PART-A

1. Implement the data link layer framing methods such as Bit Stuffing

AIM: Implement the data link layer framing methods such as Bit Stuffing

THEORY :

Security and Error detection are the most prominent features that are to be provided by any application which transfers data from one end to the other end. One of such a mechanism in tracking errors which may add up to the original data during transfer is known as Stuffing. It is of two types namely Bit Stuffing and the other Character Stuffing. Coming to the Bit Stuffing, 01111110 is appended within the original data while transfer of it. The following program describes how it is stuffed at the sender end and de-stuffed at the reciever end.

PROGRAM:

```
#include<stdio.h>

#include<string.h>
void main()
{
int a[20],b[30],i,j,k,count,n;
printf("Enter frame length:");
scanf("%d",&n);
printf("Enter input frame (0's & 1's only):");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
i=0; count=1; j=0;
while(i<n)
{
if(a[i]==1)
{
b[j]=a[i];
for(k=i+1;a[k]==1 && k<n && count<5;k++)
{
j++;
b[j]=a[k];
count++;
if(count==5)
{
j++;
b[j]=0;
}
i=k;
}}
else
{
b[j]=a[i];
}
i++;
j++;
```

```
}
printf("After stuffing the frame is:");
for(i=0;i<j;i++)
printf("%d",b[i]);
}
```

OUTPUT:

2.b Implement the data link layer framing methods such as Character Stuffing and also Destuff it

AIM: Implement the data link layer framing methods such as Character Stuffing and also Destuff it

THEORY :

Coming to the Character Stuffing, DLESTX and DLEETX are used to denote start and end of character data with some constraints imposed on repetition of charaters as shown in the program below clearly.

PROGRAM :

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

void charc(void);

void main() {

int choice;

while(1) {

printf("\n\n\n1.character stuffing");

printf("\n\n2.exit");

printf("\n\n\nenter choice");

scanf("%d",&choice);

printf("%d",choice);

 if(choice>2)

printf("\n\n invalid option... please renter");

switch(choice) {

      case 1: charc();

      break;

 case 2:

      exit(0);

}
```

```
    }
        }
void charc(void) {
char c[50],d[50],t[50];
int i,m,j;
clrscr();
printf("enter the number of characters\n");
scanf("%d",&m);
printf("\n enter the characters\n");
for(i=0;i<m+1;i++)
 {
scanf("%c",&c[i]);
 }
printf("\n original data\n");
 for(i=0;i<m+1;i++)
 printf("%c",c[i]);
 d[0]='d';
d[1]='l';
 d[2]='e';
 d[3]='s';
d[4]='t';
d[5]='x';
for(i=0,j=6;i<m+1;i++,j++) {
if((c[i]=='d'&&c[i+1]=='l'&& c[i+2]=='e')) {
d[j]='d';
j++;
d[j]='l';
```

```
 j++;
 d[j]='e';
j++;
m=m+3;
}
d[j]=c[i];
}
m=m+6;m++;
d[m]='d'; m++;
d[m]='l'; m++;
d[m]='e'; m++;
d[m]='e'; m++;
d[m]='t'; m++;
d[m]='x';
m++;
printf("\n\n transmitted data: \n");
for(i=0;i<m;i++)
 {
printf("%c",d[i]);
}
for(i=6,j=0;i<m-6;i++,j++)
 {
if(d[i]=='d'&&d[i+1]=='l'&&d[i+2]=='e'&&d[i+3]=='d'&&d[i+4]=='l'&&d[i+5]=='e')
   i=i+3;
  t[j]=d[i];
}
printf("\n\nreceived data:");
```

```
for(i=0;i<j;i++) {
printf("%c",t[i]);
}
}
```

OUTPUT:

2. Implement on a data set of characters the CRC polynomials.

AIM: Implement on a data set of characters the CRC polynomials.

THEORY :

CRC means Cyclic Redundancy Check. It is the most famous and traditionally successful mechanism used in error detection through the parity bits installed within the data and obtaining checksum which acts as the verifier to check whether the data retreived at the reciever end is genuine or not. Various operations are involved in implementing CRC on a data set through CRC generating polynomials. In the program, I have also provided the user to opt for Error detection whereby he can proceed for it. Understand the program below as it is much simpler than pretented to be so.

PROGRAM :

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#define N strlen(g)
char t[128], cs[128], g[]="100010000";
int a, e, c;
void xor() {
for(c=1;c }
void crc() {
for(e=0;e do { if(cs[0]=='1') xor(); for(c=0;c cs[c]=t[e++];
}
while(e<=a+N-1);
}
void main() {
clrscr();
printf("\nEnter poly : ");
scanf("%s",t);
printf("\nGenerating Polynomial is : %s",g);
```

```
a=strlen(t); for(e=a;e

printf("\nModified t[u] is : %s",t);

crc();

printf("\nChecksum is : %s",cs);

for(e=a;e printf("\nFinal Codeword is : %s",t);

printf("\nTest Error detection 0(yes) 1(no) ? : ");

scanf("%d",&e);

 if(e==0) {

printf("Enter position where error is to inserted : ");

scanf("%d",&e);

t[e]=(t[e]=='0')?'1':'0'; printf("Errorneous data :
%s\n",t);

}

crc();

for (e=0;(e<n-1)&&(cs[e]!='1');e++);

 if(e

else printf("No Error Detected.");

getch();

}
```

OUTPUT:

3. Implement Dijkstra‘s algorithm to compute the Shortest path through a graph.

AIM: Implement Dijkstra‘s algorithm to compute the Shortest path through a graph.

THEORY :

Dijkstra's algorithm finds the shortest path from a starting node to all other nodes in a weighted graph with non-negative weights. It initializes the distance to the start node as 0 and all others as infinity, using a priority queue to always extend the shortest known path. By repeatedly extracting the node with the smallest distance and updating the distances to its neighbors if a shorter path is found, the algorithm ensures that the shortest paths to all nodes are determined. It is optimal and efficient, with a time complexity of O((V + E) log V), where V is the number of vertices and E is the number of edges.

PROGRAM :

```
#include<stdio.h>
#define LARGE 10000
#define MAX 30
struct state
{
	int len;
	int pre;
	int label;
};
struct state states[MAX];
int a[MAX][MAX];
int main()
{
	int i,j,s,d,n,tem,min,mini;
	printf("Enter no.of vertices:");
	scanf("%d",&n);
	printf("\nEnter adjacency matrix\n");
	for(i=0;i<n;i++)
	 for(j=0;j<n;j++)
	 {
		printf("a[%d][%d]=",i,j);
		scanf("%d",&a[i][j]);
	 }
	for(i=0;i<n;i++)
	{
		states[i].len=LARGE;
		states[i].label=0;
		states[i].pre=-1;
	}
	printf("\nEnter source vertex:");
	scanf("%d",&s);
	printf("\nEnter destination vertex:");
	scanf("%d",&d);
	states[d].len=0;
	states[d].label=1;
	tem=d;
```

```
    while(tem!=s)
    {
        for(i=0;i<n;i++)
        {
         if(a[tem][i]!=0&&
            states[tem].len+a[tem][i]<states[i].len&&states[i].label==0)
         {
              states[i].len=states[tem].len+a[tem][i];
              states[i].pre=tem;
         }
        }
        min=LARGE;
        mini=0;
        for(i=0;i<n;i++)
        {
         if(states[i].len<LARGE&&states[i].label==0)
         {
              min=states[i].len;
              mini=i;
         }
        }
        states[mini].label=1;
        tem=mini;
    }
    tem=s;
    printf("\nPath length:%d",states[s].len);
    printf("\nPath\n");
    printf("%d",tem);
    do{
    tem=states[tem].pre;
    printf("-%d",tem);
    }while(tem!=d);
    return(0);
}
```

OUTPUT:

4. Take an example subnet graph with weights indicating delay between nodes. Now obtain Routing table art each node using distance vector routing algorithm

AIM: Take an example subnet graph with weights indicating delay between nodes. Now obtain Routing table art each node using distance vector routing algorithm

THEORY:

The Distance Vector Routing algorithm is a dynamic routing protocol used in packet-switched networks to calculate the shortest path between nodes. Each router maintains a distance vector table that contains the distance to every other node in the network and the next hop router to reach that destination. The algorithm works by periodically exchanging distance vectors with neighboring routers. When a router receives a distance vector from a neighbor, it updates its own table if a shorter path is found via that neighbor. This process continues iteratively until all routers have the shortest path information for the entire network. Key properties include simplicity, distributed operation, and convergence towards the optimal path, but it can suffer from slow convergence and the "count to infinity" problem in the presence of network changes.

PROGRAM:

```
#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=1;i<=nodes;i++)
    {
       for(j=1;j<=nodes;j++) {
scanf("%d",&costmat[i][j]);
costmat[i][i]=0;
rt[i].dist[j]=costmat[i][j];
rt[i].from[j]=j;
  }     }
do{
count=0;
     for(i=1;i<=nodes;i++)
     for(j=1;j<=nodes;j++)
     for(k=1;k<=nodes;k++)
     if(rt[i].dist[j]>costmat[i][k]+rt[k].dist[j]){
             rt[i].dist[j]=rt[i].dist[k]+rt[k].dist[j];
                 rt[i].from[j]=k;
                   count++; }
    }
   while(count!=0);
   for(i=1;i<=nodes;i++){
```

```
        printf("\n\n For router %d\n",i);
        for(j=1;j<=nodes;j++) {
             printf("\t\nnode %d via %d Distance %d ",j,rt[i].from[j],rt[i].dist[j]);
          }     }
 printf("\n\n");
}
```

OUTPUT:

5. Take an example subnet of hosts. Obtain broadcast tree for it.

AIM: Take an example subnet of hosts. Obtain broadcast tree for it.

THEORY:

In computer networks, a broadcast tree is a spanning tree that allows for the efficient dissemination of information to all nodes in a network. This tree structure ensures that each node receives the broadcast message exactly once, preventing redundancy and minimizing the use of network resources. The code provided constructs a broadcast tree for a given network by first creating a minimal spanning tree (sink tree) using the adjacency matrix representation of the network. It then designates a root node and propagates the broadcast message to all connected nodes while ensuring that each node is visited only once. The broadcast tree is characterized by marking each node as either a circle node (2) or an uncircled node (1), indicating its inclusion in the minimal spanning tree and its state in the broadcast process, respectively. This approach is essential in optimizing network communication, especially in large-scale networks, by reducing the likelihood of message collisions and ensuring efficient resource utilization.

PROGRAM:

```
#include <stdio.h>
#define MAX 30
int a[MAX][MAX], n, marked[MAX], k = -1, edges = 0, sink[MAX][MAX];
int visited(int);
void broadcast();
int main() {
int i, j, cv, start = 0, end = 0, distance = 0, min;
printf("Enter No. of vertices: ");
scanf("%d", &n);
printf("\nEnter adjacency matrix\n");
for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) {
printf("\na[%d][%d]=", i, j);
scanf("%d", &a[i][j]);
}
}
for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) {
sink[i][j] = 0;
}
}
while (edges < n - 1) {
if (edges == 0) {
for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) {
if (a[i][j] == 1) {
start = i; end = j;
break;
}
}
if (j < n) break;
}
```

```
marked[++k] = start;
marked[++k] = end;
} else {
min = MAX;
for (i = 0; i <= k; i++) {
cv = marked[i];
for (j = 0; j < n; j++) {
if (a[cv][j] == 1 && !visited(j)) {
start = cv;
end = j;
break;
}
}
if (j < n)
break;
}
marked[++k] = end;
distance += min;
}
sink[start][end] = 1;
sink[end][start] = 1;
edges++;
}
printf("\n\nThe sink tree is\n");
for (i = 0; i < n; i++) {
for (j = 0; j < n; j++)
printf("%3d", sink[i][j]);
printf("\n");
}
broadcast();
return 0;
} int visited(int x) {
int i;
for (i = 0; i <= k; i++)
if (marked[i] == x)
return 1;
return 0;
}
void broadcast() {
int i, root, broadcast[MAX][MAX], front = 0, rear = 0, j, b[30];
printf("Enter the root node: ");
scanf("%d", &root);
for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) {
broadcast[i][j] = 0;
}
}
for (i = 0; i < n; i++) {
if (a[root][i] == 1 && sink[root][i] == 1) {
b[rear++] = i;
```

```
broadcast[root][i] = 2;
} else if (a[root][i] == 1 && sink[root][i] == 0) {
broadcast[root][i] = 1;
} else {
broadcast[root][i] = 0;
}
}

while (front < rear) {
root = b[front];
front++;
for (i = 0; i < n; i++) {
if (a[root][i] == 1 && sink[root][i] == 1 && broadcast[i][root] != 2) {
b[rear++] = i;
broadcast[root][i] = 2;
} else if (a[root][i] == 1 && sink[root][i] == 0) {
broadcast[root][i] = 1;
} else {
broadcast[root][i] = 0;
}
}
}

printf("\n\nThe broadcast tree is (2->CircleNode, 1->Uncircled Node)\n\n");
for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) {
printf("%3d", broadcast[i][j]);
}
printf("\n");
}
}
```

OUTPUT:

PART-B

1. Implement the following forms of IPC.

AIM:

a) Pipes One way communication in one process

THEORY :

A pipe is a FIFO (first in, first out) queue or buffer with two ends, one for reading and one for writing. Pipes are efficient and reliable, but they have some limitations:

- Limited capacity: Pipes can only hold a small amount of data before their memory fills up.
- Unidirectional: In a unidirectional pipe, only one process can send data at a time.
- Lack of error checking: Pipes may not have error-checking mechanisms, which could lead to corrupted messages.

PROGRAM :

```
#include<stdio.h>

 #include<stdlib.h>

 main() {

int  pipefd[2],n;

 char buff[100];

 pipe(pipefd);

  printf("\n readfd= %d", pipefd[0]);

 printf("\n writefd= %d", pipefd[1]);

 write(pipefd[1],"Hello World",12);

 n=read(pipefd[0],buff, sizeof(buff));

 printf("\n Size of the data %d",n);

 printf("\n data from pipe: %s",buff);

}
```

OUTPUT :

One way communication in between two process :

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

main() {

int pipefd[2],n,pid;

char buff[100];

pipe(pipefd);

printf("\n readfd=%d",pipefd[0]); printf("\n
writefd=%d",pipefd[1]); pid=fork();

 if(pid==0) {

close(pipefd[0]);

printf("\n child proceesing sending data");
write(pipefd[1],"helloworld",12);

}else {

close(pipefd[1]);

printf("parent process receives data\n");
n=read(pipefd[0],buff,sizeof(buff));

 printf("\n size of the data %d",n);

 printf("\n data received from child through pipe:%s \n",buff);

}

}
```

OUTPUT:

Two way communication in between two process

```
#include<stdio.h>
#include<stdlib.h>
main() {
int p1[2],p2[2],n,pid;
char buff1[25],buff2[25];
pipe(p1);
pipe(p2);
printf("\n readfds=%d%d \n",p1[0],p2[0]);
printf("\n writefds=%d%d \n",p1[1],p2[1]);
pid=fork();
if(pid==0) {
close(p1[0]);
printf("\n child processing sendin data\n");
write(p1[1],"adity pg college",25);  close(p2[1]);
read(p2[0],buff1,25);
printf("reply from parent:%s \n",buff1);
sleep(2);
} else {
close(p1[1]);
printf("\n parent process receives data \n");
n=read(p1[0],buff2,sizeof(buff2));
printf("\n data received from child through pipe:%s \n",buff2);
sleep(3);
close(p2[0]);
write(p2[1],"in kakinada",25);  printf("\n reply
send \n");
}
```

}

OUTPUT

1. Implement the following forms of IPC.

AIM:

b) FIFO

THEORY:

A FIFO is also known as a named pipe, which is similar to a pipe but has a file name that multiple processes can open, read, and write to. This allows processes to get around one of the limitations of normal pipes, which is that you can't get one end of a pipe created by an unrelated process. To create a FIFO in C, you can use the mkfifo() function. Once created, any process can open the FIFO for reading or writing, but both ends must be open simultaneously before you can perform input or output operations.

PROGRAM:

```
#include<stdio.h>

#include<ctype.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

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

main() {

int wrfd,rdfd,n,d,ret_val,count; char buf[50];
ret_val=mkfifo("np1",0666);
ret_val=mkfifo("np2",0666);
rdfd=open("np1",O_RDONLY);
wrfd=open("np2",O_WRONLY);

n=read(rdfd,buf,50);

 buf[n]='\0';

printf("full duplex server:read from the pipe:%s\n",buf);

count=0;
```

```
while(count<n)

{

buf[count]=toupper(buf[count]);

 count++;

 } write(wrfd,buf,strlen(buf));

}
```

OUTPUT:

Client Program

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<stdlib.h>

#include<string.h>
#include<ctype.h>

main() {

int wrfd,rdfd,n; char
buf[50],line[50];

wrfd=open("np1",O_WRONLY);
rdfd=open("np2",O_RDONLY);

printf("enter line of text"); write(wrfd,line,strlen(line));

 n=read(rdfd,buf,50); buf[n]='\0';

printf("full duplex client:read from the pipe:%s\n",buf);

}
```

OUTPUT:

2. Implement file transfer using Message Queue form of IPC.

AIM: Implement file transfer using Message Queue form of IPC.

THEORY:

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. New messages are added to the end of a queue by msgsnd(). Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by msgrcv(). We don't have to fetch the messages in a first-in, first-out order. All processes can exchange information through access to a common system message queue. The sending process places a message (via some (OS) message-passing module) onto a queue which can be read by another process. Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place.

Server Program

```
#include<stdio.h>

#include<ctype.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

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

 main() {

 int wrfd,rdfd,n,d,ret_val,count; char
buf[50]; ret_val=mkfifo("np1",0666);
ret_val=mkfifo("np2",0666);
rdfd=open("np1",O_RDONLY);
wrfd=open("np2",O_WRONLY);

n=read(rdfd,buf,50);

 buf[n]='\0';
```

```
printf("full duplex server:read from the pipe:%s\n",buf);

count=0;

while(count<n) {

buf[count]=toupper(buf[count]);

count++;

} write(wrfd,buf,strlen(buf));

}
```

OUTPUT:

Client Program

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<stdlib.h>

#include<string.h>
#include<ctype.h>

main() {

int wrfd,rdfd,n; char
buf[50],line[50];

wrfd=open("np1",O_WRONLY);
rdfd=open("np2",O_RDONLY);

printf("enter line of text"); write(wrfd,line,strlen(line));

n=read(rdfd,buf,50);

buf[n]='\0';

 printf("full duplex client:read from the pipe:%s\n",buf);

}
```

OUTPUT:

3. **Write a program to producer and consumer AIM:** Write a program to producer and consumer

THEORY:

The producer-consumer problem is an example of multi-process syncronization problem. The problem describes two processes, the producer and the consumer that share a common fixed-size buffer and use it as a queue.The producer's job is to generate data, put it into the buffer, and start again.At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

PROGRAM:

```
#include  <stdio.h>
#include <stdlib.h>
int mutex = 1;
int full = 0;
int empty = 10, x = 0;
void producer()
{
   --mutex;
   ++full;
   --empty;
   x++;
   printf("\nProducer produces item %d", x);
   ++mutex;
}
void consumer(){
--mutex;
```

```
    --full;
    ++empty;
    printf("\nConsumer consumes " "item %d",x);
    x--;
    ++mutex;
}
int main()
{
    int n, i;
    printf("\n1. Press 1 for Producer"
           "\n2. Press 2 for Consumer"
           "\n3. Press 3 for Exit");
    for (i = 1; i > 0; i++) {
        printf("\nEnter your choice:");
        scanf("%d", &n);
        switch (n) {
        case 1:
            if ((mutex == 1)
                && (empty != 0)) {
                producer();
            }
            else {
                printf("Buffer is full!");
            }
            break;
```

```
        case 2:
            if ((mutex == 1)
                && (full != 0)) {
                consumer();
            }
            else {
                printf("Buffer is empty!");
            }
            break;
        case 3:
            exit(0);
            break;
        }
    }
}
```

OUTPUT:

4. Write a Program to create an integer variable using Shared Memory concept and increment the variable simultaneously by two processes. Use Semaphores to avoid Race conditions.

AIM: Write a Program to create an integer variable using Shared Memory concept and increment the variable simultaneously by two processes. Use Semaphores to avoid Race conditions.

THEORY:

Shared memory in computer networks refers to a memory that can be accessed by multiple processes or devices for communication and data exchange. By creating an integer variable in shared memory, multiple networked processes can increment this variable, ensuring that all processes have access to the latest value. Synchronization mechanisms, such as locks, are used to prevent race conditions, ensuring that only one process can modify the shared variable at a time, thus maintaining data consistency and integrity across the network.

PROGRAM:

```
#include<sys/stat.h>

#include<stdio.h>

#include<sys/types.h>

#include<sys/shm.h>

#include<sys/ipc.h>

#include<sys/sem.h>

#include<string.h>

#define SIZE 10 int *integer=0;

main() {

int shmid;

key_t key_10;

char *shm;

 int semid,pid;
shmid=shmget((key_t)10,SIZE,IPC_CREAT|0666);

 shm=shmat(shmid,NULL,0);
```

```
semid = semget(0X20,1,IPC_CREAT|0666);
integer=(int *)shm;

pid=fork();

if(pid==0) {

int i=0;

while(i<10) {

sleep(2);

printf("\n child process use shared memory");

 accessmem(semid);

 i++; } }

 else {

int j=0;

 while(j<10) {

sleep(j);

printf("\n parent versus shared memory");

 accessmem(semid);

 j++;

}

} shmctl(semid,IPC_RMID,0);

}

int accessmem(int semid) { struct
sembuf sop; sop.sem_num=0;

sop.sem_op=-1; sop.sem_flg=0;
semop(semid,&sop,1); (*integer)++;

printf("\t integer variable=%d",(*integer));

sop.sem_num=0;
```

```
sop.sem_op=1; sop.sem_flg=0;
semop(semid,&sop,1);

}
```

OUTPUT:

5. Design TCP iterative Client and Server application to reverse the given input sentence.

AIM: Design TCP iterative Client and Server application to reverse the given input sentence.

THEORY:

In a TCP iterative client-server application designed to reverse a given input sentence, the server listens on a specified port, accepting and handling one client connection at a time. When a client connects, it sends a sentence to the server, which reads the input, reverses the sentence, and sends the reversed sentence back to the client before closing the connection. The client, upon connecting to the server, sends its sentence, waits for the reversed response, displays it to the user, and then disconnects. This simple yet effective communication model ensures reliable, ordered, and error-checked data exchange, though it is limited in scalability as it handles one client at a time.

PROGRAM:

Server Program

```
#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<sys/types.h>

#define MAXLINE 20 #define
SERV_PORT 5777 main(int
argc,char *argv) {

    int i,j;

   ssize_t n;

char line[MAXLINE],revline[MAXLINE];

int listenfd,connfd,clilen;
```

```
 struct sockaddr_in servaddr,cliaddr;
listenfd=socket(AF_INET,SOCK_STREAM,0);
bzero(&servaddr,sizeof(servaddr));
servaddr.sin_family=AF_INET;
servaddr.sin_port=htons(SERV_PORT); bind(listenfd,(struct
sockaddr*)&servaddr,sizeof(servaddr)); listen(listenfd,1); for( ;
; ) { clilen=sizeof(cliaddr); connfd=accept(listenfd,(struct
sockaddr*)&cliaddr,&clilen); printf("connect to client");
while(1) {

if((n=read(connfd,line,MAXLINE))==0) break;

line[n-1]='\0'; j=0;

 for(i=n-2;i>=0;i--)
revline[j++]=line[i];
revline[j]='\0';
write(connfd,revline,n);

}

}

}
```

OUTPUT:

Client Program

```
#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<sys/types.h>

#define MAXLINE 20 #define
SERV_PORT 5777 main(int
argc,char *argv)

{

char sendline[MAXLINE],revline[MAXLINE];

 int sockfd;

 struct sockaddr_in servaddr;
sockfd=socket(AF_INET,SOCK_STREAM,0);
bzero(&servaddr,sizeof(servaddr));
servaddr.sin_family=AF_INET;
servaddr.sin_port=ntohs(SERV_PORT); connect(sockfd,(struct
sockaddr*)&servaddr,sizeof(servaddr));

printf("\n enter the data to be send");
while(fgets(sendline,MAXLINE,stdin)!=NULL)

{ write(sockfd,sendline,strlen(sendline));

printf("\n line send"); read(sockfd,revline,MAXLINE); printf("\n
reverse of the given sentence is : %s",revline);
```

```
printf("\n");

}

exit(0);

}
```

OUTPUT:

6. Design TCP client and server application to transfer file

AIM: Design TCP client and server application to transfer file

THEORY:

A TCP client-server application designed to transfer files involves the server listening on a specific port, ready to accept incoming connections from clients. When a client connects, it specifies the file it wants to send or receive. For file sending, the client reads the file from its local storage, sends the file data over the TCP connection to the server, which writes the received data to its local storage. For file receiving, the server reads the specified file from its local storage and sends the file data over the TCP connection to the client, which writes the received data to its local storage. This process ensures reliable and ordered file transfer, leveraging TCP's capabilities for error-checking and data integrity, and is typically used for applications requiring robust file exchange between remote systems.

PROGRAM:

Server program

```
#include<stdio.h>

#include<unistd.h>

#include<string.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<sys/types.h>

#define SERV_PORT 5576 main(int
argc,char **argv) {

int i,j;  ssize_t n;

FILE *fp; char s[80],f[80]; struct
sockaddr_in servaddr,cliaddr; int
listenfd,connfd,clilen;

listenfd=socket(AF_INET,SOCK_STREAM,0);
bzero(&servaddr,sizeof(servaddr));
servaddr.sin_family=AF_INET;
```

```
servaddr.sin_port=htons(SERV_PORT); bind(listenfd,(struct
sockaddr *)&servaddr,sizeof(servaddr)); listen(listenfd,1);
clilen=sizeof(cliaddr); connfd=accept(listenfd,(struct
sockaddr*)&cliaddr,&clilen); printf("\n clinet connected");
read(connfd,f,80); fp=fopen(f,"r");

printf("\n name of the file: %s",f); while(fgets(s,80,fp)!=NULL)

{ printf("%s",s); write(connfd,s,sizeof(s));

} }
```

OUTPUT:

Client Program

```
#include<stdio.h>

#include<unistd.h>

#include<string.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<sys/types.h>

#define SERV_PORT 5576 main(int
argc,char **argv)

{

int i,j; ssize_t n;

char  filename[80],recvline[80];
struct sockaddr_in servaddr; int
sockfd;

sockfd=socket(AF_INET,SOCK_STREAM,0);
bzero(&servaddr,sizeof(servaddr));
servaddr.sin_family=AF_INET;
servaddr.sin_port=htons(SERV_PORT);
inet_pton(AF_INET,argv[1],&servaddr.sin_addr);
connect(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
printf("enter the file name"); scanf("%s",filename);

write(sockfd,filename,sizeof(filename)); printf("\n
data from server: \n");
while(read(sockfd,recvline,80)!=0)

{ fputs(recvline,stdout);

}
```

```
}
```

OUTPUT:

7. Design a TCP concurrent server to convert a given text into upper case using multiplexing system call "select"

AIM: Design a TCP concurrent server to convert a given text into upper case using multiplexing system call "select"

THEORY:

A TCP concurrent server can be designed to convert text to uppercase using the `select` system call for multiplexing. The server listens on a specified port and uses `select` to monitor multiple client connections concurrently. When a client connects, the server adds the client socket to a list of monitored sockets. When data is received from any client, the server converts the text to uppercase and sends it back to the client. This non-blocking approach allows the server to handle multiple clients efficiently by processing whichever sockets are ready for reading, writing, or have exceptions, without dedicating a separate thread or process to each client.

PROGRAM:

Server Program

```
#include<stdio.h>

#include<netinet/in.h>

#include<sys/types.h>

#include<string.h>

#include<stdlib.h>

#include<sys/socket.h>

#include<sys/select.h>

#include<unistd.h>

#define MAXLINE 20  #define
SERV_PORT 7134  main(int
argc,char **argv)

{  int i,j,maxi,maxfd,listenfd,connfd,sockfd;
int nread,client[FD_SETSIZE];  ssize_t n;
fd_set rset,allset;
```

```
char line[MAXLINE]; socklen_t clilen; struct sockaddr_in
cliaddr,servaddr;
listenfd=socket(AF_INET,SOCK_STREAM,0);
bzero(&servaddr,sizeof(servaddr));
servaddr.sin_family=AF_INET;
servaddr.sin_port=htons(SERV_PORT); bind(listenfd,(struct
sockaddr *)&servaddr,sizeof(servaddr)); listen(listenfd,1);
maxfd=listenfd; maxi=-1; for(i=0;i<FD_SETSIZE;i++)

client[i]=-1;

FD_ZERO(&allset);
FD_SET(listenfd,&allset);
for(; ;) { rset=allset;

nread=select(maxfd+1,&rset,NULL,NULL,NULL);
if(FD_ISSET(listenfd,&rset))

{ clilen=sizeof(cliaddr); connfd=accept(listenfd,(struct
sockaddr*)&cliaddr,&clilen); for(i=0;i<FD_SETSIZE;i++)

if(client[i]<0)

{

client[i]=connfd; break;

}

if(i==FD_SETSIZE)

{ printf("too many clients");
exit(0);

}

FD_SET(connfd,&allset);
if(connfd>maxfd)      maxfd=connfd;
```

```
if(i>maxi)  maxi=i;     if(--nread<=0)
continue;

}  for(i=0;i<=maxi;i++)

{  if((sockfd=client[i])<0) continue;
if(FD_ISSET(sockfd,&rset))

{

if((n=read(sockfd,line,MAXLINE))==0)

{  close(sockfd);  FD_CLR(sockfd,&allset);
client[i]=-1;  }  else  {  printf("line recieved from
the client :%s\n",line);  for(j=0;line[j]!='\0';j++)
line[j]=toupper(line[j]);
write(sockfd,line,MAXLINE);

}  if(--nread<=0) break;

}

 }

 }

 }
```

OUTPUT:

Client Program

```
#include<netinet/in.h>

#include<sys/types.h>

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<sys/socket.h>

#include<sys/select.h>  #include<unistd.h>

#define MAXLINE 20  #define
SERV_PORT 7134  main(int
argc,char **argv)

{  int maxfdp1;  fd_set
rset;

char sendline[MAXLINE],recvline[MAXLINE];

int sockfd;  struct sockaddr_in
servaddr;  if(argc!=2)  {
printf("usage tcpcli <ipaddress>");
return;

 }

 sockfd=socket(AF_INET,SOCK_STREAM,0);
bzero(&servaddr,sizeof(servaddr)); servaddr.sin_family=AF_INET;
servaddr.sin_port=htons(SERV_PORT);
inet_pton(AF_INET,argv[1],&servaddr.sin_addr);
connect(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));

 printf("\n enter data to be send");
while(fgets(sendline,MAXLINE,stdin)!=NULL)
```

```
{

write(sockfd,sendline,MAXLINE);

printf("\n line send to server is %s",sendline);
read(sockfd,recvline,MAXLINE);

 printf("line recieved from the server %s",recvline);

 }  exit(0);

 }
```

OUTPUT:

9. Design UDP Client and server application to reverse the given input sentence

AIM: Design UDP Client and server application to reverse the given input sentence

THEORY:

A UDP client-server application for reversing a given input involves the server listening on a specific port and waiting for incoming datagrams from clients. The client sends a sentence to the server, which then processes this sentence by reversing it and sends the reversed sentence back to the client. The client, upon receiving the reversed sentence, displays it to the user. Unlike TCP, UDP is connectionless and does not guarantee delivery or order, making this approach suitable for simple, fast applications where occasional data loss is acceptable. This design leverages UDP's low overhead for efficient, lightweight communication, ideal for quick, non-critical data exchanges.

PROGRAM:

Server Program

```
#include<stdio.h>

#include<unistd.h>

#include<string.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<sys/types.h>

#include<stdlib.h>

#define SERV_PORT 5839

#define MAXLINE 20 main(int argc,char **argv)

{ int i,j; ssize_t n;

char line[MAXLINE],recvline[MAXLINE];  struct sockaddr_in servaddr,cliaddr;  int
sockfd,clilen; sockfd=socket(AF_INET,SOCK_DGRAM,0);
bzero(&servaddr,sizeof(servaddr));  servaddr.sin_family=AF_INET;
```

```
servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
servaddr.sin_port=htons(SERV_PORT); bind(sockfd,(struct
sockaddr*)&servaddr,sizeof(servaddr));  for( ; ; ) {

clilen=sizeof(cliaddr); while(1) { if((n=recvfrom(sockfd,line,MAXLINE,0,(struct
sockaddr*)&cliaddr,&clilen))==0) break; printf("\n line received successfully");
line[n-1]='\0'; j=0; for(i=n-2;i>=0;i--){ recvline[j++]=line[i];

} recvline[j]='\0'; sendto(sockfd,recvline,n,0,(struct
sockaddr*)&cliaddr,clilen);

}

}

}
```

OUTPUT:

Client Program

```
#include<stdio.h>

#include<unistd.h>

#include<string.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<sys/types.h>

#include<stdlib.h>

#define SERV_PORT 5839

#define MAXLINE 20 main(int argc,char **argv)

{ ssize_t n; struct sockaddr_in servaddr; char
sendline[MAXLINE],recvline[MAXLINE]; int
sockfd; if(argc!=2)

{

printf("usage:<IPADDRESS>");

exit(0); } bzero(&servaddr,sizeof(servaddr));
servaddr.sin_family=AF_INET;
servaddr.sin_port=htons(SERV_PORT);
inet_pton(AF_INET,argv[1],&servaddr.sin_addr);
sockfd=socket(AF_INET,SOCK_DGRAM,0);

printf("enter the data to be send");
while(fgets(sendline,MAXLINE,stdin)!=NULL)

{ sendto(sockfd,sendline,strlen(sendline),0,(struct
sockaddr*)&servaddr,sizeof(servaddr)); printf("line sent");
n=recvfrom(sockfd,recvline,MAXLINE,0,NULL,NULL);
```

```
recvline[n]='\0'; fputs(recvline,stdout); printf("\n reverse of the
sentense is %s",recvline); printf("\n");

} exit(0);

}
```

OUTPUT:

10. Design UDP Client Server to transfer a file.

AIM: Design UDP Client Server to transfer a file.

THEORY:

A UDP client-server application for file transfer involves the server listening on a specific port and waiting for incoming datagrams from clients. The client reads the file in chunks and sends these chunks as UDP packets to the server. The server receives these packets, reconstructs the file from the received chunks, and writes it to its local storage. Due to UDP's connectionless nature and lack of guaranteed delivery, the application must implement mechanisms for packet ordering and retransmission to ensure complete and correct file transfer, making it suitable for scenarios where speed is prioritized over reliability.

PROGRAM:

Server Program

```
#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<sys/socket.h>

#include<sys/types.h>

#include<netinet/in.h>

#define SERV_PORT 6349 main(int argc,char **argv)

{ char filename[80],recvline[80]; FILE *fp; struct
sockaddr_in servaddr,cliaddr;

int clilen,sockfd;

sockfd=socket(AF_INET,SOCK_DGRAM,0); bzero(&servaddr,sizeof(servaddr));
servaddr.sin_family=AF_INET; servaddr.sin_port=htons(SERV_PORT);

bind(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr)); clilen=sizeof(cliaddr);

recvfrom(sockfd,filename,80,0,(struct sockaddr*)&cliaddr,&clilen); printf("\n
date in the file is \n "); fp=fopen(filename,"r");

while(fgets(recvline,80,fp)!=NULL)

{
```

```
printf("\n %s\n ",recvline);

}fclose(fp);

}
```

OUTPUT:

Client Program

```
#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<unistd.h>

#define SERV_PORT 6349 main(int argc,char **argv)

{ char filename[80]; int sockfd; struct
sockaddr_in servaddr;

sockfd=socket(AF_INET,SOCK_DGRAM,0); bzero(&servaddr,sizeof(servaddr));
servaddr.sin_family=AF_INET; servaddr.sin_port=htons(SERV_PORT);
inet_pton(AF_INET,argv[1],&servaddr.sin_addr);

printf("enter the file name"); scanf("%s",filename);

sendto(sockfd,filename,strlen(filename),0,(structsockaddr*)&servaddr,sizeof (servaddr))

}
```

OUTPUT:

11. Design using poll client server application to multiplex TCP and UDP requests for converting a given text into upper case.

AIM: Design using poll client server application to multiplex TCP and UDP requests for converting a given text into upper case.

THEORY:

A client-server application can use the `poll` system call to multiplex between TCP and UDP requests for converting text to uppercase. The server listens on separate ports for TCP and UDP connections and uses `poll` to monitor these sockets for incoming data. When a TCP client connects, the server accepts the connection and handles data conversion by reading, converting the text to uppercase, and sending it back. For UDP clients, the server receives the data, converts it to uppercase, and sends the response back to the client. This approach allows the server to efficiently handle multiple TCP and UDP requests concurrently without blocking.

PROGRAM:

Server program:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>

#include <arpa/inet.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <poll.h>

#define PORT_TCP 12345

#define PORT_UDP 12346

#define MAX_CLIENTS 10

#define BUF_SIZE 1024
```

```
void convert_to_uppercase(char *str) {
    int i = 0;
    while (str[i]) {
        str[i] = toupper(str[i]);
        i++;
    }
}
int main() {
    int tcp_socket, udp_socket, max_clients = MAX_CLIENTS;
    struct sockaddr_in server_tcp, server_udp, client_tcp, client_udp;
    struct pollfd fds[2];
    char buffer[BUF_SIZE];
    int nfds = 2;
    int rc, len;
    if ((tcp_socket = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("TCP socket creation failed");
        exit(EXIT_FAILURE);
    }
    if ((udp_socket = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("UDP socket creation failed");
        exit(EXIT_FAILURE);
    }
    int opt = 1;
    if (setsockopt(tcp_socket, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt))
== -1) {
        perror("setsockopt");
```

```
        exit(EXIT_FAILURE);
    }
    if (setsockopt(udp_socket, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt))
== -1) {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }
    server_tcp.sin_family = AF_INET;
    server_tcp.sin_addr.s_addr = INADDR_ANY;
    server_tcp.sin_port = htons(PORT_TCP);
    server_udp.sin_family = AF_INET;
    server_udp.sin_addr.s_addr = INADDR_ANY;
    server_udp.sin_port = htons(PORT_UDP);
    if (bind(tcp_socket, (struct sockaddr *)&server_tcp, sizeof(server_tcp)) == -1) {
        perror("TCP bind failed");
        exit(EXIT_FAILURE);
    }
    if (bind(udp_socket, (struct sockaddr *)&server_udp, sizeof(server_udp)) == -1) {
        perror("UDP bind failed");
        exit(EXIT_FAILURE);
    }
printf("Server is listening on TCP port %d and UDP port %d...\n", PORT_TCP, PORT_UDP);
    if (listen(tcp_socket, max_clients) == -1) {
        perror("TCP listen failed");
        exit(EXIT_FAILURE); }
    fds[0].fd = tcp_socket;
```

```
    fds[0].events = POLLIN;
    fds[1].fd = udp_socket;
    fds[1].events = POLLIN;
    while (1) {
        rc = poll(fds, nfds, -1);
        if (rc == -1) {
            perror("poll error");
            continue;
        }
        if (fds[0].revents & POLLIN) {
            len = sizeof(client_tcp);
                int new_tcp_socket = accept(tcp_socket, (struct sockaddr *)&client_tcp,  (socklen_t *)&len);
            if (new_tcp_socket == -1) {
                perror("TCP accept failed");
                continue;
            }
          printf("New TCP connection from %s:%d\n", inet_ntoa(client_tcp.sin_addr), ntohs(client_tcp.sin_port));
            rc = recv(new_tcp_socket, buffer, sizeof(buffer), 0);
            if (rc <= 0) {
                perror("TCP recv failed");
                close(new_tcp_socket);
                continue;
            }
         printf("Received TCP message: %s\n", buffer);
            convert_to_uppercase(buffer);
```

```
            rc = send(new_tcp_socket, buffer, strlen(buffer), 0);
            if (rc != strlen(buffer)) {
                perror("TCP send failed");
                close(new_tcp_socket);
                continue;
            }
            printf("Sent TCP response: %s\n", buffer);
            close(new_tcp_socket);
        }
        if (fds[1].revents & POLLIN) {
            len = sizeof(client_udp);
            rc = recvfrom(udp_socket, buffer, sizeof(buffer), 0, (struct sockaddr *)&client_udp, (socklen_t *)&len);
            if (rc == -1) {
                perror("UDP recvfrom failed");
                continue;
            }
            printf("Received UDP message from %s:%d: %s\n", inet_ntoa(client_udp.sin_addr), ntohs(client_udp.sin_port), buffer);
            convert_to_uppercase(buffer);
            rc = sendto(udp_socket, buffer, strlen(buffer), 0, (struct sockaddr *)&client_udp, len);
            if (rc == -1) {
                perror("UDP sendto failed");
                continue;
            }
            printf("Sent UDP response: %s\n", buffer);
```

```
        }
    }
    close(tcp_socket);
    close(udp_socket);
    return 0;
}
```

OUTPUT:

Client program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#define PORT_TCP 12345
#define BUF_SIZE 1024
int main() {
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char buffer[BUF_SIZE] = {0};
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("TCP socket creation failed");
        exit(EXIT_FAILURE);
    }
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT_TCP);
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        perror("Invalid address/ Address not supported");
        exit(EXIT_FAILURE);
    }
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        perror("Connection failed");
```

```
        exit(EXIT_FAILURE);
    }
    printf("Enter text to convert to uppercase: ");
    fgets(buffer, BUF_SIZE, stdin);
    buffer[strcspn(buffer, "\n")] = 0;
    send(sock, buffer, strlen(buffer), 0);
    printf("Message sent to server.\n");
    valread = read(sock, buffer, BUF_SIZE);
    printf("Uppercase message received from server: %s\n", buffer);
    close(sock);
    return 0;
}
```

OUTPUT:

12. Design a RPC application to add and subtract a given pair of integers.

AIM: Design a RPC application to add and subtract a given pair of integers.

THEORY:

RPC (Remote Procedure Call) enables invoking functions or procedures on a remote server as if they were local, abstracting network complexities. In the context of designing an RPC application for arithmetic operations like addition and subtraction, a defined interface (arithmetic.x) specifies procedures (ADD and SUB) with parameters and return types. RPCGEN generates client and server code (arithmetic_client.c and arithmetic_server.c) from this interface. The server implements these procedures to perform arithmetic operations based on client requests, while the client uses generated stubs to invoke these procedures remotely, handling communication transparently.

PROGRAM:

Server program:

```
#include "arithmetic.h"

#include <stdio.h>

#include <stdlib.h>

#include <rpc/rpc.h>

int *add_1_svc(intpair *arg, struct svc_req *rqstp) {

    static int result;

    printf("Received ADD request: %d + %d\n", arg->a, arg->b);

    result = arg->a + arg->b;

    return &result;

}

int *sub_1_svc(intpair *arg, struct svc_req *rqstp) {

    static int result;

    printf("Received SUB request: %d - %d\n", arg->a, arg->b);

    result = arg->a - arg->b;

    return &result;

}
```

```
int main(int argc, char **argv) {
    register SVCXPRT *transp;
    pmap_unset(ARITHMETIC_PROG, ARITHMETIC_VERS);
    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
        fprintf(stderr, "Cannot create UDP service.\n");
        exit(1);
    }
    if(!svc_register(transp, ARITHMETIC_PROG, ARITHMETIC_VERS, arithmetic_prog_1, IPPROTO_UDP)) {
        fprintf(stderr, "Unable to register service.\n");
        exit(1);
    }
    svc_run();
    fprintf(stderr, "Error: SVC_RUN returned\n");
    exit(1);
}
```

OUTPUT:

Client program:

```
#include "arithmetic.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/rpc.h>
int main(int argc, char **argv) {
    CLIENT *cl;
    intpair input;
    int *result;
    if (argc != 5) {
        fprintf(stderr, "Usage: %s <hostname> <ADD|SUB> <int1> <int2>\n", argv[0]);
        exit(1);
    }
    cl = clnt_create(argv[1], ARITHMETIC_PROG, ARITHMETIC_VERS, "udp");
    if (cl == NULL) {
        clnt_pcreateerror(argv[1]);
        exit(1);
    }
    input.a = atoi(argv[3]);
    input.b = atoi(argv[4]);
    if (strcmp(argv[2], "ADD") == 0) {
        result = add_1(&input, cl);
        if (result == NULL) {
            clnt_perror(cl, "Addition failed");
            exit(1);
```

```
    }
        printf("Result of ADD operation: %d + %d = %d\n", input.a, input.b, *result);
    } else if (strcmp(argv[2], "SUB") == 0) {
        result = sub_1(&input, cl);
        if (result == NULL) {
            clnt_perror(cl, "Subtraction failed");
            exit(1);
        }
        printf("Result of SUB operation: %d - %d = %d\n", input.a, input.b, *result);
    } else {
        fprintf(stderr, "Invalid operation specified. Use ADD or SUB.\n");
        exit(1);
    }
    clnt_destroy(cl);
    return 0;
}
```

OUTPUT: