MALLA REDDY ENGINEERING COLLEGE
(AUTONOMOUS)

Maisammaguda, Dhulapally , Secunderabad-500 014
Department Of Computer Science and Engineering

COMPUTER NETWORKS (MR15)

List Of Experiments:
Part-A

1.Implement the data link layer framing methods:
a) Character count
b) Character stuffing
c) Bit stuffing and destuffing

2) Implement on a data set of characters the three CRC polynomials:CRC-12,CRC-16,and CRC-32.
3) Implement parity check using the followinf techniques:
a)single dimensional data
b) Multi dimensional data

4) Implement Even and Odd parity

5) Implementation of data link layers
a) Unrestricted simplex protocol
b) stop and wait protocol
c) Noisy channel

6. Implementation of sliding window protocols
a) one bit sliding window protocol
b) Go back N sliding window protocol
c) Selective repeat sliding window protocol

7)Implementation of Routing protocols
a) Dijkstras algorithm
b) Distance Vector routing protocol
c) link state routing protocols

8) Implement the congestion algorithms:
a) Token bucket algorithms
b) Leaky bucket algorithm

1)
A) character count
AIM: To develop a c program to generate character count
Procedure :
Character-count integrity is a telecommunications term for the ability of a certain link to preserve the number of characters in a message (per unit time, in the case of a user-to-user connection). Character-count integrity is not the same as character integrity, which requires that the characters delivered be, in fact, exactly the same as they were originated.
Code :
```
#include <stdio.h>

/* count characters and input using while */
main()
{
   long nc;

   nc = 0;
   while (getchar() != EOF)
      ++nc;
   printf("%ld\n", nc);
}
```

B)CHARACTER STUFFING&DESTUFFING

Aim:

To implement the data link layer framing method character stuffing.

Problem Description:

The character stuffing method gets around the problem of re synchronization after an error by having each frame start and end with special bytes.

Character Stuffing / Byte Stuffing:

Character stuffing or byte stuffing is which an escape byte (ESC) is stuffed character stream before a flag byte in the data.

Character destuffing / Byte destuffing:

Character destuffing (or) byte destuffing is the process in which the data link layer on the receiving end removes escape byte (ESC) before the data are given to network layer.

Explanation:

To provide service to network layer, data link layer must use the services provided to it by the physical layer. The bit stream is not guaranteed to be error free.The number of bits received may be less than,equal to,or more than the number of bits transmitted,and they may have different values. It is up to the data link layer to detect and, If necessary, correct errors.

The usal approach is for the data link layer to break the bit stream up into discrete frames and compute the checksum for each frame. When a frame arrives at the destination ,the checksum is recomputed. If the newly computed checksum is different from one contain in the frame, the data link layer knows than an error has occurred and takes steps to deal with it.

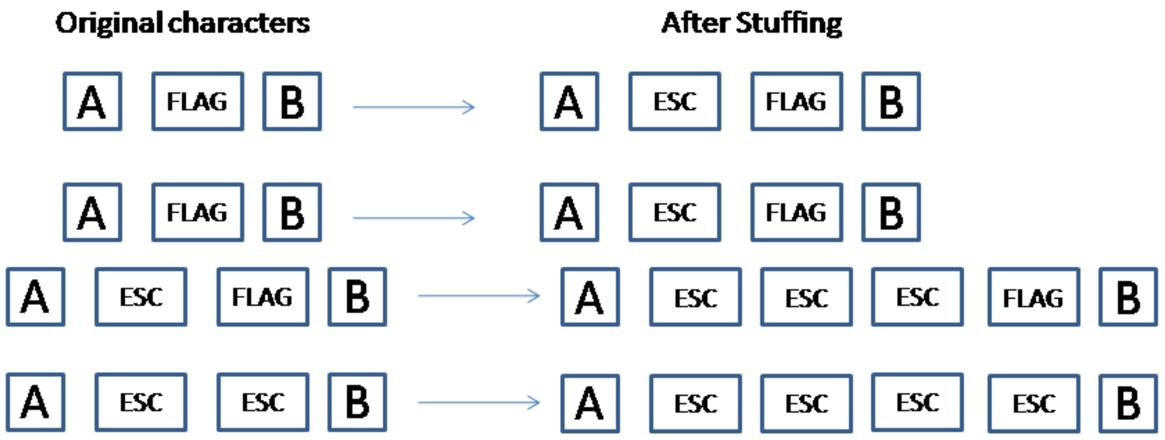
In this approach, the “flag byte” is appended at the starting and ending delimiter.

Frame Format:

FLAG	Header	Pay load field	Trailer	FLAG

Frame delimited by flag bytes

Example:



```
CharacterStuffing:
#include<stdio.h>
int pl=0;
FILE *fp,*fp1;
main()
{
void stuff();
stuff();
return;
}
void stuff()
{
int H=0,count=0,i=0,c=0,p=0,t=0;
char f[20];
char ch,prev,a[500],se[6]={'s','t','x','d','l','e'},de[6]={'e','t','x','d','l','e'};
printf("enter payload");
scanf("%d",&pl);
printf("enter the file to be stuffed");
scnaf("%c",&f);
fp=fopen("c source.txt","r");
fp1=fopen(f1,"w");
L1:while(((fscanf(fp,"%c",&ch1!=EOF)&&(w!=pl))
{
if(ch=='d')
{
a[H]=ch;
count=count+1;
}
else if(ch=='l')
{
a[H]=ch;
count=count+1;
}
else if(ch=='e')
{
a[H]=ch;
count=count+1;
if(count==3)
{
a[H+1]='d';
a[H+2]='l';
a[H+3]='e';
count=0;
if(H!=pl-3)
{
H=H+3;
count=0;
p=0;
}
else
{
prev=ch;
p=1;
}
}
}
else
{
a[H]=ch;
count=0;
}
H++;
count=0;
if(feof(fp))
c=1;
else
fseek(fp,-1,1)
L2:for(i=0;i<6;i++)
fprintf(fp,"%c",se[i]);
for(i=0;i<H;i++)
fprintf(fp1,"%c",a[i]);
for(i=0;i<6;i++)
fprintf(fp,"%c",de[i]);
fprintf(fp,"/n");
```

```
H=0;
if(p==1)
{
goto L1;
}
if(c==0)
{
p=0;
goto L1;
}
if(p==1)
{
p=0;
goto L2;
}
fcloseall();
}
```

CharacterDestuffing:

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
FILE *fp,*fp1;
main()
{
void dstuff();
dstuff();
return;
}
void dstuff()
{
int i=0,count=0,j,n=0;
char ch,a[50];
size-t read;
fp=fopen("cstuff.txt","r");
fp1=fopen("out.txt","w");
while((fscanf(fp,"%c",&ch)!=EOF))
{
n++;
if(ch=='\n')
{
n=n-1;
n=n-12;
for(j=6;j<(6+n);j++)
{
fprintf(fp1,"%C",a[j]);
}
i=0;
n=0;
}
else
{
a[i]=ch;
i++;
}
}
fcloseall();
}
```

c)
Problem: Implementing the data link layer framing methods such as the character stuffing and Bit stuffing.
Aim: To implement the data link layer framing method bit stuffing.
Problem Description: A new technique allows data frames to contain arbitrary number of bits and allows character codes with arbitrary number of bits per character.
Bit Stuffing: Bit stuffing is which an zero bit is stuffed after five consecutive ones in the input bit stream.
Bit destuffing: Bit destuffing is the process of removing the stuffed bit in the output stream.

Explanation:

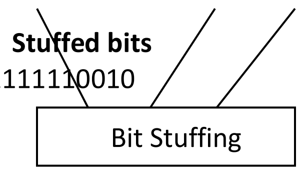
To provide service to network layer, the data link layer, must use the services provided to it by the physical layer. The bit stream is not guaranteed to be error free. The number of bits received may be less than, equal to, or more than data link layer to detect and, if necessary, correct errors.

The usual approach is for the data link layer to break the bit stream up into discrete frames and compute the checksum for each frame. When a frame arrives at the destination, the checksum is re computed. If the newly computed checksum is different from one contained in the frame, the data link layer knows than an error has occurred and takes steps to deal it.

Each frame begins and ends with a special bit pattern, 01111110.When ever the sender's data link layer encounter five consecutive 1's in the data, it automatically stuffs a 0 bit in to outgoing bit stream. This bit stuffing is analogous to byte stuffing. When ever the receiver sees five consecutive incoming ones, followed by a 0 bit, it automatically dyestuffs the 0 bit.

Example:

01101111111111111110010
0101111011111011111010010



01101111111111111110010

The Original Data
The Data as they appear on the line
The data as they are stored in the receiver's memory after destuffing.

Conclusion:

With the bit stuffing, the boundary between two frames can be unambiguously recognized by the bit pattern. Thus the receiver loses track of where it is, all it has to do is scan the input for flag sequences, since they can only occur at frame boundaries and never within data.

```
#include<stdio.h>
char a[100],se[8]={'0','1','1','1','1','1','1','0'};
void stuff();
void tobinary();
FILE *fp1,*fp2;
main()
{
tobinary();
stuff();
return;
}
void tobinary()
{
int v[10],i=0,a;
char ch;
fp1=fopen("soruce.txt","r");
fp2=fopen("binary.txt","w");
while(fscanf(fp1,"c",&ch)!=EOF)
{
FOR(I=0;i<7;i++)
v[i]=0;
a=ch;
i=0;
while(a!=0)
{
v[i]=a%2;
a=a%2;
i++;
}
i=6;
while(i>0)
{
fprintf(fp2,"%d",v[i]);
i--;
}
}
fcloseall();
}
void stuff()
{
int pl,x=0,count=0,i=0,p=0,c=0;
char ch,prev;
printf("enter payload");
```

```
scanf("%d",&pl);
fp1=fopen("binary.txt","r");
fp2=fopen("deatination.txt","W");
l1:while((fscanf(fp1,"%c",&ch)!=EOF)&&(x!=pl))
{
if(ch=='0')
{
a[x]=ch;
count=0;
}
elseif(ch=='1')
{
if(count=='5')
{
a[x]=0;
if(x!=pl-1)
{
x=x+1;
a[x]=ch;
count=0;
p=0;
}
else
{
prev=ch;
p=1;
}
count=count+1;
}
else
{
a[x]=ch;
count=count+1;
}
}
x++;
}
count=0;
if(feof(fp1))
c=1;
else
fseek(fp1,-1,1);
L2:for(i=0;i<8;i++)
fprintf(fp2,"%c",se[i]);
for(i=0;i<x;i++)
fprintf(fp2,"%c",a[i]);
for(i=0;i<8;i++)
```

```
fprintf(fp2,"%c",se[i]);
fprintf(fp2,"\n");
x=0;
if(p==1)
{
if(prev==0)
{
a[x]=prev;
x++;
count=0;
}
elseif(prev==1)
{
a[x]=prev;
count=count+1;
x++;
}
}
if(c==0)
{
p=0;
goto L1;
}
if(p==1)
{
p=0;
goto L2;
}
fcloseall();
}
```

BitDestuffing:

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
FILE *fp,*fp1;
main()
{
void dstuff();
void bintochar();
dstuff();
bintochar();
return;
}
void dstuff()
{
```

```
int i=0,count=0,pl,n=30,j;
char ch,a[50];
ssize_+ read;
printf("enter pay load");
scanf("%d",&pl);
fp=fopen("destination.txt","r");
fp=fopen("output.txt","w');
while(fgets(a,n,fp)!='\0')
{
a[strlen[a]-9]='\0';
for(i=0;a[i]!='\0';i++)
a[i]=a[i+s]
for(j=0;j<strlen(a);j++)
{
if(a[j]=='0')
{if(coumt==5)
count=0;
]
else
{
fprintf(fp1,"%',a[j]);
count=0;
}
]
if(a[j]=='1')
{
count++;
fprintf(fp1,"%c",a[j]);
}
}
}
fclose();
}
void bintochar()
{
char ch;
int a[7],p=0,j=0,i=0,c=0,sum=0;
fp=fopen("output1.txt","r");
fp=fopen("output2.txt',"w');
l1;while((fscanf(fp,"%c",&ch)!=EOF)&&(i<7))
{
if(ch=='1')
a[i]=1;
elseif(ch=='0')
a[i]=0;
i++;
}
```

```
p=6;
while(j<7)
{
if(a[j]==1)
sum=sum+pow(2,p);
j++;
p--;
}
if(feof(fp))
c=1;
if(c==0)
{
fprintf(fp,"%c",sum);
fseek(fp,-1,1);
sum=0;
j=0;
i=0;
goto l1;
}
fcloseall();
}
```

2.CRC

Problem Description:

Implement on data set of characters the there crc polynomials – crc12, crc16, crcccitt.

Aim:

To implement on data set of characters the three crc polynomials - crc12, crc16, crcccitt.

Program Description:

Error correcting codes are widely used on wireless links, which are notoriously noisy and error prone when compared to copper wire or optical fiber. The polynomial code, also know as a crc (Cycle Redundancy Check). Polynomial codes are based on treating bit strings as representations of polynomial with coefficient of 0 and 1 only.

Explanation:

A k-bit frame is regarded as the coefficient list for a polynomial with k-terms, ranging from x^{k-1} to x^0. Such a polynomial is said to be of degree k-1. The higher order bit is the coefficient of x^{k-1} .The next bit is coefficient of x^{k-2} and so on.
Polynomial arithmetic is done modulo 2,according to the rules of algebraic field theory.

Example:

```
   1 0 0 1 1 0 1 1       0 0 1 1 0 0 1 1       1 1 1 1 0 0 0 0       0 1 0 1 0 1 0 1
 + 1 1 0 0 1 0 1 0   +   1 1 0 0 1 1 0 1   -   1 0 1 0 0 1 1 0   -   1 0 1 0 1 1 1 1
_______________   ____________  ____________  _____________
 0 1 0 1 0 0 0 1         1 1 1 1 1 1 10        0 1 0 1 0 1 1 0       1 1 1 1 1 0 1 0
________________  _________________________  _____________
```

When polynomial is employed, the sender and receiver must agree up on a generated polynomial, g(x),in advance. Both high and low order bits of the generator must one. To compute the check sum for some frame with m bits,corresponding to the polynomial m(x),the frame must be longer than the generated polynomial.

PARITY TECHNIQUES

A parity check is the process that ensures accurate data transmission between nodes during communication. A parity bit is appended to the original data bits to create an even or odd bit number; the number of bits with value one. The source then transmits this data via a link, and bits are checked and verified at the destination. Data is considered accurate if the number of bits (even or odd) matches the number transmitted from the source.

Parity Check

Parity checking, which was created to eliminate data communication errors, is a simple method of network data verification and has an easy and understandable working mechanism.

As an example, if the original data is 1010001, there are three 1s. When even parity checking is used, a parity bit with value 1 is added to the data’s left side to make the number of 1s even; transmitted data becomes 11010001. However, if odd parity checking is used, then parity bit value is zero; 01010001.

If the original data contains an even number of 1s (1101001), then parity bit of value 1 is added to the data’s left side to make the number of 1s odd, if odd parity checking is used and data transmitted becomes 11101001. In case data is transmitted incorrectly, the parity bit value becomes incorrect; thus, indicating error has occurred during transmission.

Source code:

```
#include<stdio.h>
#include<string.h>
void crc();
char s[50],r[50]="";
int i,j,k,n;
main()
{
char ch;
do
{
int a=12,b=16,c=22;
printf("\t\t\t\t\n*****CRC*****\N");
printf("1.CRC12 \n 2.CRC16 \n 3.CRC CCITT\n Enter your choice:");
scnaf("%d",&n);
switch(n)
{
case 1:CRC(a);
    break;
case 2:CRC(b);
    break;
case 3:CRC(c);
    break;
default:printf("Enter correct choice \n");
break;
}
printf("\n do you want to continue:");
fflush(stdin);
scanf("%c",&ch);
}
while(ch=='y'||ch=='y');
}
void crc(int x)
{
char g[20],c;
FILE *fp,*fp1;
char fname[25],fname1[25];
printf("\n Enter file:");
scanf("%s",fname);
fp=fopen(fname,"w");
if(x==12)
strcpy(g,"1100000001111");
if(x==16)
```

```
strcpy(g,"11000000000000101");
if(x==12)
printf("\t\t \n YOU HAVE ENTERED CRC 12");
if(x==16)
printf("\t\t \n YOU HAVE ENTERED CRC 16");
if(x==22)
printf("\t\t \n YOU HAVE ENTERED CRC CCITT");
puts("\n Enter the message");
scanf("%s",s);
printf("\n entered string is: %s",s);
for(i=0;i<strlen(g)-1;i++)
strcat(s,"0");
printf("\n string after appending zero s:%S \n",s);
for(i=0;s[i]!='\0';)
{
p=strlen(r);
while((strlen(g)!=strlen(r)&&(s[i]!='\0'))
{
r[j++}==s[i++];
r[j]='\0';
}
for(j=0;strlen[g]==strlen9r00&&g[j]!='\0';j++)
{
if(g[j]==r[j])
r[j]='0';
else
r[j]='1';
}
while(r[0]=='0'&&r[0]!='\0')
{
for(j=0;r[j]!='\0';j++)
r[j]=r[j+1];
}
}
s[strlen(s)-strlen(r)]='\0';
strcat(s,r);
printf("\n string converted is :%s and stored at %s\n',s,frame);
fprintf(fp,"%s",s);
fcloseall();
printf("\n do you want to check?:");
scanf("%c",&c);
if(c=='y'||c=='y')
{
printf("\nb enter file:");
scanf("%s',frame);
fp1=fopen(frame,"r");
fgets(s,100,fp1);
```

```
strcpy(r,"\0");
for(i=0;s[i]!='0';i++)
{
j=strlen(r);
while((strlen(g)!=strlen(r)&&(s[i]!='\o'))
{
r[j++]=s[i++];
r[j]='\0';
}
for(j=0;(strlen(g)==strlen(r)&&g[i]!='\0';j++)
{

if(g[j]==r[j])
r[j]='\0';
else
r[j]='1';
}
while(r[0]='0'&& r[0]!='\0')
{
for(j=0;r[j]!='\0';i++)
r[j]=r[j+1];
}
}
s[strlen(s)-strlen(g)+1]='\0';
strcat(s,r);
fcloseall();
printf("\n transmitted string is: %s \n",s);
if(strlen(r)==0)
{
printf("\n success");
}
else
printf("\n data corrupted");
}
else
exit(0);
}
```

3) Implement parity check techniques

a) Single dimensional

```
#include<stdio.h>
#include<string.h>
void main()
{   int i;
   char name[200];
   int one=0,zero=0;
   int count=0;
   int no;
```

```
printf("Enter The Name:-");
gets(name);
printf("Enter The Parity:-");
scanf("%d",&no);
for(i=0;name[i]!='\0';i++)
  {
    if(name[i]=='1')
      {
        one++;
      }
    else
      {
        zero++;
      }
    count++;
  }
printf("\nZero Are:-%d",zero);
printf("\nOne Are:-%d",one);
if(no==0)
  {
    if(one%2==0)
    {
      printf("\nEven Parity");
      printf("\n0");
      puts(name);
    }
    else
    {
      printf("\nEven Parity");
      printf("\n1");
      puts(name);
    }
  }
else

  {
    if(one%2==0)
    {
      printf("\nOdd Parity");
      printf("\n1");
      puts(name);
    }
    else
    {
      printf("\nOdd Parity");
      printf("\n0");
      puts(name);
```

```
        }
      }
}
```

3B) AIM: C code to implement multi dimensional parity check

```
#include<iostream>
#include<stdlib.h>
using namespace std;
#define maxlength 10
#define maxmessages 10
void initialize(int arr[][10],int m,int n)
{
for(int i =0;i<m;i++)
for(int j=0;j<n;j++)
{
arr[i][j] = rand()%2;
}
}
void print(int arr[][10],int m,int n)
{
for(int i =0;i<m;i++)
{  for(int j=0;j<n;j++)
{
cout<<arr[i][j]<<" ";
}
cout<<endl;
}
}
void addparbit(int arr[][10],int m,int n)  // Even Parity
{
for(int i=0;i<m;i++)
{
int count = 0;
for(int j=0;j<n;j++)
{
if(arr[i][j] == 1)
count++;
}
if(count%2 == 0)
arr[i][n] = 0;
else
arr[i][n] = 1;
}
}
void induceerror(int arr[][10],int m,int n)
{
int k1,k2;
k1= rand()%m;
k2 = rand()%n;
if(arr[k1][k2]==0)
arr[k1][k2]=1;
else
arr[k1][k2]=0;
cout<<“Inducing error at line : “<<k1<<endl;
}
void checkerror(int arr[][10],int m,int n)
{
for(int i=0;i<m;i++)
{
int count = 0;
for(int j=0;j<n;j++)
{
if(arr[i][j] == 1)
count++;
}
if(count%2 == 0 && arr[i][n] != 0)
{
cout<<“Error here at line : ” <<i;
}
else if(count%2 == 1 && arr[i][n] != 1)
{
cout<<“Error here at line : ” <<i;
}

}
}

int main()
{   int m,n,arr[maxmessages][maxlength];
cout<<“Enter total number of messages”;
cin>>m;
cout<<“Enter length of each message”;
cin>>n;
initialize(arr,m,n);
print(arr,m,n);
addparbit(arr,m,n);
print(arr,m,n+1);
induceerror(arr,m,n);
print(arr,m,n+1);
checkerror(arr,m,n);
return 0;
}
```

4)
AIM: C code to implement even odd parity
Procedure :

4. EVEN AND ODD PARITY

Even parity refers to a parity checking mode in asynchronous communication systems in which an extra bit, called a parity bit, is set to one if there is an even number of one bits in a one-byte data item. If the number of one bits adds up to an odd number, the parity bit is set to zero.

Even parity checking may also be used in testing memory storage devices.
In asynchronous communication systems, odd parity refers to parity checking modes, where each set of transmitted bits has an odd number of bits. If the total number of ones in the data plus the parity bit is an odd number of ones, it is called odd parity. If the data already has an odd number of ones, the value of the added parity bit is 0, otherwise it is 1.

Parity bits are the simplest form of error detection. Odd parity checking is used in testing memory storage devices. The sender and receiver should agree to the use odd parity checking. Without this, successful communication is not possible. If an odd number of bits are switched during transmission, parity checks can detect that the data is corrupted. However, the method will fail to detect errors introduced when an even number of bits in the same data unit is altered, as the parity will still remain odd despite data.
Parity bits are added to transmitted messages to ensure that the number of bits with a value of one in a set of bits add up to even or odd numbers. Even and odd parities are the two variants of parity checking modes.
Odd parity can be more clearly explained through an example. Consider the transmitted message 1010001, which has three ones in it. This is turned into odd parity by adding a zero, making the sequence 0 1010001. Thus, the total number of ones remain at three, an odd number. If the transmitted message has the form 1101001, which has four ones in it, this can be turned into odd parity by adding a one, making the sequence 1 1101001.
Code :

```
# include <stdio.h>
# define  bool int

/* Function to get parity of number n. It returns 1
   if n has odd parity, and returns 0 if n has even
   parity */
bool getParity(unsigned int n)
{
    bool parity = 0;
    while (n)
    {
        parity = !parity;
        n      = n & (n - 1);
    }
    return parity;
}

/* Driver program to test getParity() */
int main()
{
    unsigned int n = 7;
    printf("Parity of no %d = %s",  n,
             (getParity(n)? "odd": "even"));

    getchar();
    return 0;
}
```

5) Implement data link protocols:
a) AIM: C code to implement unrestricted simplex protocol
procedure :
In order to appreciate the step by step development of efficient and complex protocols such as SDLC, HDLC etc., we will begin with a simple but unrealistic protocol. In this protocol:
Data are transmitted in one direction only
The transmitting (Tx) and receiving (Rx) hosts are always ready
Processing time can be ignored
Infinite buffer space is available
No errors occur; i.e. no damaged frames and no lost frames (perfect channel)

[HEADER.H]

```
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
typedef struct
{
 int seqno;
 int ackno;
 char data[50];
}frame;
void from_network_layer(char buffer[])
{
 printf("Enter Data : ");
 scanf("%s",buffer);
}
void to_physical_layer(int pid1,frame *f)
{
 write(pid1,f,sizeof(frame));
}
void from_physical_layer(int pid1,frame *f)
{
 read(pid1,f,sizeof(frame));
}
void to_network_layer(char buffer[])
{
 printf("\n%s",buffer);
}
```

[SENDER SIDE]

```
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
#include "header.h"
void main()
{
 int pid1,i,no;
 char buffer[50];
 frame f;
 system(">pipe1");
 pid1=open("pipe1",O_WRONLY);
 printf("Enter NUMBER OF DATA : ");
 scanf("%d",&no);
 write(pid1,&no,sizeof(no));
 for(i=0;i<no;i++)
 {
  from_network_layer(buffer);
  strcpy(f.data,buffer);
  to_physical_layer(pid1,&f);
 }
 close(pid1);
}
```

[RECEIVER SIDE]

```
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
#include "header.h"
void main()
{
 int pid1,no,i;
 char buffer[50];
 frame f;
 pid1=open("pipe1",O_RDONLY);
 read(pid1,&no,sizeof(no));
 printf("DATA RECEIVED : %d",no);
 printf("\nDATA");
 for(i=0;i<no;i++)
 {
  from_physical_layer(pid1,&f);
  strcpy(buffer,f.data);
  to_network_layer(buffer);
 }
 close(pid1);
 unlink("pipe1");
}
```

B)

AIM: C code to generate stop and wait protocol

Procedure :

Stop-and-wait ARQ, also referred to as alternating bit protocol, is a method in telecommunications to send information between two connected devices. It ensures that information is not lost due to dropped packets and that packets are received in the correct order. It is the simplest automatic repeat-request (ARQ) mechanism. A stop-and-wait ARQ sender sends one frame at a time; it is a special case of the general sliding window protocol with transmit and receive window sizes equal to one and greater than one respectively. After sending each frame, the sender doesn't send any further frames until it receives an acknowledgement (ACK) signal. After receiving a valid frame, the receiver sends an ACK. If the ACK does not reach the sender before a certain time, known as the timeout, the sender sends the same frame again. The timeout countdown is reset after each frame transmission. The above behavior is a basic example of Stop-and-Wait. However, real-life implementations vary to address certain issues of design.

Typically the transmitter adds a redundancy check number to the end of each frame. The receiver uses the redundancy check number to check for possible damage. If the receiver sees that the frame is good, it sends an ACK. If the receiver sees that the frame is damaged, the receiver discards it and does not send an ACK—pretending that the frame was completely lost, not merely damaged.

One problem is when the ACK sent by the receiver is damaged or lost. In this case, the sender doesn't receive the ACK, times out, and sends the frame again. Now the receiver has two copies of the same frame, and doesn't know if the second one is a duplicate frame or the next frame of the sequence carrying identical data.

Another problem is when the transmission medium has such a long latency that the sender's timeout runs out before the frame reaches the receiver. In this case the sender resends the same packet. Eventually the receiver gets two copies of the same frame, and sends an ACK for each one. The sender, waiting for a single ACK, receives two ACKs, which may cause problems if it assumes that the second ACK is for the next frame in the sequence.

To avoid these problems, the most common solution is to define a 1 bit sequence number in the header of the frame. This sequence number alternates (from 0 to 1) in subsequent frames. When the receiver sends an ACK, it includes the sequence number of the next packet it expects. This way, the receiver can detect duplicated frames by checking if the frame sequence numbers alternate. If two subsequent frames have the same sequence number, they are duplicates, and the second frame is discarded. Similarly, if two subsequent ACKs reference the same sequence number, they are acknowledging the same frame.

Stop-and-wait ARQ is inefficient compared to other ARQs, because the time between packets, if the ACK and the data are received successfully, is twice the transit time (assuming the turnaround time can be zero). The throughput on the channel is a fraction of what it could be. To solve this problem, one can send more than one packet at a time with a larger sequence number and use one ACK for a set. This is what is done in Go-Back-N ARQ and the Selective Repeat ARQ.

Code :

```
#include <cnet.h>
#include <stdlib.h>
#include <string.h>
```

```
/*  This is an implementation of a stop-and-wait data link protocol.
    It is based on Tanenbaum's `protocol 4', 2nd edition, p227
    (or his 3rd edition, p205).
    This protocol employs only data and acknowledgement frames -
    piggybacking and negative acknowledgements are not used.

    It is currently written so that only one node (number 0) will
    generate and transmit messages and the other (number 1) will receive
    them. This restriction seems to best demonstrate the protocol to
    those unfamiliar with it.
    The restriction can easily be removed by "commenting out" the line

        if(nodeinfo.nodenumber == 0)

    in reboot_node(). Both nodes will then transmit and receive (why?).

    Note that this file only provides a reliable data-link layer for a
    network of 2 nodes.
 */


typedef enum    { DL_DATA, DL_ACK }   FRAMEKIND;

typedef struct {
    char        data[MAX_MESSAGE_SIZE];
} MSG;

typedef struct {
    FRAMEKIND    kind;         /* only ever DL_DATA or DL_ACK */
    unsigned int len;          /* the length of the msg field only */
    int          checksum;     /* checksum of the whole frame */
    int          seq;          /* only ever 0 or 1 */
    MSG          msg;
} FRAME;

#define FRAME_HEADER_SIZE  (sizeof(FRAME) - sizeof(MSG))
#define FRAME_SIZE(f)      (FRAME_HEADER_SIZE + f.len)


static  MSG            *lastmsg;
static  unsigned int    lastlength          = 0;
static  CnetTimerID     lasttimer           = NULLTIMER;

static  int             ackexpected         = 0;
static  int             nextframetosend     = 0;
static  int             frameexpected       = 0;
static void transmit_frame(MSG *msg, FRAMEKIND kind,
                    unsigned int length, int seqno)
{
    FRAME      f;
    int        link = 1;

    f.kind      = kind;
    f.seq       = seqno;
    f.checksum  = 0;
    f.len       = length;

    switch (kind) {
    case DL_ACK :
       printf("ACK transmitted, seq=%d\n", seqno);
       break;

    case DL_DATA: {
       CnetTime       timeout;

       printf(" DATA transmitted, seq=%d\n", seqno);
       memcpy(&f.msg, (char *)msg, (int)length);

       timeout = FRAME_SIZE(f)*((CnetTime)8000000 / linkinfo[link].bandwidth) +
                         linkinfo[link].propagationdelay;

       lasttimer = CNET_start_timer(EV_TIMER1, 3 * timeout, 0);
       break;
     }
    }
    length      = FRAME_SIZE(f);
    f.checksum  = CNET_ccitt((unsigned char *)&f, (int)length);
    CHECK(CNET_write_physical(link, (char *)&f, &length));
}


static void application_ready(CnetEvent ev, CnetTimerID timer, CnetData data)
{
    CnetAddr destaddr;

    lastlength  = sizeof(MSG);
    CHECK(CNET_read_application(&destaddr, (char *)lastmsg, &lastlength));
    CNET_disable_application(ALLNODES);

    printf("down from application, seq=%d\n", nextframetosend);
    transmit_frame(lastmsg, DL_DATA, lastlength, nextframetosend);
```

```
    nextframetosend = 1-nextframetosend;
}


static void physical_ready(CnetEvent ev, CnetTimerID timer, CnetData data)
{
    FRAME       f;
    unsigned int len;
    int         link, checksum;

    len        = sizeof(FRAME);
    CHECK(CNET_read_physical(&link, (char *)&f, &len));

    checksum    = f.checksum;
    f.checksum  = 0;
    if(CNET_ccitt((unsigned char *)&f, (int)len) != checksum) {
        printf("\t\t\t\tBAD checksum - frame ignored\n");
        return;          /* bad checksum, ignore frame */
    }

    switch (f.kind) {
    case DL_ACK :
        if(f.seq == ackexpected) {
            printf("\t\t\t\tACK received, seq=%d\n", f.seq);
            CNET_stop_timer(lasttimer);
            ackexpected = 1-ackexpected;
            CNET_enable_application(ALLNODES);
        }
        break;

    case DL_DATA :
        printf("\t\t\t\tDATA received, seq=%d, ", f.seq);
        if(f.seq == frameexpected) {
            printf("up to application\n");
            len = f.len;
            CHECK(CNET_write_application((char *)&f.msg, &len));
            frameexpected = 1-frameexpected;
        }
        else
            printf("ignored\n");
        transmit_frame((MSG *)NULL, DL_ACK, 0, f.seq);
        break;
    }
}


static void draw_frame(CnetEvent ev, CnetTimerID timer, CnetData data)
```

```
{
   CnetDrawFrame *df   = (CnetDrawFrame *)data;
   FRAME         *f    = (FRAME *)df->frame;

   switch (f->kind) {
   case DL_ACK :
      df->colour[0]   = (f->seq == 0) ? CN_RED : CN_PURPLE;
      df->pixels[0]   = 10;
      sprintf(df->text, "%d", f->seq);
      break;

   case DL_DATA :
      df->colour[0]   = (f->seq == 0) ? CN_RED : CN_PURPLE;
      df->pixels[0]   = 10;
      df->colour[1]   = CN_GREEN;
      df->pixels[1]   = 30;
      sprintf(df->text, "data=%d", f->seq);
      break;
   }
}


static void timeouts(CnetEvent ev, CnetTimerID timer, CnetData data)
{
   if(timer == lasttimer) {
      printf("timeout, seq=%d\n", ackexpected);
      transmit_frame(lastmsg, DL_DATA, lastlength, ackexpected);
   }
}


static void showstate(CnetEvent ev, CnetTimerID timer, CnetData data)
{
   printf(
   "\n\tackexpected\t= %d\n\tnextframetosend\t= %d\n\tframeexpected\t= %d\n",
               ackexpected, nextframetosend, frameexpected);
}


void reboot_node(CnetEvent ev, CnetTimerID timer, CnetData data)
{
   if(nodeinfo.nodenumber > 1) {
      fprintf(stderr,"This is not a 2-node network!\n");
      exit(1);
   }

   lastmsg     = malloc(sizeof(MSG));
```

```
    CHECK(CNET_set_handler( EV_APPLICATIONREADY, application_ready, 0));
    CHECK(CNET_set_handler( EV_PHYSICALREADY,    physical_ready, 0));
    CHECK(CNET_set_handler( EV_DRAWFRAME,        draw_frame, 0));
    CHECK(CNET_set_handler( EV_TIMER1,           timeouts, 0));
    CHECK(CNET_set_handler( EV_DEBUG0,           showstate, 0));

    CHECK(CNET_set_debug_string( EV_DEBUG0, "State"));

    if(nodeinfo.nodenumber == 1)
        CNET_enable_application(ALLNODES);
}
```

c)

AIM: C code to implement noisy channel

Procedure :

Definition

Given an alphabet , let be the set of all finite strings over . Let the dictionary of valid words be some subset of , i.e., .

The **noisy channel** is the matrix

,

where is the intended word and is the scrambled word that was actually received.

Example

Consider the English alphabet . Some subset makes up the dictionary of valid English words.

There are several mistakes that may occur while typing, including:

Missing letters, e.g., leter instead of letter

Accidental letter additions, e.g., misstake instead of mistake

Swapping letters, e.g., recieved instead of received

Replacing letters, e.g., fimite instead of finite

To construct the noisy channel matrix , we must consider the probability of each mistake, given the intended word (for all and). These probabilities may be gathered, for example, by considering the Levenshtein distance between and or by comparing the draft of an essay with one that has been manually edited for spelling.

Error-correction

The goal of the noisy channel model is to find the intended word given the scrambled word that was received. The **decision function** is a function that, given a scrambled word, returns the intended word.

Methods of constructing a decision function include the maximum likelihood rule, the maximum a posteriori rule, and the minimum distance rule.
In some cases, it may be better to accept the scrambled word as the intended word rather than attempt to find an intended word in the dictionary. For example, the word schönfinkeling may not be in the dictionary, but might in fact be the intended word.
Code :

```
#include<iostream>
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<dos.h>
using namespace std;
#define time 5
#define max_seq 1
#define tot_pack 5
int randn(int n)
{
   return rand()%n + 1;
}
typedef struct
{
      int data;
}packet;
typedef struct
{
       int kind;
       int seq;
       int ack;
       packet info;
}frame;
typedef enum{ frame_arrival,error,time_out}event_type;
frame data1;
//creating prototype
void from_network_layer(packet *);
void to_physical_layer(frame *);
void to_network_layer(packet *);
void from_physical_layer(frame*);
void sender();
void receiver();
void wait_for_event_sender(event_type *);
void wait_for_event_receiver(event_type *);
//end


#define inc(k) if(k<max_seq)k++;else k=0;
```

```
int i=1;
char turn;
int disc=0;
int main()
{
  while(!disc)
  {  sender();
    // delay(400);
     receiver();
  }
   getchar();
}
void sender()
{
     static int frame_to_send=0;
     static frame s;
     packet buffer;
     event_type event;
     static int flag=0;       //first place
     if (flag==0)
     {
   from_network_layer(&buffer);
   s.info=buffer;
   s.seq=frame_to_send;
   cout<<"\nsender information \t"<<s.info.data<<"\n";
   cout<<"\nsequence no. \t"<<s.seq;


   turn='r';
   to_physical_layer(&s);
   flag=1;
     }


 wait_for_event_sender(&event);
 if(turn=='s')
 {
     if(event==frame_arrival)
      {
      from_network_layer(&buffer);
      inc(frame_to_send);
      s.info=buffer;
      s.seq=frame_to_send;
      cout<<"\nsender information \t"<<s.info.data<<"\n";
      cout<<"\nsequence no. \t"<<s.seq<<"\n";
      getch();
      turn='r';
      to_physical_layer(&s);
      }


 }


}                  //end of sender function


void from_network_layer(packet *buffer)
{
     (*buffer).data=i;
     i++;
}                          //end of from network layer function


void to_physical_layer(frame *s)
{


     data1=*s;
}            //end of to physical layer function


void wait_for_event_sender(event_type *e)
{
      static int timer=0;
      if(turn=='s')
      {   timer++;
    //timer=0;
    return ;
  }


  else                         //event is frame arrival
    {
       timer=0;
       *e=frame_arrival;
    }


}             //end of wait for event function
```

```
void receiver()
{
    static int frame_expected=0;
    frame s,r;
    event_type event;
    wait_for_event_receiver(&event);
    if(turn=='r')
    {  if(event==frame_arrival)
   {
        from_physical_layer(&r);
        if(r.seq==frame_expected)
    {
      to_network_layer(&r.info);
      inc (frame_expected);
    }
    else
    cout<<"\nReceiver :Acknowledgement resent \n";
    getch();
    turn='s';
    to_physical_layer(&s);
     }


    }
}                  //end of receiver function




void wait_for_event_receiver(event_type *e)
{
    if(turn=='r')
    {
   *e=frame_arrival;
    }
}


void from_physical_layer(frame *buffer)
{
   *buffer=data1;
}


void to_network_layer(packet *buffer)
{
    cout<<"\nReceiver : packet received \t"<< i-1;
    cout<<"\n Acknowledgement  sent \t";
    getch();
    if(i>tot_pack)
     { disc=1;
 cout<<"\ndiscontinue\n";
     }
}      //end of network layer function
```

6) Implementation of sliding window protocols

a) Aim: C code to implement 1-bit sliding window protocol

Procedure :

The transmitter and receiver each have a current sequence number n_t and n_r, respectively. They each also have a window size w_t and w_r. The window sizes may vary, but in simpler implementations they are fixed. The window size must be greater than zero for any progress to be made.

As typically implemented, n_t is the next packet to be transmitted, i.e. the sequence number of the first packet not yet transmitted. Likewise, n_r is the first packet not yet received. Both numbers are monotonically increasing with time; they only ever increase.

The receiver may also keep track of the highest sequence number yet received; the variable n_s is one more than the sequence number of the highest sequence number received. For simple receivers that only accept packets in order ($w_r = 1$), this is the same as n_r, but can be greater if $w_r > 1$. Note the distinction: all packets below n_r have been received, no packets above n_s have been received, and between n_r and n_s, some packets have been received.

When the receiver receives a packet, it updates its variables appropriately and transmits an acknowledgment with the new n_r. The transmitter keeps track of the highest acknowledgment it has received n_a. The transmitter knows that all packets up to, but not including n_a have been received, but is uncertain about packets between n_a and n_s; i.e. $n_a \le n_r \le n_s$.

The sequence numbers always obey the rule that $n_a \le n_r \le n_s \le n_t \le n_a + w_t$. That is:

$n_a \le n_r$: The highest acknowledgement received by the transmitter cannot be higher than the highest n_r acknowledged by the receiver.

$n_r \le n_s$: The span of fully received packets cannot extend beyond the end of the partially received packets.

$n_s \le n_t$: The highest packet received cannot be higher than the highest packet sent.

$n_t \le n_a + w_t$: The highest packet sent is limited by the highest acknowledgement received and the transmit window size.

T

Code :

```
#include<stdio.h>

int main()
{
   int w,i,f,frames[50];

```

```
    printf("Enter window size: ");
    scanf("%d",&w);

    printf("\nEnter number of frames to transmit: ");
    scanf("%d",&f);

    printf("\nEnter %d frames: ",f);

    for(i=1;i<=f;i++)
        scanf("%d",&frames[i]);

    printf("\nWith sliding window protocol the frames will be sent in the following manner (assuming no corruption of frames)\n\n");
    printf("After sending %d frames at each stage sender waits for acknowledgement sent by the receiver\n\n",w);

    for(i=1;i<=f;i++)
    {
        if(i%w==0)
        {
            printf("%d\n",frames[i]);
            printf("Acknowledgement of above frames sent is received by sender\n\n");
        }
        else
            printf("%d ",frames[i]);
    }

    if(f%w!=0)
        printf("\nAcknowledgement of above frames sent is received by sender\n");

    return 0;
}
```

b) AIM: To develop goback-N sliding window protocol

```
#include<stdio.h>
#include<conio.h>
void main()
{
char sender[50],receiver[50];
int i,winsize;
clrscr();
 printf("\n ENTER THE WINDOWS SIZE : ");
scanf("%d",&winsize);
 printf("\n SENDER WINDOW IS EXPANDED TO STORE MESSAGE OR WINDOW \n");
 printf("\n ENTER THE DATA TO BE SENT: ");
fflush(stdin);
gets(sender);
for(i=0;i<winsize;i++)
receiver[i]=sender[i];
receiver[i]=NULL;
 printf("\n MESSAGE SEND BY THE SENDER:\n");
 puts(sender);
 printf("\n WINDOW SIZE OF RECEIVER IS EXPANDED\n");
 printf("\n ACKNOWLEDGEMENT FROM RECEIVER \n");
for(i=0;i<winsize;i++);
printf("\n ACK:%d",i);
 printf("\n MESSAGE RECEIVED BY RECEIVER IS : ");
 puts(receiver);
 printf("\n WINDOW SIZE OF RECEIVER IS SHRINKED \n");
getch();
}
```

c)Aim:to develop selective repeate window protocol

```
                                                    //inculsion
#include<iostream>
#include<stdio.h>

#include<sys/types.h>
#include<netinet/in.h>
#include<netdb.h>

#define cls() printf("33[H33[J")

                                                    //structure definition for designing the packet.
struct frame
{
 int packet[40];
};

                                //structure definition for accepting the acknowledgement.
struct ack
{
 int acknowledge[40];
};


int main()
{
 int serversocket;
 sockaddr_in serveraddr,clientaddr;
 socklen_t len;
 int windowsize,totalpackets,totalframes,framessend=0,i=0,j=0,k,l,m,n,repacket[40];
 ack acknowledgement;
 frame f1;
 char req[50];
 serversocket=socket(AF_INET,SOCK_DGRAM,0);

 bzero((char*)&serveraddr,sizeof(serveraddr));
 serveraddr.sin_family=AF_INET;
 serveraddr.sin_port=htons(5018);
 serveraddr.sin_addr.s_addr=INADDR_ANY;

 bind(serversocket,(sockaddr*)&serveraddr,sizeof(serveraddr));

 bzero((char*)&clientaddr,sizeof(clientaddr));
 len=sizeof(clientaddr);

                                        //connection establishment.
 printf("\nWaiting for client connection.\n");
 recvfrom(serversocket,req,sizeof(req),0,(sockaddr*)&clientaddr,&len);
 printf("\nThe client connection obtained.\t%s\n",req);

                                        //sending request for windowsize.
 printf("\nSending request for window size.\n");
 sendto(serversocket,"REQUEST FOR WINDOWSIZE.",sizeof("REQUEST FOR
WINDOWSIZE."),0,(sockaddr*)&clientaddr,sizeof(clientaddr));

                                        //obtaining windowsize.
 printf("\nWaiting for the windowsize.\n");
 recvfrom(serversocket,(char*)&windowsize,sizeof(windowsize),0,(sockaddr*)&clientaddr,&len);
 cls();
 printf("\nThe windowsize obtained as:\t%d\n",windowsize);

 printf("\nObtaining packets from network layer.\n");
 printf("\nTotal packets obtained:\t%d\n",(totalpackets=windowsize*5));
 printf("\nTotal frames or windows to be transmitted:\t%d\n",(totalframes=5));

                                        //sending details to client.
 printf("\nSending total number of packets.\n");
 sendto(serversocket,(char*)&totalpackets,sizeof(totalpackets),0,(sockaddr*)&clientaddr,sizeof(clientad
dr));
 recvfrom(serversocket,req,sizeof(req),0,(sockaddr*)&clientaddr,&len);

 printf("\nSending total number of frames.\n");
 sendto(serversocket,(char*)&totalframes,sizeof(totalframes),0,(sockaddr*)&clientaddr,sizeof(clientadd
r));
 recvfrom(serversocket,req,sizeof(req),0,(sockaddr*)&clientaddr,&len);

 printf("\nPRESS ENTER TO START THE PROCESS.\n");
 fgets(req,2,stdin);
 cls();
```

```
j=0;
l=0;                                         //starting the process of sending
while( l<totalpackets)
{
                                             //initialising the transmit buffer.
 bzero((char*)&f1,sizeof(f1));
 printf("\nInitialising the transmit buffer.\n");
 printf("\nThe frame to be send is %d with packets:\t",framessend);
                                             //Builting the frame.
 for(m=0;m<j;m++)
 {
          //including the packets for which negative acknowledgement was received.
  printf("%d  ",repacket[m]);
  f1.packet[m]=repacket[m];
 }

 while(j<windowsize && i<totalpackets)
 {
  printf("%d  ",i);
  f1.packet[j]=i;
  i++;
  j++;
 }
 printf("\nSending frame %d\n",framessend);

                                             //sending the frame.
 sendto(serversocket,(char*)&f1,sizeof(f1),0,(sockaddr*)&clientaddr,sizeof(clientaddr));
                                             //Waiting for the acknowledgement.
 printf("\nWaiting for the acknowledgement.\n");
 recvfrom(serversocket,(char*)&acknowledgement,sizeof(acknowledgement),0,(sockaddr*)&clientaddr,
&len);
 cls();

                                             //Checking acknowledgement of each packet.
 j=0;
 k=0;
 m=0;
 n=l;
 while(m<windowsize && n<totalpackets)
 {
  if(acknowledgement.acknowledge[m]==-1)
  {
   printf("\nNegative acknowledgement received for packet: %d\n",f1.packet[m]);
   k=1;
   repacket[j]=f1.packet[m];
   j++;
  }
  else
  {
   l++;
  }
  m++;
  n++;
 }

 if(k==0)
 {
printf("\nPositive acknowledgement received for all packets within the frame: %d\n",framessend);
 }

 framessend++;
 printf("\nPRESS ENTER TO PROCEED……\n");
 fgets(req,2,stdin);
 cls();
}

printf("\nAll frames send successfully.\n\nClosing connection with the client.\n");
close(serversocket);
}
```

c) Selective window sliding protocol

Aim: To implement Selective window sliding protocol

Procedure :

Transmitter operation[edit]

Whenever the transmitter has data to send, it may transmit up to w_t packets ahead of the latest acknowledgment n_a. That is, it may transmit packet number n_t as long as $n_t < n_a+w_t$.

In the absence of a communication error, the transmitter soon receives an acknowledgment for all the packets it has sent, leaving n_a equal to n_t. If this does not happen after a reasonable delay, the transmitter must retransmit the packets between n_a and n_t.

Techniques for defining "reasonable delay" can be extremely elaborate, but they only affect efficiency; the basic reliability of the sliding window protocol does not depend on the details.

Receiver operation[edit]

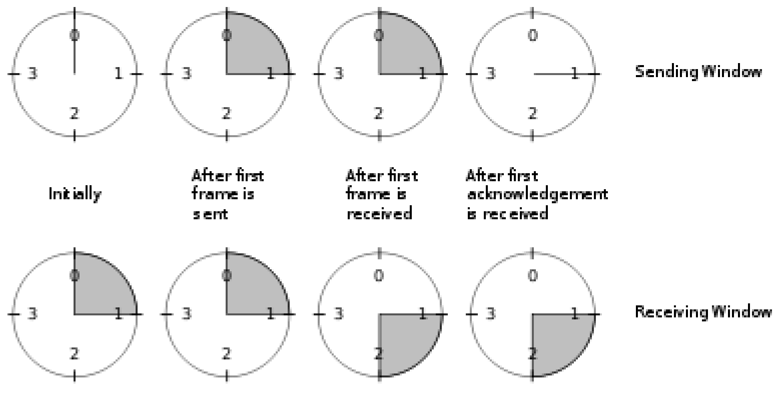
Every time a packet numbered x is received, the receiver checks to see if it falls in the receive window, $n_r \le x < n_r+w_r$. (The simplest receivers only have to keep track of one value $n_r=n_s$.) If it falls within the window, the receiver accepts it. If it is numbered n_r, the receive sequence number is increased by 1, and possibly more if further consecutive packets were previously received and stored. If $x > n_r$, the packet is stored until all preceding packets have been received.[1] If $x \ge n_s$, the latter is updated to $n_s=x+1$.

If the packet's number is not within the receive window, the receiver discards it and does not modify n_r or n_s.

Whether the packet was accepted or not, the receiver transmits an acknowledgment containing the current n_r. (The acknowledgment may also include information about additional packets received between n_r or n_s, but that only helps efficiency.)
Note that there is no point having the receive window w_r larger than the transmit window w_t, because there is no need to worry about receiving a packet that will never be transmitted; the useful range is $1 \le w_r \le w_t$.

Sequence number range required**[edit]**

Main article: serial number arithmetic


Sending Window
Receiving Window
Initially
After first frame is sent
After first frame is received
After first acknowledgement is received
0
1
2
3
A sliding window with a 2-bit sequence, of size 1

Sequence numbers modulo 4, with w_r=1. Initially, n_t=n_r=0
So far, the protocol has been described as if sequence numbers are of unlimited size, ever-increasing. However, rather than transmitting the full sequence number x in messages, it is possible to transmit only x mod N, for some finite N. (N is usually a power of 2.)
For example, the transmitter will only receive acknowledgments in the range n_a to n_t, inclusive. Since it guarantees that $n_t - n_a \le w_t$, there are at most w_t+1 possible sequence numbers that could arrive at any given time. Thus, the transmitter can unambiguously decode the sequence number as long as $N > w_t$.
A stronger constraint is imposed by the receiver. The operation of the protocol depends on the receiver being able to reliably distinguish new packets (which should be accepted and processed) from retransmissions of old packets (which should be discarded, and the last acknowledgment retransmitted). This can be done given knowledge of the transmitter's window size. After receiving a packet numbered x, the receiver knows that $x < n_a+w_t$, so $n_a > x-w_t$. Thus, packets numbered $x-w_t$ will never again be retransmitted.
The lowest sequence number we will ever receive in future is n_s-w_t
The receiver also knows that the transmitter's n_a cannot be higher than the highest acknowledgment ever sent, which is n_r. So the highest sequence number we could possibly see is $n_r+w_t \le n_s+w_t$.
Thus, there are $2w_t$ different sequence numbers that the receiver can receive at any one time. It might therefore seem that we must have $N \ge 2w_t$. However, the actual limit is lower.
The additional insight is that the receiver does not need to distinguish between sequence numbers that are too low (less than n_r) or that are too high (greater than or equal to n_s+w_r). In either case, the receiver ignores the packet except to retransmit an acknowledgment. Thus, it is only necessary that $N \ge w_t+w_r$. As it is common to have $w_r<w_t$ (e.g. see Go-Back-Nbelow), this can permit larger w_t within a fixed N.
Code :

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

```
main()
int i,m,n,j,w,1;
char c;
FILE*f;
f=fopen("text.txt","r");
printf("window size");
scanf("%d",&n);
m=n;
while(!fof(f))
{
i=rand()%n+1;
j=i;
1=i;
if(m>i)
{
m=m-i;
if(m>0)
{
printf("\n");
while(i>0 & !feof(f))
{
c=getc(f);
printf("%c",c);
i--;
}
printf("\n%d transferred"j);
if(j>3)
printf("\n 1 acknowledgement received");
else
printf("\n acknowledgement received",j+1);
}
}
m=m+j-1;}
```

7.Implementation of Routing Protocols

a).Shortest Path

Aim:To implement Shortest path

Procedure :

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.[1][2][3]

The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes,[3] but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

For a given source node in the graph, the algorithm finds the shortest path between that node and every other.[4]:196–206 It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path algorithm is widely used in network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and Open Shortest Path First (OSPF). It is also employed as a subroutine in other algorithms such as Johnson's.

Dijkstra's original algorithm does not use a min-priority queue and runs in time (where is the number of nodes). The idea of this algorithm is also given in Leyzorek et al. 1957. The implementation based on a min-priority queue implemented by a Fibonacci heap and running

in (where is the number of edges) is due to Fredman & Tarjan 1984. This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights. However, specialized cases (such as bounded/integer weights, directed acyclic graphs etc.) can indeed be improved further as detailed in § Specialized variants.
In some fields, artificial intelligence in particular, Dijkstra's algorithm or a variant of it is known as **uniform cost search** and formulated as an instance of the more general idea of best-first search.[5]

Source Code:
```
#include <stdio.h>
#define MAX 7
#define INFINITE 998
int allselected(int *selected)
{
   int i;
   for(i=0;i<MAX;i++)
      if(selected[i]==0)
         return 0;
   return 1;
}
void shortpath(int cost[][MAX],int *preced,int *distance)
{
   int selected[MAX]={0};
   int current=0,i,k,dc,smalldist,newdist;
   for(i=0;i<MAX;i++)
     distance[i]=INFINITE;
   selected[current]=1;
   distance[0]=0;
   current=0;
   while(!allselected(selected))
   {
      smalldist=INFINITE;
      dc=distance[current];
      for(i=0;i<MAX;i++)
      {
                  if(selected[i]==0)
```

```
                {
                        newdist=dc+cost[current][i];
                        if(newdist<distance[i])
                        {
                                distance[i]=newdist;

                                preced[i]=current;
                        }
                        if(distance[i]<smalldist)
                        {
                                 smalldist=distance[i];
                                 k=i;
                        }
                }
        }
        current=k;
        selected[current]=1;
  }
}

int main()
{
   int
cost[MAX][MAX]={{INFINITE,2,4,7,INFINITE,5,INFINITE},{2,INFINITE,INFINITE,6,3,INFINITE,8},{4,INFINITE,I
NFINITE,INFINITE,INFINITE,6,INFINITE},{7,6,INFINITE,INFINITE,INFINITE,1,6},{INFINITE,3,INFINITE,INFINIT
E,INFINITE,INFINITE,7},{5,INFINITE,6,1,INFINITE,INFINITE,6},{INFINITE,8,INFINITE,6,7,6,INFINITE}};
   int i,preced[MAX]={0},distance[MAX];
   shortpath(cost,preced,distance);
   for(i=0;i<MAX;i++)
      printf("%d\n",distance[i]);

   return 0;
}
```

b).Distance Vector Routing

Aim:

To implementing the distance vector routing algorithm.

Source Code:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int path[5][5],i,j,min,a[5][5],p,st=1,ed=5,stp,edp,t[5],index;
clrscr();
printf("enter the cost matrix\n");
for(i=1;i<=5;i++)
for(j=1;j<=5;j++)
scanf("%d",&a[i][j]);
printf("enter  the paths\n");
scanf("%d",&p);
printf("enter possible paths\n");
for(i=1;i<=p;i++)
for(j=1;j<=5;j++)
scanf("%d",&path[i][j]);
for(i=1;i<=p;i++)
{
t[i]=0;
stp=st;
for(j=1;j<=5;j++)
{
edp=path[i][j+1];
t[i]=t[i]+a[stp][edp];
if(edp==ed)
break;
else
stp=edp;
}
}
min=t[st];index=st;
for(i=1;i<=p;i++)
{
if(min>t[i])
{
min=t[i];
index=i;
}
}
printf("minimum cost %d",min);
printf("\n minimum cost path ");
for(i=1;i<=5;i++)
{
printf("--> %d",path[index][i]);
if(path[index][i]==ed)
break;
}
getch();
}
```

c)Link State Routing Protocol

Aim : To implement Link state Routing Protocol

Procedure :

Link-state routing protocols are one of the two main classes of routing protocols used in packet switching networks for computer communications, the other being distance-vector routing protocols. Examples of link-state routing protocols include Open Shortest Path First (OSPF) and intermediate system to intermediate system (IS-IS).

The link-state protocol is performed by every switching node in the network (i.e., nodes that are prepared to forward packets; in the Internet, these are called routers). The basic concept of link-state routing is that every node constructs a map of the connectivity to the network, in the form of a graph, showing which nodes are connected to which other nodes. Each node then independently calculates the next best logical path from it to every possible destination in the network. Each collection of best paths will then form each node's routing table.

This contrasts with distance-vector routing protocols, which work by having each node share its routing table with its neighbours. In a link-state protocol the only information passed between nodes is connectivity related. Link-state algorithms are sometimes characterized informally as each router, "telling the world about its neighbours."

Code :

```
#include<stdio.h>
struct node
{
   unsigned dist[20];
```

```
  unsigned from[20];
}rt[10];
int main()
{
  int costmat[20][20];
  int nodes,i,j,k,count=0;
  printf("\nEnter the number of nodes : ");
  scanf("%d",&nodes);//Enter the nodes
  printf("\nEnter the cost matrix :\n");
  for(i=0;i<nodes;i++)
  {
    for(j=0;j<nodes;j++)
    {
      scanf("%d",&costmat[i][j]);
      costmat[i][i]=0;
      rt[i].dist[j]=costmat[i][j];//initialise the distance equal to cost matrix
      rt[i].from[j]=j;
    }
  }
    do
    {
      count=0;
      for(i=0;i<nodes;i++)//We choose arbitary vertex k and we calculate the direct distance from the
node i to k using the cost matrix
      //and add the distance from k to node j
      for(j=0;j<nodes;j++)
      for(k=0;k<nodes;k++)
        if(rt[i].dist[j]>costmat[i][k]+rt[k].dist[j])
        {//We calculate the minimum distance
          rt[i].dist[j]=rt[i].dist[k]+rt[k].dist[j];
          rt[i].from[j]=k;
          count++;
        }
    }while(count!=0);
    for(i=0;i<nodes;i++)
    {
      printf("\n\n For router %d\n",i+1);
      for(j=0;j<nodes;j++)
      {
        printf("\t\nnode %d via %d Distance %d ",j+1,rt[i].from[j]+1,rt[i].dist[j]);
      }
    }
  printf("\n\n");
  getch();
}
```

8. Implementation of congestion algorithm

a) Token Bucket algorithm

Aim: To implement Token Bucket algorithm

Procedure :

The token bucket algorithm is based on an analogy of a fixed capacity bucket into which tokens, normally representing a unit of bytes or a single packet of predetermined size, are added at a fixed rate. When a packet is to be checked for conformance to the defined limits, the bucket is inspected to see if it contains sufficient tokens at that time. If so, the appropriate number of tokens, e.g. equivalent to the length of the packet in bytes, are removed ("cashed in"), and the packet is passed, e.g., for transmission. The packet does not conform if there are insufficient tokens in the bucket, and the contents of the bucket are not changed. Non-conformant packets can be treated in various ways:
They may be dropped.
They may be enqueued for subsequent transmission when sufficient tokens have accumulated in the bucket.
They may be transmitted, but marked as being non-conformant, possibly to be dropped subsequently if the network is overloaded.
A conforming flow can thus contain traffic with an average rate up to the rate at which tokens are added to the bucket, and have a burstiness determined by the depth of the bucket. This burstiness may be expressed in terms of either a jitter tolerance, i.e. how much sooner a packet might conform (e.g. arrive or be transmitted) than would be expected from the limit on the average rate, or a burst tolerance or maximum burst size, i.e. how much more than the average level of traffic might conform in some finite period.

Code :

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

```
#include<unistd.h>

#define NOF_PACKETS 10

int rand(int a)
{
   int rn = (random() % 10) % a;
   return  rn == 0 ? 1 : rn;
}

int main()
{
   int packet_sz[NOF_PACKETS], i, clk, b_size, o_rate, p_sz_rm=0, p_sz, p_time, op;
   for(i = 0; i<NOF_PACKETS; ++i)
      packet_sz[i] = rand(6) * 10;
   for(i = 0; i<NOF_PACKETS; ++i)
      printf("\npacket[%d]:%d bytes\t", i, packet_sz[i]);
   printf("\nEnter the Output rate:");
   scanf("%d", &o_rate);
   printf("Enter the Bucket Size:");
   scanf("%d", &b_size);
   for(i = 0; i<NOF_PACKETS; ++i)
   {
      if( (packet_sz[i] + p_sz_rm) > b_size)
         if(packet_sz[i] > b_size)/*compare the packet siz with bucket size*/
            printf("\n\nIncoming packet size (%dbytes) is Greater than bucket capacity (%dbytes)-PACKET
REJECTED", packet_sz[i], b_size);
         else
            printf("\n\nBucket capacity exceeded-PACKETS REJECTED!!");
      else
      {
         p_sz_rm += packet_sz[i];
         printf("\n\nIncoming Packet size: %d", packet_sz[i]);
         printf("\nBytes remaining to Transmit: %d", p_sz_rm);
         p_time = rand(4) * 10;
         printf("\nTime left for transmission: %d units", p_time);
         for(clk = 10; clk <= p_time; clk += 10)
         {
            sleep(1);
            if(p_sz_rm)
            {
               if(p_sz_rm <= o_rate)/*packet size remaining comparing with output rate*/
                  op = p_sz_rm, p_sz_rm = 0;
               else
                  op = o_rate, p_sz_rm -= o_rate;
               printf("\nPacket of size %d Transmitted", op);
               printf("----Bytes Remaining to Transmit: %d", p_sz_rm);
            }
            else
            {
               printf("\nTime left for transmission: %d units", p_time-clk);
               printf("\nNo packets to transmit!!");
            }
         }
      }
   }
}
```

b)Leaky Bucket Algorithm

Aim : To implement Leaky Bucket Algorithm

Procedure :

The **leaky bucket** is an algorithm based on an analogy of how a bucket with a leak will overflow if either the average rate at which water is poured in exceeds the rate at which the bucket leaks or if more water than the capacity of the bucket is poured in all at once, and how the water leaks from the bucket at an (almost) constant rate. It can be used to determine whether some sequence of discrete events conforms to defined limits on their average and peak rates or frequencies, or to directly limit the actions associated to these events to these rates, and may be used to limit these actions to an average rate alone, i.e. remove any variation from the average.

It is used in packet switched computer networks and telecommunications networks in both the traffic policing and traffic shaping of data transmissions, in the form of packets,[note 1] to defined limits on bandwidth and burstiness (a measure of the unevenness or variations in the trafficflow). It can also be used as a scheduling algorithm to determine the timing of transmissions that will comply with the limits set for the bandwidth and burstiness applied by the network: see network scheduler.[1][2][3][4] A version of the leaky bucket, the Generic Cell Rate Algorithm, is recommended for Asynchronous Transfer Mode (ATM) networks[5] in Usage/Network Parameter Control at User–Network Interfaces or Inter-Network Interfaces or Network-Network Interfaces to protect a network from excessive traffic levels on connections routed through it. The Generic Cell Rate Algorithm, or an equivalent, may also be used to shape transmissions by a Network Interface Card onto an ATM network (i.e. on the user side of the User-Network Interface), e.g. to levels below the levels set for Usage/Network Parameter Control in the network to prevent it taking action to further limit that connection. The leaky bucket algorithm is also used in leaky bucket counters, e.g. to detect when the average or peak rate of random or stochastic events or stochastic processes, such as faults or failures, exceed defined limits.

At least some implementations of the leaky bucket are a mirror image of the Token Bucket algorithm and will, given equivalent parameters, determine exactly the same sequence of events to conform or not conform to the same limits. However, there are at least two different descriptions of the leaky bucket that can and have caused confusion

Code :

```
using System;
using System.DateTime;

public class TokenBucket
```

```
{
   float _capacity = 0;
   float _tokens   = 0;
   float _fillrate = 0;
   DateTime _time_stamp;

   public TokenBucket(float tokens, float fill_rate)
   {
      _capacity  = tokens;
      _tokens    = tokens;
      _fill_rate = fill_rate;
      _time_stamp = DateTime.Now;
   }

   public bool Consume(float tokens)
   {
      if(tokens       {
         _tokens -= tokens;
      }else{
         return false;
      }
      return true;
   }

   public float GetTokens()
   {
      DateTime _now = DateTime.Now;
      if(_tokens < _capacity)
      {
         var delta = _fill_rate * (_now - _time_stamp);
         _tokens = Math.Min(_capacity, _tokens + delta);
         _time_stamp = _now;
      }
      return _tokens;
   }
}
```