COMPUTER NETWORKS JOURNAL

Name : Suraj Yadav

Roll No : 20291

Class : TyBsc

Subject : Computer Science

Paper : Computer Networks

Paper Code : CSC105

INDEX

Sr. No.	Title	Date
1	Create scenario and study the performance of network with star topology through NS-2 simulation.	12-10-2022
2	To study implementation of framing using Bit stuffing.	07-09-2022
3	To study implementation of framing using Bit destuffing	07-09-2022
4	To study implementation of framing using Character stuffing.	14-09-2022
5	To study implementation of framing using Character destuffing.	21-09-2022
6	To study Implementation of Dijkstra's algorithm for Shortest Path Routing.	19-10-2022
7	To study configuration of TCP/IP on a desktop.	18-10-2022
8	To study the use of diagnostic Network Commands: ping, traceroute, netstat, nslookup.	26-10-2022
9	Analyze network traffic using network protocol analyzer tool like ethereal (wireshark) or tcpdump.	26-10-2022
10	To study IP Address manipulation: Extract Network ID and Host ID given netmask.	19-10-2022
11	To study Implementation of IP Fragmentation and Reassembly.	19-10-2022
12	To study simple TCP client and server application to send String from Client to server.	26-10-2022
13	To study simple TCP client and server application to perform Arithmetic operations.	26-10-2022
14	To study simple TCP client and server application to reverse a string.	26-10-2022

PRACTICAL 1

Aim:
Create scenario and study the performance of network with star topology through NS-2 simulation.

Theory:
A network simulator is software that predicts the behavior of a computer network. Since communication networks have become too complex for traditional analytical methods to provide an accurate understanding of system behavior, network simulators are used. In simulators, the computer network is modeled with devices, links, applications, etc., and the network performance is reported. Simulators come with support for the most popular technologies and networks in use today such as 5G, Internet of Things (IoT), Wireless LANs, mobile ad hoc networks, wireless sensor networks, vehicular ad hoc networks, cognitive radio networks, LTE etc.

NS-2 is one of the popular Network Simulation Software.

CODE :

```
#Create a simulator object
set ns [new Simulator]

#Define different colors for data flows (for NAM)
$ns color 1 Blue
$ns color 2 Red

#Open the NAM trace file
set nf [open out.nam w]
$ns namtrace-all $nf

#Define a 'finish' procedure
proc finish {}
{
        global ns nf $ns flush-trace

        #Close the NAM trace file
        close $nf

        #Execute NAM on the trace file
        exec nam out.nam & exit 0
}

#Create four nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
```

```
#Create links between the nodes
$ns duplex-link $n0 $n2 2Mb 10ms DropTail
$ns duplex-link $n1 $n2 2Mb 10ms DropTail
$ns duplex-link $n2 $n3 1.7Mb 20ms DropTail
#Set Queue Size of link (n2-n3) to 10
$ns queue-limit $n2 $n3 10

#Give node position (for NAM)
$ns duplex-link-op $n0 $n2 orient right-down
$ns duplex-link-op $n1 $n2 orient right-up
$ns duplex-link-op $n2 $n3 orient right

#Monitor the queue for link (n2-n3). (for NAM)
$ns duplex-link-op $n2 $n3 queuePos 0.5

#Setup a TCP connection
set tcp [new Agent/TCP] $tcp set class_ 2
$ns attach-agent
$n0 $tcp set sink [new Agent/TCPSink]
$ns attach-agent
$n3 $sink $ns connect $tcp
$sink $tcp set fid_ 1

#Setup a FTP over TCP connection
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ftp set type_ FTP

#Setup a UDP connection
set udp [new Agent/UDP]
$ns attach-agent $n1 $udp
set null [new Agent/Null]
$ns attach-agent $n3 $null
$ns connect $udp $null
$udp set fid_ 2

#Setup a CBR over UDP connection
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp $cbr set type_ CBR
$cbr set packet_size_ 1000
$cbr set rate_ 1mb
$cbr set random_ false

#Schedule events for the CBR and FTP agents
$ns at 0.1 "$cbr start"
$ns at 1.0 "$ftp start"
$ns at 4.0 "$ftp stop"
$ns at 4.5 "$cbr stop"

#Detach tcp and sink agents (not really necessary)
```

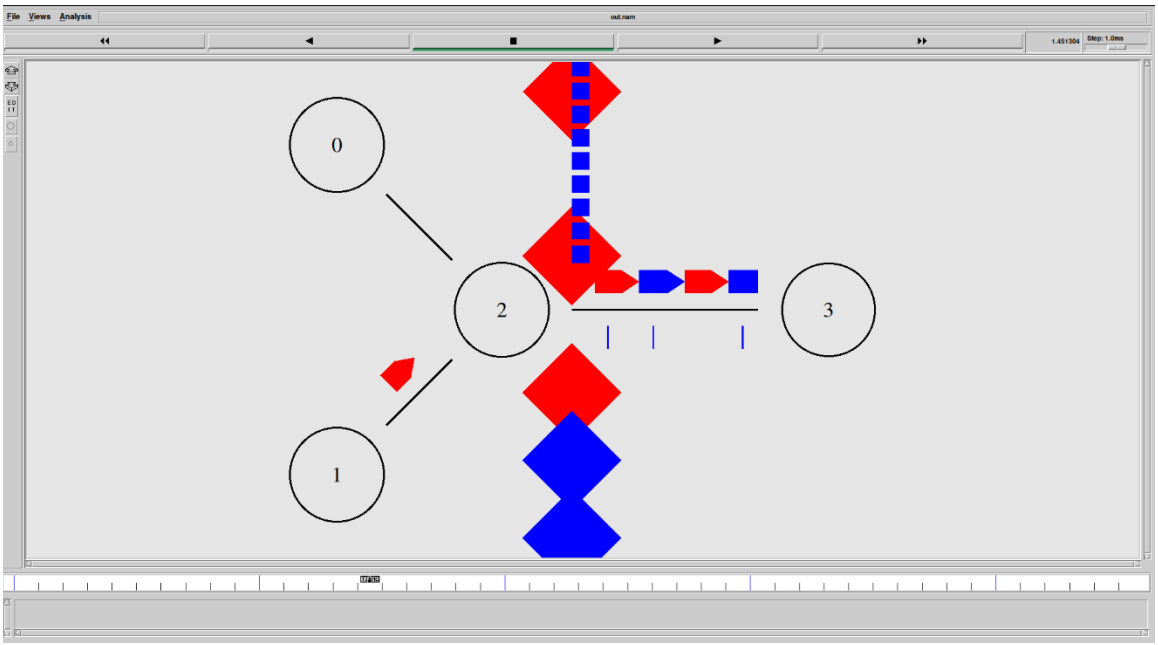
```
$ns at 4.5
"$ns detach-agent $n0 $tcp ;
$ns detach-agent $n3 $sink"

#Call the finish procedure after 5 seconds of simulation time
$ns at 5.0 "finish"

#Print CBR packet size and interval
puts "CBR packet size = [$cbr set packet_size_]"
puts "CBR interval = [$cbr set interval_]"

#Run the simulation
$ns run
```

OUTPUT :



Conclusion:
The performance of network with star topology through NS-2 simulation was studied successfully.

PRACTICAL 2

Aim:
To study implementation of framing using Bit stuffing.

Theory:
Data link layer is responsible for something called Framing, which is the division of stream of bits from network layer into manageable units (called frames). Frames could be of fixed size or variable size. In variable-size framing, we need a way to define the end of the frame and the beginning of the next frame.

Bit stuffing is the insertion of non information bits into data.

Applications of Bit Stuffing –
1. synchronize several channels before multiplexing
2. rate-match two single channels to each other
3. run length limited coding

Example of bit stuffing –
Bit sequence: 1101011111010111111010111111110 (without bit stuffing)
Bit sequence: 1101011111001011111010101111110110 (with bit stuffing)

After 5 consecutive 1-bits, a 0-bit is stuffed. Stuffed bits are marked bold.

CODE :

```
#include<stdio.h>
#include<string.h>

char * bitstuff(char str[],char stack[])
{

        int n=strlen(str),count=0,stack_index=0,i;
        for(i=0;i<n;i++)
        {
                if(str[i]=='1')
                {
                        count++;

                        stack[stack_index++]=str[i];
                        if(count==5)
                        {

                                stack[stack_index++]='0';
                                count=0;
                        }
                }
                else
                {
```

```
                        stack[stack_index++]=str[i];
                        count=0;
                }
        }
        stack[stack_index]='\0';
        return stack;
}

void main()
{
        char str[40],stack[40];

        int n=0,i=0;
         printf("\nEnter length of string");
        scanf("\n%d",&n);
        printf("\nEnter the string");
        for(int i=0;i<n;i++)
        {
                scanf("\n%c",&str[i]);
        }
        for(i;i<n;i++)
        {

                if(str[i]!='0' && str[i]!='1')
                {
                        printf("\n Enter valid input");

                }
        }
        printf("\nInput:\n");
        printf("%s",str);



        printf("\nOutput:\n");
        printf("%s",bitstuff(str,stack));
}
```

OUTPUT :

Enter length of string
7

Enter the string
1111111

Input:
1111111
Output:
11111011

Conclusion:
The implementation of framing using Bit stuffing was studied successfully.

PRACTICAL 3

Aim:
To study implementation of framing using Bit destuffing.

Theory:
When data frames are revived the non-information bits need to be removed which is called destuffing/unstuffing.

CODE :

```
#include <stdio.h>
#include <string.h>
char *destuff(char str[], char stack[])
{

   int n = strlen(str), count = 0, stack_index = 0, i;
   for (i = 0; i < n; i++)
   {
      if (str[i] == '1')
      {
         count++;

         stack[stack_index++] = str[i];

         if (count == 5)
         {

            i++;
            if (str[i] != '0')
            {
               printf("\n Invalid input");

               break;
            }
            count = 0;
         }
      }
      else
      {

         stack[stack_index++] = str[i];

         count = 0;
      }
   }
   stack[stack_index] = '\0';
   return stack;
}
void main()
{
   char str[40], stack[40];
```

```
    int n = 0, i = 0;
    printf("\nEnter length of string");
    scanf("\n%d", &n);
    printf("\nEnter the string");
    for (int i = 0; i < n; i++)
    {
        scanf("\n%c", &str[i]);
    }
    for (i; i < n; i++)
    {

        if (str[i] != '0' && str[i] != '1')
        {
            printf("\n Enter valid input");
        }
    }
    printf("\nInput:\n");
    printf("%s", str);

    printf("\nOutput:\n");
    printf("%s", destuff(str, stack));
}
```

OUTPUT :

Enter length of string
7

Enter the string
1111101

Input:
1111101
Output:
111111

Conclusion:
The implementation of framing using Bit destuffing was studied successfully.

PRACTICAL 4

Aim:
To study implementation of framing using Character stuffing.

Theory:
In variable-size framing at the data link layer, we need to define a way to separate one frame from the next. Byte stuffing is employed to accomplish the task. In byte stuffing an 8-bit flag (‘F’) is added at the beginning and at the end of the frame, thereby distinguishing one frame from the next. Therefore, every time a flag sequence (‘F’) is encountered, it signifies the beginning or end of a frame. This, ingenious scheme, however would give rise to a discrepancy, if the flag pattern (‘F’) would occur within the data carried by the frame itself. Byte stuffing comes to the rescue here, by stuffing the original data with an extra 8-bit escape sequence (‘E’) before the flag pattern, whenever it occurred within the data carried by a frame. The receiver would then have to de-stuff the escape sequence, in order to obtain the original data. A simple question that might arise at this juncture is, what if the escape sequence (‘E’) formed a part of the data to be sent! This scenario is handled in exactly the same way as described above, i.e an extra 8-bit escape sequence (‘E’) is added to the original data before the escape sequence that formed part of the data. In character-oriented protocols, where data to be carried are 8-bit characters, byte stuffing is employed to handle the problems discussed above.

CODE :

```
#include <stdio.h>
#include <string.h>
char output[40];
char *stuff(char input[])
{
   int n = strlen(input);
   int j = 1;
   output[0] = '$';
   for (int i = 0; i < n; i++)
   {
      if (input[i] == '$' || input[i] == '#')
      {
         output[j++] = '#';
      }
      output[j++] = input[i];
   }
   output[j++] = '$';
   output[j] = '\0';
   return output;
}
int main()
{
   char input[40];
   int n = 0, i = 0;
   printf("\nEnter length of string");
   scanf("\n%d", &n);
   printf("\nEnter the string");
   for (int i = 0; i < n; i++)
   {
      scanf("\n%c", &input[i]);
   }
```

```
    printf("\ninput\n");
    printf("%s", input);
    printf("\noutput\n");
    printf("%s", stuff(input));
}
```

OUTPUT :

```
Enter length of string
9

Enter the string
Dollar$$$

input
Dollar$$$   a
output
$Dollar#$#$#$
```

Conclusion:
The implementation of framing using Character stuffing was studied successfully.

PRACTICAL 5

Aim:
To study implementation of framing using Character destuffing.

CODE :

```
#include <stdio.h>
#include <string.h>
char output[40];
char *destuff(char input[])
{
   int n = strlen(input), j = 0;
   for (int i = 0; i < n; i++)
   {
      if (input[i] == '$')
      {
         continue;
      }
      if (input[i] == '#')
      {
         output[j] = input[++i];
      }
      output[j++] = input[i];
   }
   output[j] = '\0';
   return output;
}
int main()
{
   char input[40];
   int n = 0, i = 0;
   printf("\nEnter length of string");
   scanf("\n%d", &n);
   printf("\nEnter the string");
   for (int i = 0; i < n; i++)
   {
      scanf("\n%c", &input[i]);
   }
   printf("\ninput\n");
   printf("%s", input);
   printf("\noutput\n");
   printf("%s", destuff(input));
}
```

OUTPUT :

Enter length of string
10

Enter the string
$Dollar#$$

input
$Dollar#$$ a
output
Dollar$

Conclusion:
The implementation of framing using Character destuffing was studied successfully.

PRACTICAL 6

Aim:
To study Implementation of Dijkstra's algorithm for Shortest Path Routing.

Theory:
Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph. We generate a SPT (shortest path tree) with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source. Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm
1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3) While sptSet doesn't include all vertices
….a) Pick a vertex u which is not there in sptSet and has a minimum distance value.
….b) Include u to sptSet.
….c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

CODE :

```
#include <stdio.h>

void main()
{
   int cost[10][10],distance[10],path[10][10];
   int n,v,p,row,column,min,index=1,i,j;

   //enter number of nodes
   printf("Enter the number of nodes : ");
   scanf("%d",&n);

   //enter the cost matrix
   printf("Enter the cost matrix :\n");
   for(i=1;i<=n;i++)
   {
      for(j=1;j<=n;j++)
      {
         scanf("%d",&cost[i][j]);
      }
   }

   //user enters node to be visited
```

```
    printf("Enter the node to be visited : ");
    scanf("%d",&v);

    //enter the no of paths for a particular node
    printf("Enter paths for selected node : ");
    scanf("%d",&p);

    //enter the path matrix
    printf("Enter the path matrix :\n");
    for(i=1;i<=p;i++)
    {
       for(j=1;j<=n;j++)
       {
          scanf("%d",&path[i][j]);
       }
    }

    for(i=1;i<=p;i++)
    {
       distance[i]=0;
       row=1;
       for(j=1;j<n;j++)
       {
          if(row!=v)
          {
             //till i visits the last node
             column = path[i][j+1];
             distance[i] = distance[i] + cost[row][column];
          }
          row = column;
       }
    }

    //which distance to be consisdered
    min = distance[1];
    for(i=1;i<=p;i++)
    {
       if(distance[i]<=min)
       {
          min = distance[i];
          index = i;
       }
    }

    printf("min distance is %d\n",min);
    printf("min distance path is : ");
    for(i=1;i<=n;i++)
    {
       if(path[index][i]!=0)
       {
          printf("--->%d",path[index][i]);
       }
    }
}
```

OUTPUT :

```
Enter the number of nodes : 5
Enter the cost matrix :
0
4
0
8
0
4
0
3
0
0
0
3
0
4
0
8
0
4
0
7
0
0
0
7
0
Enter the node to be visited : 5
Enter paths for selected node : 2
Enter the path matrix :
1
2
3
4
5
1
4
5
0
0
min distance is 15
min distance path is : --->1--->4--->5
```

PRACTICAL 7

Aim:
To study configuration of TCP/IP on a desktop.

Theory:
The success of TCP/IP as the network protocol of the Internet is largely because of its ability to connect together networks of different sizes and systems of different types. These networks are arbitrarily defined into three main classes (along with a few others) that have predefined sizes. Each of them can be divided into smaller subnetworks by system administrators. A subnet mask is used to divide an IP address into two parts. One part identifies the host (computer), the other part identifies the network to which it belongs. To better understand how IP addresses and subnet masks work, look at an IP address and see how it's organized.

IP addresses: Networks and hosts
An IP address is a 32-bit number. It uniquely identifies a host (computer or other device, such as a printer or router) on a TCP/IP network. IP addresses are normally expressed in dotted-decimal format, with four numbers separated by periods, such as 192.168.123.132. To understand how subnet masks are used to distinguish between hosts, networks, and subnetworks, examine an IP address in binary notation. For example, the dotted-decimal IP address 192.168.123.132 is (in binary notation) the 32-bit number 11000000010100011110111000100. This number may be hard to make sense of, so divide it into four parts of eight binary digits. These 8-bit sections are known as octets. The example IP address, then, becomes 11000000.10101000.01111011.10000100. This number only makes a little more sense, so for most uses, convert the binary address into dotted-decimal format (192.168.123.132). The decimal numbers separated by periods are the octets converted from binary to decimal notation. For a TCP/IP wide area network (WAN) to work efficiently as a collection of networks, the routers that pass packets of data between networks don't know the exact location of a host for which a packet of information is destined. Routers only know what network the host is a member of and use information stored in their route table to determine how to get the packet to the destination host's network. After the packet is delivered to the destination's network, the packet is delivered to the appropriate host. For this process to work, an IP address has two parts. The first part of an IP address is used as a network address, the last part as a host address. If you take the example 192.168.123.132 and divide it into these two parts, you get 192.168.123. Network .132 Host or 192.168.123.0 - network address. 0.0.0.132 - host address.

Subnet mask
The second item, which is required for TCP/IP to work, is the subnet mask. The subnet mask is used by the TCP/IP protocol to determine whether a host is on the local subnet or on a remote network. In TCP/IP, the parts of the IP address that are used as the network and host addresses aren't fixed. Unless you have more information, the network and host addresses above can't be determined. This information is supplied in another 32-bit number called a subnet mask. The subnet mask is 255.255.255.0 in this example. It isn't obvious what this number means unless you know 255 in binary notation equals 11111111. So, the subnet mask is 11111111.11111111.11111111.00000000. Lining up the IP address and the subnet mask together, the network, and host portions of the address can be separated:
11000000.10101000.01111011.10000100 - IP address (192.168.123.132)
11111111.11111111.11111111.00000000 - Subnet mask (255.255.255.0) The first 24 bits (the number of ones in the subnet mask) are identified as the network address. The last 8

bits (the number of remaining zeros in the subnet mask) are identified as the host address. It gives you the following addresses: 11000000.10101000.01111011.00000000 - Network address (192.168.123.0) 00000000.00000000.00000000.10000100 - Host address (000.000.000.132) So now you know, for this example using a 255.255.255.0 subnet mask, that the network ID is 192.168.123.0, and the host address is 0.0.0.132. When a packet arrives on the 192.168.123.0 subnet (from the local subnet or a remote network), and it has a destination address of 192.168.123.132, your computer will receive it from the network and process it. Almost all decimal subnet masks convert to binary numbers that are all ones on the left and all zeros on the right. Some other common subnet masks are: Decimal Binary 255.255.255.192 1111111.11111111.1111111.11000000 255.255.255.224 1111111.11111111.1111111.11100000 Internet RFC 1878 (available from InterNIC-Public Information Regarding Internet Domain Name Registration Services) describes the valid subnets and subnet masks that can be used on TCP/IP networks.

Network classes

Internet addresses are allocated by the InterNIC, the organization that administers the Internet. These IP addresses are divided into classes. The most common of them are classes A, B, and C. Classes D and E exist, but aren't used by end users. Each of the address classes has a different default subnet mask. You can identify the class of an IP address by looking at its first octet. Following are the ranges of Class A, B, and C Internet addresses, each with an example address:

- Class A networks use a default subnet mask of 255.0.0.0 and have 0-127 as their first octet. The address 10.52.36.11 is a class A address. Its first octet is 10, which is between 1 and 126, inclusive.
- Class B networks use a default subnet mask of 255.255.0.0 and have 128-191 as their first octet. The address 172.16.52.63 is a class B address. Its first octet is 172, which is between 128 and 191, inclusive.
- Class C networks use a default subnet mask of 255.255.255.0 and have 192-223 as their first octet. The address 192.168.123.132 is a class C address. Its first octet is 192, which is between 192 and 223, inclusive.

In some scenarios, the default subnet mask values don't fit the organization needs for one of the following reasons:

- The physical topology of the network
- The numbers of networks (or hosts) don't fit within the default subnet mask restrictions.

The next section explains how networks can be divided using subnet masks.

Subnetting

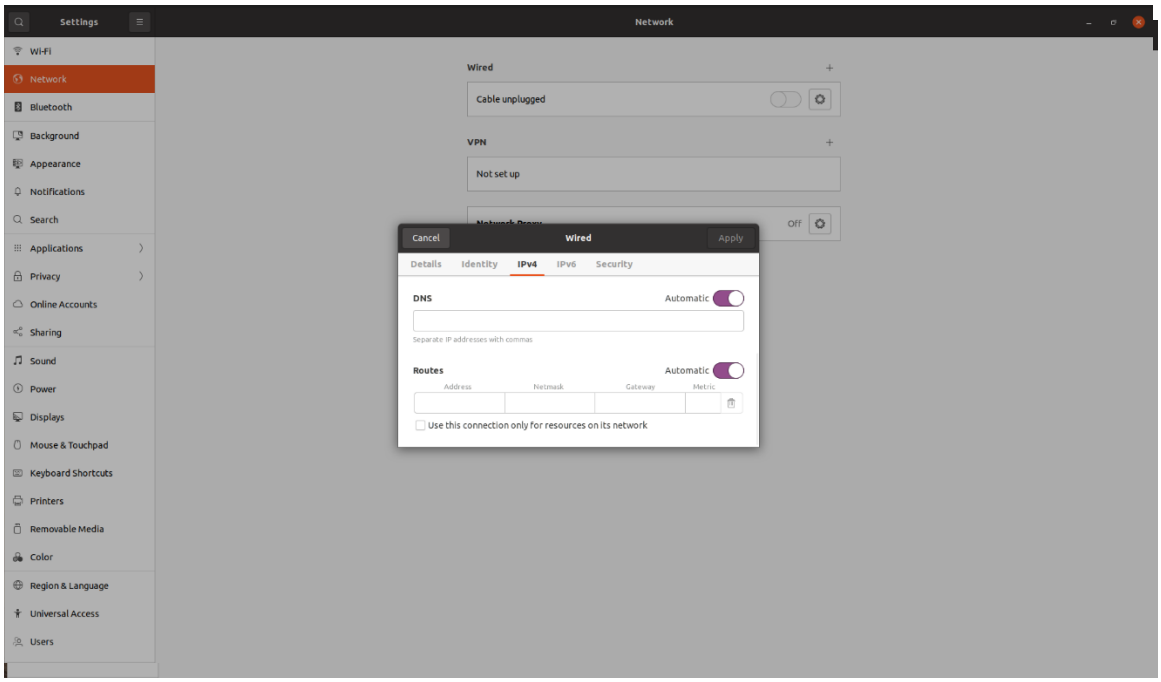
A Class A, B, or C TCP/IP network can be further divided, or subnetted, by a system administrator. It becomes necessary as you reconcile the logical address scheme of the Internet (the abstract world of IP addresses and subnets) with the physical networks in use by the real world. A system administrator who is allocated a block of IP addresses may be administering networks that aren't organized in a way that easily fits these addresses. For example, you have a wide area network with 150 hosts on three networks (in different cities) that are connected by a TCP/IP router. Each of these three networks has 50 hosts. You are allocated the class C network 192.168.123.0. (For illustration, this address is actually from a range that isn't allocated on the Internet.) It means that you can use the addresses 192.168.123.1 to 192.168.123.254 for your 150 hosts. Two addresses that can't be used in your example are 192.168.123.0 and 192.168.123.255 because binary addresses with a host portion of all ones and all zeros are invalid. The zero address is invalid because it's used to specify a network without specifying a host. The 255 address (in binary notation, a host

address of all ones) is used to broadcast a message to every host on a network. Just remember that the first and last address in any network or subnet can't be assigned to any individual host. You should now be able to give IP addresses to 254 hosts. It works fine if all 150 computers are on a single network. However, your 150 computers are on three separate physical networks. Instead of requesting more address blocks for each network, you divide your network into subnets that enable you to use one block of addresses on multiple physical networks. In this case, you divide your network into four subnets by using a subnet mask that makes the network address larger and the possible range of host addresses smaller. In other words, you are 'borrowing' some of the bits used for the host address, and using them for the network portion of the address. The subnet mask 255.255.255.192 gives you four networks of 62 hosts each. It works because in binary notation, 255.255.255.192 is the same as 1111111.11111111.1111111.11000000. The first two digits of the last octet become network addresses, so you get the additional networks 00000000 (0), 01000000 (64), 10000000 (128) and 11000000 (192). (Some administrators will only use two of the subnetworks using 255.255.255.192 as a subnet mask. For more information on this topic, see RFC 1878.) In these four networks, the last six binary digits can be used for host addresses. Using a subnet mask of 255.255.255.192, your 192.168.123.0 network then becomes the four networks 192.168.123.0, 192.168.123.64, 192.168.123.128 and 192.168.123.192. These four networks would have as valid host addresses: 192.168.123.1-62 192.168.123.65-126 192.168.123.129-190 192.168.123.193-254 Remember, again, that binary host addresses with all ones or all zeros are invalid, so you can't use addresses with the last octet of 0, 63, 64, 127, 128, 191, 192, or 255. You can see how it works by looking at two host addresses, 192.168.123.71 and 192.168.123.133. If you used the default Class C subnet mask of 255.255.255.0, both addresses are on the 192.168.123.0 network. However, if you use the subnet mask of 255.255.255.192, they are on different networks; 192.168.123.71 is on the 192.168.123.64 network, 192.168.123.133 is on the 192.168.123.128 network.

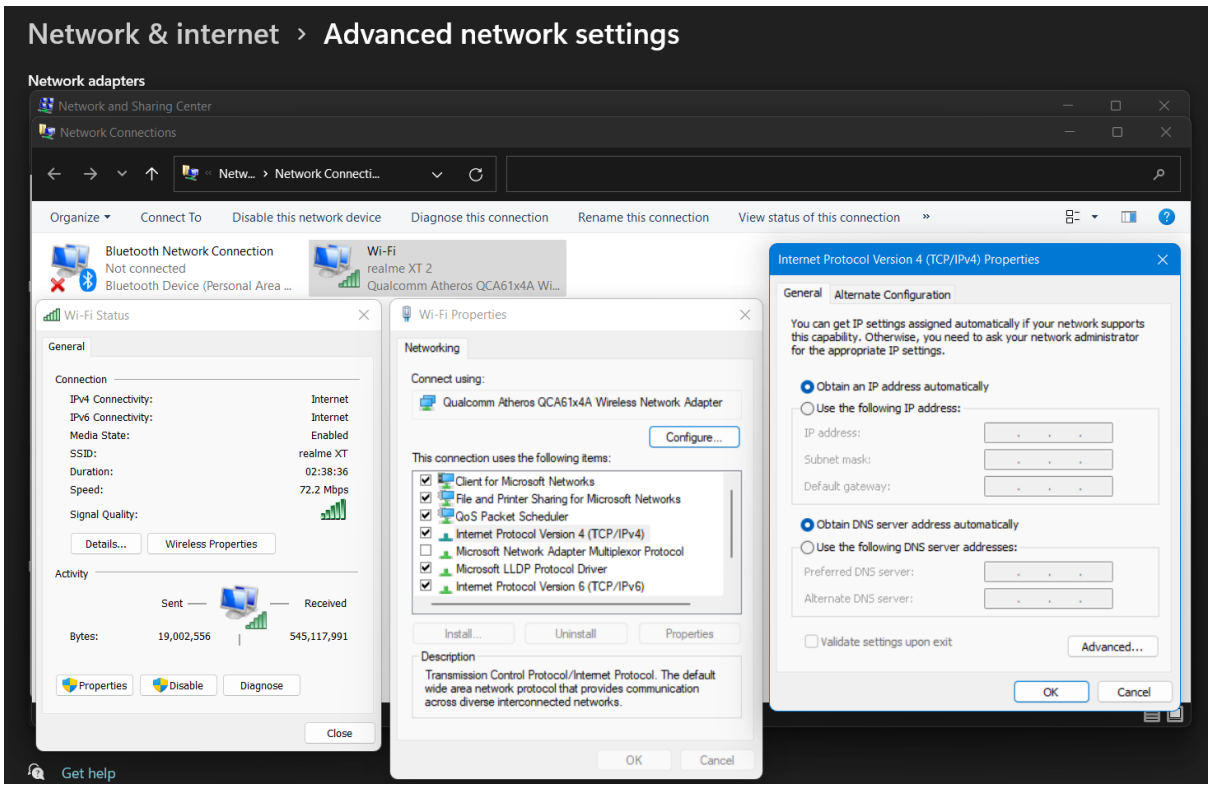
Default gateways

If a TCP/IP computer needs to communicate with a host on another network, it will usually communicate through a device called a router. In TCP/IP terms, a router that is specified on a host, which links the host's subnet to other networks, is called a default gateway. This section explains how TCP/IP determines whether or not to send packets to its default gateway to reach another computer or device on the network. When a host attempts to communicate with another device using TCP/IP, it performs a comparison process using the defined subnet mask and the destination IP address versus the subnet mask and its own IP address. The result of this comparison tells the computer whether the destination is a local host or a remote host. If the result of this process determines the destination to be a local host, then the computer will send the packet on the local subnet. If the result of the comparison determines the destination to be a remote host, then the computer will forward the packet to the default gateway defined in its TCP/IP properties. It's then the responsibility of the router to forward the packet to the correct subnet.

In Linux :



In Windows :



PRACTICAL 8

Aim:
To study the use of diagnostic Network Commands:
ping, traceroute, netstat, nslookup.

Theory:
1. ping
Ping is a program that allows a user to verify that a particular IP address exists and can accept requests. Ping is used diagnostically to ensure that a host computer the user is trying to reach is actually operating. Ping works by sending an Internet Control Message Protocol (ICMP) Echo Request to a specified interface on the network and waiting Internet for a reply. Ping can be used for troubleshooting to test connectivity and determine response time.

2. Traceroute
A traceroute procedure allows you to find out precisely how a data transmission (like a Google search) travelled from your computer to another. Essentially, the traceroute compiles a list of the computers on the network that are involved with a specific Internet activity. The traceroute identifies each computer/server on that list and the amount of time it took the data to get from one computer to the next. If there was a hiccup or interruption in the transfer of data, the traceroute will show where along the chain the problem occurred.

Aside from being somewhat interesting, performing a traceroute also has a very practical use: If someone is having difficulty accessing a particular website or computer, performing a traceroute can help find out where the problem is occurring along the network.

<u>How data travels:</u>
Each computer on the traceroute is identified by its IP address, which are the nine-digit numbers separated by periods that identifies that computer's unique network connection.

Here are a few details regarding a traceroute:

- The journey from one computer to another is known as a hop.
- The amount of time it takes to make a hop is measured in milliseconds.
- The information that travels along the traceroute is known as a packet.

A traceroute readout typically will display three separate columns for the hop time, as each traceroute sends out three separate packets of information to each computer. At the very top of the list, the traceroute will give the limit of how many lines of hops it will display—30 hops is often the maximum number. When a traceroute has difficulty accessing a computer, it will display the message "Request timed out.". Each of the hop columns will display an asterisk instead of a millisecond count.

<u>How to run a traceroute:</u>
On a PC using Windows, you can perform a traceroute using the traceroute utility on the Windows operating system (as long as you are not attempting to tap into heavily secured networks). You'll need to know the domain name, IP address or name of the specific computer you're trying to reach. Using the traceroute utility, you would type "tracert x"—where "x" stands for the IP address, the domain name or the computer name. If using Macintosh OS X or any subsequent versions, you may use either the Terminal program or the network utility to generate a traceroute. The utility will display the traceroute on your screen.

3. Netstat
Netstat is a common command line TCP/IP networking utility available in most versions of Windows, Linux, UNIX and other operating systems. Netstat provides information and statistics about protocols in use and current TCP/IP network connections. (The name derives from the words network and statistics.)

NETSTAT -a -b -e -n -o -p proto -r -s -v interval

-a Displays all connections and listening ports.

-b Displays the executable involved in creating each connection or listening port. In some cases well-known executables host multiple independent components, and in these cases the sequence of components involved in creating the connection or listening port is displayed. In this case the executable name is in [] at the bottom, on top is the component it called, and so forth until TCP/IP was reached. Note that this option can be timeconsuming and will fail unless you have sufficient permissions.

-e Displays Ethernet statistics. This may be combined with the -s option. -n Displays addresses and port numbers in numerical form.

-o Displays the owning process ID associated with each connection.

-p proto Shows connections for the protocol specified by proto; proto may be any of: TCP, UDP, TCPv6, or UDPv6. If used with the -s option to display per-protocol statistics, proto may be any of: IP, IPv6, ICMP, ICMPv6, TCP, TCPv6, UDP, or UDPv6.

-r Displays the routing table.

-s Displays per-protocol statistics. By default, statistics are shown for IP, IPv6, ICMP, ICMPv6, TCP, TCPv6, UDP, and UDPv6; the -p option may be used to specify a subset of the default.

-v When used in conjunction with -b, will display sequence of components involved in creating the connection or listening port for all executables. interval Redisplays selected statistics, pausing interval seconds between each display. Press CTRL+C to stop redisplaying statistics. If omitted, netstat will print the current configuration information once. Careful perusal of this information informs the reader that netstat not only documents active TCP and UDP connections and related port addresses but that it can also tie established TCP or UDP connections to the executable files, runtime components, and process IDs that opened or use them. Netstat can also provide counts of byte unicast and non-unicast packets, discards, errors, and unknown protocols. Netstat can also show connections for transport layer protocols for IPv4 and IPv6, display routing table contents, and can redisplay selected statistics at regular intervals. Netstat can be a helpful forensic tool when trying to determine what processes and programs are active on a computer and involved in networked communications. It can provide telltale signs of malware compromise under some circumstances and is a good tool to use to observe what kinds of communications are underway at any given time.

4. nslookup
nslookup is the name of a program that lets an Internet server administrator or any computer user enters a host name (for example, "whatis.com") and find out the corresponding IP address. It will also do reverse name lookup and find the host name for an

IP address you specify. For example, if you entered "whatis.com" (which is one of the TechTarget sites), you would receive as a response our IP address, which happens to be : 65.214.43.37 Or if you entered "65.214.43.37", it would return "sites.techtarget.com". nslookup sends a domain name query packet to a designated (or defaulted) domain name system (DNS) server. Depending on the system you are using, the default may be the local DNS name server at your service provider, some intermediate name server, or the root server system for the entire domain name system hierarchy.

COMMANDS :

ping 192.168.137.211

ping www.google.com

ping dd.com

Traceroute 192.168.137.211

Traceroute wwww.google.com

netstat -a

nslookup whatis.com

nslookup 206.19.49.165

OUTPUT :

Pinging www.google.com [2404:6800:4009:822::2004] with 32 bytes of data:
Reply from 2404:6800:4009:822::2004: time=86ms
Reply from 2404:6800:4009:822::2004: time=93ms
Reply from 2404:6800:4009:822::2004: time=110ms
Reply from 2404:6800:4009:822::2004: time=140ms

Ping statistics for 2404:6800:4009:822::2004:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 86ms, Maximum = 140ms, Average = 107ms

Conclusion:
Use of diagnostic Network Commands: ping, traceroute, netstat, nslookup were studied successfully.

PRACTICAL 9

Aim:
Using network protocol analyzer tool like ethereal (wireshark) or tcpdump to analyze network traffic.

Theory:
Wireshark is a network protocol analyzer, or an application that captures packets from a network connection, such as from your computer to your home office or the internet. Packet is the name given to a discrete unit of data in a typical Ethernet network.

Wireshark is the most often-used packet sniffer in the world. Like any other packet sniffer, Wireshark does three things:

1. Packet Capture: Wireshark listens to a network connection in real time and then grabs entire streams of traffic – quite possibly tens of thousands of packets at a time.
2. Filtering: Wireshark is capable of slicing and dicing all of this random live data using filters. By applying a filter, you can obtain just the information you need to see.
3. Visualization: Wireshark, like any good packet sniffer, allows you to dive right into the very middle of a network packet. It also allows you to visualize entire conversations and network streams.

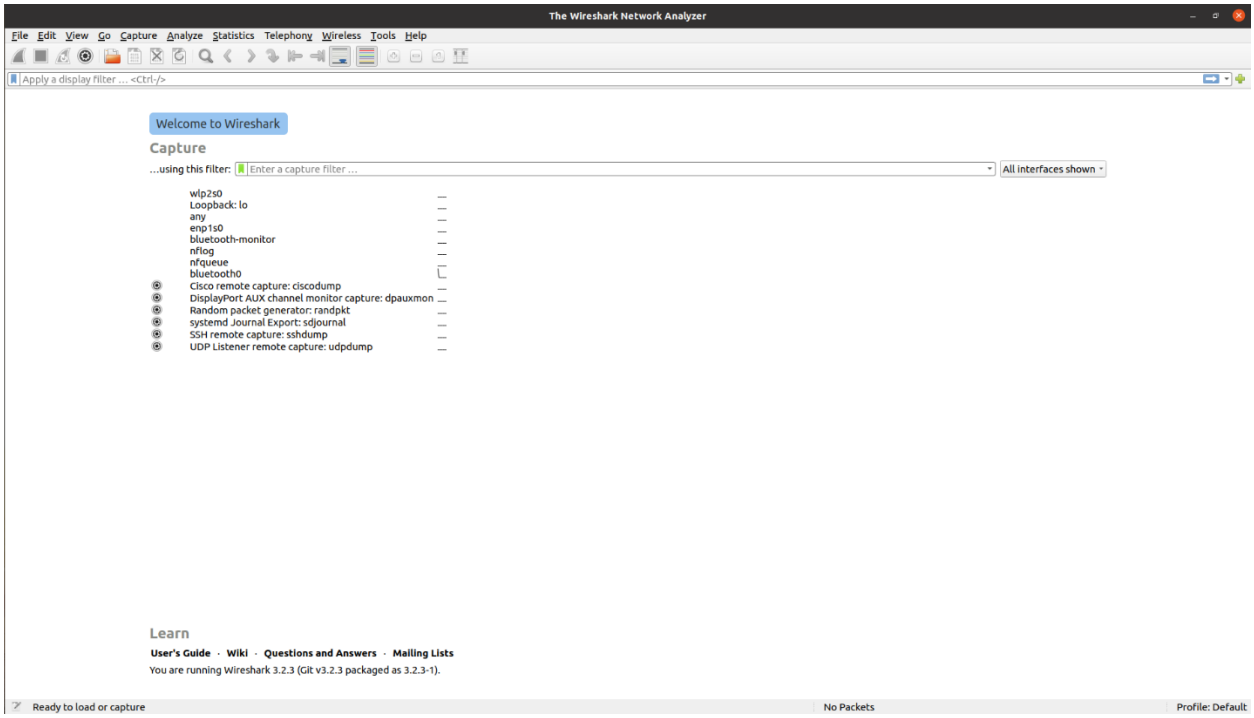
INSTALLATION OF WIRESHARK ON LINUX :

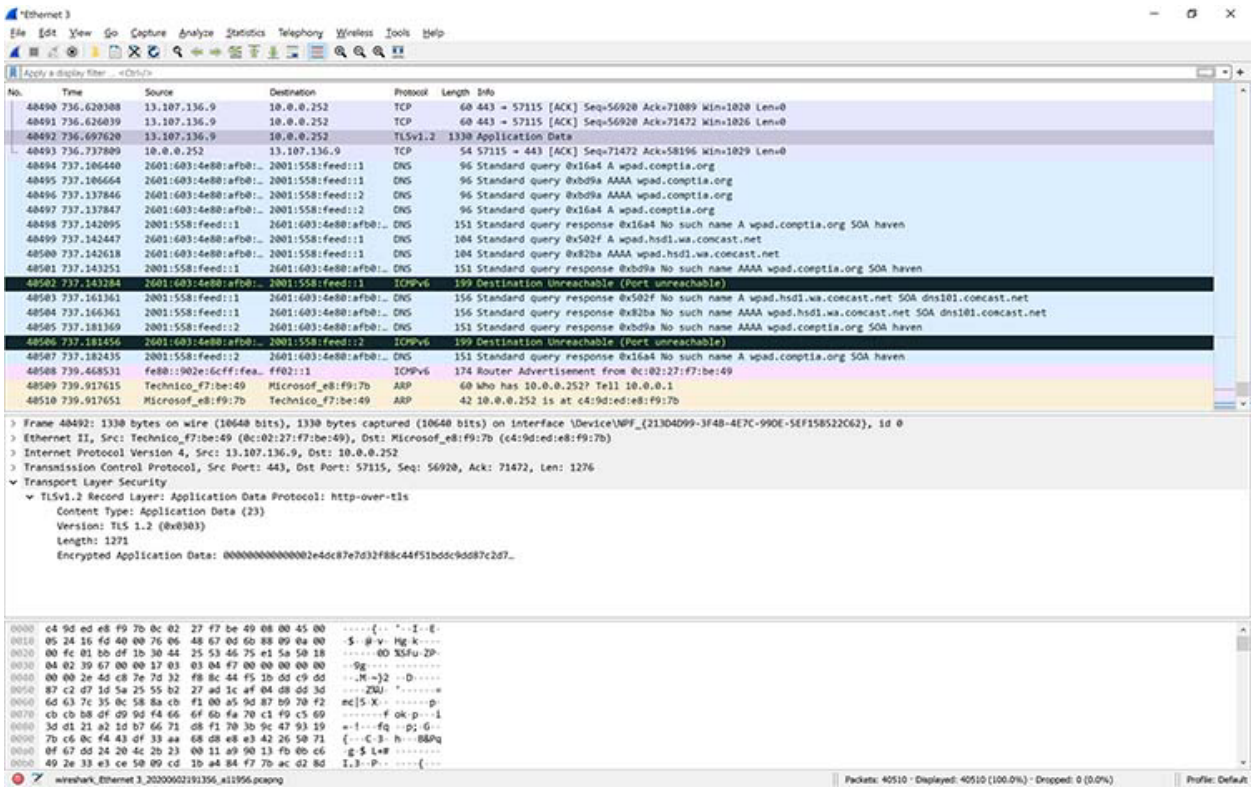
$ sudo apt-get install wireshark

(optional) :

$ sudo dpkg-reconfigure wireshark-common
$ sudo usermod -a -G wireshark $USER
$ newgrp wireshark

OUTPUT :





Conclusion:

Using network protocol analyzer tool wireshark to analyze network traffic studied successfully.

PRACTICAL 10

Aim:
To study IP Address manipulation: Extract Network ID and Host ID given netmask.

Theory:
TCP/IP defines five classes of IP addresses: class A, B, C, D, and E. Each class has a range of valid IP addresses. The value of the first octet determines the class. IP addresses from the first three classes (A, B and C) can be used for host addresses. The other two classes are used for other purposes – class D for multicast and class E for experimental purposes. The system of IP address classes was developed for the purpose of Internet IP addresses assignment. The classes created were based on the network size. For example, for the small number of networks with a very large number of hosts, the Class A was created. The Class C was created for numerous networks with small number of hosts.

Classes of IP addresses are:

Class	FIrst octet value	Subnet mask
A	0-127	8
B	128-191	16
C	192-223	24
D	224-239	-
E	240-255	-

For the IP addresses from Class A, the first 8 bits (the first decimal number) represent the network part, while the remaining 24 bits represent the host part. For Class B, the first 16 bits (the first two numbers) represent the network part, while the remaining 16 bits represent the host part. For Class C, the first 24 bits represent the network part, while the remaining 8 bits represent the host part.

CODE :

```
#include <stdio.h>

int main()
{
   int ip[4],sm[4];

   printf("\nEnter IP : ");
   scanf("%d.%d.%d.%d",&ip[0],&ip[1],&ip[2],&ip[3]);

   printf("\nEnter subnet mask : ");
   scanf("%d.%d.%d.%d",&sm[0],&sm[1],&sm[2],&sm[3]);

   printf("\nNetwork ID : %d.%d.%d.%d",(ip[0]&sm[0]),(ip[1]&sm[1]),(ip[2]&sm[2]),(ip[3]&sm[3]));

   printf("\nHost ID :
```

```
 %d.%d.%d.%d",(ip[0]&~sm[0]),(ip[1]&~sm[1]),(ip[2]&~sm[2]),(ip[3]&~sm[3]));

   if(ip[0]>=1 && ip[0]<127)
   {
      printf("\nClass A\n");
   }
   else if(ip[0]>=128 && ip[0]<192)
   {
      printf("\nClass B\n");
   }
   else if(ip[0]>=192 && ip[0]<224)
   {
      printf("\nClass C\n");
   }
   else if(ip[0]>=224 && ip[0]<240)
   {
      printf("\nClass D\n");
   }
   else if(ip[0]>=240 && ip[0]<255)
   {
      printf("\nClass E\n");
   }
   else if(ip[0]==127)
   {
      printf("\nReserved for TroubleShooting\n");
   }
   else
   {
      printf("\nInvalid\n");
   }
}
```

OUTPUT :

Enter IP : 192.168.100.11

Enter subnet mask : 255.255.255.0

Network ID : 192.168.100.0
Host ID : 0.0.0.11
Class C

Conclusion:
IP Address manipulation: Extract Network ID and Host ID given netmask was studied successfully.

PRACTICAL 11

Aim:
To study Implementation of IP Fragmentation and Reassembly.

Theory:
Fragmentation is done by the network layer when the maximum size of datagram is greater than maximum size of data that can be held in a frame i.e., its Maximum Transmission Unit (MTU). The network layer divides the datagram received from the transport layer into fragments so that data flow is not disrupted.

•Since there are 16 bits for total length in IP header so, the maximum size of IP datagram = 216 – 1 = 65, 535 bytes.
•It is done by the network layer at the destination side and is usually done at routers.
•Source side does not require fragmentation due to wise (good) segmentation by transport layer i.e. instead of doing segmentation at the transport layer and fragmentation at the network layer, the transport layer looks at datagram data limit and frame data limit and does segmentation in such a way that resulting data can easily fit in a frame without the need of fragmentation.
•Receiver identifies the frame with the identification (16 bits) field in the IP header. Each fragment of a frame has the same identification number.
•Receiver identifies the sequence of frames using the fragment offset(13 bits) field in the IP header.
•Overhead at the network layer is present due to the extra header introduced due to fragmentation.

Fields in IP header for fragmentation –
•Identification (16 bits) – use to identify fragments of the same frame.
•Fragment offset (13 bits) – use to identify the sequence of fragments in the frame. It generally indicates a number of data bytes preceding or ahead of the fragment. Maximum fragment offset possible = (65535 – 20) = 65515 {where 65535 is the maximum size of datagram and 20 is the minimum size of IP header}
So, we need ceil(log265515) = 16 bits for a fragment offset but the fragment offset field has only 13 bits. So, to represent efficiently we need to scale down the fragment offset field by 216/213 = 8 which acts as a scaling factor. Hence, all fragments except the last fragment should have data in multiples of 8 so that fragment offset N. ∈
•More fragments (MF = 1 bit) – tells if more fragments are ahead of this fragment i.e. if MF = 1, more fragments are ahead of this fragment and if MF = 0, it is the last fragment.
•Don’t fragment (DF = 1 bit) – if we don’t want the packet to be fragmented then DF is set i.e. DF = 1.

Reassembly of Fragments –
It takes place only at the destination and not at routers since packets take an independent path(datagram packet switching), so all may not meet at a router and hence a need of fragmentation may arise again. The fragments may arrive out of order also.

CODE :

```
#include <stdio.h>
#include <string.h>

struct fragment
{
   int id;
   int offset;
   int mf;
   char data[100];
};

void main()
{
   char str[100],sub[50];
   int flag=0,start=0,cnt=0;
   int mtu,netCap,noofFrag,i,id,offset,mf;
   struct fragment f[10];

   printf("\nEnter a string : ");
   gets(str);

   while(flag==0)
   {
      printf("\nEnter MTU : ");
      scanf("%d",&mtu);
      if((mtu-20) % 8 == 0)
      {
         flag=1;
      }
   }

   netCap = mtu - 20;
   noofFrag = strlen(str)/netCap;

   if(strlen(str)%netCap!=0)
   {
      noofFrag++;
   }

   for(i=0;i<noofFrag;i++)
   {
      f[i].id=10;
      f[i].offset=start/8;
      if(i<noofFrag-1)
      {
         f[i].mf=1;
         while(cnt<netCap)
         {
            sub[cnt]=str[start+cnt];
```

```
            cnt++;
         }
         sub[cnt]='\0';
         cnt=0;
         strcpy(f[i].data,sub);
         start = start + netCap;
      }

      else
      {
         f[i].mf=0;
         while(cnt<netCap)
         {
            if((start+cnt)==strlen(str))
            {
               break;
            }
            sub[cnt]=str[start+cnt];
            cnt++;
         }
         sub[cnt]='\0';
         strcpy(f[i].data,sub);
      }
   }

   for(i=0;i<noofFrag;i++)
   {
      printf("\n%d %d %d %s",f[i].id,f[i].offset,f[i].mf,f[i].data);
   }
}
```

OUTPUT :

Enter a string : 1522525252517171711131313131313131313

Enter MTU : 52

10 0 1 152252525251717171113131313131
10 4 0 31313

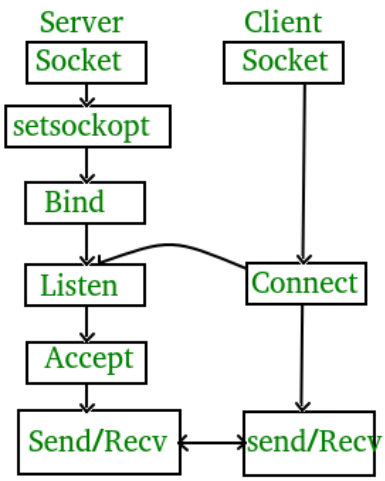
Conclusion:
Implementation of IP Fragmentation and Reassembly was studied successfully.

PRACTICAL 12

Aim:
To study simple TCP client and server application to send String from Client to server.

Theory:
If we are creating a connection between client and server using TCP then it has few functionality like, TCP is suited for applications that require high reliability, and transmission time is relatively less critical. It is used by other protocols like HTTP, HTTPs, FTP, SMTP, Telnet. TCP rearranges data packets in the order specified. There is absolute guarantee that the data transferred remains intact and arrives in the same order in which it was sent. TCP does Flow Control and requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control. It also does error checking and error recovery. Erroneous packets are retransmitted from the source to the destination.

The entire process can be broken down into following steps:



TCP Server –

1.using create(), Create TCP socket.
2.using bind(), Bind the socket to server address.
3.using listen(), put the server socket in a passive mode, where it waits for the client to approach the server to make a connection
4.using accept(), At this point, connection is established between client and server, and they are ready to transfer data.
5.Go back to Step 3.

TCP Client –

1.Create TCP socket.
2.connect newly created client socket to server.

CODE :

Client :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main()
{
   int CreateSocket = 0, n = 0, cnt = 0, clintConnt = 0, a, b;
   char dataReceived[1024];
   struct sockaddr_in ipOfServer;
   char buf1[50], str[50], oper;
   if ((CreateSocket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   {
      printf("Socket not created \n");
      return 1;
   }
   ipOfServer.sin_family = AF_INET;
   ipOfServer.sin_port = htons(2017);
   ipOfServer.sin_addr.s_addr = inet_addr("127.0.0.1");
   if (connect(CreateSocket, (struct sockaddr *)&ipOfServer, sizeof(ipOfServer)) < 0)
   {
      printf("Connection failed due to port and ip problems\n");
      return 1;
   }
   printf("\nEnter a name: ");
   scanf("%s", buf1);
   write(CreateSocket, buf1, strlen(buf1) + 1);
   n = read(CreateSocket, dataReceived, 512);
   printf("%s", dataReceived);
   close(CreateSocket);
   return 0;
}
```

Server :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
```

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main()
{
   char dataSending[1025];
   char dataReceived[1024];
   char str[50], operand[10];
   char op;
   int clintListn = 0, clintConnt = 0, sockfd = 0, cnt = 0, n = 0, len, a, b, output;
   struct sockaddr_in ipOfServer;
   clintListn = socket(AF_INET, SOCK_STREAM, 0);
   ipOfServer.sin_family = AF_INET;
   ipOfServer.sin_addr.s_addr = htonl(INADDR_ANY);
   ipOfServer.sin_port = htons(2017);
   bind(clintListn, (struct sockaddr *)&ipOfServer, sizeof(ipOfServer));
   listen(clintListn, 20);
   while (clintConnt == 0)
   {
      printf("\n\nHi, I am running server. Some Client hit me\n");
      clintConnt = accept(clintListn, (struct sockaddr *)NULL, NULL);
   }
   n = read(clintConnt, dataReceived, 512);
   printf("\n%s", dataReceived);
   close(clintConnt);
   close(clintListn);
   return 0;
}
```

OUTPUT :

Client :

Enter a name : Ak

Server :

Hi, I am running server. Some Client hit me
Ak

Conclusion:
The simple TCP client and server application to send String from Client to server was studied successfully.

PRACTICAL 13

Aim:
To study simple TCP client and server application to perform Arithmetic operations.

Theory:
Same as practical 12

CODE :

Client :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main()
{
   int CreateSocket = 0, n = 0, cnt = 0, a, b;
   char dataReceived[1024];
   struct sockaddr_in ipOfServer;
   char buf1[50], str[50], oper;
   if ((CreateSocket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   {
      printf("Socket not created \n");
      return 1;
   }
   ipOfServer.sin_family = AF_INET;
   ipOfServer.sin_port = htons(2017);
   ipOfServer.sin_addr.s_addr = inet_addr("127.0.0.1");
   if (connect(CreateSocket, (struct sockaddr *)&ipOfServer, sizeof(ipOfServer)) < 0)
   {
      printf("Connection failed due to port and ip problems\n");
      return 1;
   }
   printf("\nEnter the operand a");
   scanf("%d", &a);
   printf("\nEnter the operand b");
   scanf("%d", &b);
   printf("\nEnter the operator");
   scanf(" %c", &oper);
   sprintf(buf1, "%d %c %d", a, oper, b);
   write(CreateSocket, buf1, strlen(buf1) + 1);
   n = read(CreateSocket, dataReceived, 512);
   printf("%s", dataReceived);
   close(CreateSocket);
   return 0;
}
```

Server :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main()
{
   char dataReceived[1024];
   char op;
   int clintListn = 0, clintConnt = 0, sockfd = 0, cnt = 0, n = 0, len, a, b, output;
   struct sockaddr_in ipOfServer;
   clintListn = socket(AF_INET, SOCK_STREAM, 0);
   ipOfServer.sin_family = AF_INET;
   ipOfServer.sin_addr.s_addr = htonl(INADDR_ANY);
   ipOfServer.sin_port = htons(2017);
   bind(clintListn, (struct sockaddr *)&ipOfServer, sizeof(ipOfServer));
   listen(clintListn, 20);
   while (clintConnt == 0)
   {
      printf("\n\nHi, I am running server. Some Client hit me\n");
      clintConnt = accept(clintListn, (struct sockaddr *)NULL, NULL);
   }
   n = read(clintConnt, dataReceived, 512);
   sscanf(dataReceived, "%d %c %d", &a, &op, &b);
   printf("%d %c %d", a, op, b);
   switch (op)
   {
   case '+':
      output = a + b;
      break;
   case '-':
      output = a - b;
      break;
   case '*':
      output = a * b;
      break;
   case '/':
      if (b == 0)
      {
         printf("\n Invalid input!");
         break;
      }
      output = a / b;
      break;
   default:
      printf("\nInvalid Input!");
   }
   sprintf(dataReceived, "%d", output);
   write(clintConnt, dataReceived, strlen(dataReceived) + 1);
   close(clintConnt);
```

```
    close(clintListn);
    return 0;
}
```

OUTPUT :

Client :

```
Enter the operand a
10

Enter the operand b
11

Enter the operator
+

21
```

Server :

```
Hi, I am running server. Some Client hit me
10 + 11
```

Conclusion:
The simple TCP client and server application to perform Arithmetic operations was studied successfully.

PRACTICAL 14

Aim:
To study simple TCP client and server application to reverse a string.

Theory:
Same as practical 12

CODE :

Client :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main()
{
   int CreateSocket = 0, n = 0, cnt = 0, clintConnt = 0, a, b;
   char dataReceived[1024];
   struct sockaddr_in ipOfServer;
   char buf1[50], str[50], oper;
   if ((CreateSocket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   {
      printf("Socket not created \n");
      return 1;
   }
   ipOfServer.sin_family = AF_INET;
   ipOfServer.sin_port = htons(2017);
   ipOfServer.sin_addr.s_addr = inet_addr("127.0.0.1");
   if (connect(CreateSocket, (struct sockaddr *)&ipOfServer, sizeof(ipOfServer)) < 0)
   {
      printf("Connection failed due to port and ip problems\n");
      return 1;
   }
   printf("\nEnter a name: ");
   scanf("%s", &buf1);
   write(CreateSocket, buf1, strlen(buf1) + 1);
   close(CreateSocket);
   return 0;
}
```

Server :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main()
{
   char dataSending[1025];
   char dataReceived[1024];
   char buffer[50];
   char str[50], operand[10];
   char op;
   int clintListn = 0, clintConnt = 0, sockfd = 0, cnt = 0, n = 0, len, a, b, output, i, j = 0;
   struct sockaddr_in ipOfServer;
   clintListn = socket(AF_INET, SOCK_STREAM, 0);
   ipOfServer.sin_family = AF_INET;
   ipOfServer.sin_addr.s_addr = htonl(INADDR_ANY);
   ipOfServer.sin_port = htons(2017);
   bind(clintListn, (struct sockaddr *)&ipOfServer, sizeof(ipOfServer));
   listen(clintListn, 20);
   while (clintConnt == 0)
   {
      printf("\n\nHi, I am running server. Some Client hit me\n");
      clintConnt = accept(clintListn, (struct sockaddr *)NULL, NULL);
   }
   n = read(clintConnt, dataReceived, 1024);
   printf("\nThe received string is: %s", dataReceived);
   len = strlen(dataReceived);
   printf("\nThe length of received string is: %d", len);
   for (i = 0; i < len; i++)
   {
      buffer[i] = dataReceived[len - i - 1];
   }
   buffer[i] = '\0';
   printf("\nThe reversed string is: %s", buffer);
   close(clintConnt);
   close(clintListn);
   return 0;
}
```

OUTPUT :

Client :

```
Enter a name : THE_HUMAN_SPIDER
```

Server :

Hi, I am running server. Some Client hit me

The received string is : THE_HUMAN_SPIDER
The length of received string is : 16
The reversed string is : REDIPS_NAHUM_EHT

Conclusion:
The simple TCP client and server application to to reverse a string was studied successfully.