FINAL YEAR PROJECT REPORT ON

IMPLEMENTATION OF FILE TRANSFER PROTOCOL IN CLIENT SERVER MODEL

Submitted in Partial fulfillment for the Requirement of
Degree of Bachelor of Engineering in Computer Engineering

By:

NEELABH SAXENA 2K1/COE/035
NISHANT GUPTA 2K1/COE/037
PARAG MEHRA 2K1/COE/038

Under the guidance of
Dr GOLDIE GABRANI

Asst Prof
Department of Computer Engineering
Delhi College of Engineering

DEPARTMENT OF COMPUTER ENGINEERING
DELHI COLLEGE OF ENGINEERING
UNIVERSITY OF DELHI, DELHI

2001-05

CONTENTS

CERTIFICATE

This is to certify that the dissertation entitled "Implementation of File Transfer Protocol in Client- Server model" being submitted by Neelabh Saxena, Nishant Gupta and Parag Mehra towards the partial fulfillment of the requirement for the award of the degree of Bachelor of Engineering in the Department of Computer Engineering at the Delhi College of Engineering, Delhi is a bona fide record of their work carried out under our supervision.

Further it is also certified that the matter and results in this dissertation are original and have not been submitted for any degree or diploma in any other college to the best of our knowledge.

Dated: May 25, 2005

Dr Goldie Gabrani
Supervisor
Deptt of Computer Engineering
Delhi College of Engineering
Delhi

Prof D Roy Choudhry
Head
Deptt of Computer Engineering
Delhi College of Engineering
Delhi

ACKNOWLEDGEMENT

We wish to express our hearty and sincere gratitude and indebtness to Dr Goldie Gabrani, Department of Computer Engineering, Delhi College of Engineering, Delhi for her invaluable guidance and wholehearted cooperation .She has been a major source of inspiration throughout the project as she has not only guided us through the project but encouraged us to solve the problems that arose during this project.

We would also like to thank Prof D Roy Choudhury for his able guidance, valuable suggestions, constant encouragement and untiring efforts at every stage of our project, without which it would have been impossible to bring this work to completion.

The conclusion of our project with combined achievements & failures was a great moment of pleasure for us. This project is a result of the contribution of many people. They have enriched our knowledge & made suggestions based on their experience. We would like to thank the entire faculty of Computer Science department for their co-operation and tremendous help rendered in numerous ways for the successful completion of this work.

Most importantly, we would like to thank our parents, friends and colleagues for invaluable suggestions and critical review of our project.

Neelabh Saxena
Nishant Gupta
Parag Mehra

ABSTRACT

The FTP is basic and common service to exchange files between computers, namely hosts, over TCP/IP networks e.g., private networks or Internet. The FTP supports file transmission and character code conversion when exchanging text or binary files. The use of FTP is effective in exchanging or distributing of large volume of data over private networks and/or the Internet. FTP uses TCP as a transport protocol to provide reliable end-to-end connections. As it is necessary to log into the remote host, the user must have a user name and a password to access files and directories. The user who initiates the connection assumes the client function, while the server function is provided by the remote host.

The thesis starts with an introduction to the basic networking and the OSI model, giving background to to grasp the rest of the contents. Then next is the explanantion of the Client Server Model.Follwing this is a report on File Transfer Protocol. It explains the various functions that the File Transfer Protocol performs. FTP can be used to download the files from a server node to the client node or upload a file from a client node to the server node. FTP can also be used to change the current working directory of the server/client and also can delete files on the server station. After this there are details of socket programming. Socket programming has been used extentensively throughtout the project. The various functions that have been frequently used like socket ( ), bind ( ), accept ( ), send ( ) etc have been explained along with all the possible forms that these functions can take.

Next is the implementation of such a File Transfer Protocol using the programming language C. The functions being implemented in the the code are open, quit, exit, username, password, download, upload, change directory, show present working directory, list the contents of the current working directory. The code can be run on a Linux platform.

In the next section, File Transfer Protocol is implemented using JAVA. The functions being implemented are Connect, Disconnect, Download, Upload and Delete. This code is platform independent and thus can be run on a variety of platforms like Windows and Linux. This gives its users an added advantage of platform independence.

The C and Java implementations are followed by their respective possible outputs and errors. Also a brief comparison between the the codes is also done highlighting the major advantages and disadvantages of using C and JAVA.

INTRODUCTION

HISTORY-

Computer networks have revolutionized our use of computers. They pervade our everyday life, from automated teller machines, to airline reservation system, to e-mail services, to electronic mail services, to electronic boards. There are many reasons for the explosive growth in computer networks.

The proliferation of personal computers and work stations during 1980`s helped fuel the interest and need for networks. Computer networks used to be expensive and were restricted to large universities, govt. research sites and large corporations. Technology has greatly reduced the cost of establishing computer networks, and networks are now found in organizations of every size.

Many computer manufacturers now package networking software as part of the basic operating system. Networking software is no longer regarded as an add-on that only a few customers will want. It is now considered essential as a text editor. We are in an information age and computer networks are becoming an integral part in the dissemination of the information.

Computer systems used to be stand-alone entities. Each computer was self contained and had all the peripherals and software required to do a particular job. If a particular feature was needed, such as line printer output, a line printer was attached to the system. If large amount of disk storage were needed, disk was added to the system. What helped change this is the realization that computers and their users need to share information and resources.

Information sharing can be electronic mail or file transfer. Resource sharing can involve accessing a peripheral on another system. Today computers can be connected together by various electronic techniques called networks. A network can be as simple as 2 personal computers connected together using a 1200 baud modem, or as complex as the tcp/ip internet, which connects over 150000 systems together. Some typical network applications are.

- Exchange e-mail with users on other computers.

- Exchange files between systems. For many applications it is just as easy to distribute the application electronically, instead of mailing disketts or magnetic tapes. File transfer across network also provides faster delivery.

- Share peripheral devices. A large push towards the sharing of peripheral devices has come from the personal computer and workstation market, since often the cost of peripheral can exceed the cost of computer. In an organization with many pc`s or workstations, sharing peripheral make sense.

- Execute a program on another computer. There are cases where some other computer is better suited to run a particular program. This is often the case with programs that require special features, such as parallel processing or vast amount of storage

- Remote login. If a network connects two computers, we should be able to login to one from another. It is usually easier to connect computers together using a network and provide a remote login application, than to connect every terminal in an organization to every computer.

LAYERING-

Given a particular task that we want to solve, such as providing a way to exchange files between two computers that are connected with a network, we divide the task into pieces and solve each piece. In the end we connect the pieces back together to form a final solution. We could write a single monolithic system to solve a problem, but experience has shown that solving the problem in pieces leads to a better, and more extensible solution.
It is possible that part of the solution developed for a file transfer program can also be used for a remote printing program. Also, if we`re writing the file transfer program assuming the computers are connected with an Ethernet, it could turn out that part of this program is usable for computers connected with a leased telephone line.

In the context of networking, this is called “layering”. We divide the communication problem into pieces and let each layer concentrate on providing a particular function. Well-defined interfaces are provided between the layers.

OSI MODEL-

The starting point for describing the layers in the network is the international standards organization open system interconnection model for computer communications. This is a 7-layer model shown in fig.

7 Application
6 Presentation
5 Session
4 Transport
3 Network
2 Data link
1 Physical

The OSI model provides a detailed standard for describing a network. Most computer networks today are described using OSI model. The transport layer is important because it is the lowest of the seven layers that provide reliable data transfer between two systems. The layer above transport layer can assume they have an error free communication path. Details such as sequence control, error detection, retransmission and flow control are handles by the transport layer and layers below it.

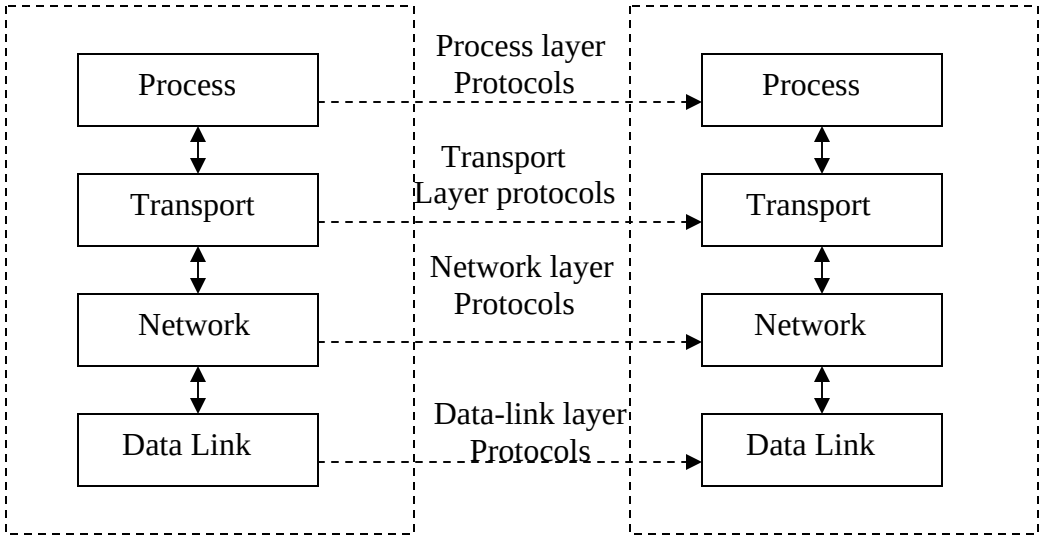
PROCESSES-

A fundamental entity in a computer network is a process. A process is a program that is being executed by the computers operating system. When we say that two computers are communicating with each other, we mean two processes, one running on each computer, are in communication with each other.

A Simplified Model-

The actual connection between two systems is between the two data-link layers. Although it appears to the two processes that they are communicating with each other, the actual data flows from the process layer, down to the transport layer, down to the network layer, down to the data link layer, up through the network layer, then the transport layer to other processes.

Layering leads to the definitions of protocols at each layer. Even though physical communication takes place at the lowest layer, protocol exist at every layer .The protocols which are used at a given layer are also called peer-to-peer protocols to reiterate that they are used between two entities at the same layer.
Since data always flows between a given layer and the layer immediately above it or immediately below it, the interface between the layers are also important for network programming



Simplified 4-layer model

Client-Server Model

The standard model for network application is the client –server model. A server is a process that is waiting to be contacted by a client process so that the server can do something for the client.

• The server process is started on some computer systems. It initializes itself then goes to sleep waiting for a client process to contact it requesting some service.

• The client process is started, either on the same system or on another system that is connected to the server system with a network. An interactive user entering a command to a time-sharing system often initiates client process. The client process sends a request across the network to the server requesting service of some form. Some examples of type of service that server can provide:

- Return the time-of-day to the client,
- Print a file on a printer for the client,
- Read or write a file on the server's system for the client,
- Allow the client to login to the server's system,
- Execute a command for the client on the server's system.

• When the server process has finished providing its services to the client, the server goes back to sleep, waiting for the next client request to arrive.

We can further divide the server's processes into two types:

1. When a client's request can be handled by the server in a known, short amount of time, the server process handles the request itself. We call these *iterative servers.*

2. When the amount of time to service a request depends on the request itself, the server typically handles it in concurrent fashion. These are called *concurrent servers.*

File Transfer Protocol (FTP)

In this section, abstractional mechanism of FTP is illustrated.

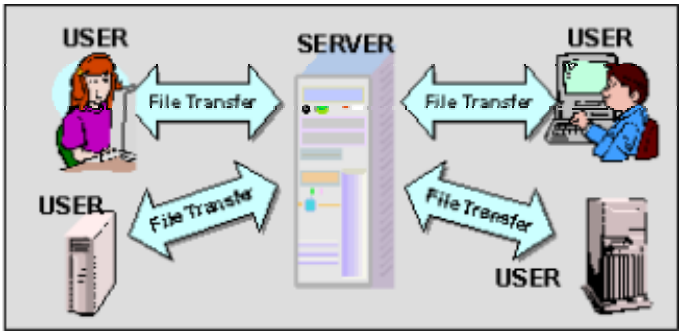
What is the FTP

The FTP is basic and common service to exchange files between computers, namely hosts, over TCP/IP networks e.g., private networks or Internet. FTP is a *standard protocol* with STD Number 9. Its status is *recommended*. It is described in *RFC 959 - File Transfer Protocol (FTP).*

Copying files from one machine to another is one of the most frequently used operations. The data transfer between client and server can be in either direction. The client may send a file to the server machine. It may also request a file from this server. To access remote files, the user must identify himself to the server. At this point the server is responsible for authenticating the client before it allows the file transfer.

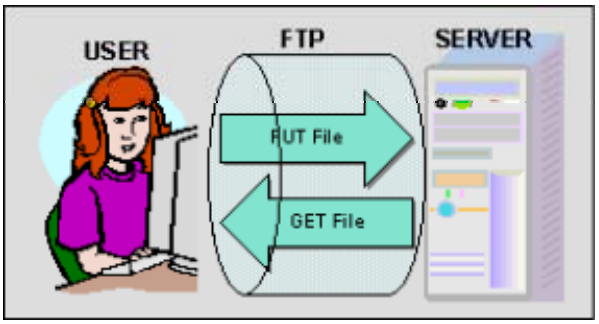
From an FTP user's point of view, the link is connection-oriented. In other words, it is necessary to have both hosts up and running TCP/IP to establish a file transfer.

The use of FTP is effective in exchanging or distributing of large volume of data over private networks and/or the Internet. A structural outline of FTP service is illustrated in Figure



Structural outline of FTP service

Basically, FTP is defined in the RFC959 as a communication protocol between *Server* and *User* for exchanging files. The FTP Server stores files to be exchanged. Users, who want to exchange files, will login to the server and PUT/GET files to/from the server (Figure). A User may be a person or a process on behalf of a person wishing to obtain file transfer service.

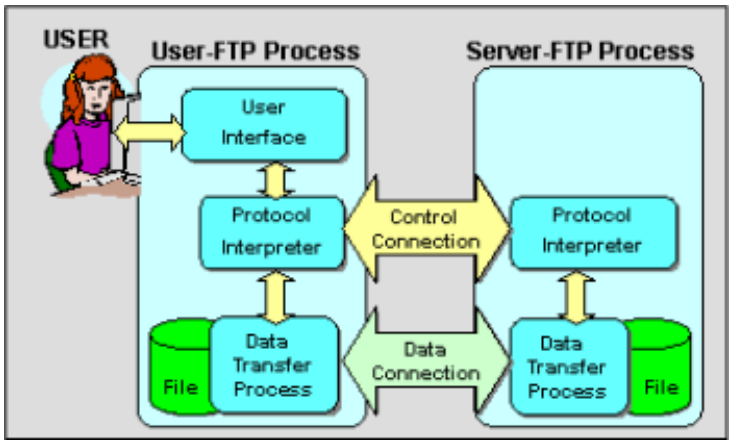


The File Transfer Protocol

To exchange files by FTP, establishing an FTP server is essential (or you could outsourcing FTP service if your budget allows). Although there is a slight difference between FTP over the Internet and FTP over the GTS in need for a proxy or staging function between an FTP server and data source, essentials of establishing the FTP server are common to both.

How FTP works

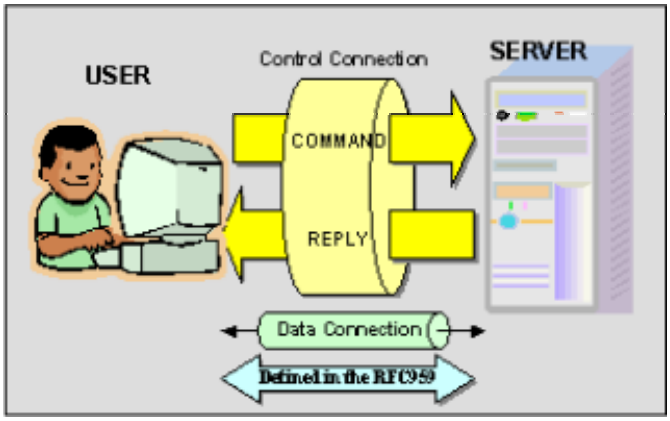
The File Transfer Protocol (FTP) is defined by the RFC959 based on the FTP Model illustrated in the Figure 2-3. The FTP uses two TCP/IP connections between User and Server, i.e., *control connection* and *data connection*. The control connection manages and controls the Server to transfer files between the Server and User through the data connection.



The FTP Model

Control Connection

The control connection is a full duplex communication path between server and user for exchanging commands and replies (see Figure). The FTP uses the Telnet protocol, defined by the RFC854, for the control connection. An FTP Server passively waits for the establishment of control connection, initiated by User, to the tcp port 21 of the Server. Once the control connection is established, the Server replies a ready message and wait for login by accepting *username* and *password* for authentication according to the replies from the Server. After recognizing the legal login of the user, the Server replies login message and wait for commands for further operation.



Control Connection

(1) FTP Command

An FTP command string is a command word may followed by a parameter string and terminated by line delimiter (crlf). Each command word is a word of three or four capital letters. Format of some FTP command strings are shown below.

USER <SP> <username> <CRLF>
PASS <SP> <password> <CRLF>
QUIT <CRLF>
PORT <SP> <host-port> <CRLF>
PASV <CRLF>
TYPE <SP> <type-code> <CRLF>
RETR <SP> <pathname> <CRLF>
STOR <SP> <pathname> <CRLF>
ABOR <CRLF>
LIST [<SP> <pathname>] <CRLF>
HELP [<SP> <string>] <CRLF>
NOOP <CRLF>

(2) FTP Reply

Each reply message send from FTP server always consists of a three digit number (reply code; transmitted as three numeric characters, xyz) followed by text string. There are five values for the first digit of the reply code:

1yz Positive Preliminary reply
2yz Positive Completion reply
3yz Positive Intermediate reply
4yz Transient Negative Completion reply
5yz Permanent Negative Completion reply

The second digit encodes function groupings as:

x0z Syntax
x1z Information
x2z Connections
x3z Authentication and accounting
x5z File system

The third digit gives a finer gradation of meaning in each of the function categories, specified by the second digit.

(3) An Example

The example below shows an FTP operation on a Linux platform using traditional FTP client with character based user interface, where the red texts denote inputs typed by user, the green for reply or prompt messages from the client software, and the blue for replies from the Server.
No FTP command, sent to the Server from the client software, are appeared on the CRT, but "USER *username*","PASS *xxxxxx*", and "QUIT" commands are presumed to be issued in this case.

An example of character based user interface:

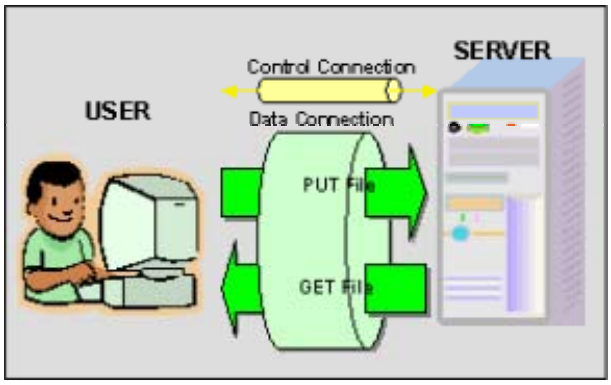
$ ftp *servername* ; user: start FTP client and connect *servername*
Connected to *servername*. ; client: connection established
220 *servername* FTP server ready. ; server: connection acknowledged and wait for login
Name (*servername*:*defaultusername*):*username* ; user: send "USER *username*" command
331 Password required for *username*. ; server: waits password for *username*
Password:*xxxxxx(not displayed)* ; user: send "PASS *xxxxxx*" command
230 User *username* logged in. ; server: accepted login of the user *username*

```
Remote system type is UNIX.                    ; client: reports current status
Using binary mode to transfer files.
ftp> bye                                       ; user:  send "QUIT" command
221 Goodbye.                                   ; server:  QUIT acknowledged
$
```

Data Connection

Data connection is a full duplex connection over which data (file) are transferred between Server and User by a specified type and mode. The data connection is established temporally for data or file transfer on demand (see Figure). Through the data connection, not only content of files to be transferred but also other data or messages, as

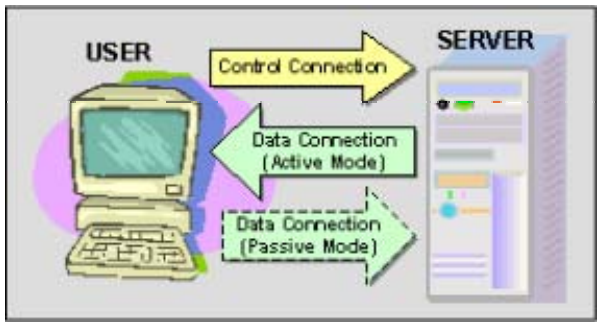
the result of FTP command execution, e.g., directory listing, help messages, are transferred.



Data Connection

Active mode and Passive mode

Normally, data connection is actively initiated by Server side toward User, with no regard to the direction of the transfer, and User passively wait for establishment of connection (*active* or *normal* mode). But in some cases, mainly at User side, initiation of connection from outer network is disallowed due to security reasons and blocked by filtering software or firewall. In this case, the FTP transfer will result in fail. To cope well with such environment, *passive* mode, a mode that initiates connection from the User side, is introduced. In the passive mode, data connections are initiated from the User side as similar to the control connection (see Figure 2-6) and do not blocked for security.



Direction of FTP Connections

Code Conversion

FTP provides very limited data type representation and conversion. The two major data types that most FTP could provides are *ASCII* and *IMAGE* .

The ASCII type is the default type and must be implemented by all FTP implementations. The sender converts the data from an internal character to the standard 8-bit ASCII representation. The receiver will convert the data from the standard form to its internal form.
Be careful that most FTP implementation doesn't implement other character types, especially multibyte-characters.

The IMAGE type is highly recommended to implement all FTP softwares. On the IMAGE type, file is treated as a sequence of 8-bit data bytes (or octets) and must be stored as contiguous bytes without modification. In short, file is transferred as it is by IMAGE transfer mode.

Although the uniqueness of the transferred file to the original one, there are possibility of incompatibility of data caused from the difference of data arrangement of multi-byte data on memory that comes from CPU design, e.g., *big endian* and *little endian, difference of floating point expressions*. Don't worry on this, because all WMO binary codes, i.e., GRIB and BUFR, are byte oriented and have no flaws on this point.

FTP Operations

When using FTP, the user will perform some or all of the following operations:

- Connect to a remote host

- Select a directory
- List files available for transfer
- Define the transfer mode
- Copy files to or from the remote host
- Disconnect from the remote host

Connect to the Remote Host

To execute a file transfer, the user begins by logging into the remote host. This is the primary method of handling the security. The user must have a user ID and password for the remote host, unless using Anonymous FTP which is described in Anonymous FTP.

There are three commands which are used:

Open
: Selects the remote host and initiates the login session

User
: Identifies the remote user ID

Pass
: Authenticates the user

Select a Directory

When the control link is established, the user may use the cd (change directory) subcommand to select a remote directory to work with. Obviously, user can only access directories for which the remote user ID has the appropriate authorization. The user may select a local directory with the lcd (local change directory) command. The syntax of theses commands depends upon the operating system in use.

List Files Available for Transfer

This is done using the dir or ls subcommands.

Specifying the Transfer Mode

Transferring data between dissimilar systems often requires transformations of the data as part of the transfer process. The user has to decide on two aspects of the data handling:

- The way the bits will be moved from one place to another.
- The different representations of data upon the system's architecture.

This is controlled using two subcommands:

Mode
: Specifies whether the file is to be treated as having a record structure in a byte stream format.

Block
Logical record boundaries of the file are preserved.
Stream
The file is treated as a byte stream. This is the default, and provides more efficient transfer but may not produce the desired results when working with a record-based file system.

Type
Specifies the character sets used for the data.
ASCII
Indicates that both hosts are ASCII-based, or that if one is ASCII-based and the other is EBCDIC-based, that ASCII-EBCDIC translation should be performed.
EBCDIC
Indicates that both hosts use an EBCDIC data representation.
Image
Indicates that data is to be treated as contiguous bits packed in 8-bit bytes.

Because these subcommands do not cover all possible differences between systems, the SITE subcommand is available to issue implementation-dependent commands.

Copy Files

Get
Copies a file from the remote host to the local host.

Put
Copies a file from the local host to the remote host.

End the Transfer Session

Quit
Disconnects from the remote host and terminates FTP. Some implementations use the BYE subcommand.

Close
Disconnects from the remote host but leaves the FTP client running. An open command may be issued to work with a new host.

Sockets

A socket is a communication mechanism. A socket is normally idenitifed by a small integer which may be called the socket descriptor. The socket mechanism was first introduced in the 4.2 BSD Unix system in 1983 in conjunction with the TCP/IP protocols that first appeared in the 4.1 BSD Unix system in late 1981.

Formally a **socket** is defined by a group of four numbers, these are

- The remote host identification number or address
- The remote host port number
- The local host identification number or address
- The local host port number

Users of Internet applications are normally aware of all except the local port number, this is allocated when connection is established and is almost entirely arbitrary unlike the **well known** port numbers associated with popular applications.

To the application programmer the sockets mechanism is accessed via a number of functions. These are.

socket()	create a socket
bind()	associate a socket with a network address
connect()	connect a socket to a remote network address
listen()	wait for incoming connection attempts
accept()	accept incoming connection attempts

In addition the functions setsockopt(), getsockopt(), fcntl() and ioctl() may be used to manipulate the properties of a socket, the function select() may be used to identify sockets with particular communication statuses. The function close() may be used to close a socket liaison.

Data can be written to a socket using any of the functions write(), writev(), send(), sendto() and sendmsg(). Data can be read from a socket using any of the functions read(), readv(), recv(), recvfrom() and recvmsg()

These notes have been written in the context of SUN's Solaris 2 operating system, in particular version 2.5. Other operating systems and environments will provide similar facilities and functions but reference to the documentation is advised. In particular many of the functions described here may well be system calls, i.e. direct entries to the operating system.

Daemon Processes

The sockets mechanism is usually used to implement client-server applications. The client process is directly or indirectly user driven whereas the server process sits on a host waiting for incoming connections. A server process will run unattended and continuously. In the Unix environment such processes are called **daemon** processes.

A daemon process can be initiated as part of the system boot up sequence. Alternatively a daemon process can be initiated by a user in such a way that it carries on running after the

user logs out. The latter can be achieved using the nohup(1) command in conjunction with an ampersand (&) at the end of the command line but special coding techniques can also be used.

To create a daemon observe the following steps.

- Ensure that the return value from functions is **always** checked.
- Remember that the daemon inherits various things from the shell from which it was invoked.
- Close all open files. This includes stdin, stdout and stderr. Open suitable special files for stdin, stdout and stderr if necessary.
- Change the working directory. Remember that if the daemon crashes it will drop core in the current working directory. Also the working directory, and under some systems the executable file, are open when the daemon is running. This may cause difficulties if the system manager wants to unmount the relevant partition whilst the daemon is running.
- Reset the file creation mask using umask().
- To run a daemon in background after interaction to start it up, which may be useful during development, use the fork() function.
- Run the daemon in a separate process group using the setpgrp() function. This avoids unexpected signals.
- Catch all likely signals. The SIGTERM signal is sent to all processes when the system closes down, arrange to catch this for a graceful shut down of the daemon.
- Ensure that there is no control terminal.
- For development you may wish to log events. To avoid information being lost in buffers use unbuffered output either using sprintf() and write() or use fopen() and fclose() before and after every logged event. ANSI C compilers offer further options for un-buffered output. Log every signal.
- Use lock files to avoid multiple instances of daemons. I.e. when a daemon starts up it checks for the existence of a lock file which it creates if it doesn't exist and if it does exist the daemon refuses to run.

Creating a socket

A socket is created using the function socket(). The prototype is

```
#include    <sys/types.h>
#include    <sys/socket.h>

int socket(int domain, int type, int protocol)
```

domain is either AF_UNIX or AF_INET. This parameter specifies whether the socket is to be used for communicating between Unix file system like objects or Internet objects.

Actually this parameter is intended to allow sockets to be used with a wide variety of networking protocols and products. Examination of the socket.h will show that there are lrage number of possible values for networking domains such as CCITT/X.25, Novell and many others. The information in these notes is only intended to cover the Internet domain.

type specifies the communications semantics. There are a number of possible values.

SOCK_STREAM	stream based full-duplex communication
SOCK_DGRAM	datagram based communication
SOCK_RAW	use raw IP sockets (must be super-user)
SOCK_SEQPACKET	sequenced reliable datagrams
SOCK_RDM	reliably delivered messages

protocol is normally set to zero.

The return value from socket() is a small integer that may be used to refer to a socket in subsequent calls. It may be called the socket descriptor or handle and is analogous to a file descriptor.

Socket Options

The behaviour of a socket can be modified using the functions setsockopt(), fcntl() and ioctl(). The function getsockopt() can be used to detemine the current values of certain aspects of the behaviour of a socket. The default options are satisfactory for most applications.

With fcntl() the FNDELAY option can be associated with the command F_SETFL to make reads non-blocking.

The prototype of setsockopt() is

```
#include    <sys/types.h>
#include    <sys/socket.h>

int setsockopt(int s, int level, int optname, char *optval, int *optlen)
```

s is the socket number

level is set to SOL_SOCKET

optname indicates which option to modify. There are several options, see the manual for details.

optval/optname are used to provide new values for the various options

The select() function

If an application is using several sockets then the select() function may be used to find out which ones are active, i.e. which ones have outstanding incoming data or which can now accept further data for transmission or which have outstanding exceptional conditions. This function would probably only be used with server daemon programs using multiplexing to handle multiple clients.

select() is normally used in conjunction with the macros FD_SET, FD_CLR, FD_ISSET and FD_ZERO. The prototype is

```
#include    <sys/types.h>
#include    <sys/time.h>

int select(int width, fd_set *readfds, fd_set *writefds, fd_set exceptfds, struct timeval *timeout)
```

Before calling select() FD_SET, FD_CLEAR and FD_ZERO should be used to put the relevant descriptors into objects of type fd_set. After the call the return value gives the number of "active" sockets and the FD_ISSET macro may be used to determine whether a particular socket is active, width specifies the number of the largest descriptor to be checked. The information in timeout specifies how long to wait to complete the selection.

The connect() function

The connect() function is used by a client program to establish communication with a remote entity. The prototype is

```
#include    <sys/types.h>
#include    <sys/socket.h>

int connect(int s, struct sockaddr *name, int namelen)
```

s specifies the socket

name points to an area of memory containing the address information and **namelen** give the length of the address information block. This is done because addresses in some addressing domains are far longer than in others. connect() is normally only used for SOCK_STREAM sockets.

The form of a struct sockaddr is

```
struct sockaddr
{
    u_short   sa_family;       /* address family */
    char      sa_data[14];     /* actual address */
```

```
}
```

IP Addresses for connect()

If the AF_INET domain is being used this will be the address of an object of type struct sockaddr_in whose internal structure is

```
struct  sockaddr_in
{
     short    sin_family;
     u_short  sin_port;
     struct   in_addr sin_addr;
     char     sin_zero;
}
```

A struct in_addr in turn consists of

```
struct in_addr
{
      union
      {
           struct { u_char s_b1, s_b2, s_b3, s_b4; };
           struct { u_short s_w1, s_w2; };
           u_long S_addr;
     } S_un;
}
```

As a convenience the following definition can be used

```
#define s_addr S_un.S_addr
```

The use of the othe "views" of the internal structure of struct in_addr is obsolete.

Howere, the function connect() requires the address of an area of memory containing a data object of type struct sockaddr. The conflict between an IP address stored in an object of type struct sockaddr_in and the object of type struct sockaddr required by connect() and related routines can be resolved by the use of a *union* thus.

```
union sock
{
          struct sockaddr          s;
          struct sockaddr_in i;
} sock;
```

The usual way of opening a connection is by the user supplying a fully qualified DNS name (e.g. scitsc.wlv.ac.uk) and using the library call gethostbyname() to determine the associated Internet address. The prototype is

```
#include    <sys/socket.h>
#include    <netdb.h>
```

```
struct hostent *gethostbyname(char *)
```

This returns the address of an object of type struct hostent. The relevant fields need to be copied into an object of type struct sockaddr_in as shown below

```
union sock
{
        struct      sockaddr s;
        struct  sockaddr_in i;
} sock;
struct in_addr      internet_address;
struct hostnet     *hp;

hp = gethostbyname(remote address);
memcpy(&internet_address, *(hp->h_addr_list),sizeof(struct in_addr);
```

The relevant components of the struct sockaddr_in can now be filled in. Some older references on sockets programming may use the function *bcopy()* rather than *memcpy()*.

```
sock.i.sin_family = AF_INET;
sock.i.sin_port = required port number;
sock.i.sin_addr = internet_address;
```

In practice the final line of code above may be written thus

```
        memcpy(&(sock.i.sin_addr),*(hp->h_addr_list),sizeof(struct in_addr));
```

connect() can now be called

```
connect(socket number, &sock.s, sizeof(struct sockaddr));
```

Domain Name Service

Domain name service (DNS) is a mechanism that provides easily remembered names for Interent hosts rather than using the dotted decimal addresses.

It can be accessed via the program /usr/etc/nslookup thus

```
/usr/etc/nslookup
Default Server: ccub.wlv.ac.uk
Address: 134.220.1.20

>unix.hensa.ac.uk
Server: ccub.wlv.ac.uk
Address: 134.220.1.20

Non-authoritative answer:
Name: unix.hensa.ac.uk
Address: 129.12.21.7
> Ctrl-D
```

Incoming connections bind()

To accept incoming connection requests a server process must first create a socket using socket() and then use bind() to associate a port number with the socket. The prototype of bind() is

```
#include  <sys/types.h>
#include  <sys/socket.h>

int bind(int s, struct sockaddr *name, int namelen)
```

This is similar to the connect() function except that, when binding in the AF_INET domain, the components of the struct sockaddr_in are filled in differently

```
union sock
{
        struct sockaddr s;
        struct sockaddr_in i;
} sock;

sock.i.sin_family = AF_INET;
sock.i.sin_port = port number;
sock.i.sin_addr.s_addr = INADDR_ANY;
```

The final value simply means that connections will be accepted from any remote host.

Incoming Connections listen()

Once an address has been bound to a socket it is then necessary to indicate the socket is to be listened to for incoming connection requests. This is done using the listen() function. Its prototype is

```
int listen(int s, int backlog)
```

s specifies the socket.

backlog specifies the maximum number of outstanding connection requests in listen()'s input queue. listen() can only be associated with SOCK_STREAM or SOCK_SEQPACKET type sockets.

Incoming connections accept()

Once the listen() call has returned the accept() call should be issued, this will block until a connection request is received from a remote host. The prototype is

```
#include<sys/types.h>
#include<sys/socket.h>

int accept(int s, struct sockaddr *addr, int *addrlen)
```

s is the socket number

addr points to a struct sockaddr that will be filled in with the address of the remote (calling) system.

addrlen points to a location that will be filled in with the amount of significant information in the remote address, initially it should specify the size of the space set aside for the incoming address.

The return value of the call is the number of a new socket descriptor that should be used for all subsequent communication with the remote host. You can, and should, carry on listening on the original socket number. There is no way of rejecting a connection request, you must accept it then close() it.

Receiving Data

There are a variety of functions that may be used to receive incoming messages.

read() may be used in the exactly the same way as for reading from files. There are some complications with non-blocking reads. This call may only be used with SOCK_STREAM type sockets.

readv() may be used in the same way as read() to read into several separate buffers. This is called a scatter read. This call may only be used with SOCK_STREAM type sockets.

recv() may be used in the same way as read(). The prototype is

```
#include <sys/types.h>
#include <sys/socket.h>

int recv(int s, char *buff, int len, int flags)
```

buff is the address of a buffer area.

len is the size of the buffer.

flags is formed by ORing MSG_OOB and MSG_PEEK allowing receipt of out of band data and allowing peeking at the incoming data. This call may only be used with SOCK_STREAM type sockets.

recvfrom() and recvmsg() may be used with all types of socket. recvfrom() is similar to recv() with extra parameters specifying the remote system address and recvmsg() receives into a message structure.

In all cases the return value is the number of bytes received or -1. The value -1 indicates an error.

It is tempting and a common beginner's error to think that data sent using a single write() to a socket can be read at the other end using a single read(). There is no guarantee that this will happen, data may well arrive in "dribs and drabs". If the application is a network server, it is not possible to impose any constraint on clients to write the data using a single call to write().

The following code shows how to receive a message whose termination is indicated by the character pair CR+LF (a common internet convention).

```
char        buff[BUFSIZ+1];                 /* +1 for string terminator */
char        *bptr;
int         i;

bptr = buff;
do
{
            i = read(sd,bptr,BUFSIZ-(bptr-buff));
            if(i<=0) break;
            bptr += i;
            *bptr = '\0';               /* make what we've got into a string */
            if((cp = strstr(buff,"\r\n"))
            {
                        *cp = '\0';
                        break;
            }
            if(bptr >= buff+BUFSIZ)
            {
                        /* message too long */
            }
} while(1);
```

The code also converts the received message into a string. If part of the next message is received in the same "chunk" of data as the end of the current message, this code is likely to loose the initial part of the next message.

Another common requirement of network programming is to implement a time out if a read() call does not return in a defined time. [An alternative technique is make the socket **non-blocking**.] This is normally done using the alarm() system call in conjunction with a signal catching routine. Unfortunately on many operating systems (such as Solaris), after

catching the signal, the system call is re-issued. If this happens with time-outs the program will loop indefinitely. The problem can be overcome either using the setjmp() and longjmp() routines or by using sigaction() and sigsetops to modify the behaviour associated with the signal.

Here's some sample Solaris code

```
void (*func)() = timeout;

struct        sigaction action;

action.sa_handler = func;
action.sa_flags = 0;
sigemptyset(&(action.sa_mask));  /* ignore all known signals */
sigaction(SIGALRM,&action,NULL));  /* ensures that SA_RESTART is NOT set */
alarm(TIMEOUT);
```

The *sigaction()* uses the information in *action* to define the handling of the SIGALRM signal.

Sending Data

There are a variety of functions that may be used to send outgoing messages.

write() may be used in exactly the same way as it is used to write to files. This call may only be used with SOCK_STREAM tpye sockets.

writev() is similar to write() except that it writes from a set of buffers. This is called a gather write. This call may only be used with SOCK_STREAM type sockets.

send() may be used in the same way as write(). The prototype is

```
#include<sys/types.h>
#include<sys/socket.h>

int send(int s, char *msg, int len, int flags)
```

s is the socket number.

msg and **len** specify the buffer holding the text of the messagee.

flags may be formed by ORing MSG_OOB and MSG_DONTROUTE. The latter is only useful for debugging. This call may only be used with SOCK_STREAM type sockets.

sendto() can be used to send messages to sockets of any type and sendmsg() sends a message held in a message structure.

In all cases the return value is the number of bytes sent or -1.

Multiple Server Sessions

If sockets programming is being used to proved a server facility it is important to ensure that multiple simultaneous sessions are handled correctly. Once a connection request has been accepted the program will be engaged in handling the associated dialogue, further connection requests will be held until the listener program gets round to issuing the accept() call again, so only one client can be handled at a time.

There are several ways of handling this issue. Under Unix the simplest solution is to fork a separate process to handle the client/server dialogue once a connection request has been received.

Here's an outline of the basic code.

```
listen(sd,2);               /* at most 2 pending connections */
do
{
        nsd = accept(sd,&(work.s),&addrlen);
        pid = fork();
        if(pid == 0)
        {
                /* Child process handles the dialogue
    using descriptor 'nsd'
                */
                close(nsd);
                exit(0);      /* end of child process */
        }
        else        close(nsd); /* parent won't be using 'nsd' */
} while(1);
```

The overheads associated with forking a separate process everytime a client connects can be significant. Alternative approaches use either **multiplexing** or **multithreading**.

Multiplexing requires that all the code be capable of talking on serveral different sockets simultanesously, typically there will be an array of descriptors and it is the responsibility of the program to keep track of the state of the dialogue associated with each active socket. The poll() can be used to check for new connection requests. Multiplexing is the most complex and most efficient way of handling multiple dialogues.

Multithreading is in many ways similar to forking a separate process for each dialogue with the exception that thread creation overheads are much lower than process creation overheads. Synchronisation between threads is usually easier than synchronisation between processes.

Miscellaneous Routines

There are various useful miscellaneous routines.

bcmp(), bcopy() and bzero() (see bstring(3)) manipulate blocks of memory. They are equivalent to the ANSI functions memcmp(), memcpy() and memset().

htonl(), htons(), ntohl() and ntohs() convert strings of bytes to/from host order to network order. They should always be used in a heterogeneous environment. On SUN systems they are NULL macros but on systems such as VAXes and PCs they convert local byte order to/from network order. See byteorder(3N).

The routines inet_addr(), inet_network(), inet_makeaddr(), inet_lnaof() and inet_ntoa() converts an Internet address to a string. See inet(3N).

The library routines gethostbyaddr() and gethostbyname() can be used for conversions between dotted decimal and DNS addresses. See gethostent(3N).

The function getpeername() determines the address of the remote system on a socket. This should be used in conjunction with gethostbyname() to determine the DNS address of the remote system.

SYSTEM RESOURCES

HARDWARE DESCRIPTION

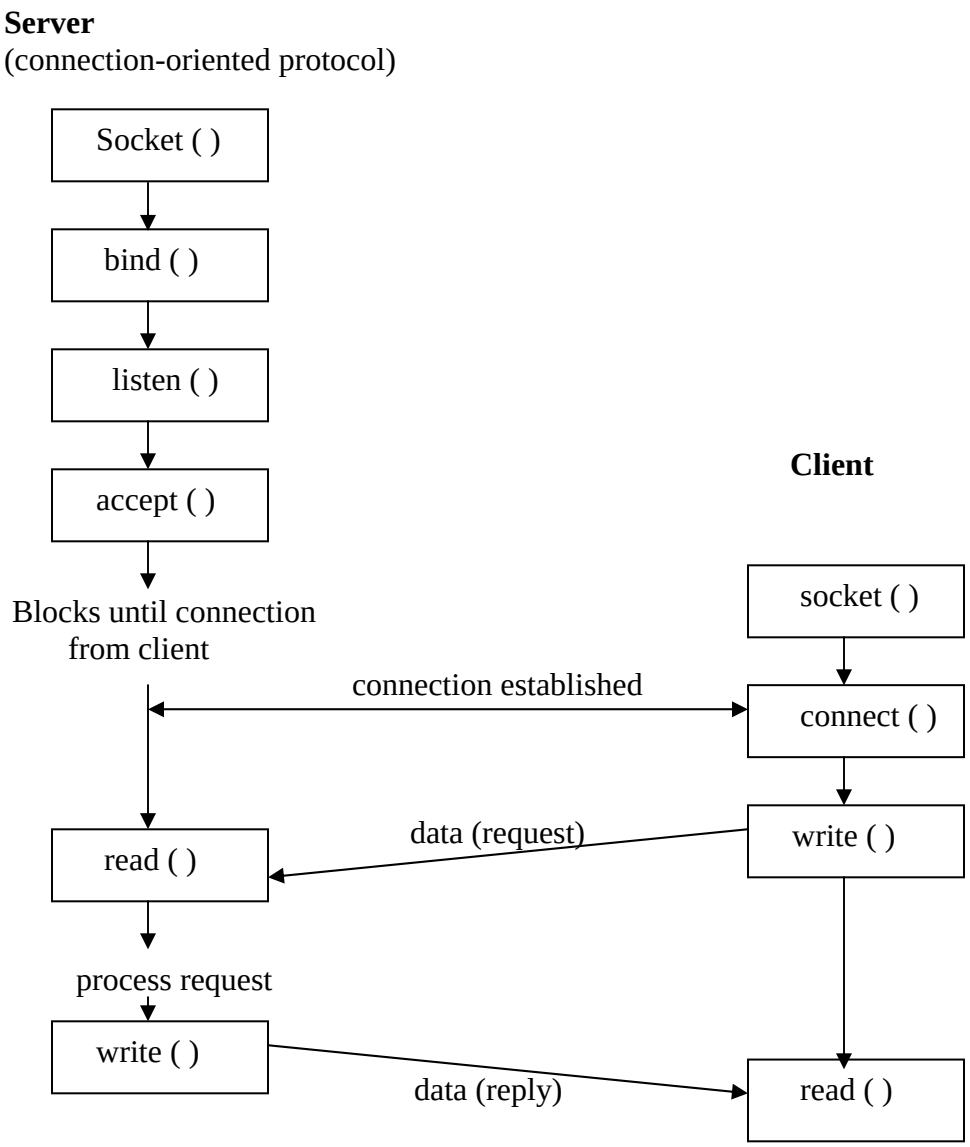
Configuration: -

CPU	Intel Pentium
RAM	128 MB
Hard Disk	10 GB
Drives	1.44 FDD & 48 X CD - Rom Drive
SMPS	250 Watts
Key Board	108 keys
Mouse	Logitech
Cache Memory	256 K

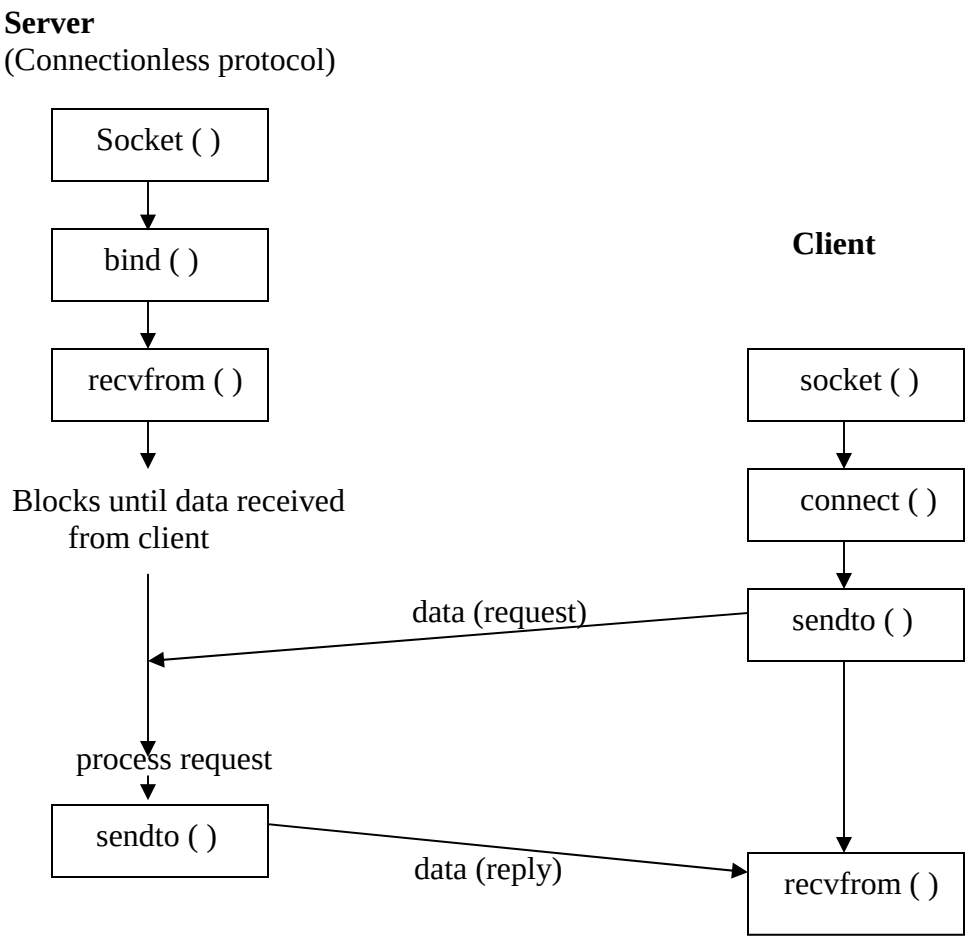
SOFTWARE DESCRIPTION

Language Used	C and JAVA
Documentation	MS WORD 2000

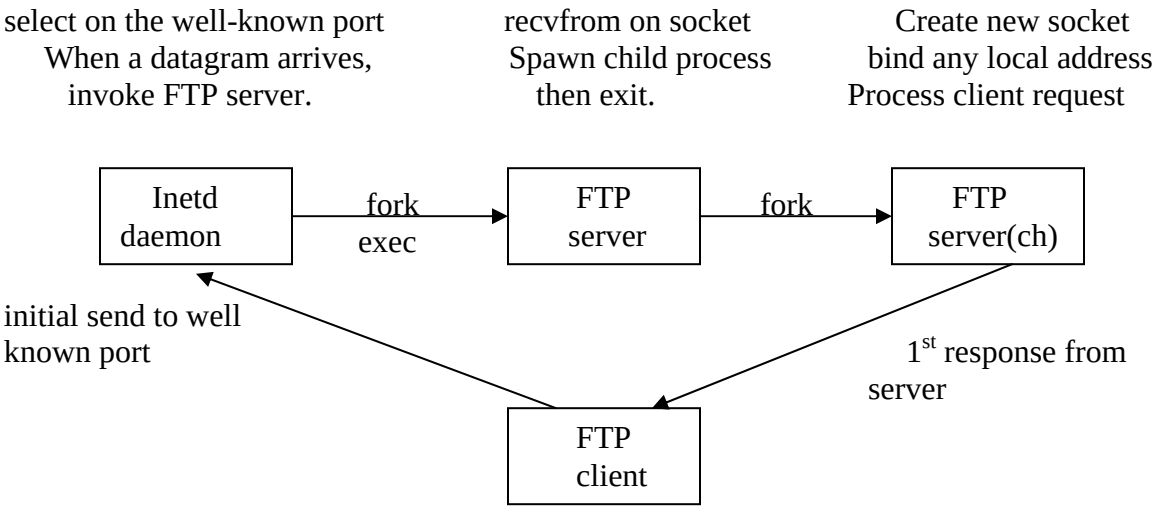
DESIGN



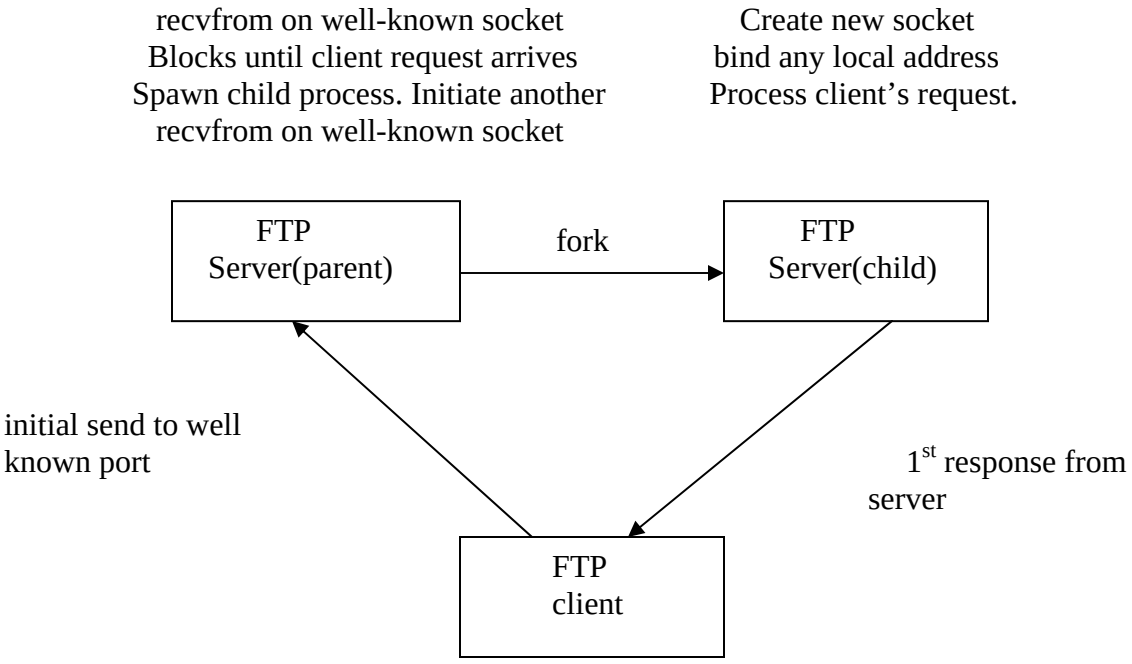
Socket system calls for connection-oriented protocol.



Socket system calls for connectionless protocol.



FTP server, when invoked.



FTP server, when run as daemon.

Why TCP Implementation????

FTP can be implemented using UDP but we chose TCP because:

In many ways, the UDP implementation is the worst-case example. The transfer protocol UDP, is unreliable, so we have to handle timeout and retransmission ourselves. We also have to maintain a pseudo connection between the client and server, since multiple datagrams hace to be exchanged for the transfer of a single file, using the connectionless UDP.

CODE

Implementation in C

The code written in programming language has been written for implementation on Linux platform only. But it can be made to run in Windows platform also by making some minor changes. We have employed file handling and socket programming extensively. The files used in program are:

- **hostname.c**
  This file is used to accept hostnames and their respective ip addresses from the user and save them to a file hostname.txt. It gives error if there is a problem in opening the file hostname.txt and finally prints the names of all the hosts and their ip addresses.
- **password.c**
  This file is used to accept all the usernames and their respective passwords from the user and saves them to the file password.txt. It gives error if there is a problem in opening the file password.txt. This is used for the authentication purpose so that unauthorized user can not avail the services provided by the software.
- **server.c**
  This file implements our server. It accepts all the rquests from the client program and responds accordingly. The main functions being implemented her are:
  - get ( )
    This function is used to implement the download function of the program. It receives a signal from the client program and if the connection is established, it accepts the path of the file requsted from the clientand then sends it.
  - put ( )
    This function is ised to implement the upload functionality of the program. It receives a signal from the client program and if the connection is established, it accepts the name of the file being sent by the client and then stores it in its current working directory.
  - ls ( )
    This function is used to list all the contents of the current working directory of the server.
  - pwd ( )
    This function is used to tell the client the complete path of the current working directory.
  - change directory
    This function is implemented by using the chdir ( ) function directly.

- **client.c**
  This file implements our client. It sends all the requests to the server and receives replies from it. It also makes calls to the functions in the hostname.c and

password.c files for the logging and authentication purposes. All these functiuonalities are implemented wit the help of follwing functions:

- gethostname ( )
  This function is used to establish the connection with the server. It accepts the name of the server and returns the status of the connection.
- establish ( )
  This function is used to establish the connection with the server by creating a socket to interact with the server.
- interact ( )
  This function is used to interact with the server at the socket created. It takes the descriptor of the socket created and returns the status according to the username and password. It gives error if the username or password is incorrect.
- cd ( )
  It is used to give the implementation for “cd” command by accepting the descriptor of the socket created. It changes the current working directory of the server.
- ls ( )
  It is used to list the contents of the current working directory. It takes in as input the descriptor of the socket created and lists the files in the current working directory of the server.
- put ( )
  The objective of this function is to implement the upload functionality. The local files are copied into the remote directory.
- get ( )
  This function is used to give the implementation of the download functionality. It copies the remote files into the local directories.

Hostname.c

```
#include <stdio.h>
struct host_ip
{
   char hostname[100];
   char address[20];
};

int main()
{
  struct host_ip temp_host;
  struct host_ip temp_host1;

  FILE *fp;
  fp=fopen("hostname.txt","ab");
  if(!fp)
  {
    printf("\nerror in opening the file...\n");
    exit(0);
  }
  char choice[2];
  do
  {
    printf("\n Enter the host name : ");
    scanf("%s",temp_host.hostname);
    printf("\n Enter the ip address : ");
    scanf("%s",temp_host.address);
    fwrite(&temp_host,sizeof(temp_host),1,fp);
    printf("\nDo you want to enter any other host name(y/n):");
    scanf("%s",choice);
  }while((choice[0]!='n')&&(choice[0]!='N'));

  fclose(fp);

  fp=fopen("hostname.txt","rb");
  if(!fp)
  {
    printf("\nerror in opening the file...\n");
    exit(0);
  }

  while(fread(&temp_host1,sizeof(temp_host1),1,fp))
    printf("\nhostname : %s  ip address :
%s\n",temp_host1.hostname,temp_host1.address);
```

```
  fclose(fp);
  return 0;
}
```

password.c

```
#include <stdio.h>
struct identity
{
   char username[100];
   char password[100];
};

int main()
{
  struct identity temp_id;
  struct identity temp_id1;

  FILE *fp;
  fp=fopen("passwd.txt","ab");
  if(!fp)
  {
    printf("\nerror in opening the file...\n");
    exit(0);
  }
  char choice[2];
  do
  {
    printf("\n Enter the user name : ");
    scanf("%s",temp_id.username);
    printf("\n Enter the password : ");
    scanf("%s",temp_id.password);
    fwrite(&temp_id,sizeof(temp_id),1,fp);
    printf("\nDo you want to enter any other identity(y/n)
                          : ");
    scanf("%s",choice);
  }while((choice[0]!='n')&&(choice[0]!='N'));

  fclose(fp);

  fp=fopen("passwd.txt","rb");
  if(!fp)
  {
    printf("\nerror in opening the file...\n");
    exit(0);
  }

  while(fread(&temp_id1,sizeof(temp_id1),1,fp))
    printf("\nusername : %s  password : %s\n",temp_id1.username,temp_id1.password);
```

```
  fclose(fp);
  return 0;
}
```

server.c

```
/********  server program   **********************/

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

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

#define PROTOPORT 5194
#define QLEN 6

#define END -1

struct data
{
  char buffer;
};

struct identity
{
  char username[100];
  char password[100];
};


void get(int);
void put(int);
void ls(int);

int main(int argc,char *argv[])
{

 struct hostent *ptrh;

 struct sockaddr_in sad;
 struct sockaddr_in cad;
 int sd_old,sd_new;
 char buf[1000],buf1[1000];
 char host[100];
 int alen;
```

```
int childpid;

//opening a tcp socket
sd_old = socket(AF_INET ,SOCK_STREAM , 0);
if(sd_old<0)
{
  printf("\n socket creation failed ");
  exit(1);
}

//filling in the vlues for server address
memset((char *)&sad,0,sizeof(sad));
sad.sin_family=AF_INET;
sad.sin_port=htons(PROTOPORT);
sad.sin_addr.s_addr=INADDR_ANY;

//binding the server
if(bind(sd_old,(struct sockaddr *)&sad,sizeof(sad)) < 0)
{
  printf("\n bind failed ");
  exit(1);
}

//specifying the queue length
if(listen(sd_old,QLEN) < 0)
{
  printf("\n listen failed ");
  exit(1);
}


//infinite loop to listen to client's request
while(1)
{
  alen=sizeof(cad);

  memset((char *)&cad,0,sizeof(cad));

  //accepting client's request and creating new socket for it
  sd_new=accept(sd_old, (struct sockaddr *)&cad,&alen);
  if(sd_new<0)
  {
    printf("\n accept failed ");
    exit(1);
  }
```

```
    //creating a child process to interact with the client
    if((childpid=fork())<0)
      printf("server : fork error ");

    //code for child process
    if(childpid==0)
    {
        int n;

        //child destroying the old socket because it doesn't need it
        close(sd_old);

        //code for session with the client

        //asking for the username
        send(sd_new,"username : ",strlen("username : "),0);
        n = recv(sd_new,buf,sizeof(buf),0);
        *(buf+n)='\0';

        //asking for the password
        send(sd_new,"password : ",strlen("password : "),0);
        n = recv(sd_new,buf1,sizeof(buf1),0);
        *(buf1+n)='\0';


           FILE *fp;
        fp=fopen("passwd.txt","rb");
        if(!fp)
        {
          printf("\nerror in opening the password file...\n");
             send(sd_new,"not_ok",strlen("not_ok"),0);
             close(sd_new);
          exit(1);
        }

           struct identity temp_id;
           int flag=0;
        while(fread(&temp_id,sizeof(temp_id),1,fp))
        {

if((strcmp(buf,temp_id.username)==0)&&(strcmp(buf1,temp_id.password)==0))
           {
             send(sd_new,"ok",strlen("ok"),0);
                flag=1;
             break;
               }
```

```
}
fclose(fp);
if(flag==0)
{
  send(sd_new,"not_ok",strlen("not_ok"),0);
  close(sd_new);
     exit(1);
}

    //authentication over : ftp session starts

n = recv(sd_new,buf,sizeof(buf),0);
*(buf+n)='\0';

while((strcmp(buf,"close")!=0) && (strcmp(buf,"quit")!=0))
{
   //code for implementation of get
   if(strcmp(buf,"get")==0)
      get(sd_new);

   else
   if(strcmp(buf,"put")==0)
      put(sd_new);

   else
   if(strcmp(buf,"ls")==0)
      ls(sd_new);

      else
   if(strcmp(buf,"cd")==0)
   {
         n = recv(sd_new,buf,sizeof(buf),0);
      *(buf+n)='\0';
            int s=chdir(buf);
            if(s!=0)
              send(sd_new,"error",strlen("error"),0);
            else
              send(sd_new,"no_error",strlen("no_error"),0);
      }

   else
   if(strcmp(buf,"pwd")==0)
   {
      char *temp;
      temp=(char *)malloc(100*sizeof(char));
      getcwd(temp,100);
```

```
            send(sd_new,temp,strlen(temp),0);
          }


          n = recv(sd_new,buf,sizeof(buf),0);
          *(buf+n)='\0';
        }


        //after the session with the client child destroys the new socket as well as itself
        printf("\nclosing a client....");
        close(sd_new);
        exit(1);
    }
    //parent destroying the new socket because it doesn't need it
    close(sd_new);
  }
}

/*************************************************************************
****
objective : to give the implementation for "put" command
input     : the descriptor for the socket created
output    : the remote file will be copied into local file
*************************************************************************
***/
void put(int sd_new)
{
  int n;
  char buf[512];
  n = recv(sd_new,buf,sizeof(buf),0);
  *(buf+n)='\0';

  FILE *fp;
  fp=fopen(buf,"w");
  if(!fp)
    send(sd_new,"not_ok",strlen("not_ok"),0);

  else
  {
    send(sd_new,"ok",strlen("ok"),0);
    n = recv(sd_new,buf,sizeof(buf),0);
    *(buf+n)='\0';
    if(strcmp(buf,"ok")==0)  //if local file is also opened successfully
    {
      struct data temp_data;
```

```
        n = recv(sd_new,&temp_data,sizeof(temp_data),0);
        while(temp_data.buffer!=END)
        {
          fwrite(&temp_data,sizeof(temp_data),1,fp);
          n = recv(sd_new,&temp_data,sizeof(temp_data),0);
        }
        fclose(fp);
      }
    }
}



/*************************************************************************
****
objective : to give the implementation for "get" command
input     : the descriptor for the socket created
output    : the local file will be copied into remote file
*************************************************************************
***/
void get(int sd_new)
{
  int n;
  char buf[512];
  n = recv(sd_new,buf,sizeof(buf),0);
  *(buf+n)='\0';
  FILE *fp;
  fp=fopen(buf,"r");
  if(!fp)
    send(sd_new,"not_ok",strlen("not_ok"),0);

  else
  {
    send(sd_new,"ok",strlen("ok"),0);
    n = recv(sd_new,buf,sizeof(buf),0);
    *(buf+n)='\0';
    if(strcmp(buf,"ok")==0)  //if local file is also opened successfully
    {
      struct data temp_data;
      while(fread(&temp_data,sizeof(temp_data),1,fp))
        send(sd_new,&temp_data,sizeof(temp_data),0);
      fclose(fp);
      temp_data.buffer=END;
      send(sd_new,&temp_data,sizeof(temp_data),0);
    }
```

```
   }
}



/*************************************************************************
****
objective : to give the implementation for "ls" command
input     : the descriptor for the socket created
output    : the output of "ls" command will be sent to the client
*************************************************************************
***/
void ls(int sd)
{
  if(fork()==0)
  {
    char*args[2];
    close(1);
    dup(sd);
    args[0] = (char *) malloc ( 10 * sizeof(char));
    args[1]=NULL;
    strcpy(args[0],"ls");
    execvp(args[0],args);
    exit(1);
  }
  send(sd,"end",strlen("end"),0);
```

client.c

```
/**************  client program ************************/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <curses.h>


//specifying the server's port number
#define PROTOPORT 5193

int establish_con(char *server_name);
int interact(int sd);
void get(int);
void put(int);
void ls(int);
void cd(int,char[100]);

#define END -1

struct data
{
   char buffer;
};

struct host_ip
{
   char hostname[100];
   char address[20];
};

//main function to control the flow of the program
int main(int argc,char *argv[])
{
  char input[100];
  char *tok;
  int pos,pos1;
  int sd=0;
```

```
int status;

pos:printf("\nFTP_client@Project ");
fgets(input,100,stdin);
input[strlen(input)-1]='\0';

while(strcmp(input,"ftp")!=0)
{
  if(strcmp(input,"exit")==0)
  {
     printf("\nquitting...\n");
     exit(1);
  }
  printf("\nthis program works only for ftp.....\n");
  printf("\nftp_client@Project>");
  fgets(input,100,stdin);
  input[strlen(input)-1]='\0';
}


//starting the ftp session
pos1:printf("\nftp>");
fgets(input,100,stdin);
input[strlen(input)-1]='\0';


tok=strtok(input," ");
while((strcmp(tok,"open")!=0) && (strcmp(input,"quit")!=0))
{
  printf("\nFirst of all you need to do \'open\'.....\n");
  printf("\nftp>");
  fgets(input,100,stdin);
  input[strlen(input)-1]='\0';
  tok=strtok(input," ");
}

if(strcmp(input,"quit")==0)
  goto pos;  //going back to the prompt


tok=strtok(NULL," ");

if(tok != NULL)
{
  sd=establish_con(tok);
  if(sd==0)
```

```
    {
      printf("\nConnection not established....\n");
      goto pos1;
    }
  }
  else
  {
    printf("\n(to)");
    fgets(input,100,stdin);
    input[strlen(input)-1]='\0';
    while(strcmp(input,"")==0)
    {
     printf("\nYou need to enter the name of the server.....\n");
     printf("\n(to)");
     fgets(input,100,stdin);
     input[strlen(input)-1]='\0';
    }
    sd=establish_con(input);
    if(sd==0)
    {
      printf("\nConnection not established....\n");
      goto pos1;
    }
  }

  status=interact(sd);
  if(status==0)
     goto pos1;

  else
  if(status==2)
  {
   printf("\nGood Bye");
   close(sd);
   goto pos1;
  }
  goto pos;
  return 0;
}//end of main
```

```
/*************************************************************************
****
objective : to establish the connection with the server
input     : the name of the server
output    : the status for the connection
          0 - connection failure
          socket descriptor - connection successful
*************************************************************************
***/
void Gethostbyname(char *server_name,char server_addr[100])
{
  FILE *fp;
  fp=fopen("hostname.txt","r");
  if(!fp)
  {
    printf("\nError in opening the file...\n");
    return ;
  }

  struct host_ip temp_host;
  while(fread(&temp_host,sizeof(temp_host),1,fp))
  {

    if(strcmp(temp_host.hostname,server_name)==0)
    {
        fclose(fp);
        strcpy(server_addr,temp_host.address);
        return ;
    }
    if(strcmp(temp_host.address,server_name)==0)
    {
       fclose(fp);
       strcpy(server_addr,temp_host.address);
       return ;
    }

  }

  fclose(fp);
  return ;

}
```

```
/**************************************************************************
****
objective : to establish the connection with the server
input     : the name of the server
output    : the status for the connection
          0 - connection failure
          socket descriptor - connection successful
**************************************************************************
***/
int establish_con(char *server_name)
{
  struct hostent *ptrh;
  struct sockaddr_in sad;
  int sd;
  char buf[512];
  char input[100];
  char server_addr[100];

  //creating the structure for server's address
  memset((char *)&sad,0,sizeof(sad));
  sad.sin_family=AF_INET;
  sad.sin_port=htons(PROTOPORT);
  Gethostbyname(server_name,server_addr);
  sad.sin_addr.s_addr=inet_addr(server_addr);

  //creating the socket to interact with the server
  sd = socket(AF_INET ,SOCK_STREAM ,0);
  if(sd<0)
  {
    printf("\n Socket creation failed\n ");
    return 0;
  }

  //sending the connection request to the server
  if(connect(sd,(struct sockaddr *)&sad,sizeof(sad)) < 0)
  {
    printf("\n Connect failed\n ");
    return 0;
  }
  printf("\nConnected to : %s\n",server_name);
  return sd;

}//end of establish_con
```

```
/*************************************************************************
****
objective : to interact with the server at the socket created
input     : the descriptor for the socket created
output    : the status according to the username and password
          0 - if invalid username and password
          1 - if valid username and password
*************************************************************************
***/
int interact(int sd)
{
  char buf[512];
  int msg_len;
  char input[100],password[100];


  msg_len=recv(sd,buf,sizeof(buf),0);
  *(buf+msg_len)='\0';
  while( (strcmp(buf,"ok")!=0)  && (strcmp(buf,"not_ok")!=0) )
  {
    printf("\n%s",buf);
    fgets(input,100,stdin);
    input[strlen(input)-1]='\0';
    send(sd,input,strlen(input),0);
    msg_len=recv(sd,buf,sizeof(buf),0);
    *(buf+msg_len)='\0';
  }

  if(strcmp(buf,"not_ok")==0)
  {
    printf("\n%s","Authentication failed.....\n");
    return 0;
  }

  printf("\n%s","\nLogin ok,access restrictions apply.\n");

  //authentication over : ftp session starts
  printf("\n%s","ftp>");
  fgets(input,100,stdin);
  input[strlen(input)-1]='\0';

  while(strcmp(input,"quit")!=0)
  {
   if(strcmp(input,"get")==0)
     get(sd);
```

```
    else
    if(strcmp(input,"put")==0)
      put(sd);

    else
    if(strcmp(input,"ls")==0)
       ls(sd);

    else
    if(strcmp(input,"pwd")==0)
    {
       send(sd,input,strlen(input),0);
       msg_len=recv(sd,buf,sizeof(buf),0);
       *(buf+msg_len)='\0';
       printf("\n%s\n",buf);
    }

    else
    if(strstr(input,"cd")==input)
       cd(sd,input);


    else
    if(strcmp(input,"close")==0)
    {
        send(sd,input,strlen(input),0);
        return 2;
    }

    else
       printf("\nInvalid command .....\n");

     printf("\n%s","ftp>");
     fgets(input,100,stdin);
     input[strlen(input)-1]='\0';
  }
  send(sd,input,strlen(input),0);
  return 1;

}//end of interact


/*************************************************************************
****
objective : to give the implementation for "cd" command
input     : the descriptor for the socket created
```

```
output    : the server's current working directory will be changed
*****************************************************************************
***/
void cd(int sd,char input[100])
{
  char *tok,buf[512];
  int msg_len;
  tok=strtok(input," ");
  if(strcmp(tok,"cd")!=0)
  {
    printf("\n\nInvalid command");
    return;
  }
  send(sd,"cd",strlen("cd"),0);
  send(sd,input+3,strlen(input+3),0);

  msg_len=recv(sd,buf,sizeof(buf),0);
  *(buf+msg_len)='\0';
  if(strcmp(buf,"Error")==0)
    printf("\n\nUnable to change the directory....");
  else
    printf("\n\nPresent working directory of the server changed....");

}//end of cd


/*****************************************************************************
****
objective : to give the implementation for "ls" command
input     : the descriptor for the socket created
output    : the files present in server's current working directory will be
            listed on the screen
*****************************************************************************
***/
void ls(int sd)
{
  char buf[512];
  int msg_len;
  send(sd,"ls",strlen("ls"),0);
  msg_len=recv(sd,buf,sizeof(buf),0);
  *(buf+msg_len)='\0';
  while(strcmp(buf,"end")!=0)
  {
    printf("\n%s\n",buf);
    msg_len=recv(sd,buf,sizeof(buf),0);
    *(buf+msg_len)='\0';
```

```
  }
}//end of ls

/*************************************************************************
****
objective : to give the implementation for "put" command
input     : the descriptor for the socket created
output    : the local file will be copied into remote file
*************************************************************************
***/
void put(int sd)
{

  char buf[512];
  char input[100],lcl_file[100];
  strcpy(input,"put");
  send(sd,input,strlen(input),0);
  int msg_len;

  //asking for the local file  whose contents have to be stored
  printf("\n(Local-file) ");
  fgets(input,100,stdin);
  input[strlen(input)-1]='\0';
  strcpy(lcl_file,input);

  //asking for the name of remote file to be accessed
  printf("\n(Remote-file) ");
  fgets(input,100,stdin);
  input[strlen(input)-1]='\0';
  if(strcmp(input,"")==0)
    strcpy(input,lcl_file);
  send(sd,input,strlen(input),0);



  msg_len=recv(sd,buf,sizeof(buf),0);
  *(buf+msg_len)='\0';

  if(strcmp(buf,"ok")==0)
  {
    FILE *fp;
    fp=fopen(lcl_file,"r");
    if(!fp)
    {
      printf("\nError in opening the local file...\n");
      send(sd,"not_ok",strlen("not_ok"),0);
```

```
    }
    else
    {
      send(sd,"ok",strlen("ok"),0);
      struct data temp_data;
      while(fread(&temp_data,sizeof(temp_data),1,fp))
        send(sd,&temp_data,sizeof(temp_data),0);
      fclose(fp);
      temp_data.buffer=END;
      send(sd,&temp_data,sizeof(temp_data),0);
         printf("\n\nFile written to the server.....");
    }
  }
  else
   printf("\nCannot open the file on server side...\n");

}


/*************************************************************************
****
objective : to give the implementation for "get" command
input     : the descriptor for the socket created
output    : the remote file will be copied into local file
*************************************************************************
***/
void get(int sd)
{
  char buf[512];
  char input[100],rmt_file[100];;
  strcpy(input,"get");
  send(sd,input,strlen(input),0);
  int msg_len;

  //asking for the name of remote file to be accessed
  printf("\n(Remote-file) ");
  fgets(input,100,stdin);
  input[strlen(input)-1]='\0';
  strcpy(rmt_file,input);
  send(sd,input,strlen(input),0);

  //asking for the local file in which the content will be stored
  printf("\n(Local-file) ");
  fgets(input,100,stdin);
  input[strlen(input)-1]='\0';
  if(strcmp(input,"")==0)
```

```
    strcpy(input,rmt_file);

  msg_len=recv(sd,buf,sizeof(buf),0);
  *(buf+msg_len)='\0';

  if(strcmp(buf,"ok")==0)
  {
    FILE *fp;
    fp=fopen(input,"w");
    if(!fp)
    {
      printf("\nError in opening the local file...\n");
      send(sd,"not_ok",strlen("not_ok"),0);
    }
    else
    {
      send(sd,"ok",strlen("ok"),0);
      struct data temp_data;
      msg_len = recv(sd,&temp_data,sizeof(temp_data),0);
      while(temp_data.buffer!=END)
      {
        fwrite(&temp_data,sizeof(temp_data),1,fp);
        msg_len = recv(sd,&temp_data,sizeof(temp_data),0);
      }
      fclose(fp);
         printf("\n\nFile accepted from the server.....");
    }
  }
  else
    printf("\nCannot open the file on server side...\n");
}
```

Output

[root@localhost root]# cd /mnt/D:/project
[root@localhost project]# gcc hostname.c
[root@localhost project]# ./a.out

Enter the host name : host

Enter the ip address : 127.0.0.1

Do you want to enter any other host name(y/n) : n

hostname : host ip address : 127.0.0.1

[root@localhost project]# gcc password.c
[root@localhost project]# ./a.out

Enter the user name : user

Enter the password : pass

Do you want to enter any other identity(y/n) : n

username : user password : pass

[root@localhost project]# gcc server.c
[root@localhost project]# ./a.out &
[1] 4721
[root@localhost project]# gcc client.c
[root@localhost project]# ./a.out

FTP_client@Project ftp

ftp>open

(to)host

Connected to : host

username : user

password : pass

Login ok,access restrictions apply.

ftp>get

(Remote-file) /mnt/D:/project/hostname.txt

(Local-file) hostnam.txt

File accepted from the server.....

ftp>put

(Local-file) /mnt/D:/project/passwd.txt

(Remote-file) pass.txt

File written to the server.....

ftp>ls

a.out
client.c
client.c~
hostna.txt
hostnam.txt
hostname.c
hostname.txt
pass.txt
passwd.txt
password.c
server.c
server.c~

ftp>pwd

/mnt/D:/project

ftp>cd /mnt/D:

Present working directory of the server changed....

ftp>pwd

/mnt/D:

ftp>ls

3668bd3
Documents and Settings
Folder Settings
My Documents
Nishant
Program Files
Recycled
System Volume Information
Thumbs.db
UTILITIES
UniScan
WUTemp
Web pages of homoeopathy
doctor
floppy
hiberfil.sys
hompath
i386
msdownld.tmp
pagefile.sys
project
songs
windows

ftp>quit

FTP_client@Project
closing a client....

ftp_client@Project>exit

quitting...

[root@localhost project]#

Implementation in JAVA

Interface.java: This class job is to create the introduction window where the buttons "START" and "CANCEL" appear. On pressing the " START" button the server session starts where the server sets to listen for any client seeking to make the connection .As and when the connection is complete, the server and the client are connected and the FTP session starts. Now when any of the functions are requested from the client side, accordingly server calls the particular function. These requests are fetched by the server through the input and output streams being maintained between the client and server.

Lserver.java: This class is being called in Interface.java where after the "START" is pressed and the server sets to listen , a black window appears onscreen which is made by this class

Lclient.java: This displays the interactive window for the client to give the various commands to the server like CONNECT, DISCONNECT,DOWNLOAD,UPLOAD,DELETE

CONNECT: for initiating the request for the connection to the server

DISCONNECT: to quit the session and ends the connection between the client and the Server. Here only client dies but the server is still listening to any client seeking the session.

DOWNLOAD: it pops a window asking for the file to be downloaded from the server And also the path and name of the new file where that is to be Downloaded.

UPLOAD: just the reverse of the download command where the file from the client is being transferred to the server.

DELETE: asks the path and name of the file to be deleted on the server and then performs the task.

Ftps1.java: This class is called by the Interface class after the request for DOWNLOAD comes from the client side. This class makes the data connection through which the file data will be transferred. First through the input stream it gets the name and the path of the file and through the output stream it transfers the file content which is obtained by the client

Ftpc1.java: This class is called by the Lclient.java after the button "DOWNLOAD" is pressed . now this class makes a window to enter the name of the source and the destination file to be downloaded .This connects through the data socket set in the ftps1.java .and gets the contents of the source file through the input stream.

Ftps2.java: Again this is called by Interface. java when "UPLOAD" button is pressed at the client side. This connects to the socket set in the ftpc2.java and receives the file contents from the client.

Ftpc2.java: Called by the Lclient.java for the upload function. It makes a new socket for the data transfer, that is, the file to the server, by setting an input stream and the output stream.

Ftpcd1.java: When the operation "DELETE" is requested by the client, this class is called by the Lclient.java, which pops a window to enter the name of the file to be deleted at the server side. This passes the name of the file to the server through a output stream which reacts accordingly.

Ftpsd1.java: When it receives the file name, through its input stream, to be deleted it opens a file pointer of the type file and manipulate it accordingly, that is, deleting the contents of the file requested to be deleted.

LServer.java

```
import java.io.*;
import java.awt.*;
import java.net.*;
import java.awt.event.*;
import javax.swing.*;

class LServer extends JFrame
{
 private JDesktopPane idesk;
 public LServer()
 {
  super("LSERVER");
  Container c = getContentPane();
  c.setBackground(Color.black);
  idesk = new JDesktopPane();
  idesk.setBackground(Color.black);
  c.add(idesk,BorderLayout.CENTER);
 }
 public JDesktopPane share()
 {
  return(idesk);
 }
}
```

ftpc1.java

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class ftpc1 extends JFrame
{
        final static int clint_port=28;
        final static int server_port=28;
        private JTextField tf1,tf2;
      private JButton b;
        private JLabel l1,l2;
        public ftpc1()
        {
                Container c = getContentPane();
                c.setBackground(Color.orange);
                tf1 = new JTextField(10);
                tf2 = new JTextField(10);
                l1 = new JLabel("Enter the file wanted:");
                l2 = new JLabel("Enter the filename to be saved:");
                b = new JButton("Start Downloading");
                JPanel p1 = new JPanel();
                p1.add(l1);
                p1.add(tf1);
                JPanel p2 = new JPanel();
                p2.add(l2);
                p2.add(tf2);
                JPanel p3 = new JPanel();
                p3.add(b);
                c.add(p1,BorderLayout.NORTH);
                c.add(p2,BorderLayout.CENTER);
                c.add(p3,BorderLayout.SOUTH);
                setSize(300,200);
                setVisible(true);
                b.addActionListener(
                        new ActionListener(){
                                public void actionPerformed(ActionEvent e){
                                        run();
                                        setVisible(false);
                                }
```

```
                        }
                );
        }
        public void run()
        {
                try{
                Socket client = new
Socket(InetAddress.getByName("localhost"),clint_port);
                ObjectOutputStream out = new
ObjectOutputStream(client.getOutputStream());
                ObjectInputStream in = new ObjectInputStream(client.getInputStream());

                String mesg1="";
                String mesg2="";
                String mesg3="";
                mesg2=tf1.getText();
                mesg3=tf2.getText();
                out.writeObject(mesg2);
                out.flush();

                FileOutputStream fs=new FileOutputStream(mesg3);
              ObjectOutputStream outf = new ObjectOutputStream(fs);
                while(true)
                {
                        mesg1=(String)in.readObject();
                        outf.writeObject(mesg1);
                        outf.flush();
                        if(mesg1.equals("EOF"))
                                break;
                }
                outf.close();
                fs.close();
                out.close();
                in.close();
                client.close();
                }
                catch(Exception e)
                {
                        System.out.println(e+"error");
                }
        }

}
```

ftpc2.java

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class ftpc2 extends JFrame
{
        final static int clint_port=28;
        final static int server_port=28;
        private JTextField tf1,tf2;
      private JButton b;
        private JLabel l1,l2;
        public ftpc2()
        {
                Container c = getContentPane();
                c.setBackground(Color.orange);
                tf1 = new JTextField(10);
                tf2 = new JTextField(10);
                l1 = new JLabel("Enter the file wanted:");
                l2 = new JLabel("Enter the filename to be saved:");
                b = new JButton("Start Uploading");
                JPanel p1 = new JPanel();
                p1.add(l1);
                p1.add(tf1);
                JPanel p2 = new JPanel();
                p2.add(l2);
                p2.add(tf2);
                JPanel p3 = new JPanel();
                p3.add(b);
                c.add(p1,BorderLayout.NORTH);
                c.add(p2,BorderLayout.CENTER);
                c.add(p3,BorderLayout.SOUTH);
                setSize(300,200);
                setVisible(true);
                b.addActionListener(
                        new ActionListener(){
                                public void actionPerformed(ActionEvent e){
                                        run();
                                        setVisible(false);
                                }
```

```
                }
            );
        }
        public void run()
        {
            try{
            Socket client = new
Socket(InetAddress.getByName("localhost"),clint_port);
            ObjectOutputStream out = new
ObjectOutputStream(client.getOutputStream());
            ObjectInputStream in = new ObjectInputStream(client.getInputStream());
            String str="";
            String mesg1="";
            String mesg2="";
            String mesg3="";
            mesg2=tf1.getText();
            mesg3=tf2.getText();
            out.writeObject(mesg3);
            out.flush();

            FileInputStream fs=new FileInputStream(mesg2);
          BufferedReader br = new BufferedReader(new InputStreamReader(fs));
            long l = fs.available();
            while(l>=0)
            {
                out.writeObject(br.readLine());
                out.flush();
                l--;
            }

            fs.close();
             str="EOF";
            out.writeObject(str);
            out.flush();
            in.close();
            out.close();
                        fs.close();
            out.close();
            in.close();
            client.close();
            }
            catch(Exception e)
            {
                System.out.println(e+"error");
            }
        }
```

```
}
```

ftps1.java

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ftps1
{
        final static int clint_port=28;
        final static int server_port=28;
        private JTextArea ta;
        private JTextField tf;
        public ftps1()
        {}

        public void run()
        {
                try
                {
                ObjectOutputStream out;
                ObjectInputStream in;
                ServerSocket socket=new ServerSocket(server_port);
                Socket ftpc = socket.accept();
                byte tbyt[]=new byte[4]         ;
                byte mesg[]=new byte[1];
                String mesg1;
                byte as[]=new byte[40];


                out = new ObjectOutputStream(ftpc.getOutputStream());
           out.flush();
                in = new ObjectInputStream(ftpc.getInputStream());

                String str ="";
                str=(String) in.readObject();
                char buf[]=new char[str.length()];
                str.getChars(0,str.length(),buf,0);
                CharArrayReader ini=new CharArrayReader(buf);
                BufferedReader outi=new BufferedReader(ini);
                mesg1=outi.readLine();
```

```
                FileInputStream fs = new FileInputStream(mesg1);


                BufferedReader br = new BufferedReader(new InputStreamReader(fs));
                long l = fs.available();
                while(l>=0)
                {
                        out.writeObject(br.readLine());
                        out.flush();
                        l--;
                }

                fs.close();
                 str="EOF";
                out.writeObject(str);
                out.flush();
                in.close();
                out.close();
                ftpc.close();
                socket.close();
                }
                catch(Exception e){System.out.println("FTP :"+e);}
        }
}
```

ftps2.java

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ftps2
{
	final static int clint_port=28;
	final static int server_port=28;
	private JTextArea ta;
	private JTextField tf;
	public ftps2()
	{}

	public void run()
	{
		try
		{
		ObjectOutputStream out;
		ObjectInputStream in;
		ServerSocket socket=new ServerSocket(server_port);
		Socket ftpc = socket.accept();
		byte tbyt[]=new byte[4]	;
		byte mesg[]=new byte[1];
		String mesg1;
		byte as[]=new byte[40];


		out = new ObjectOutputStream(ftpc.getOutputStream());
	   out.flush();
		in = new ObjectInputStream(ftpc.getInputStream());

		String str ="";
		str=(String) in.readObject();
		char buf[]=new char[str.length()];
		str.getChars(0,str.length(),buf,0);
		CharArrayReader ini=new CharArrayReader(buf);
		BufferedReader outi=new BufferedReader(ini);
```

```
                mesg1=outi.readLine();
                FileOutputStream fs = new FileOutputStream(mesg1);
                ObjectOutputStream outf = new ObjectOutputStream(fs);
                while(true)
                {
                        mesg1=(String)in.readObject();
                        outf.writeObject(mesg1);
                        outf.flush();
                        if(mesg1.equals("EOF"))
                                break;
                }



                fs.close();
                 str="EOF";

                in.close();
                out.close();
                ftpc.close();
                }
                catch(Exception e){System.out.println(""+e);}
        }
}
```

ftpcd1.java

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class ftpcd1 extends JFrame
{
        final static int clint_port=28;
        final static int server_port=28;
        private JTextField tf1;
      private JButton b;
        private JLabel l1;
        public ftpcd1()
        {
                Container c = getContentPane();
                c.setBackground(Color.orange);
                tf1 = new JTextField(10);

                l1 = new JLabel("Enter the file to be deleted:");

                b = new JButton("Start Deleting");
                JPanel p1 = new JPanel();
                p1.add(l1);
                p1.add(tf1);
                JPanel p3 = new JPanel();
                p3.add(b);
                c.add(p1,BorderLayout.NORTH);

                c.add(p3,BorderLayout.SOUTH);
                setSize(300,200);
                setVisible(true);
                b.addActionListener(
                        new ActionListener(){
                                public void actionPerformed(ActionEvent e){
                                        run();
                                        setVisible(false);
                                }
                        }
                );
```

```
	}
	public void run()
	{
		try{
		Socket client = new
Socket(InetAddress.getByName("localhost"),clint_port);
		ObjectOutputStream out = new
ObjectOutputStream(client.getOutputStream());
		ObjectInputStream in = new ObjectInputStream(client.getInputStream());

		String mesg1="";
		String mesg2="";

		mesg2=tf1.getText();

		out.writeObject(mesg2);
		out.flush();



		out.close();
		in.close();
		client.close();
		}
		catch(Exception e)
		{
			System.out.println(e+"error");
		}
	}

}
```

ftpsd1.java

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ftpsd1
{
	final static int clint_port=28;
	final static int server_port=28;
	private JTextArea ta;
	private JTextField tf;
	public ftpsd1()
	{}

	public void run()
	{
		try
		{
		ObjectOutputStream out;
		ObjectInputStream in;
		ServerSocket socket=new ServerSocket(server_port);
		Socket ftpc = socket.accept();
		byte tbyt[]=new byte[4];
		byte mesg[]=new byte[1];
		String mesg1;
		byte as[]=new byte[40];


		out = new ObjectOutputStream(ftpc.getOutputStream());
			out.flush();
		in = new ObjectInputStream(ftpc.getInputStream());

		String str ="";
		str=(String) in.readObject();
```

```
                char buf[]=new char[str.length()];
                str.getChars(0,str.length(),buf,0);
                CharArrayReader ini=new CharArrayReader(buf);
                BufferedReader outi=new BufferedReader(ini);
                mesg1= outi.readLine();
            FileOutputStream fs= new FileOutputStream(mesg1);
                File ft = new File(mesg1);
            ft.delete();
                out.flush();
                in.close();
                out.close();
                ftpc.close();
                socket.close();
                }
                catch(Exception e){System.out.println("FTP: "+e);}
        }
}
```

Interface.java

```
import java.net.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Interface extends JFrame
{
 private static LServer server;
 private static JDesktopPane desk;
 public static void main(String args[])
 {
  final JFrame frame = new JFrame("INTRODUCTION");
  final Container c = frame.getContentPane();
  server = new LServer();
  c.setLayout(new FlowLayout());
  JTextArea intro = new JTextArea();
  intro.setFont(new Font("TimesRoman", Font.BOLD, 17));

  intro.setText("\t FTP IMPLEMENTATION\t\n\nProject Guide:\t\t\t\tDeveloped
by:\nDR GOLDIE GABRANI\n\t\t\t\tNEELABH SAXENA\n\t\t\t\tNISHANT
GUPTA\n\t\t\t\tPARAG MEHRA");
  intro.setBackground(new Color(Integer.parseInt("AEECCF",16)));
  intro.setForeground(Color.black);
  intro.setEditable(false);
  JPanel p1 = new JPanel();
  p1.setLayout(new FlowLayout(FlowLayout.LEFT));
  p1.add(intro);
  JButton b1 = new JButton("START");
  b1.setFont(new Font("Comic Sans", Font.BOLD, 14));
  JButton b2 = new JButton("CANCEL");
  b2.setFont(new Font("Comic Sans", Font.BOLD, 14));
  JPanel p2 = new JPanel();

  p2.setLayout(new FlowLayout(FlowLayout.CENTER));
```

```
p2.setBackground(new Color(Integer.parseInt("ACCCFF",16)));
p2.add(b1);
p2.add(b2);
c.add(p1,BorderLayout.CENTER);
c.add(p2,BorderLayout.SOUTH);
c.setBackground(new Color(Integer.parseInt("AACCFF",16)));
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(600,280);
frame.setLocation(50,50);
frame.setForeground(Color.green);
frame.setVisible(true);
b1.addActionListener(
 new ActionListener(){
  public void actionPerformed(ActionEvent e){
   frame.setVisible(false);
   server.setSize(600,500);
   server.setLocation(50,50);
   server.setVisible(true);
   server.addWindowListener(
    new WindowAdapter()
    {
     public void windowClosing(WindowEvent e)
     {
      System.exit(0);
     }
    }
   );
   desk = server.share();
  }
 }
);
b2.addActionListener(
 new ActionListener(){
  public void actionPerformed(ActionEvent e){
   System.exit(0);
  }
 }
);
try
{
ServerSocket serve  = new ServerSocket(5009);
int i = 1;
while(true)
{
 Socket conn = serve.accept();
 Proxy pro = new Proxy(conn,desk,i);
```

```
   pro.start();
   i++;
  }
  }
  catch(Exception a){System.out.println("main::"+a);}

 }
}

class Proxy extends Thread
{
 private Socket connect;
 private int count;
 private JDesktopPane desk;
 private String msg = "";
 private ObjectOutputStream output;
 private ObjectInputStream input;
 private JTextArea tb;
 private JTextField tg;
 private JInternalFrame client;
 Proxy(Socket c,JDesktopPane d,int i)
 {
  connect=c;
  desk=d;
  count=i;
 }
 public void run()
 {
  try
  {
   InetAddress cladd = connect.getInetAddress();
   String name = cladd.getHostName();
   output = new ObjectOutputStream(connect.getOutputStream());
   output.flush();
   input= new ObjectInputStream(connect.getInputStream());

   client = new JInternalFrame(name+count,true,true,true,true);
   Container c = client.getContentPane();
   c.setLayout(new FlowLayout());
   tb = new JTextArea(7,15);
   tg = new JTextField(15);
   tb.setEditable(false);
   tg.addActionListener(
    new ActionListener(){
     public void actionPerformed(ActionEvent e){
      sendData();
```

```
    }
   }
  );
  c.add(new JScrollPane(tb),BorderLayout.CENTER);
  c.add(tg,BorderLayout.SOUTH);
  client.setSize(200,200);
  client.setLocation((10*count)+50,(10*count)+50);
  client.setBackground(Color.blue);
  client.show();
  desk.add(client);
  desk.setVisible(true);

  do
  {
   msg = (String) input.readObject();
   tb.append(name+">>"+msg+'\n');
   if(msg.equals("download."))
    askedhttp();
   if(msg.equals("upload."))
    askedftp();
   if(msg.equals("delete."))
    askedsmtp();
  }while((!msg.equals("quit"))||(!(tg.getText()).equals("quit")));

  tb.append("Connection Terminated by client "+count+'\n');

  output.close();
  input.close();
  connect.close();
  quit();
 }
 catch(SocketException ex)
 {
  client.setVisible(false);
 }
 catch(Exception e)
 {
  System.out.println("run::"+e);
 }
}
public void sendData()
{
 try
 {
  String mesg = tg.getText().trim();
  tb.append(mesg+'\n');
```

```
   output.writeObject(mesg);
   output.flush();
  }
  catch(Exception d)
  {
   System.out.println("sendData::"+d);
  }
 }
 public void quit()
 {
  client.setVisible(false);
 }
 public void askedhttp()
 {
ftps1 obj = new ftps1();
obj.run();
 }
 public void askedftp()
 {
ftps2 obj2=new ftps2();
obj2.run();
 }
 public void askedsmtp()
 {ftpsd1 obj3=new ftpsd1();
  obj3.run();
 }
}
```

LClient.java

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class LClient extends JFrame
{
 private static JTextArea tb;
 private static JTextField tg;
 private static String msg = "";
 private static ObjectOutputStream out;
 private static ObjectInputStream in;
 private static String name = "";
 private static Socket conn;
 public LClient()
 {
  super("FTP CLIENT");
  JLabel label1 = new JLabel("Status");
  JButton b = new JButton("Connect");
  JButton df = new JButton("Download File");
  JButton uf = new JButton("Upload File");
  JButton x = new JButton("Delete");
  JButton d = new JButton("Disconnect");
b.setBackground(new Color(Integer.parseInt("EEAACF",16)));
df.setBackground(new Color(Integer.parseInt("EEAACF",16)));
uf.setBackground(new Color(Integer.parseInt("EEAACF",16)));
x.setBackground(new Color(Integer.parseInt("EEAACF",16)));
d.setBackground(new Color(Integer.parseInt("EEAACF",16)));
  Container c = getContentPane();
  c.setBackground(new Color(Integer.parseInt("AABCFF",16)));
  tg = new JTextField(25);
  tb = new JTextArea(8,25);
tg.setBackground(new Color(Integer.parseInt("ABCDFF",16)));
tb.setBackground(new Color(Integer.parseInt("BAACDF",16)));
```

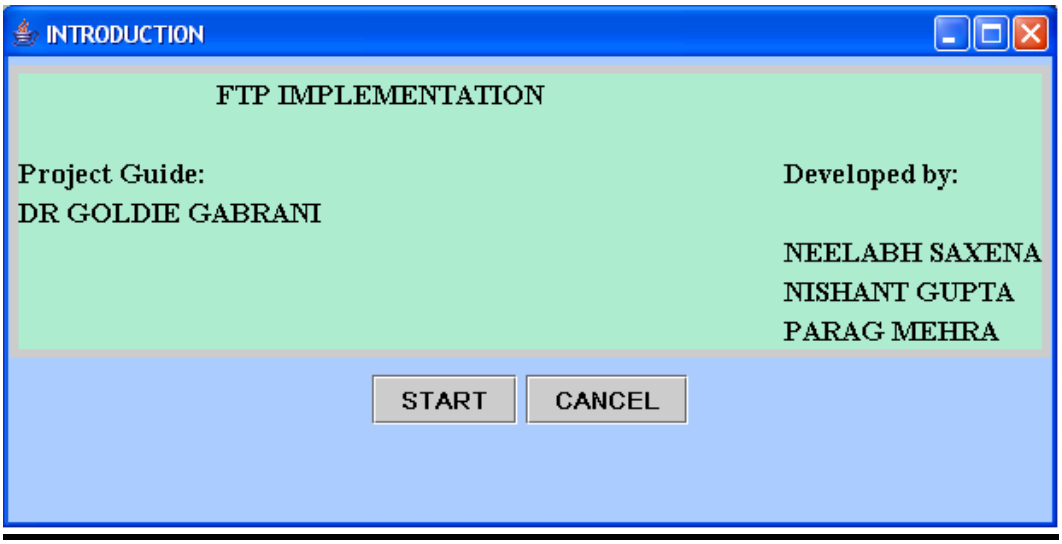
```
  tb.setEditable(false);
//JPanel p1 = new JPanel();
//p1.setLayout(new FlowLayout(FlowLayout.LEFT));
//p1.add(x);
JPanel p2 = new JPanel();
  p2.setLayout(new FlowLayout(FlowLayout.LEFT));
  p2.add(df);
  p2.add(uf);
  p2.add(x);

  JPanel p3 = new JPanel();
p3.setBackground(new Color(Integer.parseInt("EABCDF",16)));
  p3.setLayout(new FlowLayout(FlowLayout.LEFT));
  p3.add(label1);
  p3.add(new JScrollPane(tb));
  p3.add(tg);
p3.add(b);
  p3.add(d);
  c.add(p3,BorderLayout.CENTER);
  c.add(p2,BorderLayout.SOUTH);
  //c.add(p1,BorderLayout.SOUTH);
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  setSize(300,400);
  setLocation(100,50);
  setVisible(true);
  tg.addActionListener(
   new ActionListener(){
    public void actionPerformed(ActionEvent e){
     sendData();
    }
   }
  );
  b.addActionListener(
   new ActionListener(){
    public void actionPerformed(ActionEvent e){
try{
 String mesg = tg.getText().trim();
   tb.append("Connect to "+mesg);
  Socket client = new Socket(InetAddress.getByName(mesg),5009);
  out = new ObjectOutputStream(client.getOutputStream());
   in = new ObjectInputStream(client.getInputStream());
    }
catch(Exception f){System.out.println(e);}
    }
   }
  );
```

```
 df.addActionListener(
  new ActionListener(){
   public void actionPerformed(ActionEvent e){
    askhttp();
   }
  }
 );
 uf.addActionListener(
  new ActionListener(){
   public void actionPerformed(ActionEvent e){
    asksmtp();
   }
  }
 );
 x.addActionListener(
  new ActionListener(){
   public void actionPerformed(ActionEvent e){
    askftp();
   }
  }
 );
 d.addActionListener(
  new ActionListener(){
   public void actionPerformed(ActionEvent e){
    quit();
   }
  }
 );
}
public static void main(String s[])throws IOException
{
    LClient client = new LClient();
}
public static void sendData()
{
 try
 {
  String mesg = tg.getText().trim();
  tb.append("Connect to "+mesg);
 Socket client = new Socket(InetAddress.getByName(mesg),5009);
out = new ObjectOutputStream(client.getOutputStream());
  in = new ObjectInputStream(client.getInputStream());

 }
 catch(Exception d)
 {
```

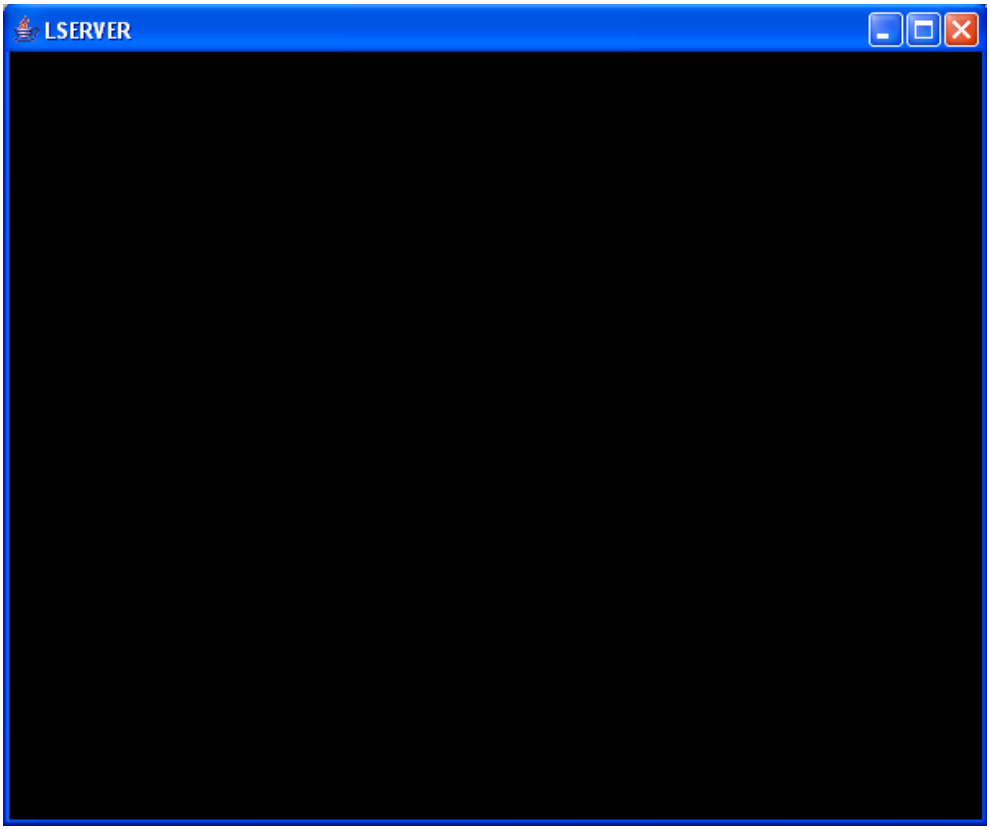
```
   System.out.println(d);
  }
 }
 public static void quit()
 {
  try{out.writeObject("quit");
  out.close();
  in.close();
  conn.close();}catch(Exception j){System.out.println(j);}
 }
 public static void finexit()
 {
  System.exit(0);
 }
 public static void askhttp()
 {
  try{
  out.writeObject("download.");
  out.flush();
ftpc1 obj = new ftpc1();

  }catch(Exception t){System.out.println("a:"+t);}
 }
 public static void asksmtp()
 {
  try{
  out.writeObject("upload.");
  out.flush();
  ftpc2 obj2 = new ftpc2();
  }catch(Exception i){System.out.println(i);}
 }
 public static void askftp()
 {
  try{
  out.writeObject("delete.");
  out.flush();
  ftpcd1 obj = new ftpcd1();

  }catch(Exception h){System.out.println(h);}
 }
}
```

Output

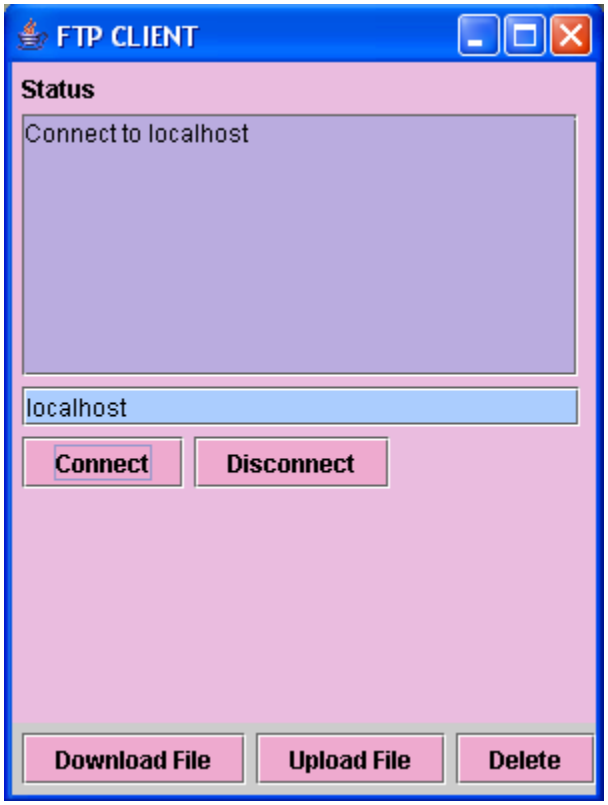
On Running Interface.java



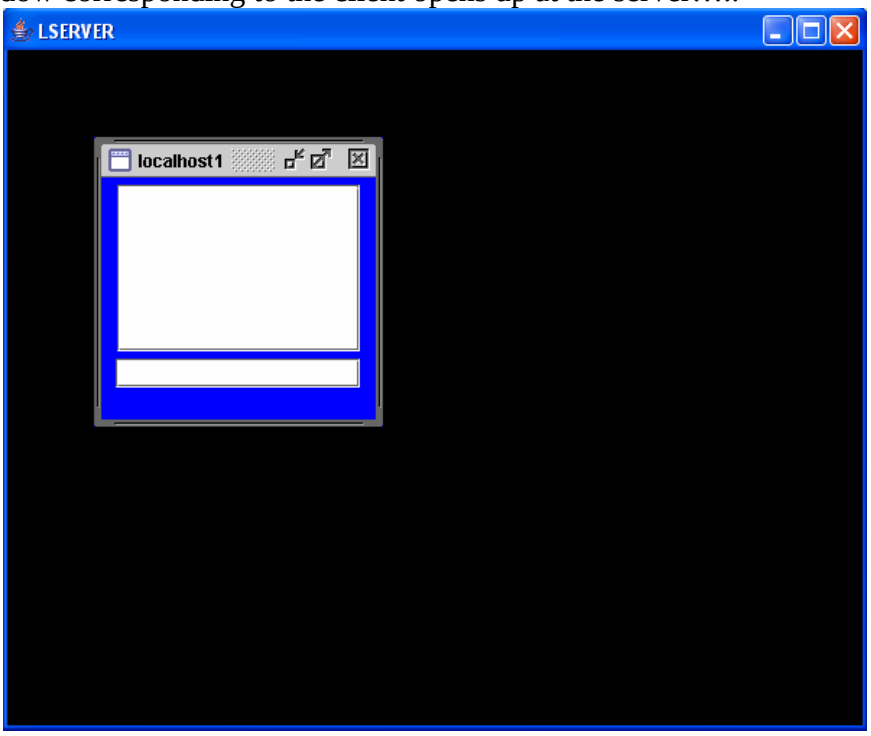
Then on Clicking Start….



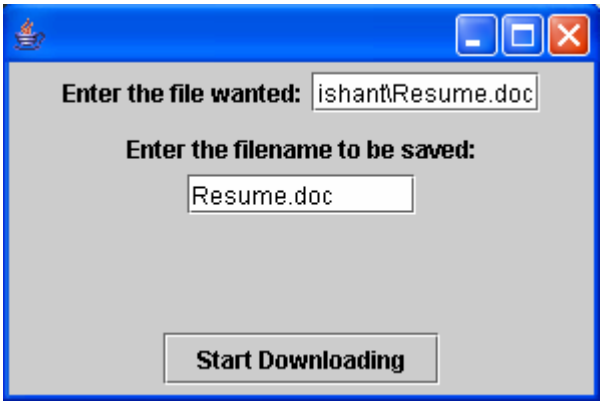
Now we run LCLient.java and connect to localhost



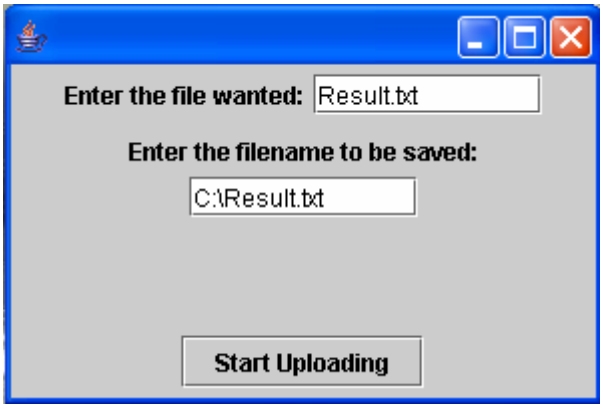
A new window corresponding to the client opens up at the server…..



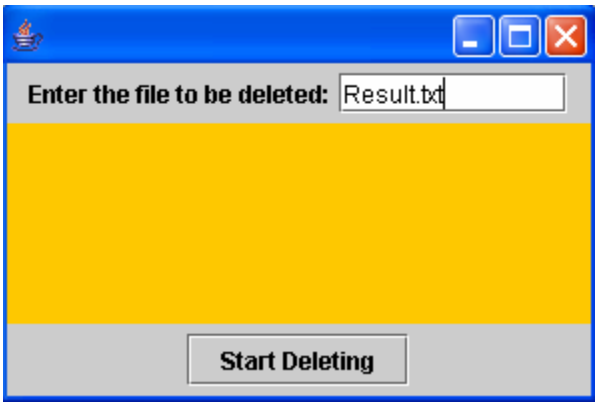
Now click on the download button…



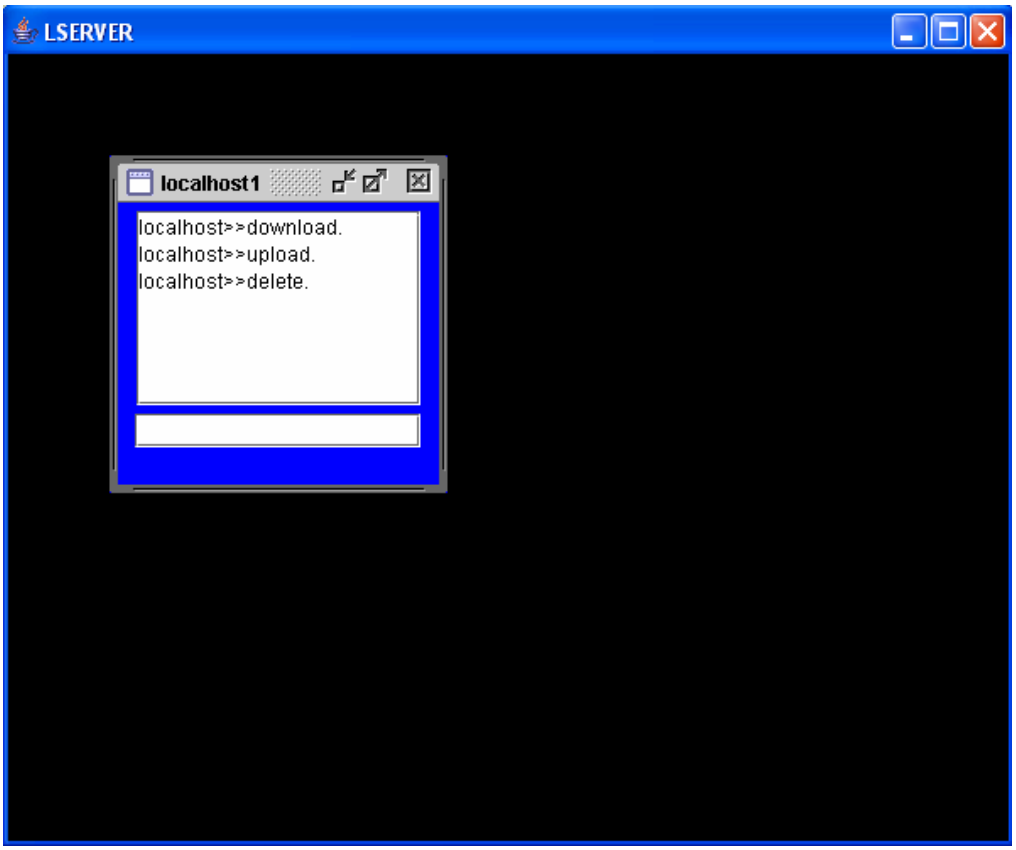
Now selecting to Upload a file….



Similalarly to delete a file….



The status of the Server during this time…..



Limitations

Despite our best efforts, there were some limitations which we observed during the course of our project. Some of them are:

- The implementation of the File Transfer Protocol in the programming language C is platform dependent, i.e. it has been made with Linux in mind. Though only small changes would be required to make the code compatible with windows as well.

- All the functionalities of the File Transfer Protocol could not be implemented. Though all the basic as well as the commonly used functionalities have been implemented in both C and JAVA making it a complete software.

- In the JAVA implementation of the code, the transferred files get some unidentified characters prepended and appended to their contents.

CONCLUSION

The current version of the File Transfer Protocol application has met all the expectations that had been hoped for, with only a few problems encountered. The project helped us to understand the concepts involved in socket programming and implement them in a client server model. Although there are some limitations in the software we are sure that they can be removed.

References

1. ***Unix Network Programming***
   By: W. Richard Stevens

2. ***Programming in JAVA***
   By: Herbert Schildt

3. ***TCP/IP Internetworking***
   By: Douglas E. Comer