TABLE OF CONTENTS

ENVIRONMENT USED

GNU/Linux environment has been used here throughout for programming. GNU is a computer operating system composed entirely of free software. Its name is a recursive acronym for GNU's Not Unix; it was chosen because its design is Unix-like, but differs from Unix by being free software and containing no Unix code. A GNU System's basic components include the GNU Compiler Collection (GCC), the GNU Binary Utilities (binutils), the bash shell, the GNU C Library (glibc) and GNU Core Utilities (coreutils). GNU is in active development. Although nearly all components have been completed long ago and have been in production use for a decade or more, its official kernel, GNU Hurd, is incomplete and not all GNU components work with it. Thus, the third-party Linux kernel is most commonly used instead. The Linux Kernel was first released to the public on 17 September, 1991.

Linux distributions have mainly been used as server operating systems, and have risen to prominence in that area. Linux distributions are the cornerstone of the LAMP server-software combination (Linux, Apache, MySQL, Perl/PHP/Python) which has achieved popularity among developers, and which is one of the more common platforms for website hosting. There are many Linux Distributions available these days which consist of the Linux kernel and, usually, a set of libraries and utilities from the GNU project, with graphics support from the X Window System. One can distinguish between commercially backed distributions, such as Fedora (Red Hat), openSUSE (Novell), Ubuntu (Canonical Ltd.), and Mandriva Linux and community distributions such as Debian and Gentoo, though there are other distributions that are driven neither by a corporation nor a community; perhaps most famously, Slackware.

The user/programmer can control a Linux-based system through a command line interface (or CLI), a graphical user interface (or GUI), or through controls attached to the associated hardware (this is common for embedded systems). For desktop systems, the default mode is usually graphical user interface (or GUI). Most Linux distributions support dozens of programming languages. The most common collection of utilities for building both Linux applications and operating system

programs is found within the GNU toolchain, which includes the GNU Compiler Collection (GCC) and the GNU build system. Amongst others, GCC provides compilers for Ada, C, C++, Java, and Fortran. The Linux kernel itself is written to be compiled with GCC.

a) COMPILING USING GCC

While compiling using gcc, there are several options available. Generally, compiling a file is done by entering a command at the terminal in the following syntax:

gcc –o <output file> <input file>

To execute the compiled file:

./<output file>

What the option –o<output file> does:

Place output in file <output file>. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code. Since only one output file can be specified, it does not make sense to use -o when compiling more than one input file, unless you are producing an executable file as output. If -o is not specified, the default is to put an executable file in a.out. In this case the execution is done by specifying the command:

. /a.out

Another method of compiling is done by:

gcc -c <input file>

What the option –c does:

Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

To compile multiple files in gcc:

Write the commands for compiling the individual files in a shell script with .sh extension and run this script file.

All the files will get compiled.

For eg: to compile the files server.c, client.c, finger_clnt.c, finger_svc.c and finger_xdr.c specify the following commands inside a shell script file say, compile.sh.

```
gcc -c server.c -o ser
gcc -c client.c -o cl
gcc -c finger_clnt.c -o f_c
gcc -c finger_svc.c -o f_s
gcc -c finger_xdr.c -o f_x
gcc -o server ser f_s f_x
gcc -o client cl f_c f_x
```

To execute the compile.sh file, give the following command in command line

```
./compile.sh
(or)
sh compile.sh
```

b) EDITORS IN LINUX

Many editors are available inbuilt in Linux itself for the purpose of editing. One of those may be used for programming too. Some of the common editors are gedit, vi, vim, nano, kedit etc. All of them provide different options for editing. Here, we use vi editor for coding purpose. The commands used in vi are:

Creating/Opening an already existing file:

vi <filename>

Exiting vi:

After pressing Esc key,

:q<return> - quit vi

:wq<return> - save the current file and quit vi

There are mainly five modes in vi.

- Insert mode
- Command mode
- Replace mode

- Execute mode
- Visual mode

Command mode

To go to command mode, press Ctrl+c at any time. In command mode, some of the common operations and corresponding commands are:

- To move to the end of file : press G
- To move to beginning of file : press gg
- To move to a particular line : enter line number and then press G
- To copy a line of text : press yy at the start of the line
- To copy a group of lines : enter number of lines to be copied and then press yy

  Eg: 5yy
- To copy a word : press yw at the start of the word
- To cut a line : press cc at the start of the line
- To cut a group of lines : enter number of lines to be copied and then press cc

  Eg: 5cc
- To cut a word : press cw at the start of the word
- For paste operation : press p
- To delete a line: press dd at the start of the line
- To delete a group of lines : enter number of lines to be copied and then press dd

  Eg: 5dd
- To delete a word : press dw at the start of the word
- To search for a particular word : press / and then enter the word and press Enter key

- To go to next word – press n
- To go backwards – press N
- To undo last action press u
- To redo last action press Ctrl+r

Insert Mode

Normally when we open a file in vi, it will be in command mode. Now to enter text into the file, we need to be in Insert Mode. To switch from command to insert mode simply press Insert key or ‘i’. This mode is used to manually edit the file.

THREADS

Theory

Threads allows a program to execute multiple parts of itself simultaneously in the same address space. On a single processor, multithreading generally occurs by time division multiplexing : the processor switches between different threads. This generally happens frequently enough that the user perceives the threads or tasks as running context switching at the same time.

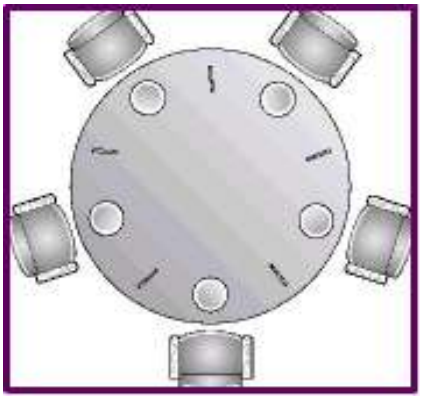
POSIX Threads, or Pthreads, is a POSIX standard for . The standard defines an API for creating and manipulating threads. Pthreads defines a set of programming C language types ,functions and constants. It is implemented with a header and a thread library. Programmers can use Pthreads to create, manipulate and manage threads, as well as between threads using mutexes.

Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.A mutex variable acts like a "lock" protecting access to a shared data resource. The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful. No other thread can own that mutex until the owning thread unlocks that mutex. Threads must "take turns" accessing protected data.

1. DINING PHILOSOPHERS PROBLEM USING PTHREAD

PROBLEM DESCRIPTION

The dining philosophers problem is summarized as five philosophers sitting at a table doing one of two things: eating or thinking. While eating, they are not thinking, and while thinking, they are not eating. The five philosophers sit at a circular table with a large bowl of spaghetti in the center. A fork is placed in between each pair of adjacent philosophers,



and as such, each philosopher has one fork to his left and one fork to his right. As spaghetti is difficult to serve and eat with a single fork, it is assumed that a philosopher must eat with two forks. The philosopher can only use the fork on his immediate left or right.

Note: while compiling this program use the following:

[root@Linux smtp]# gcc –o dining dining.c -lpthread

FUNCTIONS AND DATA STRUCTURES USED

pthread_mutex_t : describes a thread mutex

pthread_t : acts as a handle for a thread

pthread_create( ) : creates a new thread

Syntax :int pthread_create (pthread_t *thread_id, const pthread_attr_t *attributes,void *(*thread_function)(void *), void *arguments);

This function creates a new thread. pthread_t is an opaque type which acts as a handle for the new thread.attributes is another opaque data type which allows you to fine tune various parameters, to use the defaults pass NULL.

thread_function() :

syntax : void * thread_function(int i);

thread_function is the function the new thread is executing, the thread will terminate when this function terminates, or it is explicitly killed. arguments is a void * pointer which is passed as the only argument to the thread_function.

pthread_join()

syntax: int pthread_join(pthread_t *thread*, void ***value_ptr*);

The function suspends execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated.

pthread_mutex_init()

syntax: int pthread_mutex_init(pthread_mutex_t **mutex*,
const pthread_mutexattr_t **attr*).

The function initialises the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used

pthread_mutex_destroy()

syntax : int pthread_mutex_destroy(pthread_mutex_t **mutex*);

The function destroys the mutex object referenced by *mutex*; the mutex object becomes, in effect, uninitialised.

pthread_mutex_lock()

Syntax : int pthread_mutex_lock(pthread_mutex_t **mutex*);

The mutex object referenced by *mutex* is locked by calling *pthread_mutex_lock()*. If the mutex is already locked, the calling thread blocks until the mutex becomes available

pthread_mutex_unlock()

syntax : int pthread_mutex_unlock(pthread_mutex_t **mutex*);

The function releases the mutex object referenced by *mutex* .

pthread_exit()

Syntax : void pthread_exit(void **value_ptr*);

The function terminates the calling thread .

ALGORITHM

1. Create variable philosopher[5] of type pthread_t .
2. Create chopstick[5] of type pthread_mutex_t .
3. Create 5 threads corresponding to philosppher[5] using pthread_create() .
4. Each ith thread executes thread_function()
   4.1. chopstick[i] is locked indicating its the chopstick at the left.
   4.2. chopstick[(i+1)%5] is locked indicating its the chopstick at the right.
   4.3. Allot some time to eat.
   4.4.Now unlock the left chopstick followed by the right one
5. Stop

CODE

```
#include<pthread.h>
#include<stdio.h>
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
 pthread_mutex_t chopstick[5];
 pthread_t philosopher[5];
  void *msg;

  void * threadfunc(int i)
   {
     pthread_mutex_lock(&chopstick[i]);
     pthread_mutex_lock(&chopstick[(i+1)%5]);
     printf("\nphilosopher %d taken his chopsticksticks and started
eating...",i+1);
     sleep(5);
     printf("\nphilosopher %d finished eating..",i+1);
     pthread_mutex_unlock(&chopstick[i]);
     pthread_mutex_unlock(&chopstick[(i+1)%5]);
     pthread_exit(NULL);
  }
 int main()
  {
     int k,i=0;
```

```
     pthread_mutex_t chopstick[5];



    pthread_t philosopher[5];
     void *msg;
     for(i=0;i<5;i++)
     {
         k=pthread_mutex_init(&chopstick[i],NULL);
          if(k==-1)
            printf("\nerror");
     }
     for(i=0;i<5;i++)
        {

k=pthread_create(&philosopher[i],NULL,(void*)threadfunc,(int *)i);
          if(k==-1)
              printf("\nerror");
     }
     for(i=0;i<5;i++)
     {
         k=pthread_join(philosopher[i],&msg);
         if(k==-1)
             printf("\nerror....");
     }
     for(i=0;i<5;i++)
     {
         k=pthread_mutex_destroy(&chopstick[i]);
         if(k==-1)
             printf("\nerror....");
    }
   }
```

SAMPLE INPUT/OUTPUT

philosopher 1 taken his chopsticks and started eating...

philosopher 3 taken his chopsticks and started eating...

philosopher 1 finished eating...

philosopher 3 finished eating...

philosopher 2 taken his chopsticks and started eating...

philosopher 5 taken his chopsticks and started eating...

philosopher 2 finished eating...

philosopher 5 finished eating...

philosopher 4 taken his chopsticks and started eating...

philosopher 4 finished eating...

2. BANKER'S ALGORITHM

PROBLEM DESCRIPTION

The banker's algorithm is a resourc allocation and deadlock avoidace algoritm.Whenever a process requests for resources the algorithm tests for safety by simulating the allocation of pre determined maximum possible amounts of all resources and then makes a “safe-state”check to test for possible deadlock conditions for all other pending activities before deciding whether allocation should be allowed to continue.The algorithm prevents deadlock by denying the request if it determines that accepting the request could put the system in an unsafe state (one where deadlock could occur).

FUNCTIONS AND DATA STRUCTURES USED

available[m] : no.of resource instances of each of the m different resource types.

max[n][m] : maximum demand of n processes for each of the m resource types

allocation[n][m] : no.of resource instances of each of the m types currently allocated to each of the n processes.

need[n][m] : remaing resource need of each of the n processes.

need[n][m] = max[n][m] – allocation[n][m]

work[m] : To keep track of the currently available resources

finish[n] : to keep track of which processes been allocated all its requirements

ALGORITHM

1 . Start

2. Enter the values of max, allocation and available.
3. Compute the need matrix using need=max- allocation.
4. Initialsie the vector work with available.
5. Initialise the finish vector as false.
6. Find a process i such that finish[i] is false and need[i]<= work.If no such process

exists then goto step 9.

7. Set work->work+allocation[i]

   finish[i]->true

8. Repeat step 6.

9. If finish[i] is true for all n processes then the system is in safe state.Otherwise the system is in unsafe state.

CODE

```
#include<stdio.h>
int available[30],max[30][30],allocation[30][30],need[30][30];
int main()
{
              int j,flag=0,finish[2],work[30],i,m,n;
              printf("\nEnter the number of processes:");
              scanf("%d",&n);
              printf("\nEnter the number of resources:");
              scanf("%d",&m);
              printf("\nenter available vector for %d resource types
: ",m);
  for(i=0;i<m;i++)
  {
     scanf("%d",&available[i]);
  }
  printf("\n maximum deamnd of each resource type..");
  for(i=0;i<n;i++)
  {
    printf("\n process %d :",i+1);
    for(j=0;j<m;j++)
    {

       scanf("%d",&max[i][j]);
    }
  }
  printf("\n allocated resources for each process..");
  for(i=0;i<n;i++)
  {
    printf("\n process %d :",i+1);
    for(j=0;j<m;j++)
    {
       scanf("%d",&allocation[i][j]);
    }
  }
   for(i=0;i<n;i++)
  {
    for(j=0;j<m;j++)
    {
      need[i][j]=max[i][j]-allocation[i][j];

    }
```

```
  }
              for(i=0;i<n;i++)
  {
       for(j=0;j<m;j++)
        {
           need[i][j]=max[i][j]-allocation[i][j];
              }
        }
              printf("need matrix\n");

              for(i=0;i<n;i++)
        {     printf("\n");
               for(j=0;j<m;j++)
                {
                       printf("%d\t",need[i][j]);
                }
        }
for(i=0;i<m;i++)
              {
                work[i]=available[i];
                finish[i]=0;
              }
              for(i=0;i<n;i++)
              {
                for(j=0;j<m;j++)
                {
                      if((finish[i]==0) && (need[i][j]<= work[i]))
                      {
                            work[i]=work[i]+allocation[i][j];
                            finish[i]=1;
                      }
                      else
                      {
                      if(finish[i]==1)
                            {
                                  flag=1;
                            }
                      }
                }
              }
              if(flag==0)
              {
                printf("\nSystem is in unsafe state....\n");
              }
              else
              {
                printf("\nSystem is in Safe state\n");
              }
}
```

SAMPLE INPUT/OUTPUT

Enter the number of processes:5

Enter the number of resources :3

enter available vector for 3 resource types : 3 3 2

maximum demand of each resource type..

process 1 :7 5 3

process 2 :3 2 2

process 3 :9 0 2

process 4 :2 2 2

process 5 :4 3 3

allocated resources for each process..

process 1 :0 1 0

process 2 :2 0 0

process 3 :3 0 2

process 4 :2 1 1

process 5 :0 0 2

need matrix

7 4 3

1 2 2

6 0 0

0 1 1

4 3 1

System is in safe state....

INTERPROCESS COMMUNICATION

IPC (Inter-process communication) facilities provide a method for multiple processes to communicate with one another. There are several methods of IPC :

1. pipes
2. message queues
3. shared memory
4. sockets

Pipes

Pipes then are unidirectional byte streams which connect the standard output from one process into the standard input of another process. They provide a one-way communications between processes. When a process creates a pipe the kernel sets up two file descriptors for use by the pipe. One descriptor is used to allow a path of input into the pipe(write),while the other is used to obtain data from the pipe(read).

Message Queues

Message queues can be best described as an internal linked list within the kernel's addressing space. Messages can be sent to the queue in order and retrieved from the queue in several different ways. Each message queue (of course) is uniquely identified by an IPC identifier. Two (or more) processes can exchange information via access to a common system message queue. The *sending* process places via some (OS) message-passing module a message 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 .

Shared Memory

Shared memory can best be described as the mapping of an area (segment) of memory that will be mapped and shared by more than one process. This is by far the

fastest form of IPC, because there is no intermediation (i.e. a pipe, a message queue, etc). Instead, information is mapped directly from a memory segment, and into the addressing space of the calling process. A segment can be created by one process, and subsequently written to and read from by any number of processes.

Sockets

Sockets provide point-to-point, two-way communication between two processes. Sockets are very versatile and are a basic component of interprocess and intersystem communication. A socket is an endpoint of communication to which a name can be bound. It has a type and one or more associated processes. Sockets exist in communication domains. A socket domain is an abstraction that provides an addressing structure and a set of protocols. Sockets connect only with sockets in the same domain

FORKS

The fork() function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except as detailed below:

- The child process shall have a unique process ID.
- The child process ID also shall not match any active process group ID.
- The child process shall have a different parent process ID, which shall be the process ID of the calling process.
- The child process shall have its own copy of the parent's file descriptors. Each of the child's file descriptors shall refer to the same open file description with the corresponding file descriptor of the parent.
- The child process shall have its own copy of the parent's open directory streams. Each open directory stream in the child process may share directory stream positioning with the corresponding directory stream of the parent.
- pid=fork()
  child process has pid=0 and parent process gets the process id of child in its pid variable. Hence if pid>0 it is parent process.

After fork(), both the parent and the child processes shall be capable of executing independently before either one terminates.

3. EXPRESSION EVALUATION USING FORK & SHARED MEMORY

PROBLEM DESCRIPTION

Implement an expression evaluation (a*b)+(c*d),where a*b is evaluated in child process and c*d in parent process and result is computed in child process.

FUNCTIONS AND DATA STRUCTURES USED

pid_t: It is capable of representing a process ID.

key_t: Unix requires a key of type key_t defined in file sys/types.h for requesting resources such as shared memory segments, message queues and semaphores.

shmget():

Syntax:

shmid=shmget (key, size, IPC_CREAT|0666);

Used to obtain access to a shared memory segment and returns a positive integer as its unique identifier.

shmat():

Syntax:

shmat(int shmid, const void *shmaddr, int shmflg);

Returns a pointer, shmaddr, to the head of the shared segment associated with a valid shmid.

fork():

Syntax: pid_t id

id = fork();

The fork() function shall create a new process and unique identifier is returned. If id=0,child process is executed. If id>0,parent process is executed.

ALGORITHM

1. Declare a variable id of type pid_t and key of type key_t
2. fork() is called and an unique identifier will be returned to id.
3. If id =0, child process is executed.
4. Create a shared memory using shmget() and attach it using shmat()
5. Compute a*b.
6. If id>0,parent process is executed.
7. Create a shared memory using shmget() and attach it using shmat()
8. Compute c*d.
9. Computed value of c*d is obtained from memory and added it to a*b in child process.

CODE

```
#include<unistd.h>
#include <sys/shm.h>
#include <stdio.h>
#include<string.h>
#define MEMSIZE 27
main()
{
 pid_t id;
 key_t key;
 int sid1;
 int *shm,*s;
 int *s1,*sh;
 int a,b,c,d,f;
 int e,res=0;
 key=1432;
 id=fork();
  if(id>0)
    {
          sleep(4);
           printf("\n Enter c and d:");
           scanf("%d%d",&c,&d);
          e=c*d;
         sid1=shmget(key,MEMSIZE,IPC_CREAT|0666);
       if(sid1==-1)
        {
              printf("\n error creating");
               exit(1);
         }
         if((s1=shmat(sid1,NULL,0))==(int *)-1)
        {
```

```
       printf("\n error attaching");
               exit(1);

            }
              sh=s1;
             *sh=e;
               printf("\nThe value of (c*d) in memory :%d",*sh);
               sleep(2);
           }
            else if(id==0)

           {
               printf("\n Enter a and b:");
               scanf("%d%d",&a,&b);
               f=a*b;
               sleep(5);
                 if((sid1=shmget(key,MEMSIZE,0666))<0)
               {
                       printf("\n Error in creating");
                        exit(1);
                 }
                 if((shm=shmat(sid1,NULL,0))==(int *)-1)
                 {
                          printf("\nerror attaching");
                           exit(1);
                 }
                s=shm;
                res=f+(*s);
                printf("Result=%d",res);
        }
}
```

SAMPLE INPUT/OUTPUT

Enter a and b: 1 2

Enter c and d: 3 4

The value of (c*d) in memory:12

Result=14

4. IPC USING PIPES

PROBLEM DESCRIPTION

We are required to implement the working of a pipe that connects two processes.First process is needed to send data to the second process which reverses the data and send it back to the first process.This inter process communication is to be implemented by means of a pipe.

FUNCTIONS AND DATA STRUCTURES USED

access() :

syntax : int access(const char *pathname, int mode)

The function checks whether the calling process can access the file *pathname*.The *mode* specifies the accessibility check(s) to be performed, and is either the value F_OK, or a mask consisting of the bitwise OR of one or more of R_OK, W_OK, and X_OK. F_OK tests for the existence of the file.

open() :

syntax : int open(const char ***pathname*, int *flags*);

Given a *pathname* for a file, **open**() returns a file descriptor, a small, non-negative integer for use in subsequent system calls . The argument *flags* must include one of the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

read() :

syntax : read(int *fd***,** void ***buf***,** size_t *count*);

The function attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

write() :

syntax : read(int fd, void ***buf,** size_t count);

The function writes up to *count* bytes from the buffer pointed *buf* to the file referred to by the file descriptor *fd*.

ALGORITHM

Sender

1. Start
2. Declare writepipe and readpipe.
3. If writepipe is accessible, create pipe using mkfifo()
4. If readpipe is accessible, create pipe using mkfifo()
5. Open the writepipe in write-only mode
6. Open the readpipe in read-pipe mode
7. Write data into writepipe using write() function
8. Read data from readpipe using read() function
9. Stop

Client

1. Declare readpipe and writepipe in reverse as that of sender side
2. If readpipe is accessible, create pipe using mkfifo()
3. If writepipe is accessible, create pipe using mkfifo()
4. Open the readpipe in read-pipe mode
5. Open the writepipe in write-only mode
6. Read data from readpipe using read() function

7. Write data into writepipe using write() function

8. Stop

CODE

Pipes:Process a

```
#include<fcntl.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
int main()
{
       char readpipe[100]="pipe2";
       char writepipe[100]="pipe1";
             int flg1,flg2,rd,wr;
             char snd[25],rec[25],rev[25];
             int i,j=0;
             while(1)
             {
                    memset(snd,'\0',strlen(snd)+1);
             printf("\n enter data to be written..");
             gets(snd);
             if(access(writepipe,F_OK)==-1)
                              {
                     flg2=mkfifo(writepipe,0777);
                    if(flg2!=0)
                    {
                              printf("\nerror");
                         exit(0);
                     }
                   }
                   if(access(readpipe,F_OK)==-1)
            {
                            flg1=mkfifo(readpipe,0777);
                      if(flg1!=0)
                            {

printf("\n error..");
                                                                        exit(0);
                         }
              }
                    wr=open(writepipe,O_WRONLY);
                    if(wr==-1)
             {
                         printf("\nerror..");
                         exit(0);
                               }

                           rd=open(readpipe,O_RDONLY);
                          if(rd==-1)
                          {
                                       printf("\n error..");
                          }
                    write(wr,&snd,sizeof(snd));
                         if(strcmp(snd,"quit")==0)
                                      exit(0);
                          printf("\n reversed data from process b..");
```

```
                        read(rd,&rec,sizeof(rec));
                        printf("%s",rec);
                        printf("\n");
         }
         close(wr);
         close(rd);
        return(0);
}

Pipes:Process b
#include<fcntl.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
int main()
{
  int i,j=0;
  int flag=0;
  char readpipe[100]="pipe1";
  char writepipe[100]="pipe2";
  int flg1,flg2,r,w;
  char snd[25],rec[25],rev[25];
  while(1)
  {
         memset(rec,'\0',strlen(rec)+1);
         memset(rev,'\0',strlen(rev)+1);
         if(access(readpipe,F_OK)==-1)
        {
             flg1=mkfifo(readpipe,0777);
             if(flg1!=0)
             {
                  printf("\n error..");
                  exit(0);
             }
   }
  if(access(writepipe,F_OK)==-1)
  {
    flg2=mkfifo(writepipe,0777);
    if(flg2!=0)
   {
      printf("\nerror");
      exit(0);
    }
  }
   r=open(readpipe,O_RDONLY);
   if(r==-1)
   {
       printf("\n error..");exit(0);}
       w=open(writepipe,O_WRONLY);
       if(w==-1)
       {
           printf("error..");
           exit(0);
       }
       read(r,&rec,sizeof(rec));
       if(strcmp(rec,"quit")==0)
             exit(0);
        printf("\nreceived data from process a ...");
        j=0;
        for(j=0;j<strlen(rev)+1;j++)
                rev[j]='\0';
```

```
          j=0;
        for(i=strlen(rec)-1;i>=0;i--)
                      rev[j++]=rec[i];
        write(w,&rev,sizeof(rev));
        printf("\n");
   }
close(r);
close(w);
return(0);
}
```

SAMPLE INPUT/OUTPUT

Process a **Process b**

enter data to be written..good morning

received data from process a ...

reversed data from process b..gninrom doog

enter data to be written..hello

received data from process a ...

reversed data from process b..olleh

enter data to be written..quit

5. IPC USING MESSAGE QUEUE

PROBLEM DECRIPTION

Implement a message queue where the sender sends a message to receiver receiver receives the message , reverses it and retransmit to the sender. Sender has to print the reversed message.
This process is continued till “quit” is send as the message.

FUNCTIONS AND DATA STRUCTURES USED

1. A structure with two variables, a message type of long int type and message of character array is used.

```
struct mymsg {
   long      mtype;    /* message type */
   char mtext[MSGSZ]; /* message text of length MSGSZ */
}
```

2. msgget() :
   Syntax:
   msqid = msgget(ftok("/tmp",key), (IPC_CREAT | IPC_EXCL | 0400));
   Creates a message queue and returns a positive integer as its unique identifier.
3. Msgsnd()
   Syntax:
   int msgsnd(int msqid, const void *msgp, size_t msgsz,int msgflg);
   This function sends the message to the reciever.
4. Msgrcv():
   Syntax:
   int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,int msgflg);
   This function receives message from the message queue.

5. IPC_RMID

   Syntax:

   k=msgctl(id,IPC_RMID,0);

   -- Remove the message queue specified by the msqid argument.

ALGORITHM

Sender

1.Define a structure with an integer variable and a character array.

2.Create a message queue using msgget().

3.Set message type as 1.

4.Accept the message.

5.Send the message using msgsnd().

6.Receive the reversed message using msgrev().

7.Repeat the process until “quit” is send.

Receiver

1.Define a structure with an integer variable and a character array.

2.Create a message queue using msgget().

3.Set message type as 1.

4.Receive the message using msgrev();

5.Print the message.

6.Reverse the message and send it back to the sender.

7.Repeat the process until “quit” is received.

8.Remove the messageid using msgctl().

CODE

Sender

```
#include<sys/msg.h>
#include<sys/ipc.h>
#include<sys/types.h>
#include<stdio.h>
struct amsgbuf{
        int mtype;
```

```
        char mes[512];
};

struct amsgbuf mbuf,m;
main()
{
        int id,k;
        id=msgget((key_t)1234,0666|IPC_CREAT);
        //printf("%d",id);
        mbuf.mtype=1;
        m.mtype=1;
        do
        {
        printf("enter the message");
        fgets(mbuf.mes,50,stdin);
        k=msgsnd(id,(void*)&mbuf,512,0);
        if(k==-1)
        {
                printf("error");
                exit(1);
        }
        strcpy(mbuf.mes,"reset");
         k=msgrcv(id,(void*)&m,512,0,0);
        if(k==-1)
        {
                printf("cant rcv error");
                exit(1);
        }
        printf("\n reversed message is :");
        printf("\n%s",m.mes);
        }while(strcmp(mbuf.mes,"quit")!=0);
}
```

Receiver

```
#include<sys/msg.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<stdio.h>
#include<string.h>
struct amsgbuf{
        int mtype;
        char mes[512];
};
struct amsgbuf mbuf,m;
main()
{
        int id,k,i,j=0;
        id=msgget((key_t)1234,0666|IPC_CREAT);
        mbuf.mtype=1;
       do
        {
        k=msgrcv(id,(void*)&mbuf,512,0,0);
        if(k==-1)
        {
                printf("cant rcv error");
                exit(1);
        }
        printf("%s",mbuf.mes);
        for(i=strlen(mbuf.mes)-1;i>=0;i--)
        {
                m.mes[j]=mbuf.mes[i];
                j++;
        }
```

```
        m.mes[j]='\0';
        m.mtype=1;
        k=msgsnd(id,(void*)&m,512,0);
        if(k==-1)
        {
                printf("error");
                exit(1);
        }
        }while(strcmp(mbuf.mes,"quit")!=0);
        k=msgctl(id,IPC_RMID,0);
}
```

SAMPLE INPUT/OUTPUT

Sender

enter the message hello
reversed message is :olleh
enter the message hi
reversed message is :ih
enter the message hwru
reversed message is :urwh
enter the message quit

Receiver

hello
hi
hwru

SOCKET PROGRAMMING

A socket is a communications connection point (endpoint) that you can name and address in a network. The processes that use a socket can reside on the same system or on different systems on different networks. Sockets are useful for both stand-alone and network applications. A socket allow you to exchange information between processes on the same machine or across a network, distribute work to most efficient machine and allows access to centralized data easily.

The connection that a socket provides can be **connection-oriented or connectionless.**

Connection-oriented communication implies that a connection is established, and a dialog between the programs will follow. The program that provides the service (the server program) establishes the available socket which is enabled to accept incoming connection requests. Optionally, the server can assign a name to the service that it supplies which allows clients to identify where to obtain and how to connect to that service. The client of the service (the client program) must request the service of the server program. The client does this by connecting to the distinct name or to the attributes associated with the distinct name that the server program has designated. It is similar to dialing a telephone number (an identifier) and making a connection with another party that is offering a service (for example, a plumber). When the receiver of the call (the server, in this example, a plumber) answers the telephone, the connection is established. The plumber verifies that you have reached the correct party, and the connection remains active as long as both parties require it.

Connectionless communication implies that no connection is established over which a dialog or data transfer can take place. Instead, the server program designates a name that identifies where to reach it (much like a post office box). If you send a letter to a post office box, you cannot be absolutely sure the receiver got the letter. You may need to wait for a response to your letter. There is no active, real time connection in which data is exchanged.

SOCKET TYPE

Stream (SOCK_STREAM)

This type of socket is connection-oriented. Establish an end-to-end connection by using the bind(), listen(), accept(), and connect() functions. SOCK_STREAM sends data without errors or duplication, and receives the data in the sending order. SOCK_STREAM builds flow control to avoid data overruns. It does not impose record boundaries on the data. SOCK_STREAM considers the data to be a stream of bytes

Datagram (SOCK_DGRAM)

In Internet Protocol terminology, the basic unit of data transfer is a datagram. This is basically a header followed by some data. The datagram socket is connectionless. It establishes no end-to-end connection with the transport provider (protocol). The socket sends datagrams as independent packets with no guarantee of delivery. You can lose or duplicate data. Datagrams can arrive out of order. The size of the datagram is limited to the data size that you can send in a single transaction. For some transport providers, each datagram can use a different route through the network. You can issue a connect() function on this type of socket. However, on the connect() function, you must specify the destination address that the program sends to and receives from.

Creating a connection-oriented socket

A connection-oriented server uses the following sequence of function calls:
socket() bind() connect() listen() accept() send()
recv() close()

A connection-orientated client uses the following sequence of function calls:
socket() gethostbyname() connect() read() write()
close()

Creating a connectionless socket

A connectionless client illustrate the socket APIs that are written for User Datagram Protocol (UDP).

The server use the following sequence of function calls:

socket() bind() sendto() recvfrom() close()

The client example uses the following sequence of function calls:

socket() gethostbyname() sendto() recvfrom() close().

6. CLIENT SERVER COMMUNICATION USING TRANFER CONTROL PROTOCOL (TCP)

PROBLEM DESCRIPTION

Using TCP socket, create a connection between a client and server and exchange data. The data sent by the client should be reversed at the server and sent back to client.

FUNTIONS AND DATA STRUCTURES

struct sockaddr: This structure holds socket address information for many types of sockets:

```
struct sockaddr {
  unsigned short    sa_family;    // address family, AF_xxx
  char              sa_data[14];  // 14 bytes of protocol address
};
```

sa_family can be a variety of things, but it'll be AF_INET for everything we do in this document. sa_data contains a destination address and port number for the socket. To deal with struct sockaddr, programmers created a parallel structure:

struct sockaddr_in ("in" for "Internet".)

```
struct sockaddr_in {
   short int          sin_family;  // Address family
   unsigned short int sin_port;    // Port number
   struct in_addr     sin_addr;    // Internet address
   unsigned char      sin_zero[8]; // Same size as struct sockaddr
};
```

This structure makes it easy to reference elements of the socket address. Note that sin_zero (which is included to pad the structure to the length of a struct sockaddr) should be set to all zeros with the function memsetFinally, the sin_port and sin_addr must be in Network Byte Order!

System calls that allow to access the network functionality of a Unix box are as given below. When you call one of these functions, the kernel takes over and does all the work for you automatically

Server Side

socket()

Syntax: socket(int domain, int type, int protocol);

bind()

Syntax: bind(int sockfd, struct sockaddr *my_addr, int addrlen);

connect()

Syntax: connect(int sockfd, struct sockaddr *serv_addr, int addrlen);

listen()

Syntax: listen(int sockfd, int backlog);

accept()

Syntax: accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

send()

Syntax: send(int sockfd, const void *msg, int len, int flags);

recv()

Syntax: recv(int sockfd, void *buf, int len, unsigned int flags);

close()

Syntax: close(sockfd)

Client Side

socket()

Syntax: socket(int domain, int type, int protocol);

gethostbyname()

Syntax: struct hostent *gethostbyname(const char *name);

connect()

Syntax: connect(int sockfd, struct sockaddr *serv_addr, int addrlen);

send()

Syntax: send(int sockfd, const void *msg, int len, int flags);

recv()

Syntax: recv(int sockfd, void *buf, int len, unsigned int flags);

close()

Syntax: close(sockfd)

ALGORITHM

TCP Server

1. Create a socket using the function socket(int domain, int type, int protocol).
2. Bind it to the operating system using (int sockfd, struct sockaddr *my_addr, int addrlen).
3. Listen over it using listen(int sockfd, int backlog).
4. Accept connections connect(int sockfd, struct sockaddr *serv_addr, int addrlen);.
5. Receive data from client and reverse it, send it back to client using send(int sockfd, const void *msg, int len, int flags) and recv(int sockfd, void *buf, int len, unsigned int flags)
6. Close the socket.

TCP Client

1. Create a tcp socket using socket(int domain, int type, int protocol).
2. Connect using the function connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
3. Receive data using recv(int sockfd, void *buf, int len, unsigned int flags)
4. Send the data using send(int sockfd, const void *msg, int len, int flags).
5. Close the socket.

CODE

//CLIENT

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<sys/socket.h>
#define PORT 1425
#define MAXDATASIZE 100

int main()
{
int sockfd,numbytes;
char buf[MAXDATASIZE];
struct sockaddr_in their_addr;
their_addr.sin_family=AF_INET;
their_addr.sin_port=htons(PORT);
their_addr.sin_addr.s_addr=INADDR_ANY;
bzero(&(their_addr.sin_zero),8);
while(1)
{
  sockfd=socket(AF_INET,SOCK_STREAM,0);
  if(connect(sockfd,(struct sockaddr*)&their_addr, sizeof(struct
sockaddr))==-1)
     {
       perror("error_connect");
       exit(1);
       }
printf("\n enter the string\n");
scanf("%s",buf);
if(send(sockfd,buf,15,0)==-1)
    {
      perror("error_send");

      close(sockfd);

      exit(0);
    }
if((numbytes==recv(sockfd,buf,MAXDATASIZE,0))==-1)
   {
    perror("error_receive");
    exit(1);
}
//buf[numbytes]='\0';
printf("received%s\n",buf);
close(sockfd);
}
return(0);
}
```

//SERVER

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<sys/socket.h>
```

```
#include<string.h>
#define MYPORT 1425
#define BACKLOG 10

int main()
{
 int sockfd,new_fd,i,j,len;
 int sin_size,number,count;
 char str[20],rev[20];
 struct sockaddr_in my_addr;
 struct sockaddr_in their_addr;
 sockfd=socket(AF_INET,SOCK_STREAM,0);
 if(sockfd==-1)
 {
      perror("Socket");
      exit(1);
 }
 my_addr.sin_family=AF_INET;
 my_addr.sin_port=htons(MYPORT);
 my_addr.sin_addr.s_addr=INADDR_ANY;
 bzero(&(my_addr.sin_zero),8);
 if(bind(sockfd,(struct sockaddr *)&my_addr,sizeof(struct sockaddr))==-1)
 {
      perror("bind");
      exit(1);
 }
 if(listen(sockfd,BACKLOG)==-1)
 {
      perror("listen");
      exit(1);
 }
 while(1)
 {
      sin_size=sizeof(struct sockaddr_in);
      if((new_fd=accept(sockfd,(struct sockaddr
*)&their_addr,&sin_size))==-1)
      {
            perror("accept");
            exit(0);
        }
      if((number==recv(new_fd,str,25,0))==-1)
      {
            perror("Error-receive");
            exit(0);
      }
      //str[number]='\0';
      printf("\nReceived:%s\n",str);
      i=0;
      while(str[i]!='\0')
      {
            len=i;
            i++;
      }
      j=0;
      while(len>=0)
      {
            rev[j]=str[len--];
            j++;
      }
      rev[j]='\0';
      if(send(new_fd,rev,14,0)==-1)
      {
            perror("Error-send");
```

```
            close(new_fd);
            exit(0);
      }
 }
 close(new_fd);
 return(0);
}
```

SAMPLE INPUT/OUTPUT

Client	**Server**
enter the data:how **are you**	
	Received:how are you
received: uoy era woh	
enter the data:hello	
	Received:hello

7. CLIENT SERVER COMMUNICATION USING USER DATAGRAM PROTOCOL (UDP)

PROBLEM DESCRIPTION

Using UDP socket, create a connection between a client and server and exchange data. The data sent by the client should be reversed at the server and sent back to client.

FUNTIONS AND DATA STRUCTURES

struct sockaddr: This structure holds socket address information for many types of sockets:

```
struct sockaddr {
  unsigned short    sa_family;    // address family, AF_xxx
  char              sa_data[14];  // 14 bytes of protocol address
};
```

sa_family can be a variety of things, but it'll be AF_INET for everything we do in this document. sa_data contains a destination address and port number for the socket.
To deal with struct sockaddr, programmers created a parallel structure:
struct sockaddr_in ("in" for "Internet".)

```
struct sockaddr_in {
   short int          sin_family;  // Address family
   unsigned short int sin_port;    // Port number
   struct in_addr     sin_addr;    // Internet address
   unsigned char      sin_zero[8]; // Same size as struct sockaddr
};
```

This structure makes it easy to reference elements of the socket address. Note that sin_zero (which is included to pad the structure to the length of a struct sockaddr) should be set to all zeros with the function memset. Finally, the sin_port and sin_addr must be in Network Byte Order!

System calls that allow to access the network functionality of a Unix box are as given below. When you call one of these functions, the kernel takes over and does all the work for you automatically

The server use the following sequence of function calls:

socket()

Syntax: socket(int domain, int type, int protocol);

bind()

Syntax: bind(int sockfd, struct sockaddr *my_addr, int addrlen);

sendto()

Syntax: sendto(int sockfd, const void *msg, int len, unsigned int flags,
const struct sockaddr *to, socklen_t tolen);

recvfrom()

Syntax: recvfrom(int sockfd, void *buf, int len, unsigned int flags,
struct sockaddr *from, int *fromlen);

close()

Syntax: close(sockfd)

The client example uses the following sequence of function calls:

socket()

Syntax: socket(int domain, int type, int protocol);

gethostbyname()

Syntax: struct hostent *gethostbyname(const char *name);

sendto()

Syntax: sendto(int sockfd, const void *msg, int len, unsigned int flags,
const struct sockaddr *to, socklen_t tolen);

recvfrom()

Syntax: recvfrom(int sockfd, void *buf, int len, unsigned int flags,

struct sockaddr *from, int *fromlen);

close().

Syntax: close(sockfd)

ALGORITHM

UDP Server

1. Create the internal host *server and initialize
2. Initialize the members of sin_addr stucture and port address
3. Create the socket so with parameters (AF_INET, SOCK_DGRAM, 0) for datagram communication
4. Bind the socket to its address
5. Receive message sent from client, reverse it and send it back.
6. Display message
7. Close the socket

UDP Client

1. Initialize port address coresponding to the server
2. Initialize the members of sin_addr stucture
3. Create the socket so with parameters (AF_INET, SOCK_DGRAM, 0) for datagram communication
4. Send message to server
5. Close the socket

CODING

// SERVER

```
#include"sys/socket.h"
#include"arpa/inet.h"
#include"netdb.h"
#include"unistd.h"
#include"string.h"
#include"stdio.h"
#include"stdlib.h"

int main(){
```

```
  char buf[100];
  int i=1,k;
  int sock_desc;
  struct sockaddr_in server;

  memset(&server,0,sizeof(server)); //initialising de structure object
'server' with 0 bytes

  sock_desc=socket(AF_INET,SOCK_DGRAM,0); //creating socket descriptor
  if(sock_desc==-1){
    printf("Error in creating socket");
    exit(1);
  }

  server.sin_family=AF_INET; //initialising sockaddr_in parameters like
family, address & port no:
  server.sin_addr.s_addr=inet_addr("127.0.0.1");
  server.sin_port=3000;

  while(1){
    sprintf(buf,"Data Pkt %d",i);
    k=sendto(sock_desc,buf,100,0,(struct sockaddr
*)&server,sizeof(server)); //sending the data pkts to client
    if(k==-1){
      printf("\nError in sending data pkts");
      exit(1);
    }
    sleep(1);
    if(i>=20){
      strcpy(buf,"end");
      k=sendto(sock_desc,buf,100,0,(struct sockaddr
*)&server,sizeof(server));
      break;
    }
    i++;
  }
  close(sock_desc); //closing the socket descriptor after use(optional)
  exit(0);
  return 0;
}
```

//CLIENT

```
#include"sys/socket.h"
#include"sys/types.h"
#include"arpa/inet.h"
#include"netdb.h"
#include"unistd.h"
#include"stdio.h"
#include"string.h"
#include"stdlib.h"

int main(){
  socklen_t len;
  char buf[100];
  int sock_desc;
  int k;
  struct sockaddr_in client;

  memset(&client,0,sizeof(client));
```

```
  sock_desc=socket(AF_INET,SOCK_DGRAM,0);
  if(sock_desc==-1){
    printf("Error in creating socket");
    exit(1);
  }

  client.sin_family=AF_INET;
  client.sin_addr.s_addr=INADDR_ANY; //first difference
  client.sin_port=3000;

  k=bind(sock_desc,(struct sockaddr*)&client,sizeof(client)); //second
difference
  if(k==-1){
    printf("\nError in binding socket");
    exit(1);
    }

  while(1){
    //len=sizeof(client);
    k=recvfrom(sock_desc,buf,100,0,(struct sockadddr *)&client,&len);
//third difference
    /*if(k==-1){
      printf("\nError in receiving pkt");
      exit(1);
    }*/
    if((strcmp(buf,"end")==0)){
      printf("End of transmission from server");
      break;
    }
    printf("\n%s\n",buf);
  }
  close(sock_desc);
  exit(0);
  return 0;
}
```

SAMPLE INPUT/OUTPUT

Client **Server**

enter the data:how **are you**

Received:how are you

received: uoy era woh

enter the data:hello

Received:hello

MAC PROTOCOLS

Theory

The Media Access Control (MAC) data communication protocol sub-layer , also known as the medium Access Control, is a part of the data link layer specified in the seven layer OSI model (layer 2) . It provides addressing and channel access control mechanisms that make it possible for several terminal or network nodes to communicate within a multipoint network (LAN) or Metropolitan Area Network (MAN). AMAC protocol is not required in full-duplex point –to- point communication. In single channel point –to- point communications full-duplex can be emulated . This emulation can be considered a MAC layer.

The MAC sub-layer acts as an interface between the Logical Link Control sublayer and the network's physical layer.

The MAC layer provides an addressing mechanism called physical address or MAC address. This is a unique serial number assigned to each network adapter, making it possible to deliver data packets to a destination within a subnetwork, i.e a physical network without routers, for example an Ethernet network.

Media access control is often used as a synonym to multiple access protocol, since the MAC sublayer provides the protocol and control mechanisms that are required for a certain channel access method. This makes it possible for several stations connected to the same physical medium to share it . Example of shared physical medium are bus networks , ring networks , hub networks , wireless networks and half – duplex point – to –point links.

SLIDING WINDOW PROTOCOL

In the simplex, stop and wait protocols , data frames were transmitted in one direction only. In most practical solutions , there is a need for transmitting data in both

directions . One way of achieving full-duplex data transmission is to have to separate communication channels and use one for simplex data traffic. If this is done, we have two separate physical circuits, each with a "forward: channel (for data) and a "reverse" channel (for acknowledgments). In both case the bandwidth of the reverse channel is almost wasted.

A better idea is to use same circuit for data in both directions. In this model data frames from A to B are intermixed with the acknowledgment frames A to B. By looking at the field in the header of and incoming frame, the receiver can tell whether the frame is data or acknowledgment. When a data from arrives, instead of immediately sending a separate control frame, the receiver retains itself and waits until the network layer passes it the next packet. The acknowledgment is attached to the outgoing data frame. The technique of temporarily delaying outgoing acknowledgments so that they can be hooked onto the next outgoing data frame known as piggybacking. The principal advantage of using piggybacking over having distinct acknowledgment frames is a better use of the available channel bandwidth.

Bidirectional protocols that belongs to a class called sliding window protocols. They differ among themselves in terms of efficiency , complexity,and buffer requirements. All sliding window protocols, each outbound frame contains a sequence number ,ranging from 0 up to some maximum. The essence of all sliding window protocols is that at any instant of time , the sender maintains a set of sequence numbers corresponding frames it is permitted to send . These frames are said to fall within the sending window, the receiver also maintains a receiving window corresponding to the set of frames it is permitted to accept.

A one – bit sliding window protocol

A sliding window protocol with a maximum window size of 1. Such a protocol use a stop – and – wait since the sender transmits a frame and waits from its acknowledgment before sending the next one. The starting machine fetches the first packet from its network layer , builds a frame from it , and sends it. The acknowledgment field contains the number of last frame received without error. If this number agrees with the sequence number of the frame the sender trying to send , the sender knows it is done with the frame stored in buffer and can fetch the next packet

from its network layer. If the sequence number disagrees, it must continue trying to send the same dame frame whenever a frame is received, a frame also sent back.

A Protocol Using Go back N

Two basic approaches are available for dealing with errors in the presence of pipelining. One way , called go back N , is for the receiver simply to discard all subsequent frames, sending no acknowledgments for the discarded frames. In other words , the data link layer refuses to accept any frame except the next one it must give to the network layer.

A Protocol Using Selective Repeat

In this protocol , both the sender and receiver maintain a window of acceptable sequence numbers. The sender's window size starts out at 0 and grows to some predefined maximum. The receiver's window , in contrast , is always fixed in size and equal to maximum. The receiver has a buffer reserved for each sequence number within its fixed window. Whenever a frame arrives , its sequence number is checked by the function between to see if falls within the window. If so and if it has not already been received , it is accepted and stored. This action is taken without regard to whether or not it contains the next packet expected by the network layer.

8. 1-BIT SLIDING WINDOW PROTOCOL

PROGRAM DESCRIPTION

This program is a simulation of one bit Sliding window protocol. After establishing connection, the server sends 5 packets to the client. If the packet is received successfully, an acknowledgement message is sent back.then only server sends the next packet. For simulation purpose, packet 3 is supposed to be lost for the first time. Then client sends a RETRANSMIT request along with the current packet number. On receiving this, the server retransmits the 3rd packet.

FUNCTIONS AND DATA STRUCTURES USED

Client

Sockfd and newSockFd are integer socket descriptors.currentPacket is an integer that denotes the packet that is sent last. Data is a string that is used to store the message. struct sockaddr_in client -->.

Server

Sockfd and newSockFd are integer socket descriptors.currentPacket is an integer that denotes the packet that is sent last. Data is a string that is used to store the message. struct sockaddr_in server, client -->.

ALGORITHM

Server

1. Start.
2. Establish connection with the client . (UDP/ TCP recommended UDP)
3. Accept the window size from the client.
4. Accept the packet from the network layer.
5. Combine the packets to form frames / window. (depending on window size.).

6. Initialize the transmit buffer.
7. Send the frame and wait for the acknowledgment.
8. If a negative acknowledgment is received repeat the transmission of the previous frame .

   Else increment the frame to be transmitted.
9. Repeat steps 5 to 8 until all packets are transmitted successfully.
10. Close the connection.
11. Stop.

Client

1. Start.
2. Establish a connection with the server. (recommended UDP).
3. Send the window size on server request.
4. Accept the details from the server (total packets, total frames).
5. Initialize the receive buffer with the expected frame and packets.
6. Accept the frame and check for its validity.
7. Send the positive or negative acknowledgment as required.
8. Repeat steps 5 to 7 until all packets are received.
9. Close the connection with server.
10. Stop.

CODE

```
/**************/
/* CLIENT */
/**************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 3599

int main() {

      int sockfd, newSockFd, size, firstTime = 1, currentPacket;
      char data[100];

      struct sockaddr_in client;
```

```
	memset(&client, 0, sizeof(client));

	sockfd = socket(AF_INET, SOCK_STREAM, 0);
	if(sockfd == -1) {
		printf("Error in socket creation...");
	}
	else {
		printf("\nSocket Created...");
	}

	client.sin_family = AF_INET;
	client.sin_port = PORT;
	client.sin_addr.s_addr = inet_addr("127.0.0.1");

	printf("\nStarting up...");

	size = sizeof(client);

	printf("\nEstablishing Connection...");

	if(connect(sockfd, (struct sockaddr *)&client, size) == -1) {
		printf("\nError in connecting to server...");
		exit(1);
	} else {
		printf("\nConnection Established!");
	}

	memset(&data, 0, sizeof(data));
	sprintf(data, "REQUEST");

	if(send(sockfd, data, strlen(data), 0) == -1) {
		printf("Error in sending request for data...");
		exit(1);
	}

	do {
		memset(&data, 0, sizeof(data));
		recv(sockfd, data, 100, 0);
		currentPacket = atoi(data);
		printf("\nGot packet: %d", currentPacket);
		if(currentPacket == 3 && firstTime) {
			printf("\n*** Simulation: Packet data corrupted or
incomplete.");
			printf("\n*** Sending RETRANSMIT.");
			memset(&data, 0, sizeof(data));
			sprintf(data, "RETRANSMIT");
			if(send(sockfd, data, strlen(data), 0) == -1) {
				printf("\nError in sending RETRANSMIT...");
				exit(1);
			}
			firstTime = 0;
		}
		else {
			printf("\n*** Packet Accepted -> Sending ACK");
		}

		memset(&data, 0, sizeof(data));
		sprintf(data, "ACK");
		if(send(sockfd, data, strlen(data), 0) == -1) {
			printf("\nError in sending ACK...");
			exit(1);
		}
```

```
	} while(currentPacket != 5);

	printf("\nAll packets recieved... Exiting.");

	close(sockfd);
	return(0);
}

/*********************************/
/* SLIDING WINDOW ONE BIT SERVER */
/*********************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 3599

void itoa(int number, char numberString[]) {
	numberString[0] = (char)(number + 48);
	numberString[1] = '\0';
}

int main() {

	int sockfd, newSockFd, size, currentPacket = 1;
	char buffer[100];
	socklen_t len;

	struct sockaddr_in server, client;

	memset(&server, 0, sizeof(server));
	memset(&client, 0, sizeof(client));

	if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
		printf("\nError in socket creation...");
		exit(1);
	}
	else {
		printf("\nSocket created successfully...");
	}

	server.sin_family = AF_INET;
	server.sin_port = PORT;
	server.sin_addr.s_addr = INADDR_ANY;

	printf("\nStarting up...");

	if(bind(sockfd, (struct sockaddr *)&server, sizeof(server)) == -1) {
		printf("\nBinding Error...");
		exit(1);
	}
	else {
		printf("\nBinding completed successfully. Waiting for
connection..");
	}

	len = sizeof(client);
```

```
      if(listen(sockfd, 20) != -1) {
            if((newSockFd = accept(sockfd, (struct sockaddr *)&client,
&len)) == -1) {
                  printf("Error in accepting connection...");
                  exit(1);
            }

            memset(&buffer, 0, sizeof(buffer));
            if(recv(newSockFd, buffer, 100, 0) == -1) {
                  printf("\n Recieve Error! Exiting...");
                  exit(1);
            }

            printf("\nRecieved a request from client. Sending packets one
by one...");
            do {
                  memset(&buffer, 0, sizeof(buffer));
                  itoa(currentPacket, buffer);
                  send(newSockFd, buffer, 100, 0);
                  printf("\nPacket Sent: %d", currentPacket);

                  memset(&buffer, 0, sizeof(buffer));
                  recv(newSockFd, buffer, 100, 0);
                  currentPacket++;
                  if(strcmp(buffer, "RETRANSMIT") == 0) {
                        currentPacket--;
                        printf("\n** Received a RETRANSMIT packet.
Resending last packet...");
                  }
                  sleep(1);
            } while(currentPacket != 6);
      }
      else {
            printf("\nError in listening...");
            exit(1);
      }

      close(sockfd);
      close(newSockFd);
      printf("\nSending Complete. Sockets closed. Exiting...\n");

      return(0);
}
```

SAMPLE INPUT/OUTPUT

server

Socket created successfully...

Starting up...

Binding completed successfully. Waiting for connection..

Recieved a request from client. Sending packets one by one...

Packet Sent: 1

Packet Sent: 2

Packet Sent: 3

** Received a RETRANSMIT packet. Resending last packet...

Packet Sent: 3

Packet Sent: 4

Packet Sent: 5

Sending Complete. Sockets closed. Exiting...

client

Socket created successfully...

Starting up...

Binding completed successfully. Waiting for connection..

Recieved a request from client. Sending packets one by one...

Packet Sent: 1

Packet Sent: 2

Packet Sent: 3

** Received a RETRANSMIT packet. Resending last packet...

Packet Sent: 3

Packet Sent: 4

Packet Sent: 5

Sending Complete. Sockets closed. Exiting...

9. GO-BACK-N SLIDING WINDOW PROTOCOL

PROBLEM DESCRIPTION

This program is a simulation of Go Back N Sliding window protocol. After establishing connection, the server sends 10 packets to the client. For this non blocking send is used. i.e., the packets are sent continuously without waiting for the ACK. If the packet is received successfully, an acknowledgement message is sent back by the client. For simulation purpose, packet 3 is supposed to be lost for the first time. Then client sends a RETRANSMIT request along with the current packet number. On receiving this, the server moves the window Start back to 3. Then it retransmits packets from 3 onwards.

FUNCTIONS AND DATA STRUCTURES USED

Client

Sockfd and newSockFd are integer socket descriptors.currentPacket is an integer that denotes the packet that is sent last. Data is a string that is used to store the message.struct sockaddr_in client -->, Digit is used to store the packet no for ACK or Retransmit message.

Server

Sockfd and newSockFd are integer socket descriptors.currentPacket is an integer that denotes the packet that is sent last.WindowStart and WindowEnd are use to sore the starting and ending of window. Data is a string that is used to store the message. struct sockaddr_in server, client -->.

ALGORITHM

Server

1. Start
2. Establish connection

3. Accept the window size from the client
4. Accept the packet from the network layer.
5. Calculate the total frames / windows required.
6. Send the details to the client (total packets , total frames .)
7. Initialize the transmit buffer.
8. Built the frame / window depending on the window size.
9. Transmit the frame.
10. Wait the acknowledgment frame.
11. Check for the acknowledgment of each packet and repeat the process from the packet for which the first negative acknowledgment is received.
    Else continue as usual.
12. Increment the frame count and repeat steps 7 to 12 until all packets are transmitted.
13. Close the connection.
14. Stop.

Client

1. Start.
2. Establish a connection. (recommended UDP).
3. Send the windowsize on server request.
4. Accept the details from the server (total packets, total frames)
5. Initialize the receive buffer with the expected packets.
6. Accept the frame / window from the server.
7. Check the validity of the packet and construct the acknowledgment frame depending on the validity. (Here the acknowledgment is accepted from the users)
8. Depending on the acknowledgment frame readjust the process.
9. Increment the frame count and repeat steps 5-9 until all packets are received.
10. Close the connection
11. Stop.

CODE

```
/*********************************/
/* SLIDING WINDOW GO-BACK CLIENT */
/*********************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 3599

int main() {

      int sockfd, newSockFd, size, firstTime = 1, currentPacket, wait = 3;
      char data[100], digit[2];

      struct sockaddr_in client;

      memset(&client, 0, sizeof(client));

      sockfd = socket(AF_INET, SOCK_STREAM, 0);
      if(sockfd == -1) {
            printf("Error in socket creation...");
      }
      else {
            printf("\nSocket Created...");
      }

      client.sin_family = AF_INET;
      client.sin_port = PORT;
      client.sin_addr.s_addr = inet_addr("127.0.0.1");

      printf("\nStarting up...");

      size = sizeof(client);

      printf("\nEstablishing Connection...");

      if(connect(sockfd, (struct sockaddr *)&client, size) == -1) {
            printf("\nError in connecting to server...");
            exit(1);
      } else {
            printf("\nConnection Established!");
      }

      memset(&data, 0, sizeof(data));
      sprintf(data, "REQUEST");

      if(send(sockfd, data, strlen(data), 0) == -1) {
            printf("Error in sending request for data...");
            exit(1);
      }

      do {
            memset(&data, 0, sizeof(data));
            recv(sockfd, data, 100, 0);
            currentPacket = atoi(data);
            printf("\nGot packet: %d", currentPacket);
```

```
            if(currentPacket == 3 && firstTime) {
                  printf("\n*** Simulation: Packet data corrupted or
incomplete.");
                  printf("\n*** Sending RETRANSMIT for packet 1.");
                  memset(&data, 0, sizeof(data));
                  sprintf(data, "R1");
                  if(send(sockfd, data, strlen(data), 0) == -1) {
                        printf("\nError in sending RETRANSMIT...");
                        exit(1);
                  }
                  firstTime = 0;
            }
            else {
                  wait--;
                  if(!wait) {
                        printf("\n*** Packet Accepted -> Sending ACK");
                        wait = 3;
                        memset(&data, 0, sizeof(data));
                        sprintf(data, "A");
                        digit[0] = (char)(currentPacket + 48);
                        digit[1] = '\0';
                        strcat(data, digit);
                        send(sockfd, data, strlen(data), 0);
                  }
            }

      } while(currentPacket != 9);

      printf("\nAll packets recieved... Exiting.");

      close(sockfd);
      return(0);
}

/*********************************/
/* SLIDING WINDOW GO-BACK SERVER */
/*********************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <fcntl.h>

#define PORT 3599

void itoa(int number, char numberString[]) {
      numberString[0] = (char)(number + 48);
      numberString[1] = '\0';
}

int main() {

      int sockfd, newSockFd, size, windowStart = 1, windowCurrent = 1,
windowEnd = 4, oldWindowStart, flag;
      char buffer[100];
      socklen_t len;

      struct sockaddr_in server, client;
```

```
	memset(&server, 0, sizeof(server));
	memset(&client, 0, sizeof(client));

	if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
		printf("\nError in socket creation...");
		exit(1);
	}
	else {
		printf("\nSocket created successfully...");
	}

	server.sin_family = AF_INET;
	server.sin_port = PORT;
	server.sin_addr.s_addr = INADDR_ANY;

	printf("\nStarting up...");

	if(bind(sockfd, (struct sockaddr *)&server, sizeof(server)) == -1) {
		printf("\nBinding Error...");
		exit(1);
	}
	else {
		printf("\nBinding completed successfully. Waiting for
connection..");
	}

	len = sizeof(client);

	if(listen(sockfd, 20) != -1) {

		if((newSockFd = accept(sockfd, (struct sockaddr *)&client,
&len)) == -1) {
			printf("Error in accepting connection...");
			exit(1);
		}


		memset(&buffer, 0, sizeof(buffer));
		if(recv(newSockFd, buffer, 100, 0) == -1) {
			printf("\n Recieve Error! Exiting...");
			exit(1);
		}

		fcntl(newSockFd, F_SETFL, O_NONBLOCK);

		printf("\nRecieved a request from client. Sending packets one
by one...");
		do {
			if(windowCurrent != windowEnd) {
			memset(&buffer, 0, sizeof(buffer));
			itoa(windowCurrent, buffer);
			send(newSockFd, buffer, 100, 0);
			printf("\nPacket Sent: %d", windowCurrent);
			windowCurrent++;
			}

			/*DEBUG*/ printf("\n**%d||%d**", windowCurrent,
windowEnd);

			memset(&buffer, '\0', sizeof(buffer));
			if(recv(newSockFd, buffer, 100, 0) != -1) {
				if(buffer[0] == 'R') {
```

```
                              printf("\n** Received a RETRANSMIT packet.
Resending packet no. %c...", buffer[1]);
                              itoa((atoi(&buffer[1])), buffer);
                              send(newSockFd, buffer, 100, 0);
                              windowCurrent = atoi(&buffer[0]);
                              windowCurrent++;
                        }
                        else if(buffer[0] == 'A') {
                              oldWindowStart = windowStart;
                              windowStart = atoi(&buffer[1]) + 1;
                              windowEnd += (windowStart - oldWindowStart);
                              printf("\n** Recieved ACK %c. Moving window
boundary.", buffer[1]);
                        }
                  }

                  sleep(1);
            } while(windowCurrent != 10);
      }
      else {
            printf("\nError in listening...");
            exit(1);
      }

      close(sockfd);
      close(newSockFd);
      printf("\nSending Complete. Sockets closed. Exiting...\n");

      return(0);
}
```

SAMPLE INPUT/OUTPUT

Client

Socket Created...

Starting up...

Establishing Connection...

Connection Established!

Got packet: 1

Got packet: 2

Got packet: 3

*** Simulation: Packet data corrupted or incomplete.

*** Sending RETRANSMIT for packet 1.

Got packet: 1

*** Packet Accepted -> Sending ACK

Got packet: 2

Got packet: 3

Got packet: 4

*** Packet Accepted -> Sending ACK

Got packet: 5

Got packet: 6

Got packet: 7

*** Packet Accepted -> Sending ACK

Got packet: 8

Got packet: 9

All packets recieved... Exiting.

Server

Socket created successfully...

Starting up...

Binding completed successfully. Waiting for connection..

Recieved a request from client. Sending packets one by one...

Packet Sent: 1

2||4

Packet Sent: 2

3||4

Packet Sent: 3

4||4

4||4

** Received a RETRANSMIT packet. Resending packet no. 1...

Packet Sent: 2

3||4

** Recieved ACK 1. Moving window boundary.

Packet Sent: 3

4||5

Packet Sent: 4

5||5

5||5

** Recieved ACK 4. Moving window boundary.

Packet Sent: 5

6||8

Packet Sent: 6

7||8

Packet Sent: 7

8||8

8||8

** Recieved ACK 7. Moving window boundary.

Packet Sent: 8

9||11

Packet Sent: 9

10||11

Sending Complete. Sockets closed. Exiting...

10. SELECTIVE REPEAT

PROBLEM DESCRIPTION

This program is a simulation of Selective Repeat Sliding window protocol. After establishing connection, the server sends 10 packets to the client. For this non blocking send is used i.e., the packets are sent continuously without waiting for the ACK. If the packet is received successfully, an acknowledgement message is sent back by the client. For simulation purpose, packet 3 is supposed to be lost for the first time. Then client sends a RETRANSMIT request along with the current packet number. On receiving this, the server retransmits the 3rd packet. After that it resumes sending packets from where it stopped.

FUNCTIONS AND DATA STRUCTURES USED

Client

Sockfd and newSockFd are integer socket descriptors.currentPacket is an integer that denotes the packet that is sent last. Data is a string that is used to store the message.struct sockaddr_in client -->, Digit is used to store the packet no for ACK or Retransmit msg

Server

Sockfd and newSockFd are integer socket descriptors. CurrentPacket is an integer that denotes the packet that is sent last. WindowStart and WindowEnd are use to sore the starting and ending of window. Data is a string that is used to store the message. struct sockaddr_in server, client -->.

ALGORITHM

Server

1. Start.
2. Establish connection (recommended UDP)
3. Accept the window size from the client (should be <= 40)

4. Accept the packet from the network layer.
5. Calculate the total frames / windows required.
6. Send the details to the client (total packets , total frames .)
7. Initialize the transmit buffer.
8. Built the frame / window depending on the window size.
9. Transmit the frame.
10. Wait the acknowledgment frame.
11. Check for the acknowledgment of each packet and repeat the process from the packet for which the first negative acknowledgment is received.
    Else continue as usual.
12. Increment the frame count and repeat steps 7 to 12 until all packets are transmitted.
13. Close the connection.
14. Stop.

Client

1. Start.
2. Establish a connection. (recommended UDP).
3. Send the windowsize on server request.
4. Accept the details from the server (total packets, total frames)
5. Initialize the receive buffer with the expected packets.
6. Accept the frame / window from the server.
7. Check the validity of the packet and construct the acknowledgment frame depending on the validity. (Here the acknowledgment is accepted from the users)
8. Depending on the acknowledgment frame readjust the process.
9. Increment the frame count and repeat steps 5-9 until all packets are received.
10. Close the connection
11. Stop.

CODE

```
/**************************/
/* SELECTIVE REJECT CLIENT */
/**************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 3599

int main() {

	int sockfd, newSockFd, size, firstTime = 1, currentPacket, wait = 3;
	char data[100], digit[2];

	struct sockaddr_in client;

	memset(&client, 0, sizeof(client));

	sockfd = socket(AF_INET, SOCK_STREAM, 0);
	if(sockfd == -1) {
		printf("Error in socket creation...");
	}
	else {
		printf("\nSocket Created...");
	}

	client.sin_family = AF_INET;
	client.sin_port = PORT;
	client.sin_addr.s_addr = inet_addr("127.0.0.1");

	printf("\nStarting up...");

	size = sizeof(client);

	printf("\nEstablishing Connection...");

	if(connect(sockfd, (struct sockaddr *)&client, size) == -1) {
		printf("\nError in connecting to server...");
		exit(1);
	} else {
		printf("\nConnection Established!");
	}

	memset(&data, 0, sizeof(data));
	sprintf(data, "REQUEST");

	if(send(sockfd, data, strlen(data), 0) == -1) {
		printf("Error in sending request for data...");
		exit(1);
	}

	do {
		memset(&data, 0, sizeof(data));
```

```
            recv(sockfd, data, 100, 0);
            currentPacket = atoi(data);
            printf("\nGot packet: %d", currentPacket);
            if(currentPacket == 3 && firstTime) {
                  printf("\n*** Simulation: Packet data corrupted or
incomplete.");
                  printf("\n*** Sending RETRANSMIT.");
                  memset(&data, 0, sizeof(data));
                  sprintf(data, "R3");
                  if(send(sockfd, data, strlen(data), 0) == -1) {
                        printf("\nError in sending RETRANSMIT...");
                        exit(1);
                  }
                  firstTime = 0;
            }
            else {
                  wait--;
                  if(!wait) {
                        printf("\n*** Packet Accepted -> Sending ACK");
                        wait = 3;
                        memset(&data, 0, sizeof(data));
                        sprintf(data, "A");
                        digit[0] = (char)(currentPacket + 48);
                        digit[1] = '\0';
                        strcat(data, digit);
                        send(sockfd, data, strlen(data), 0);
                  }
            }

      } while(currentPacket != 9);

      printf("\nAll packets recieved... Exiting.");

      close(sockfd);
      return(0);
}


/****************************/
/*  SELECTIVE REJECT SERVER */
/****************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <fcntl.h>

#define PORT 3599

void itoa(int number, char numberString[]) {
      numberString[0] = (char)(number + 48);
      numberString[1] = '\0';
}

int main() {

      int sockfd, newSockFd, size, windowStart = 1, windowCurrent = 1,
windowEnd = 4, oldWindowStart, flag;
      char buffer[100];
```

```
      socklen_t len;

      struct sockaddr_in server, client;

      memset(&server, 0, sizeof(server));
      memset(&client, 0, sizeof(client));

      if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            printf("\nError in socket creation...");
            exit(1);
      }
      else {
            printf("\nSocket created successfully...");
      }

      server.sin_family = AF_INET;
      server.sin_port = PORT;
      server.sin_addr.s_addr = INADDR_ANY;

      printf("\nStarting up...");

      if(bind(sockfd, (struct sockaddr *)&server, sizeof(server)) == -1) {
            printf("\nBinding Error...");
            exit(1);
      }
      else {
            printf("\nBinding completed successfully. Waiting for
connection..");
      }

      len = sizeof(client);

      if(listen(sockfd, 20) != -1) {

            if((newSockFd = accept(sockfd, (struct sockaddr *)&client,
&len)) == -1) {
                  printf("Error in accepting connection...");
                  exit(1);
            }


            memset(&buffer, 0, sizeof(buffer));
            if(recv(newSockFd, buffer, 100, 0) == -1) {
                  printf("\n Recieve Error! Exiting...");
                  exit(1);
            }

            fcntl(newSockFd, F_SETFL, O_NONBLOCK);

            printf("\nRecieved a request from client. Sending packets one
by one...");
            do {
                  if(windowCurrent != windowEnd) {
                  memset(&buffer, 0, sizeof(buffer));
                  itoa(windowCurrent, buffer);
                  send(newSockFd, buffer, 100, 0);
                  printf("\nPacket Sent: %d", windowCurrent);
                  windowCurrent++;
                  }

                  /*DEBUG*/ printf("\n**%d||%d**", windowCurrent,
windowEnd);
```

```
                  memset(&buffer, '\0', sizeof(buffer));
                  if(recv(newSockFd, buffer, 100, 0) != -1) {
                        if(buffer[0] == 'R') {
                              printf("\n** Received a RETRANSMIT packet.
Resending packet no. %c...", buffer[1]);
                              itoa((atoi(&buffer[1])), buffer);
                              send(newSockFd, buffer, 100, 0);
                        }
                        else if(buffer[0] == 'A') {
                              oldWindowStart = windowStart;
                              windowStart = atoi(&buffer[1]) + 1;
                              windowEnd += (windowStart - oldWindowStart);
                              printf("\n** Recieved ACK %c. Moving window
boundary.", buffer[1]);
                        }
                  }

                  sleep(1);
            } while(windowCurrent != 10);
      }
      else {
            printf("\nError in listening...");
            exit(1);
      }

      close(sockfd);
      close(newSockFd);
      printf("\nSending Complete. Sockets closed. Exiting...\n");

      return(0);
}
```

SAMPLE INPUT/OUTPUT

Server

Socket created successfully...

Starting up...

Binding completed successfully. Waiting for connection..

Recieved a request from client. Sending packets one by one...

Packet Sent: 1

2||4

Packet Sent: 2

3||4

Packet Sent: 3

4||4

4||4

** Received a RETRANSMIT packet. Resending packet no. 3...

4||4

** Recieved ACK 3. Moving window boundary.

Packet Sent: 4

5||7

Packet Sent: 5

6||7

Packet Sent: 6

7||7

7||7

** Recieved ACK 6. Moving window boundary.

Packet Sent: 7

8||10

Packet Sent: 8

9||10

Packet Sent: 9

10||10

Sending Complete. Sockets closed. Exiting...

Client

Socket Created...

Starting up...

Establishing Connection...

Connection Established!

Got packet: 1

Got packet: 2

Got packet: 3

*** Simulation: Packet data corrupted or incomplete.

*** Sending RETRANSMIT.

Got packet: 3

*** Packet Accepted -> Sending ACK

Got packet: 4

Got packet: 5

Got packet: 6

*** Packet Accepted -> Sending ACK

Got packet: 7

Got packet: 8

Got packet: 9

*** Packet Accepted -> Sending ACK

All packets recieved... Exiting.

11. SMTP USING UDP

About SMTP

SMTP is a relatively simple, text-based protocol, where one or more recipients of a message are specified and then the message text is transferred. It is a client-server protocol, where the client transmits an email message to the server. Either an end-user's email client, a.k.a MUA (Mail User Agent), or a relaying server's MTA (Mail Transfer Agent) can act as an SMTP *client.*

An email client knows the *outgoing mail* SMTP server from its configuration. A relaying server typically determines wich SMTP server to connect to by looking up the MX (Mail eXchange) DNS record for each recipient's domain name (the part of the email address to the right of the **at** (@) symbol). Conformant MTAs (not all) fall back to a simple A record in the case of an MX. Some current mail transfer agents will also use SRV records, a more general form of MX, though these are not widely adopted.

The SMTP client initates a TCP connection to server's port number 25 (unless overridden by configuration). It is quite easy to test an SMTP server using the telnet program. SMTP is a "push" protocol that does not allow one to "pull" messages from a remote server on demand. To do this a mail client must use POP3 or IMAP. Another SMTP server can trigger a delivery in SMTP using ETRN.

PROBLEM DESCRIPTION

Develop a program to simulate SMTP using UDP connection. The input to the program from the client side is the sender’s and receiver’s email id, which is of the form user@url.com. The content of the mail is also accepted from the user. The server validates the domain and the URL from which the mail is sent and displays the content of the mail.

FUNCTIONS AND DATA STRUCTURES

The data structures used are:

- A structure named sockaddr_in which can store the required information about a client or a server.
- A character array for storing the message to be communicated.

The functions used are:

- socket () – used to create a socket for communication.
- sendto () – used to send data.
- recvfrom () – used to receive data.
- bind () – used to bind the socket with the sockaddr_in object.

ALGORITHM

Client Side

1. Start
2. Create Socket
3. Fill up Socket Address
4. Get 'Helo' from user and send to server via the socket
5. Get other details such as destination email address, from email address and message body from the client and send to server via the socket
6. Receive acknowledgment from client
7. Stop

Server Side

1. Start

2. Create Socket
3. Fill up the socket address
4. Bind socket to port
5. Wait for 'Helo' from client and acknowledge it back
6. Receive other details from the client using recvfrom () function
7. Acknowledge back to server using sendto () function
8. Stop

CODE

Smtpclient.c

```
#include"sys/socket.h"
#include"netinet/in.h"
#include"stdio.h"
#include"string.h"
#include"stdlib.h"

int main()
{
      char buf[100];
      int k;
      int sock_desc;
      struct sockaddr_in client;

      memset(&client,0,sizeof(client));

      sock_desc=socket(AF_INET,SOCK_DGRAM,0);
      if(sock_desc==-1)
      {
            printf("\nError in socket creation");
            exit(1);
      }

      client.sin_family=AF_INET;
      client.sin_addr.s_addr=inet_addr("127.0.0.1");
      client.sin_port=3500;

      printf("\nMAIL TO : ");
      gets(buf);
      k=sendto(sock_desc,buf,sizeof(buf),0,(struct sockaddr
*)&client,sizeof(client));
      if(k==-1)
      {
            printf("\nError in sending");
            exit(1);
      }
      strcpy(buf,"\0");
      printf("\nFROM : ");
```

```
      gets(buf);
      k=sendto(sock_desc,buf,sizeof(buf),0,(struct sockaddr
*)&client,sizeof(client));
      if(k==-1)
      {
            printf("\nError in sending");
            exit(1);
      }

      strcpy(buf,"\0");
      printf("\nSUBJECT : ");
      gets(buf);
      k=sendto(sock_desc,buf,sizeof(buf),0,(struct sockaddr
*)&client,sizeof(client));
      if(k==-1)
      {
            printf("\nError in sending");
            exit(1);
      }
      strcpy(buf,"\0");
      printf("\nMSG BODY : ");
      while(strcmp(buf,".")!=0)
      {
            strcpy(buf,"\0");
            gets(buf);
            k=sendto(sock_desc,buf,sizeof(buf),0,(struct sockaddr
*)&client,sizeof(client));
            if(k==-1)
            {
                  printf("\nError in sending");
                  exit(1);
            }
      }
      close(sock_desc);
      return 0;
}
```

smtpserver.c

```
#include"sys/socket.h"
#include"netinet/in.h"
#include"stdio.h"
#include"string.h"
#include"stdlib.h"

int main()
{
      char buf[100],domain[20],snd[25];
      int k,i=0,j,m=0;
      socklen_t len;
      int sock_desc,temp_sock_desc;
      struct sockaddr_in server,client;

      memset(&server,0,sizeof(server));
      memset(&client,0,sizeof(client));

      sock_desc=socket(AF_INET,SOCK_DGRAM,0);
      if(sock_desc==-1)
      {
            printf("\nError in socket creation");
            exit(1);
      }
```

```
      server.sin_family=AF_INET;
      server.sin_addr.s_addr=INADDR_ANY;
      server.sin_port=3500;


      k=bind(sock_desc,(struct sockaddr*)&server,sizeof(server));
      if(k==-1)
      {
            printf("\nError in binding");
            exit(1);
      }

      len=sizeof(server);
        k=recvfrom(sock_desc,buf,sizeof(buf),0,(struct
sockaddr*)&server,&len);
      if(k==-1)
      {
            printf("\nError in receiving");
            exit(1);
      }
      strcpy(snd,buf);
      while(i<(strlen(buf)))
      {
            if(buf[i]=='@')
            {
                  for(j=i+1;j<strlen(buf);j++)
                  domain[m++]=buf[j];
                  break;
            }
            i++;
      }
      printf("Receiving Mail...");
      printf("\nDomain verified << %s >>",domain);

      len=sizeof(server);
        k=recvfrom(sock_desc,buf,sizeof(buf),0,(struct
sockaddr*)&server,&len);
        if(k==-1)
        {
                printf("\nError in receiving");
                exit(1);
        }
        printf("\nFROM: %s\n",buf);

      len=sizeof(server);
        k=recvfrom(sock_desc,buf,sizeof(buf),0,(struct
sockaddr*)&server,&len);
        if(k==-1)
        {
                printf("\nError in receiving");
                exit(1);
        }
        printf("\nSUBJECT: %s\n",buf);

      printf("\nMSG BODY: \n\t");
      len=sizeof(server);
        k=recvfrom(sock_desc,buf,sizeof(buf),0,(struct
sockaddr*)&server,&len);
        if(k==-1)
        {
                printf("\nError in receiving");
                exit(1);
        }
```

```
      while(strcmp(buf,".")!=0)
      {
            printf("%s\n\t",buf);
            len=sizeof(server);
                k=recvfrom(sock_desc,buf,sizeof(buf),0,(struct
sockaddr*)&server,&len);
              if(k==-1)
            {
                  printf("\nError in receiving");
                      exit(1);
            }
        }
      printf("\nMail received successfully from %s\n",snd);
      close(temp_sock_desc);
      exit(0);
      return 0;
}
```

SAMPLE INPUT AND OUTPUT

Client

Helo

Server acknowledged: 250 – Request command completed

MAIL FROM: <mail@example.com>

RCPT TO: <mail@linux.com>

MSGBODY: This is a sample email.

Sending the mail…

QUIT

Server Closed Successfully

Server

Client Send a Helo

Sending Helo Reply

MAIL FROM: mail@example.com

RCPT TO: mail@linux.com <<Domain Verified>>

MSGBODY: This is a sample email.

Received QUIT message from client.

Server Shutting Down

12. FTP USING TCP

About FTP

File Transfer Protocol (FTP), a standard Internet protocol, is the simplest way to exchange files between computers on the Internet., FTP is an application protocol that uses the Internet's TCP/IP protocols. FTP is commonly used to transfer Web page files from their creator to the computer that acts as their server for everyone on the Internet. It's also commonly used to download programs and other files to your computer from other servers. An FTP client may connect to an FTP server to manipulate files on that server.

The objectives of FTP, are:

1. To promote sharing of files (computer programs and/or data).
2. To encourage indirect or implicit use of remote computers
3. To shield a user from variations in file storage systems among different hosts
4. To transfer data reliably, and efficiently.

An FTP server defaults to listen on port 21for incoming connections from FTP clients. A connection to this port from the FTP Client forms the control stream on which commands are passed from the FTP client to the FTP server and on occasion from the FTP server to the FTP client. FTP uses out-of-band-control, which means it uses a separate connection for control and data. Thus, for the actual file transfer to take place, a different connection is required which is called the data stream. Port 21 for control (or program), port 20 for data.

PROBLEM DEFINITION

Creation of a simple simulator for performing FTP operations of LIST,LOAD ,STORE where the LIST operation gets the list of files present in the server and displays it in the client .The LOAD operation downloads a particular file from the server to the client and the STORE operation uploads a file from the client to the server.

FUNCTIONS AND DATA STRUCTURES USED

FTP uses the TCP protocol so all the functions and data structures used in TCP are present in FTP. The functions and data structures specific to FTP are as follows:

- <sys/stat.h>

  This header file defines the structure of the data returned by the function *stat()*. It contains the status of the various features allocated for an object of type stat.for example the size of file in blocks:

  struct stat obj;
  obj st_size;

- <sys/sendfile.h>

  copy data directly from one file descriptor to another.

  sendfile():

  Syntax:
  sendfile(temp_descr,filehandle,NULL,obj.st_size);
  used to send an file from the server to the client.

- system("ls -al>list.txt");

  this function is used to list out the files present, along with the attributes into the file list.txt

- system("cat list2.txt");

  this function is used to copy the details listed out onto the file named list2.txt

ALGORITHM

Client

For LIST operation::

1. Open a file 'list2.txt ' in the read/write or create mode
2. Create a socket and notify the server on which operation is to be performed, by using the send command
3. From the server the size of the file as well as the file is received in the variable length and fil frespectively
4. Write the received details into the filehandle which is the descriptor for the opened file list2.txt
5. Now write the contents into list2.txt

For LOAD operation::

1. Open a file 'list3.txt ' in the read/write or create mode
2. Create a socket and notify the server that LOAD is to be performed, by using the send command. Along with it send the name of the file to be downloaded .
3. From the server the size of the file as well as the file is received in the variable length and fil frespectively
4. Write the received details into the filehandle which is the descriptor for the opened file list3.txt
5. Now write the contents into list3.txt

Server

For LIST operation::

1. Create a socket and bind to the port which is given as input from the user
2. Listen and accept the request from the client
3. For the LIST operation , using the command system("ls -al>list.txt") get the details of all the files along with its attributes into the file named list.txt
4. Get the size of the file in blocks using the commands :

   stat("list.txt",&obj);

sprintf(length,"%d",(int)obj.st_size);

5. Send the size to the client .
6. Using the function sendfile() send the contents of list.txt to the client

For LOAD operation::

1. For the LOAD operation the filename to be downloaded is specified by the client
2. Get the size of the file in blocks using the commands :

   stat(filename,&obj);

   sprintf(length,"%d",(int)obj.st_size);
3. Send the size to the client .
4. Using the function sendfile() send the contents to the client

CODE

Client side:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include<sys/stat.h>
#include<sys/sendfile.h>
#include <fcntl.h>



int main(int arc,char *argv[])
{
  struct sockaddr_in address;
  int sock_descr;
  int k,choice,len,l,c=0,filehandle;
  char buf[100],length[100],fil[100],filename[100];

  memset(&address,0,sizeof(address));

  sock_descr=socket(AF_INET,SOCK_STREAM,0);
  if(sock_descr==-1)
    {
      printf("socket creation failed");
      exit(1);
    }

  address.sin_family=AF_INET;
  address.sin_port=atoi(argv[1]);
  address.sin_addr.s_addr=INADDR_ANY;

  k=connect(sock_descr,(struct sockaddr*)&address,sizeof(address));
  if(k==-1)
```

```
    {
      printf("connecting failed");
      exit(1);
    }

  while(1)
  {
   printf("\n1:LIST\n2:LOAD\n3:Exit : ");
   scanf("%d",&choice);

   switch(choice){
   case 1:
          filehandle=open("list2.txt",O_RDWR|O_CREAT,0700);
          strcpy(buf,"LIST");
          k=send(sock_descr,buf,strlen(buf),0);
          if(k==-1)
          {
            printf("send failed");
            exit(1);
          }

          k=recv(sock_descr,length,100,0);
          if(k==-1)
          {
            printf("receive failed");
            exit(1);
          }

          len=atoi(length);

          while(c<len)
          {
           l=read(sock_descr,fil,100);
           if(l==0)
           break;
           write(filehandle,fil,100);
           c+=l;
          }
          system("cat list2.txt");
          close(filehandle);
          break;

   case 2:
          filehandle=open("list3.txt",O_RDWR|O_CREAT,0700);
          strcpy(buf,"LOAD");
          printf("Enter filename : ");
          scanf("%s",filename);
          strcat(buf,filename);
          k=send(sock_descr,buf,strlen(buf),0);
          if(k==-1)
          {
            printf("send failed");
            exit(1);
          }
          k=recv(sock_descr,length,100,0);
          if(k==-1)
          {
            printf("receive failed");
            exit(1);
          }

          len=atoi(length);
          c=0;
```

```
          while(c<len)
          {
           l=read(sock_descr,fil,strlen(fil));
           if(l==0)
           break;
           write(filehandle,fil,strlen(fil));
           c+=l;
          }
          system("cat list3.txt");
          break;
    case 3:
        exit(0);

    default:
             break;


   }
  }
exit(0);
return 0;
}
```

Server side:

```
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/sendfile.h>
#include <fcntl.h>

int main(int argc,char *argv[])
{
  struct sockaddr_in address1,address2;
  struct stat obj;
  int sock_descr,temp_descr;
  socklen_t len;
  char buf[100],choice[10],length[100],filename[100];
  int k,i;
  int filehandle;

  memset(&address1,0,sizeof(address1));
  memset(&address2,0,sizeof(address2));

  sock_descr=socket(AF_INET,SOCK_STREAM,0);
  if(sock_descr==-1)
    {
      printf("socket creation failed");
      exit(1);
    }

  address1.sin_family=AF_INET;
  address1.sin_port=atoi(argv[1]);
  address1.sin_addr.s_addr=INADDR_ANY;//inet_addr(argv[1]);

  k=bind(sock_descr,(struct sockaddr*)&address1,sizeof(address1));
  if(k==-1)
    {
```

```
      printf("binding error");
      exit(1);
    }

  k=listen(sock_descr,5);
  if(k==-1)
    {
      printf("listen failed");
      exit(1);
    }

  len=sizeof(address2);
  temp_descr=accept(sock_descr,(struct sockaddr*)&address2,&len);
  if(temp_descr==-1)
    {
      printf("temp: socket creation failed");
      exit(1);
    }

   while(1)
   {
    k=recv(temp_descr,buf,100,0);
    if(k==-1)
    {
      printf("receive failed");
      exit(1);
    }

   for(i=0;i<4;i++)
      choice[i]=buf[i];
   choice[4]='\0';
   printf("\n%s",choice);
   if(strcmp(choice,"LIST")==0)
   {

    system("ls -al>list.txt");
    filehandle=open("list.txt",O_RDONLY);//
    stat("list.txt",&obj);//
    sprintf(length,"%d",(int)obj.st_size);
//sending size of file
   k=send(temp_descr,length,strlen(length),0);
    if(k==-1)
    {
      printf("send failed");
      exit(1);
    }
//send entire file
    k=sendfile(temp_descr,filehandle,NULL,obj.st_size);
    if(k==-1)

    {
      printf("file sendingfailed");
      exit(1);
    }
   }
   if(strcmp(choice,"LOAD")==0){
   //strcpy(filename,"./");
   //strcat(filename,buf+4);
   strcpy(filename,buf+4);
   stat(filename,&obj);
   filehandle=open(filename,O_RDONLY);
   if(filehandle==-1)
   {
```

```
    printf("NO SUCH FILE\n");
    //exit(1);
   }
   sprintf(length,"%d",(int)obj.st_size);
   printf("\n%s\n",length);
   k=send(temp_descr,length,strlen(length),0);
   if(k==-1)
   {
      printf("send failed");
      exit(1);
   }
   sendfile(temp_descr,filehandle,NULL,obj.st_size);
   if(k==-1)
    {
      printf("file sendingfailed");
      exit(1);
    }
   }

   exit(0);
  }
exit(0);
return 0;
 }
```

SAMPLE INPUT/OUTPUT

LIST
17:50 ..
-rwx------ 1 sample sample 6874 2009-05-09 17:50 1bitclient
-rwx------ 1 sample sample 2396 2009-05-09 17:50 1bitclient.c
-rwx------ 1 sample sample 6833 2009-05-09 17:50 1bitserver
-rwx------ 1 sample sample 2653 2009-05-09 17:50 1bitserver.c
-rwx------ 1 sample sample 6874 2009-05-09 17:50 1c
-rwx------ 1 sample sample 6833 2009-05-09 17:50 1s
-rwxr-xr-x 1 sample sample 9417 2009-05-09 17:52 a.out
-rwx------ 1 sample sample 7804 2009-05-09 17:50 ban
-rwx------ 1 sample sample 7804 2009-05-09 17:50 bank
-rwx------ 1 sample sample 1533 2009-05-09 17:50 bankers.c
-rwx------ 1 sample sample 7804 2009-05-09 17:50 bankexit
-rwxr-xr-x 1 sample sample 9117 2009-05-09 17:52 c
LOAD
The chosen file has been downloaded.
STORE
The chosen file has been uploaded

VIRTUAL FILE SYSTEM

The main data item in any Unix-like system is the “file”, and a unique pathname identifies each file within a running system. Every file appears like any other file in the way is is accessed and modified: the same system calls and the same user commands apply to every file. This applies independently of both the physical medium that holds information and the way information is laid out on the medium. Abstraction from the physical storage of information is accomplished by dispatching data transfer to different device drivers; abstraction from the information layout is obtained in Linux through the VFS implementation.

The Unix way

Linux looks at its file-system in the way Unix does: it adopts the concepts of super-block, inode, directory and file in the way Unix uses them. The tree of files that can be accessed at any time is determined by how the different parts are assembled together, each part being a partition of the hard driver or another physical storage device that is ``mounted'' to the system.

While the reader is assumed to be confident with the idea of mounting a file-system, I'd better detail the concepts of super-block, inode, directory and file.

- The **super-block** owes its name to its historical heritage, from when the first data block of a disk or partition was used to hold meta-information about the partition itself. The super-block is now detached from the concept of data block, but still is the data structure that holds information about each mounted file-system. The actual data structure in Linux is called struct super_block and hosts various housekeeping information, like mount flags, mount time and device blocksize. The 2.0 kernel keeps a static array of such structures to handle up to 64 mounted file-systems.
- An **inode** is associated to each file. Such an ``index node'' encloses all the information about a named file except its name and its actual data. The owner, group, permissions and access times for a file are stored in its inode, as well as the size of the data it holds, the number of links and other information. The

idea of detaching file information from filename and data is what allows to implement hard-links -- and to use the `dot' and `dot-dot' notations for directories without any need to treat them specially. An inode is described in the kernel by a struct inode.

- The **directory** is a file that associates inodes to filenames. The kernel has no special data strcture to represent a directory, which is treated like a normal file in most situations. Functions specific to each filesystem-type are used to read and modify the contents of a directory independently of the actual layout of its data.
- The **file** itself is something that is associated to an inode. Usually files are data areas, but they can also be directories, devices, FIFO's or sockets. An ``open file'' is described in the Linux kernel by a struct file item; the structure encloses a pointer to the inode representing the file. file structures are created by system calls like *open*, *pipe* and *socket*, and are shared by father and child across *fork*.

13. DISK STATUS USAGE REPORT

PROBLEM DESCRIPTION

Program to find the status information of a file system

FUNCTION AND DATASTRUCTURES USED

struct statvfs : data structure containing detailed file system information

The implementation makes use of the following fields in statvfs

f_bsize : File system block size.

f_blocks : Total number of blocks on the file system

f_bfree : Total number of free blocks.

statvfs()

syntax : int statvfs(const char *path, struct statvfs *buf);

The function obtains information about the file system containing the file referred to by the specified path name.

ALGORITHM

1. Define an object Data of the structure statvfs.
2. The path of the filename whose status is to be found is stored in variable Path.
3. Using the function statvfs() status of the file system is obtained in the object data.
4. Now the required status info can be retrieved using Data.f_bsize,Data.f_blocks etc.
5. Display the block size,free blocks size and used blocks in MB.
6. Stop

CODE

```
#include <stdio.h>
#include <sys/statvfs.h>
#include <string.h>
float a ;
int main( int argc, char* argv[] )
{
        struct statvfs Data;
        char Path[128];
        float n,k,l;
      int i;
        if( argc < 2 ) {
                printf("Usage, <executable> DEVICE0 ..... DEVICEX\n");
                return(2);
        }

        for( i = 1 ; i<argc; i++ ) {
                strcpy(Path, argv[i]);
                if((statvfs(Path,&Data)) < 0 ) {
                        printf("Failed to stat %s:\n", Path);


                } else {
                n=((float)Data.f_bsize/(1024.0*1024.0));

                        k=((float)Data.f_blocks*n);


                        l=((float)Data.f_bfree*n);

                        a=k-l;
                        printf("Disk %s: \n", Path);

                  printf("\tblock size in MB: %f\n", n);
                  printf("\tBlock size in bytes: %lu\n",Data.f_bsize);
                  printf("\ttotal size in
bytes:%lu\n",Data.f_blocks*Data.f_bsize);
                  printf("\ttotal size in MB: %f\n", k);
                        printf("\tfree blocks in MB: %f\n", l);
                  printf("\tfree blocks in bytes:
%lu\n",Data.f_bfree*Data.f_bsize);
                        printf("\tused blocks in MB: %f\n", a);

                  printf("\tused blocks in bytes:
%lu\n",(Data.f_blocks*Data.f_bsize)-(Data.f_bfree*Data.f_bsize));
             }
        }
}
```

SAMPLE INPUT &OUTPUT

INPUT:

/root

OUTPUT:

Disk /root:

block size in MB: 0.003906

Block size in bytes: 4096

total size in bytes:4088434688

total size in MB: 3899.035156

free blocks in MB: 1644.042969

free blocks in bytes: 1723904000

used blocks in MB: 2254.992188

used blocks in bytes: 2364530688

REMOTE PROCEDURE CALL

What Is RPC

RPC is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

RPC makes the client/server model of computing more powerful and easier to program. When combined with the ONC RPCGEN protocol compiler clients transparently make remote calls through a local procedure interface.

How RPC Works

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. Figure shows the flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

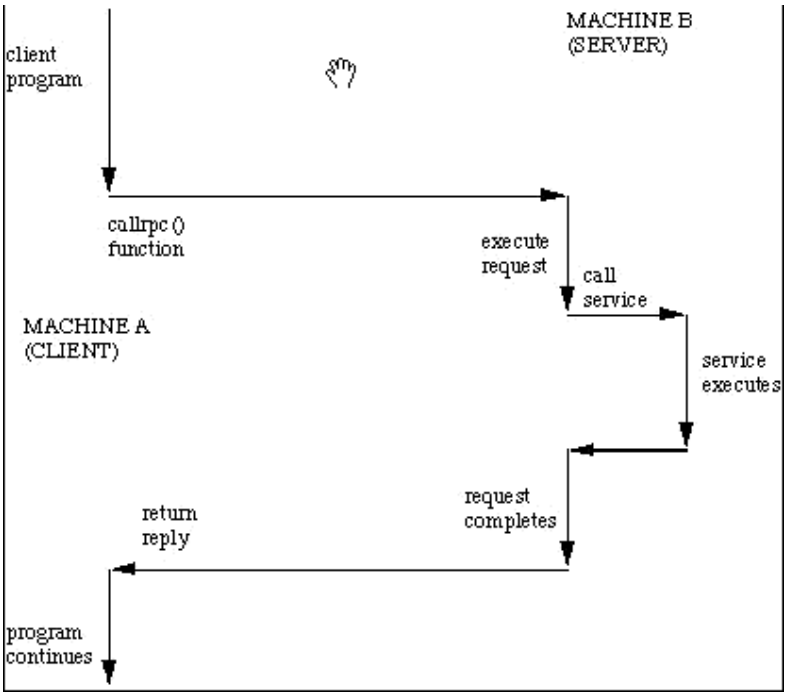


Fig Remote Procedure Calling Mechanism

A remote procedure is uniquely identified by the triple: (program number, version number, procedure number) The program number identifies a group of related remote procedures, each of which has a unique procedure number. A program may consist of one or more versions. Each version consists of a collection of procedures which are available to be called remotely.

Defining the Protocol

The easiest way to define and generate the protocol is to use a protocol complier such as rpcgen. For the protocol you must identify the name of the service procedures, and data types of parameters and return arguments.The protocol compiler reads a definitio and automatically generates client and server stubs. rpcgen uses its own language (RPC language or RPCL) which looks very similar to preprocessor directives. rpcgen exists as a standalone executable compiler that reads special files denoted by a .x prefix.

So to compile a RPCL file you simply do rpcgen rpcprog.x

This will generate possibly four files:

- rpcprog_clnt.c -- the client stub
- rpcprog_svc.c -- the server stub
- rpcprog_xdr.c -- If necessary XDR (external data representation) filters
- rpcprog.h -- the header file needed for any XDR filters.

The external data representation (XDR) is an data abstraction needed for machine independent communication. The client and server need not be machines of the same type.

External Data Representation

XDR is a standard for the description and encoding of data. It is useful for transferring data between different computer architectures, and has been used to communicate data between such diverse machines as the SUN WORKSTATION*, VAX*, IBM-PC*, and Cray*. XDR fits into the ISO presentation layer, and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference between these two is that XDR uses implicit typing, while X.409 uses explicit typing.

XDR uses a language to describe data formats. The language can only be used only to describe data; it is not a programming language. This language allows one to describe intricate data formats in a concise manner. The alternative of using graphical representations (itself an informal language) quickly becomes incomprehensible when faced with complexity. The XDR language itself is similar to the C language, just as Courier [4] is similar to Mesa. Protocols such as ONC RPC (Remote Procedure Call) and the NFS* (Network File System) use XDR to describe the format of their data.

The XDR standard makes the following assumption: that bytes (or octets) are portable, where a byte is defined to be 8 bits of data. A given hardware device should encode the bytes onto the various media in such a way that other hardware devices may decode the bytes without loss of meaning. For example, the Ethernet* standard suggests that bytes be encoded in "little-endian" style [2], or least significant bit first.

What is rpcgen

The rpcgen tool generates remote program interface modules. It compiles source code written in the RPC Language. RPC Language is similar in syntax and structure to C. rpcgen produces one or more C language source modules, which are then compiled by a C compiler.

The default output of rpcgen is:

- A header file of definitions common to the server and the client
- A set of XDR routines that translate each data type defined in the header file
- A stub program for the server
- A stub program for the client

rpcgen can optionally generate (although we ***do not*** consider these issues here -- see man pages or receommended reading):

- Various transports
- A time-out for servers
- Server stubs that are MT safe
- Server stubs that are not main programs
- C-style arguments passing ANSI C-compliant code
- An RPC dispatch table that checks authorizations and invokes service routines

rpcgen significantly reduces the development time that would otherwise be spent developing low-level routines. Handwritten routines link easily with the rpcgen output. The “finger.x” file is passed to rpcgen which is given by:

```
struct finger_out{
      char message[1024];
};

program FINGER{
      version FINGER_VERSION{
            finger_out MyFinger() = 1;
      } = 1;
} = 0x21230000;
```

14. FINGER UTILITY USING RPC

PROBLEM SPECIFICATION

The finger server at the remote side is invoked using rpc and the options to finger utility are accepted from user and passess on to the remote system using TCP connection.

FUNCTIONS AND DATASTRUCTURES USED

RPCGEN

Rpcgen-an RPC protocol compiler

Rpcgen is a tool that generates C code to implement an RPC protocol. The input to rpcgen is a language similar to C known as, rpc language.

ALGORITHM

1. Start
2. Client makes a TCP connection to the server (default port 79) and transfers parameters.
3. If input parameter is {c} only, print list of all users online and some additional information like
   - terminal location
   - home location
   - idle time [no. of minutes since last typed input or since last job activity]
4. If input parameter is {u}{c}, only print above mentioned information of a specified user {u} along with the following information:
   - login information
   - amount of time after being logged in

- new mail information

5. If input is {u}{H}{c} where {h} is host H, then transfer the finger request to the computer on behalf of this host.

6. Stop

CODE

```
struct finger_out{
      char message[1024];
};

program FINGER{
      version FINGER_VERSION{
            finger_out MyFinger() = 1;
      } = 1;
} = 0x21230000;
```

Client

```
#include <rpc/rpc.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "finger.h"

void err_quit(char *msg)
{
      printf("%s\n", msg);
      exit(1);
}

int main(int argc, char* argv[])
{
      CLIENT *c1;
      finger_out *outp;

      if(argc!=2)
            err_quit("usage: client <hostname>");

      c1 = clnt_create(argv[1], FINGER, FINGER_VERSION, "tcp");

      if( (outp=myfinger_1(NULL, c1))==NULL )
            err_quit(clnt_sperror(c1, argv[1]));

      printf("result: %s\n", outp->message);
      exit(0);
}
```

Server

```
#include <rpc/rpc.h>
#include <stdio.h>
#include <stdlib.h>
#include "finger.h"
```

```
finger_out* myfinger_1_svc(void *dummy, struct svc_req *rqstp)
{
      static finger_out fo;
      char buffer[1024];
      system("finger > result.txt");
      FILE *fp = fopen("result.txt", "r");
      int i=0;
      while( !feof(fp) ){
            buffer[i++] = fgetc(fp);
      }
      buffer[i] = '\0';
      strcpy(fo.message, buffer);
      system("rm -f result.txt");
      return &fo;
}
```

SAMPLE INPUT/OUTPUT

Open the executable server file
Open the executable client file with arguments
./client 127.0.0.1 5000

root @Shark:~/Desktop/RPC$ ./client 127.0.0.1 5000
Login Name Tty Idle Login Time Office Office Phone
root root tty7 May 9 15:52 (:0)
root root pts/0 May 9 16:06 (:0.0)
root @Shark:~/Desktop/RPC$