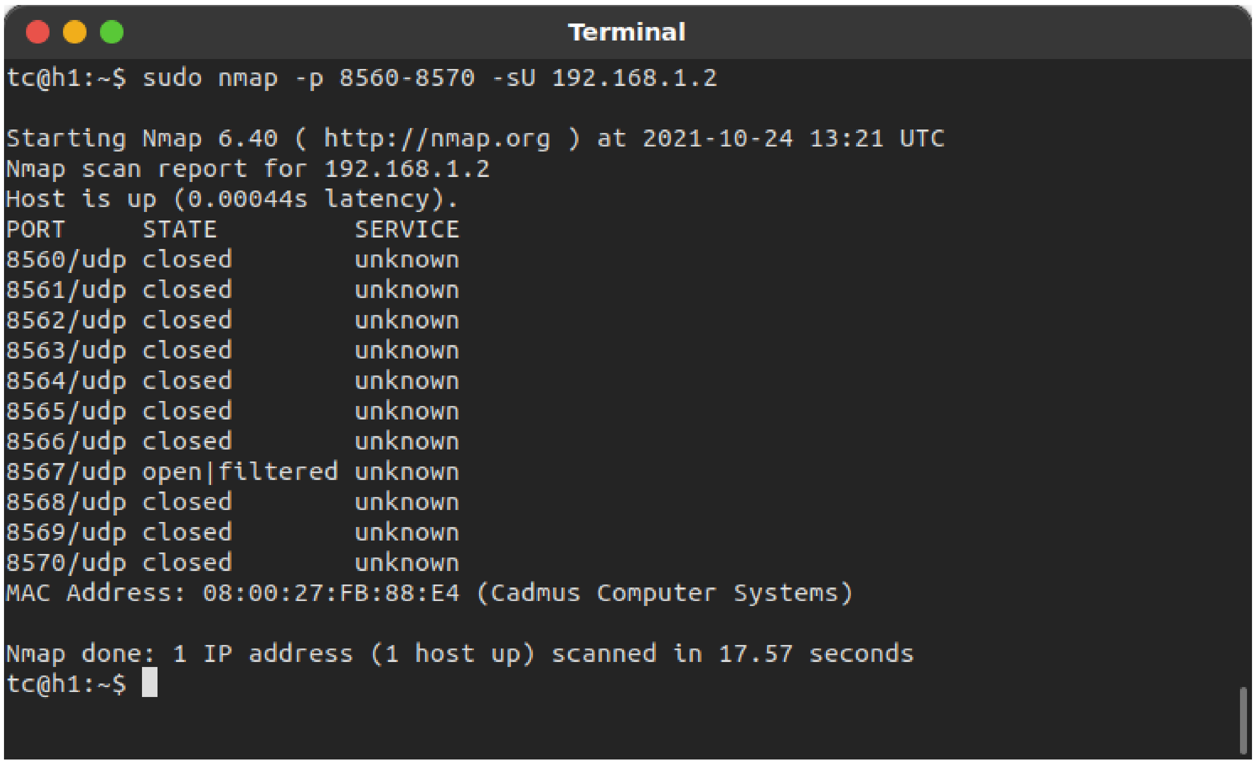
Alisetti Sai Vamsi

24/10/2021

111801002

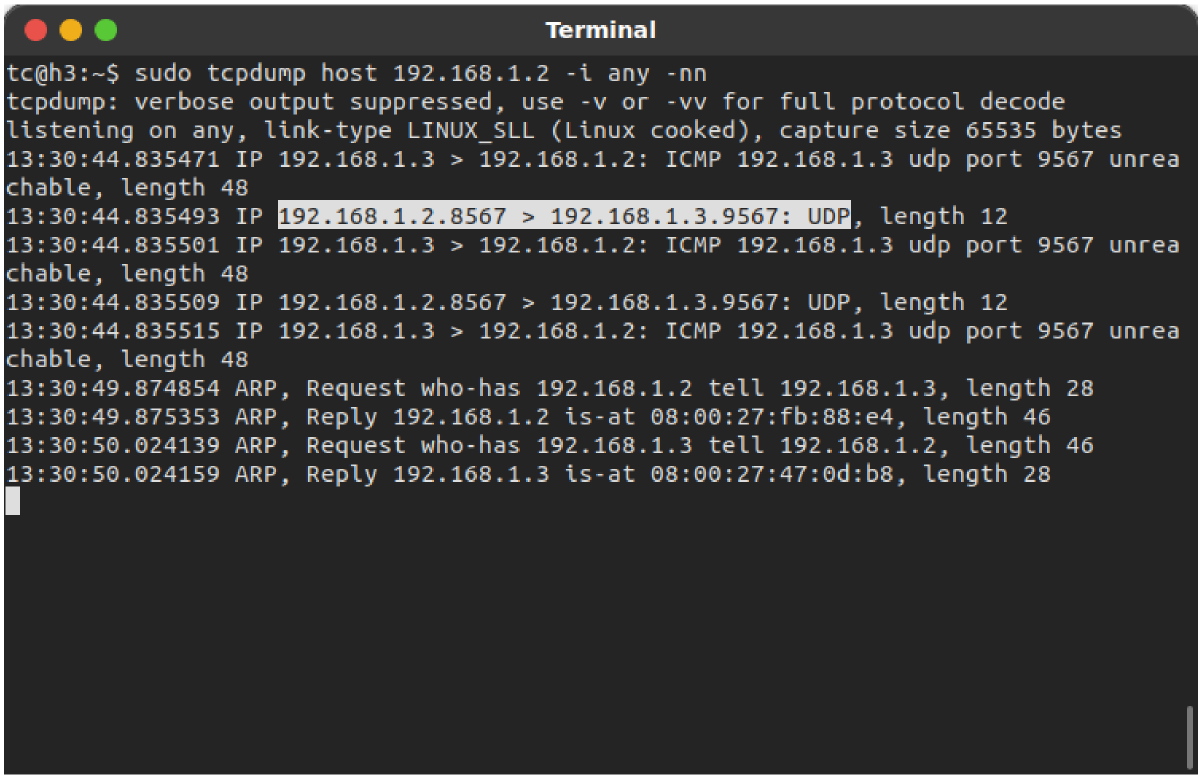
1. Finding Ports X & Y.

For finding the port X, we can use the functionality of nmap to scan active UDP ports by using the -sU flag.

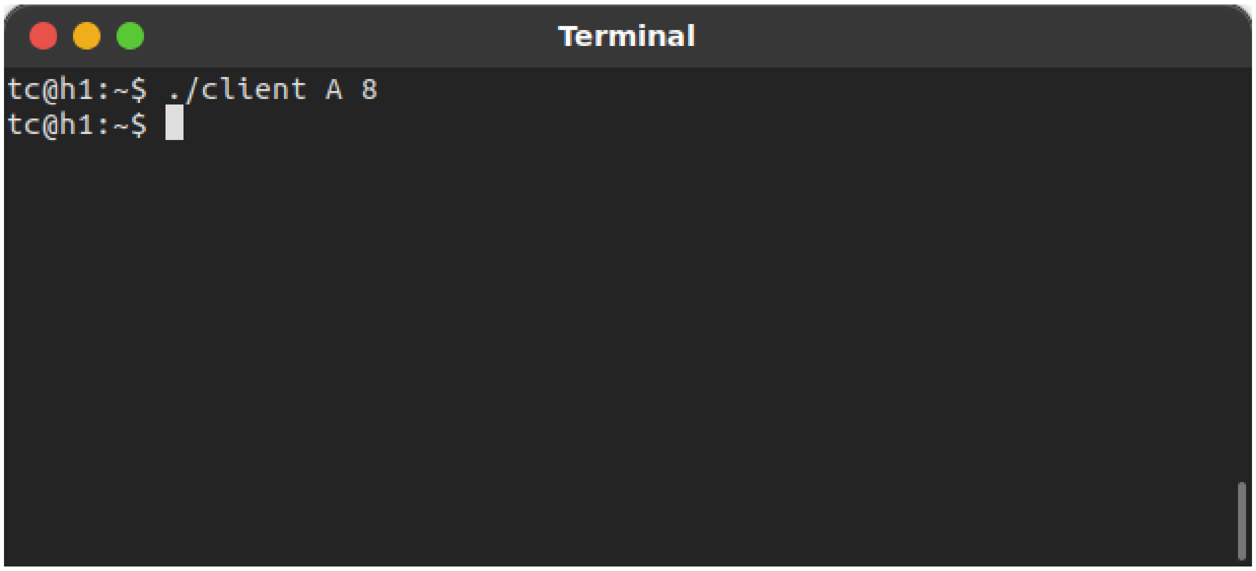


```
tc@h1:~$ sudo nmap -p 8560-8570 -sU 192.168.1.2

Starting Nmap 6.40 ( http://nmap.org ) at 2021-10-24 13:21 UTC
Nmap scan report for 192.168.1.2
Host is up (0.00044s latency).
PORT     STATE         SERVICE
8560/udp closed        unknown
8561/udp closed        unknown
8562/udp closed        unknown
8563/udp closed        unknown
8564/udp closed        unknown
8565/udp closed        unknown
8566/udp closed        unknown
8567/udp open|filtered unknown
8568/udp closed        unknown
8569/udp closed        unknown
8570/udp closed        unknown
MAC Address: 08:00:27:FB:88:E4 (Cadmus Computer Systems)

Nmap done: 1 IP address (1 host up) scanned in 17.57 seconds
tc@h1:~$
```

We can see that the active UDP port on h2(192.168.1.2) is 8567. The state is open | filtered because h2 does not send any response back to h1.



```
tc@h3:~$ sudo tcpdump host 192.168.1.2 -i any -nn
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 65535 bytes
13:30:44.835471 IP 192.168.1.3 > 192.168.1.2: ICMP 192.168.1.3 udp port 9567 unrea
chable, length 48
13:30:44.835493 IP 192.168.1.2.8567 > 192.168.1.3.9567: UDP, length 12
13:30:44.835501 IP 192.168.1.3 > 192.168.1.2: ICMP 192.168.1.3 udp port 9567 unrea
chable, length 48
13:30:44.835509 IP 192.168.1.2.8567 > 192.168.1.3.9567: UDP, length 12
13:30:44.835515 IP 192.168.1.3 > 192.168.1.2: ICMP 192.168.1.3 udp port 9567 unrea
chable, length 48
13:30:49.874854 ARP, Request who-has 192.168.1.2 tell 192.168.1.3, length 28
13:30:49.875353 ARP, Reply 192.168.1.2 is-at 08:00:27:fb:88:e4, length 46
13:30:50.024139 ARP, Request who-has 192.168.1.3 tell 192.168.1.2, length 46
13:30:50.024159 ARP, Reply 192.168.1.3 is-at 08:00:27:47:0d:b8, length 28
```



```
tc@h1:~$ ./client A 8
tc@h1:~$ 
```

For finding port Y we use the functionality of tcpdump and listen on the incoming packets of h3. Therefore once we send the UDP packet from h1 to h2, h2 will be sending a response to h3 through a UDP packet which can be displayed by tcpdump.

2. Client: (Determining End of Transmission using Byte Stuffing)

Invocation of the program:

```
$ gcc client.c -o client -lm
$ ./client <string> <m value>
```

Algorithm Outline:

1. Calculate value of r from m
2. Create socket and link it with the port and IP of h2(192.168.1.2)
3. Generate FLAG and ESC frames based on the value of m
4. Convert the message from command line to binaryString
5. if m is equal to the length the binaryString
    a. check if the binaryString matches either FLAG or ESC
    b. if yes then encode and send an ESC flag prior to this (indicates that this is not the end of transmission)
    c. encode the binaryString and send to the server (Note that encoding is done in the sendToServer Function)
6. Else
    a. m is greater than the length of the binaryString then
    b. Perfom padding to the binaryString upto m
    c. check if the binaryString matches either FLAG or ESC
        i. if yes then send an ESC flag prior to this
    d. encode the paddedString and send to the server
7. m is less than the length of the binaryString then
    a. Cut the binary string into strings of length m (frames)
    b. Check if this frame matches either FLAG or ESC
        i. if it does then encode and send an ESC flag prior to this
    c. Encode and send the frame to the server
8. Send the final FLAG frame

3. Server:

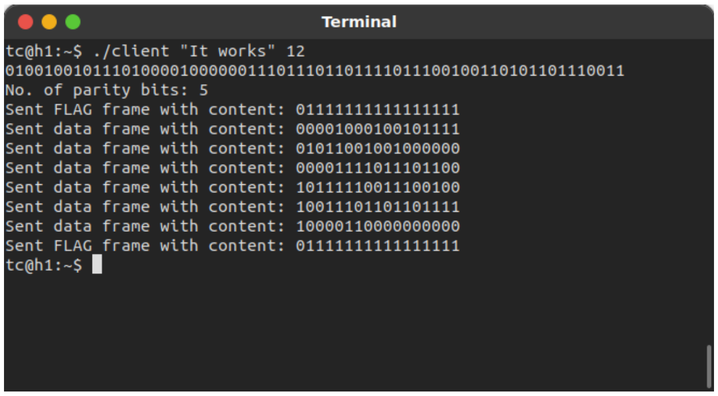
Invocation of the program:

```
$ gcc server.c -o server -lm
$ ./server
```

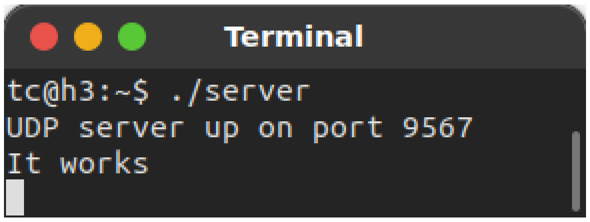
Algorithm Outline:

```
1. Loop Infinite
   a. Receive the first FLAG Frame and compute m and r from the length(n) of
      received codeword
   b. Generate the FLAG and ESC frames from the value of m
   c. Loop Infinite
      i.    current frame = Receive the next frame
      ii.   decode frame
      iii.  Handle the FLAG ESC cases and appropriately concatenate the
            decoded frames
      iv.   prev frame = current frame (handled by using an integer flag)
      v.    Once you find the finish FLAG then
      vi.   Check for padding
      vii.  if padding found then truncate the final string
      viii. print the final message
      ix.   break out of this loop
```

All the functions have been explained in detail in the comments written in the code, and hence omitting the explanation of the working of function in the report.



```
tc@h1:~$ ./client "It works" 12
0100100101110100001000000111011101101111011100100110101101110011
No. of parity bits: 5
Sent FLAG frame with content: 01111111111111111
Sent data frame with content: 00001000100101111
Sent data frame with content: 01011001001000000
Sent data frame with content: 00001111011101100
Sent data frame with content: 10111110011100100
Sent data frame with content: 10011101101101111
Sent data frame with content: 10000110000000000
Sent FLAG frame with content: 01111111111111111
tc@h1:~$ 
```



```
tc@h3:~$ ./server
UDP server up on port 9567
It works
```