FILE TRANSFER USING SOCKET PROGRAMMING ON LINUX...

BY

ANKIT PRADHAN – 041036

SAURABH KUMAR GUPTA – 041048

MAY-2008

Submitted in partial fulfillment of the Degree Bachelors of Technology.

DEPARTMENT OF ELECTRONICS AND COMMUNICATION

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT.

CERTIFICATE

This is to certify that the work entitled, "File Transfer Using Socket programming On Linux" submitted by Ankit Pradhan and Saurabh Kumar Gupta in partial fulfillment for the award degree of Bachelor of Technology in Electronics and Communication of Jaypee University of Information technology has been carried under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

PROJECT SUPERVISOR

Mr. Vinay Kumar
Sr.Lecturer,
Department of Electronics and Communication
JUIT

Dr. S.V.Bhooshan
Head.Of.Department
E.C.E
JUIT

ACKNOWLEDGEMENTS

The project in this report is an outcome of continual work over a period of four months and intellectual support from various sources. Obligations thus occurred in completing the work have been many. It is therefore almost impossible to express adequately the debts owed to many persons who have been instrumental in imparting this work a successful status. It is however a matter of pleasure to express our gratitude and appreciation to those who have been contributing to bring about this project.

We take this opportunity to thank our esteemed mentor and supervisor Mr. Vinay Kumar, Lecturer Department of Electronics and Communication, JUIT, for lending us stimulating suggestions, innovative quality guidance and creative thinking. His practicality, constructive criticism, constant encouragement and advice helped us in all the stages of the project. His scientific views and scientific approach will always be the source of motivation for us. We are grateful to him for the support, he provided us in doing things at our pace and for being patient with our mistakes.

We would like to express our gratitude to Mr. S.V.Bhooshan, Head of Department Of Electronics and Communication Engineering, JAYPEE UNIVERSITY of INFORMATION TECHNOLOGY for providing us the opportunity and facilities to undergo this project.

We were fortunate to have very supportive friends with whom, it was pleasure to work. This unreserved help, adjusting nature and friendly atmosphere will always be cherished and remembered.

SAURABH KUMAR GUPTA
041048
E.C.E

(AMKIT PRADHAN)
(041036)

ABSTRACT

This is the era of Hi-tech communication. We know this very well that revolutions do not born on streets. They are born in our mind. There are lots of technologies that are continuously emerging in the technology field to provide us much more convenience and reliability. Networking is in one of those fields that has a huge application everywhere, whether it is engineering field, corporate field or medical field. Now a day we want our communication without any sort of interruption and with full security. We designed the language for simple and intuitive description of communication protocols. So that we can transfer our data easily without any threat of data – stealing.

Communication performance between two processes in their own domains on the same physical machine gets improved but it does not reach our expectation. This paper presents the design and implementation of high-performance inter-domain communication mechanism that maintains binary compatibility for applications written in standard socket interface. We found that three overheads mainly contribute to the poor performance; those are TCP/IP processing cost in each domain, and long communication path between both sides of a socket. Our project achieves high performance by bypassing TCP/IP stacks and providing a direct, accelerated communication path between domains in the same machine. Moreover, we introduce the socket architecture to support full binary compatibility with as little effort as possible.

We implemented our design on Linux (Fedora 3.0) and evaluated basic performance, and binary compatibility using binary image of real socket applications. In our tests, we have proved that our project realizes the high performance that is comparable to UNIX domain socket and ensures full binary compatibility. Those applications worked perfectly well.

Table of Contents:

Chapter 1: Introduction and TCP/IP

1.1 Platform and Compiler:

The code contained in this Report was complied on a Linux P.C using a GCC compiler. It could, however, build on any platform that uses GCC compiler. Now we'll explain you that how to compile a C code using GCC compiler in LINUX/FEDORA/SUSE and also some important commands of GCC. The information starts with the command and a list of options (including the sub-processes). The following is a list of file extensions. Typically, the executable doesn't have extension.

File extension	Description
file_name.c	C source code which must be preprocessed.
file_name.i	C source code which should not be preprocessed.
file_name.ii	C++ source code which should not be preprocessed.
file_name.h	C header files (not to be compiled or linked).
file_name.cc file_name.cp file_name.cxx file_name.cpp file_name.c++ file_name.C	C++ source code which must be preprocessed. For file_name.cxx, the xx must both be literally character x and file_name.C, is capital c.
file_name.s	Assembler code.
file_name.S	Assembler code which must be preprocessed.
file_name.o	Object file by default, the object file name for a source file is made by replacing the extension .c, .i, .s etc with .o

1.2 Some Background Story:

This background story tries to introduce the terms used in network programming and also to give you the big picture.

The following figure is a typical physical network devices connection.

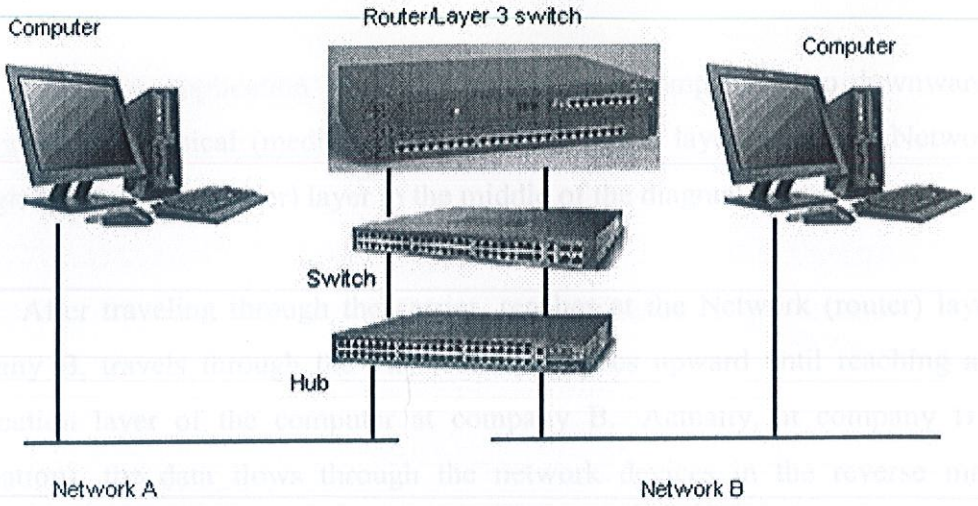


Figure 1

- Using a simple network shown in the above Figure, let trace the data stream flow from Network A to Network B, by assuming that Network A is company A's network and Network B is company B's network.

- Physically, the flow of the data stream is from a computer in Network A (source) will go through the hub, switch and router.

- Then the stream travel through the carrier such as Public Switch Telephone Network (PSTN) and leased line (copper, fiber or wireless – satellite) and finally reach Network B's router, go through the switch, hub and finally reach at the computer in company B (destination).

- The OSI (Open System Interconnection) 7 layer stack mapping is shown next.

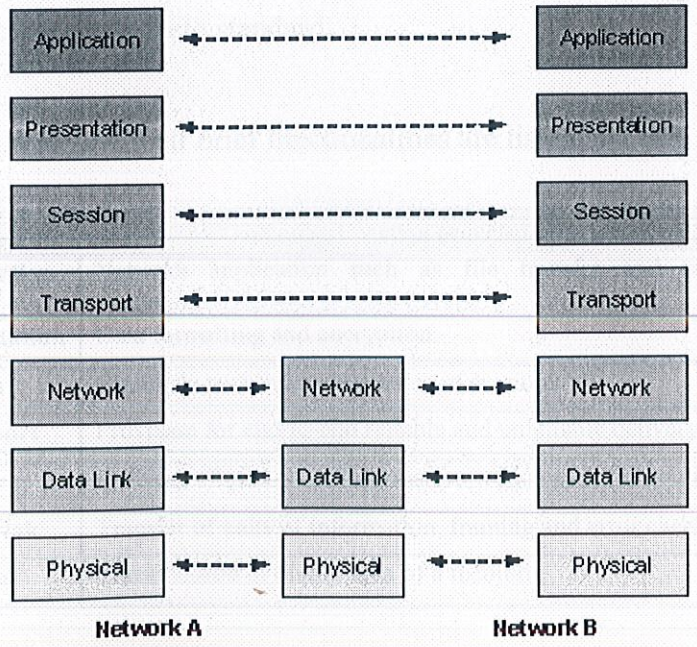


Figure 2

- From the Application layer of a computer at company A go downward the layer until the Physical (medium such as Cat 5 cable) layer, then exit Network A through the Network (router) layer in the middle of the diagram.

- After traveling through the carrier, reaches at the Network (router) layer of company B, travels through the Physical layer, goes upward until reaching at the Application layer of the computer at company B. Actually, at company B (the destination), the data flows through the network devices in the reverse manner compared to what happened at company A (the source).

- In contrast to TCP/IP, the OSI approach started from a clean slate and defined standards, adhering tightly to their own model, using a formal committee process without requiring implementations.

- Internet protocols use a less formal but more practical engineering approach, where anybody can propose and comment on Request for Comment (RFC) documents, and implementations are required to verify feasibility.

- The OSI protocols developed slowly, and because running the full protocol stack is resource intensive, they have not been widely deployed, especially in the desktop and small computer market. In the meantime, TCP/IP and the internet were developing rapidly, with deployment occurring at a very high rate, which is why the TCP/IP suite becomes a de facto standard.

- The OSI layer and their brief functionalities are listed in Table 1

OSI Layer	Function provided
Application	Network application such as file transfer and terminal emulation
Presentation	Data formatting and encryption.
Session	Establishment and maintenance of sessions.
Transport	Provision for end-to-end reliable and unreliable delivery.
Network	Delivery of packets of information, which includes routing.
Data Link	Transfer of units of information, framing and error checking.
Physical	Transmission of binary data of a medium.

Table 1.

- In the practical implementation, the standard used is based on TCP/IP stack. This TCP/IP stack is a de facto standard, not a pure standard but it is widely used and adopted.

- The equivalent or mapping of the OSI and TCP/IP stack is shown below. It is divided into 4 layers. The Session, Presentation and Application layers of OSI have been combined into one layer, Application layer.

- Physical and data link layers also become one layer. Different books or documentations might use different terms, but the 4 layers of TCP/IP are usually referred.

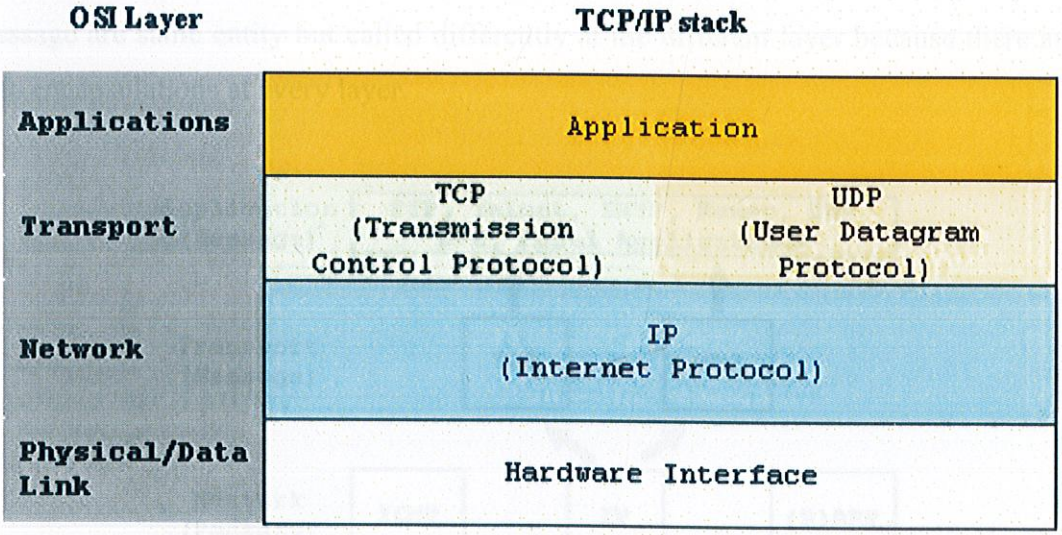


Figure 3

- Now we will concentrate more on the Transport and Network layer of the TCP/IP stack.

- More detailed TCP/IP stack with typical applications is shown next.

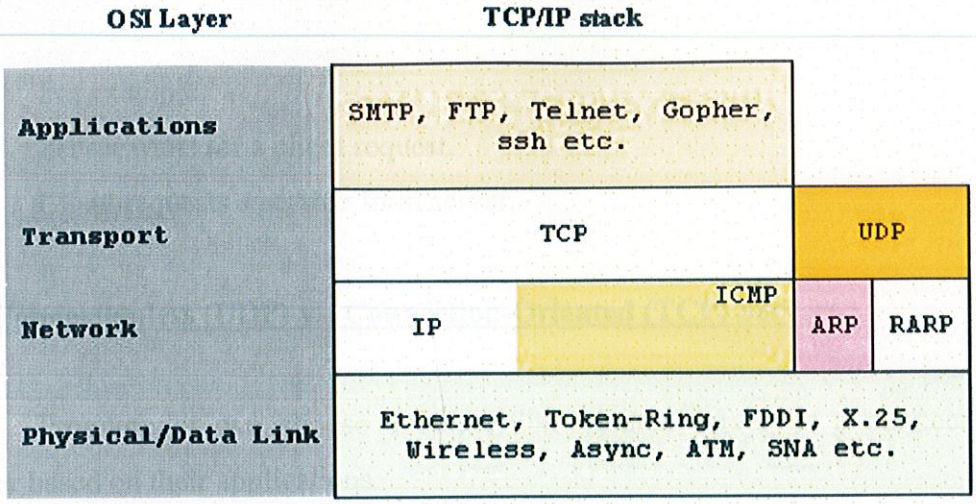


Figure 4

➢ The following figure is a TCP/IP architectural model. Frame, packet and message are same entity but called differently at the different layer because there are data encapsulations at every layer.

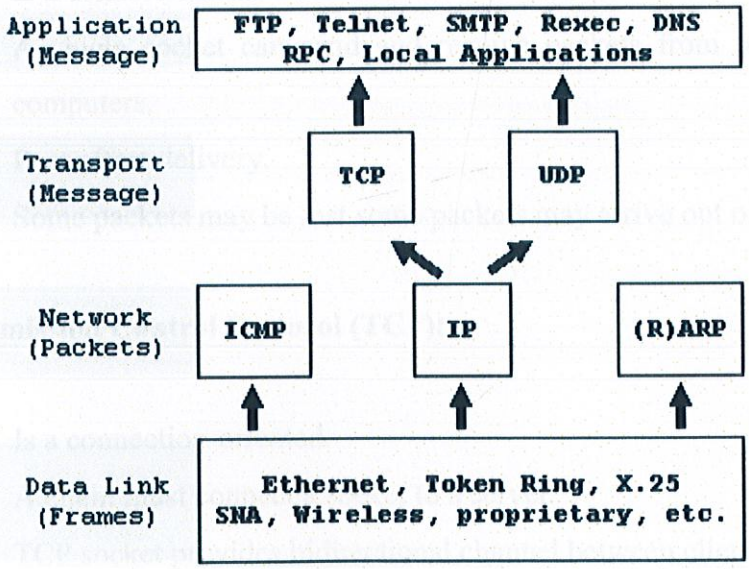


Figure 5

1.3 Client – Server Model :

➢ TCP/IP enables peer-to-peer communication.

➢ Computers can cooperate as equals or in any desired way.

➢ Most distributed applications have special roles.

For example:

Server waits for a client request.

Client requests a service from server.

1.4 Connectionless (UDP) vs. Connection-Oriented (TCP) Servers :

➢ Programmer can choose a connection-oriented server or a connectionless server based on their applications.

➢ In Internet Protocol terminology, the basic unit of data transfer is a **datagram**. This is basically a header followed by some data. The datagram socket is connectionless.

➢ **User Datagram Protocol (UDP):**

- Is a connectionless.
- A single socket can send and receive packets from many different computers.
- Best effort delivery.
- Some packets may be lost some packets may arrive out of order.

➢ **Transmission Control Protocol (TCP):**

- Is a connection-oriented.
- A client must connect a socket to a server.
- TCP socket provides bidirectional channel between client and server.
- Lost data is re-transmitted.
- Data is delivered in-order.
- Data is delivered as a stream of bytes.
- TCP uses flow control.

Chapter 2: Elementary Sockets and Structures

2.1 Types of Internal Sockets:

First of all we'll see what is a socket and how do they work???

Socket - Socket is an Application Programming Interface (API) used for Interprocess Communications (IPC).
It is a well defined method of connecting two processes, locally or across a network. It is a Protocol and Language Independent. Often referred to as Berkeley Sockets or BSD Sockets.

Sockets are commonly used for client –server interaction. Typical system configuration places the server on one machine, with the clients on other machine. The clients connect to the server, exchange information and then disconnect.

A socket has a typical flow of events in connection- oriented client –to – server model, the socket on the server waits for requests from a client. To do this the server first establishes an address that clients can use to find the server. When the address is established, the server waits for clients to request a service. The client –to-server data exchange takes place when a client connects to the server through a socket. The server performs the client request and sends the reply back to the client.

There are 2 types of internal sockets:

One is "stream socket" and another is "datagram socket",which may also be referred as "SOCK_STREAM" and "SOCK_DGRAM" respectively.

Stream sockets are reliable two way connected communication stream.if you output two items into the socket in the order "1, 2". Then they'll arrive in the order "1, 2" at the opposite end.

- Socket types are listed in the following Table.

Socket type	Protocol	TYPE
STREAM	TCP, Systems Network Architecture (SNA-IBM), Sequenced Packet eXchange (SPX-Novell).	SOCK_STREAM
SEQPACKET	SPX.	SOCK_SEQPACKET
DGRAM	UDP, SNA, Internetwork Packet eXchange (IPX-Novell).	SOCK_DGRAM
RAW	IP.	SOCK_RAW

Table 2: socket combinations.

2.2 various socket structures :

In this section and that follows we will discuss the socket APIs details: the structures, functions, macros and types.

1.) Struct sockaddr

```
{
        u_char   sa_len ;
        u_short  sa_family ;
        char     sa_data[14] ;  //14 bytes of protocol address.
} ;
```

- The short integer that defines the address family .The value that is specified for address family on the **socket ()** call.
- Fourteen bytes that are reserved to hold the address itself.
- Depending on the address family, sa_data could be a file name or a socket end point.
- sa_family can be a variety of things, but it,ll be AF_INET for everything we do here.
- sa_data contains a destination address and port number for the socket.
- To deal with struct sockaddr, a parallel structure is created: struct sockaddr_in ("in" for "Internet".)

2.) Struct sockaddr_in

```
{
        Short int    sin_family ;              // Address Family
        Unsigned short int    sin_port ;       // port number
        Struct in_addr    sin_addr ;           // internet address
        Unsigned char    sin_zero[8] ;         // same size as struct sockaddr
} ;
```

The sockaddr_in data structure contains an IP Address and a port number.

- The sin_family field is the address family (always AF_INET for TCP and UDP).
- The sin_port field is the port number, and the sin_addr field is the Internet address. The sin_zero field is reserved, and you must set it to hexadecimal zeroes.
- Data type struct in_addr - this data type is used in certain contexts to contain an Internet host address. It has just one field, named s_addr, which records the host address number as an unsigned long int.
- sockaddr_in is used to specify an endpoint.
- sin_addr could be u_long.
- sin_addr is 4 bytes and 8 bytes are unused.
- The sin_port and sin_addr must be in **Network Byte Order**.

3.) Struct in_addr

```
{
        unsigned long int   s_addr ;
} ;
```

- This structure is used in certain contexts to contain an Internet host address. It has just one field, named `s_addr`, which records the host address number as an `unsigned long int`.

4.) Struct hostent

```
{
        char  *h_name ;
```

```
        char  **h_aliases ;
        int    h_addrtype ;
        int    h_length ;
        char  **h_addr_list ;
    } ;
```

Data Type	Description
char *h_name	This is the "official" name of the host.
char **h_aliases	These are alternative names for the host, represented as a null-terminated vector of strings.
int h_addrtype	This is the host address type; in practice, its value is always AF_INET. In principle other kinds of addresses could be represented in the data base as well as Internet addresses; if this were done, you might find a value in this field other than AF_INET.
int h_length	This is the length, in bytes, of each address.
char **h_addr_list	This is the vector of addresses for the host. Recall that the host might be connected to multiple networks and have different addresses on each one. The vector is terminated by a null pointer.
char *h_addr	This is a synonym for h_addr_list[0]; in other words, it is the first host address.

Table 3

2.3 IP Addresses and Their Concept :

➢ Numeric IP Addresses :

5.) Ipv4 Internet addresses are 32 bit integers.

6.) For convenience they are displayed in "dotted decimal" format.

7.) Each byte is presented as a decimal number.

8.) Dots separate the bytes for example: 172.16.7.79

➢ IP Addresses Classification:

- An IP address has two parts: The network portion and the host portion.

- The network portion is unique to each Company/organization/domain/group/network, and the host portion is unique to each system (host) in the network.
- Where the network portion ends and the host portion begin is different for each class of IP address.
- You can determine this by looking at the two high-order bits in the IP address.

192.168.1.100			
xxxxxxxx.xxxxxxxx.xxxxxxxx.xxxxxxxx Byte 1.Byte 2.Byte 3.Byte 4			
Class and Network size	**Range (decimal)**	**Network ID**	**Host ID**
Class A (Large)	1 -127	Byte 1	Bytes 2, 3, 4
Class B (Medium)	128 – 191	Bytes 1, 2	Bytes 3, 4
Class C (Small)	192 – 223	Bytes 1, 2, 3	Bytes 4

Table 4

- The first four bits (bits 0-3) of an address determine its class:

0xxx = class A bits 1-7 define a network. bits 8-31 define a host on that network. 128 networks with 16 million hosts.
10xx = class B bits 2-15 define a network. bits 16-31 define a host on that network. 16384 networks with 65536 hosts.
110x = class C bits 3-23 define a network. bits 24-31 define a host on that network. 2 million networks with 256 hosts.

Table 5

- The IP network portion can represent a very large network that may spans multiple geographic sites.
- To make this situation easier to manage, you can use subnetworks. Subnetworks use the two parts of the address to define a set of IP addresses that are treated as group. The subnetting divides the address into smaller networks.

- You configure a subnetwork by defining a mask, which is a series of bits. Then, the system performs a logical AND operation on these bits and the IP address.
- The 1 bit defines the subnetwork portion of the IP address (which must include at least the network portion). The 0 bits define the host portion.
- **Class D** is a multicast addresses and **class E** is reserved. This you can realize through the following figure.

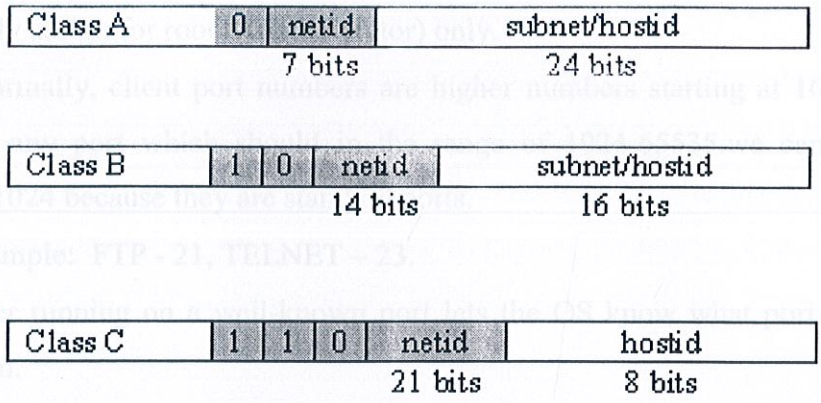


Figure 6

- Nowadays we use classless IP address. That means we subnet the class type IP into smaller subnet or smaller group of IP addresses creating smaller networks.
- Before the IPV4 run out of the IP addresses, now we have IPV6 with 128 bits.

➢ **IPv6 :**

- Current IP is IPv4 (Internet Protocol version 4).
- IPv4 has 32 bit addresses.
- Due to splitting addresses, 32 bits is not enough.
- IPv6 will have 128 bit addresses.
- Addresses will be shown in a colon hexadecimal format with internal strings of 0s omitted. For example:
  "69DC:88F4:FFFF:0:ABCD:DBAC:1234:FBCD:A12B::F6"
- New service types exist to accommodate IPv6 such as multimedia and wireless.

2.4 Port Numbers:

- It is 16 bit integers. So we have $2^{16} = 65536$ ports maximum.
- It is unique within a machine/IP address..
- To make a connection we need an IP address and port number of the protocol.
- The connection defined by : "IP address & port of server + IP address & port of client ".
- Normally, server port numbers are low numbers in the range 1 – 1023 and normally assign for root (Administrator) only.
- And normally, client port numbers are higher numbers starting at 1024. we can choose any port which should in the range of 1024-65535.we cannot choose below 1024 because they are standard ports.
- For example: FTP - 21, TELNET – 23.
- A server running on a well-known port lets the OS know what port it wants to listen on.
- Whereas a client normally simply lets the operating system picks a new port that isn't already in use.

2.5 Inet Functions:

There are some inet functions that we,ll use in our code.these inet functions are described below.

inet_addr , inet_aton , and inet_ntoa functions :

`inet_aton`, `inet_ntoa`, and `inet_addr` convert an IPv4 address from a dotted-decimal string (e.g., `"206.168.112.96"`) to its 32-bit network byte ordered binary value. You will probably encounter these functions in lots of existing code.

#include <arpa/inet.h>
int inet_aton(const char *strptr, struct in_addr *addrptr);
Returns: 1 if string was valid, 0 on error

in_addr_t inet_addr(const char *strptr);
Returns: 32-bit binary network byte ordered IPv4 address; INADDR_NONE if error
char *inet_ntoa(struct in_addr inaddr);
Returns: pointer to dotted-decimal string

- `inet_aton`, converts the C character string pointed to by strptr into its 32-bit binary network byte ordered value, which is stored through the pointer addrptr. If successful, 1 is returned; otherwise, 0 is returned.

- `inet_addr` does the same conversion, returning the 32-bit binary network byte ordered value as the return value. The problem with this function is that all 2^{32} possible binary values are valid IP addresses (0.0.0.0 through 255.255.255.255), but the function returns the constant `INADDR_NONE` (typically 32 one-bits) on an error. This means the dotted-decimal string 255.255.255.255 (the IPv4 limited broadcast address) cannot be handled by this function since its binary value appears to indicate failure of the function.

- The `inet_ntoa` function converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string. The string pointed to by the return value of the function resides in static memory. This means the function is not reentrant,finally notice that this function takes a structure as its argument, not a pointer to a structure.

Chapter 3: Header files used in socket programming

Here is the list of all header files that we have used in our C code of socket programming:

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <string.h>

#include <netdb.h>

#include <fcntl.h>

#include <sys/stat.h>

Now we,ll discuss each of them precisely.

#include <stdio.h>

stdio.h, which stands for "standard input/output header", is the header in the C standard library that contains macro definitions, constants, and declarations of functions and types used for various standard input and output operations.

Variables defined in the stdio.h header include:

- **Stdin** – a pointer to file which refers to the standard input stream, usually a keyboard.
- **Stdout** – a pointer to file which refers to the standard output stream, usually a display terminal.
- **Stderr** –a pointer to file which refers to the standard error stream, often a display terminal.

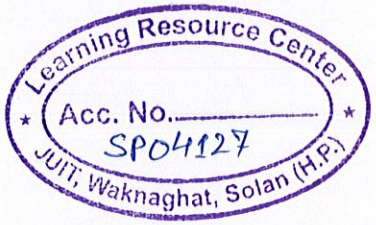
Functions declared in stdio.h are extremely popular, since as a part of the C standard library, they are guaranteed to work on any platform that supports C. Applications on

a particular platform might, however, have reasons to use the platform's I/O routines, rather than the stdio.h routines.

#include <stdlib.h>

stdlib.h - is the header of the **general purpose standard library** of C programming language which includes functions involving memory allocation, process control, conversions and others.

The stdlib.h and stddef.h header files define the macro NULL, which yields a null pointer constant, and represents a pointer value that is guaranteed not to point to a valid address in memory. NULL may be defined as a constant expression equal to int zero, long int zero, or zero cast to a void * pointer:




```
#define NULL 0
#define NULL 0L
#define NULL ((void*) 0)
```

Member data type:

A datatype called size_t is defined in the stdlib.h library, which is used to represent the size of an object. Library functions that take sizes expect them to be of type size_t, and the sizeof operator evaluates to size_t.

#include <sys/types.h>

Defines various data types. Also includes prototypes, macros, variables, and structures that are associated with the **select** () function. You must include this file in all socket applications.

#include <sys/socket.h>

This is an internet protocol family.

<sys/socket.h> makes available a type, **socklen_t**, which is an unsigned opaque integral type of length of at least 32 bits. To forestall portability problems, it is recommended that applications should not use values larger than $2^{32} - 1$.

The *<sys/socket.h>* header defines the unsigned integral type **sa_family_t**.

The *<sys/socket.h>* header defines the **sockaddr** structure that includes at least the following members:

```
sa_family_t    sa_family      address family
char           sa_data[]      socket address (variable-length data)
```

Defines socket prototypes, macros, variables, and the following structures:

- **sockaddr**
- **msghdr**
- **Linger**

You must include this file in all socket applications.

#include <netinet/in.h>

This is also an internet protocolfamily.

When header file <*netinet/in.h*> is included, the following types are defined through **typedef.**

in_port_t

An unsigned integral type of exactly 16 bits.

in_addr_t

An unsigned integral type of exactly 32 bits.

The <*netinet/in.h*> header defines the **in_addr** structure that includes at least the following member:

- in_addr_t s_addr

The <*netinet/in.h*> header defines the **sockaddr_in** structure that includes at least the following member:

- sa_family_t sin_family
- in_port_t sin_port
- struct in_addr sin_addr
- unsigned char sin_zero[8]

Defines prototypes, macros, variables and the **sockaddr_in** structure to use with Internet domain sockets.

#include <arpa/inet.h>

This header file includes definitions for internet operations.

The <**arpa/inet.h**> header makes available the type **in_port_t** and the type **in_addr_t** as defined in the description of <*netinet/in.h*>.

The <**arpa/inet.h**> header makes available the **in_addr** structure, as defined in the description of <*netinet/in.h*>.

Defines prototypes for those network library routines that convert Internet address and dotted-decimal notation. Example: **inet_makeaddr ()**.

#include <unistd.h>

This header file includes standard symbolic constants and types.

The *<unistd.h>* header defines miscellaneous symbolic constants and types, and declares miscellaneous functions.

Contains macros and structures that are defined by the integrated file system. Needed when the system uses the **read ()** and **write()** system functions.

#include <string.h>

string.h is the header in the C standard library for the C programming language which contains macro definitions, constants, and declarations of functions and types used not only for string handling but also various memory handling functions; the name is thus something of a misnomer.

Functions declared in string.h are extremely popular, since as a part of the C standard library, they are guaranteed to work on any platform which supports C. However, some security issues exist with these functions, such as buffer overflows, leading programmers to prefer safer, possibly less portable variants. Also, the string functions only work with ASCII or character sets that extend ASCII in a compatible manner such as ISO-8859-1; multibyte ASCII-compatible character sets such as UTF-8 will work with the caveat that string "length" is to be interpreted as the count of bytes in the string rather than the count of Unicode characters. Non-ASCII compatible string handling is generally achieved through wchar.h.

null	Macro expanding to the null pointer constant; that is, a constant representing a pointer value which is guaranteed **not** to be a valid address of an object in memory.

#include <netdb.h>

This header file includes definitions for network database operations.

- The *<netdb.h>* header may make available the type **in_port_t** and the type **in_addr_t** as defined in the description of *<netinet/in.h>*.
- The *<netdb.h>* header defines the **hostent** structure.
- The *<netdb.h>* header provides a declaration of *h_errno* as a modifiable l-value of type **int**.
- The *<netdb.h>* header defines the macro IPPORT_RESERVED with the value of the highest reserved Internet port number.

Contains data definitions for the network library routines.

Defines the following structures:

- **hostent** and **hostent_data**
- **netent** and **netent_data**
- **servent** and **servent_data**
- **protoent** and **protoent_data.**

#include <fcntl.h>

This header file includes file control options.

Defines prototypes, macros, variables, and structures for control-type functions, for example, **fcntl()**.

#include <sys/stat.h>

This header file includes data returned by the stat() function.

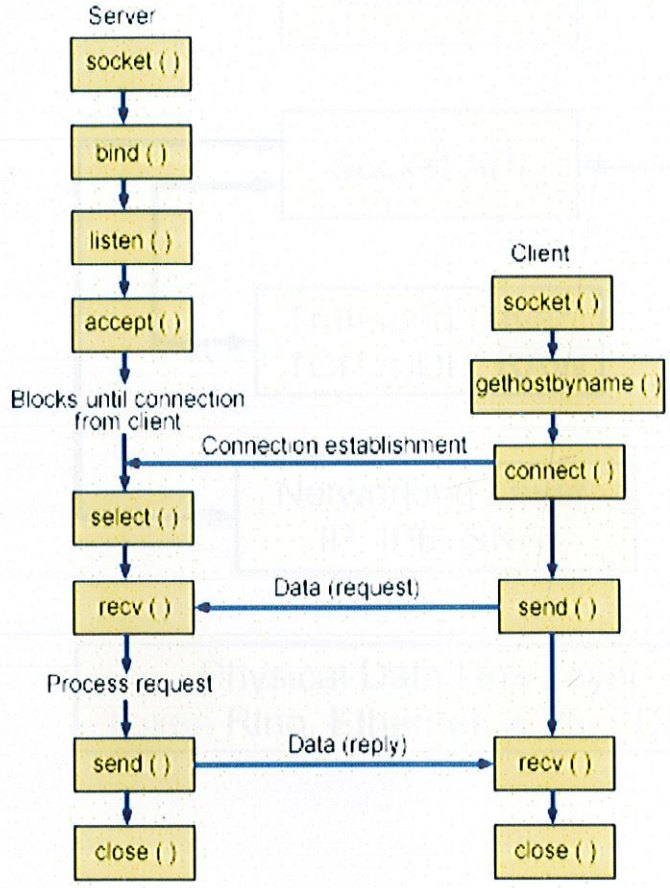
The *<sys/stat.h>* header shall define the structure of the data returned by the functions *fstat*(), *lstat*(), and *stat*().

Chapter 4: Elementary TCP sockets and system calls

4.1 socket ( )

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

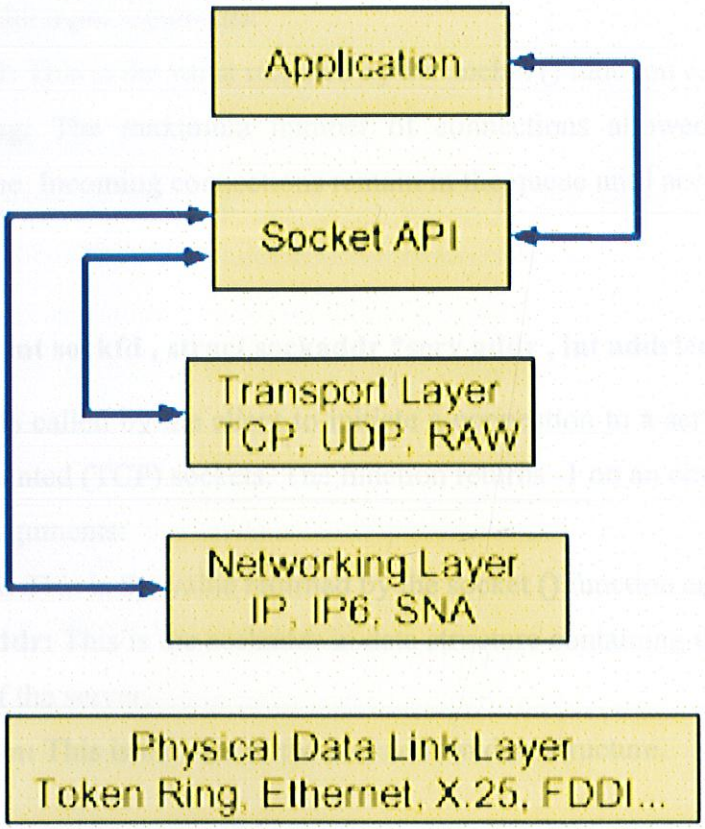
- This function creates the socket and returns a unique file descriptor to socket.the socket can either be a stream (T.C.P) or a datagram (U.D.P) socket depending on the input arguments.
- The following figure illustrates the example of client/server relationship of the socket APIs for connection-oriented protocol (TCP).



- **Domain:** set this to AF_INET.
- **Type** : Specifies what kind of socket.set this to SOCK_STREAM for TCP stream (telnet , ftp , https .etc) and SOCK_DGRAM for UDP datagrams.
- **Protocol:** takes one of the following values.
- **0:** automatically selects correct protocol based on type.
- **IPPROTO_TCP:** selects TCP protocol.
- **IPPROTO_UDP:** selects UDP protocol.

Note:

The socket APIs are located in the communications model between the application layer and the transport layer. The socket APIs are not a layer in the communication model. Socket APIs allow applications to interact with the transport or networking layers of the typical communications model. The arrows in the following figure show the position of a socket, and the communication layer that the socket provides.



4.2 bind ( )

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

This function associates socket sockfd with a port on the local machine. This function only needs to be called for incoming connections on the server. Returns -1 if there is an error. The following are the input arguments:

- **Sockfd:** This is the value returned by the socket () function call.

- **My addr:** This is the sockaddr in data structure containing the IP address and port number on the local machine.
- **Addrlen:** This is the size of the sockaddr in data structure.

3.3 Listen ( )

int listen (int sockfd , int backlog) ;

This function listens for incoming connections. It only needs to be called by the server for connection oriented (TCP) sockets. The function returns -1 on an error. The following are the input arguments:

- Sockfd: This is the value returned by the socket () function call.
- **Backlog:** The maximum number of connections allowed to wait in the incoming queue. Incoming connections remain in the queue until accept () is called.

4.4 Connect ( )

int connect (int sockfd , struct sockaddr *serv addr , int addrlen) ;

This function is called by the client to initiate a connection to a server. It is used for connection-oriented (TCP) sockets. The function returns -1 on an error. The following are the input arguments:

- **Sockfd:** This is the value returned by the socket () function call.
- **serv addr:** This is the sockaddr in data structure containing the IP address and port number of the server.
- **Addrlen:** This is the size of the sockaddr in data structure.

4.5 Accept ( )

int accept (int sockfd , void *addr , int *addrlen) ;

This function accepts a connection from the incoming queue associated with the socket sockfd. The function is used by the server for connection-oriented (TCP) sockets. The function returns a new socket descriptor which can be used to send and receive information on the connection. The function returns -1 on an error. The following are the input arguments:

- **Sockfd:** This is the socket _le descriptor for the socket that is listening for connections.
- **Addr:** This is a pointer to a local sockaddr in data structure that can be used to hold the IP address and port number of the incoming connecting client. This

data structure is di_erent from the one that contains the IP address and port number of the server.

- **Addrlen:** This is the size of the above sockaddr in data structure.

4.6 Read ( )

int read (int sockfd , const void *msg , int len) ;

Once a connection has been established (the client has connect () ed and the server has accept ()ed), this function is used to read information from client to server or from server to client. If one is using read ( ), the other should be using write (). This function either returns the number of bytes sent out or returns -1 if there is an error.

- **Read ( ):** function reads a specified number of bytes from a file handle and stores the data in a scalar variable.
- **Sockfd:** This is the socket _le descriptor for the socket being used to send the data. On the client side, sockfd is the same socket used when calling connect (). On the server side, sockfd is the sockets returned from accept ().
- **Msg:** This is a pointer to the data that is being sent.
- **Len:** This is the length of the data in bytes.

4.7 Write ( ) Function

int write (int sockfd , void *buf , int len) ;

This function writes data that is to be sent to a client. The function returns the number of bytes actually written, returns -1 if there is an error, or returns 0 if the other end has closed the connection.

- **Sockfd:** This is the socket _le descriptor for the socket from which the data is being write. On the client side, sockfd is the same socket used when calling connect (). On the server side, sockfd is the sockets returned from accept ().
- **Buf:** This is a pointer to the buffer into which the data will be writing.
- **Len:** This is the maximum length of the buffer in bytes.

4.8 Close ( )

void close(sockfd) ;

This function closes a socket.

4.9 Getpeername ( )

int getpeername (int sockfd , struct sockaddr *addr , int *addrlen) ;

- **Sockfd:** is the descriptor of the connected stream socket.
- **Addr**: is a pointer to a struct sockaddr (or a struct sockaddr_in) that will hold the information about the other side of the connection.
- **Addrlen**: is a pointer to an int, that should be initialized to sizeof(struct sockaddr).

The function returns -1 on error and sets *errno* accordingly.

Once you have their address, you can use inet_ntoa() or gethostbyaddr() to print or get more information.

4.10 Gethostname ( )

int gethostname (char *hostname , size_t size) ;

The arguments are simple: *hostname* is a pointer to an array of chars that will contain the hostname upon the function's return.

The function returns 0 on successful completion, and -1 on error, setting *errno* as usual.

- Below is a table which summarizes the various functions that need to be called at the client and server for TCP and UDP sockets.

	client	server
connection-oriented TCP	socket()	socket()
		bind()
		listen()
	connect()	
		accept()
	send()	recv()
	recv()	send()
connectionless UDP	socket()	socket()
		bind()
	sendto()	recvfrom()
	recvfrom()	sendto()

4.11 fopen ( ) Function

FILE *fopen (const char *filename, const char *mode) ;

- fopen opens a stream.
- fopen opens a file and associate a stream with it. The function returns a pointer that identifies the stream in subsequent operations.

Mode:

String	Description
r	Open for reading only
w	Create for writing If a file by that name already exists, it will be overwritten.
a	Append; open for writing at end of file, or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing)
w+	Create a new file for update (reading and writing). If a file by that name already exists, it will be overwritten.
a+	Open for append; open for update at the end of the file, or

To specify that a given file is being opened or created in text mode, append "t" to the string (rt, w+t, etc.).

To specify binary mode, append "b" to the string (wb, a+b, etc.).

If "t" or "b" is not given in the string, the mode is governed by _fmode.

- If _fmode is set to O_BINARY, files are opened in binary mode.
- If _fmode is set to O_TEXT, they are opened in text mode.
- These O_... constants are defined in FCNTL.H.

4.12 fclose ( ) Function

int fclose (FILE *stream) ;

- fclose closes the named stream.
- All buffers associated with the stream are flushed before closing.
- System-allocated buffers are freed upon closing.
- **Return Value :**

- On success, returns 0.
- On error, returns EOF.

4.13 fread ( ) Function

size_t fread (void *ptr , size_t size , size_t n , FILE *stream) ;

- Files are often processed not by lines but they are read/written in large chunks.
- fread reads a specified number of equal-sized data items from an input stream into a block.

Argument	What It Is/Does
ptr	Points to a block into which data is read
size	Length of each item read, in bytes
n	Number of items read
stream	Points to input stream

- The total number of bytes read is (n * size).
- **Return Value :**
- On success, fread returns the number of items (not bytes) actually read.
- On end-of-file or error, fread returns a short count (possibly 0).

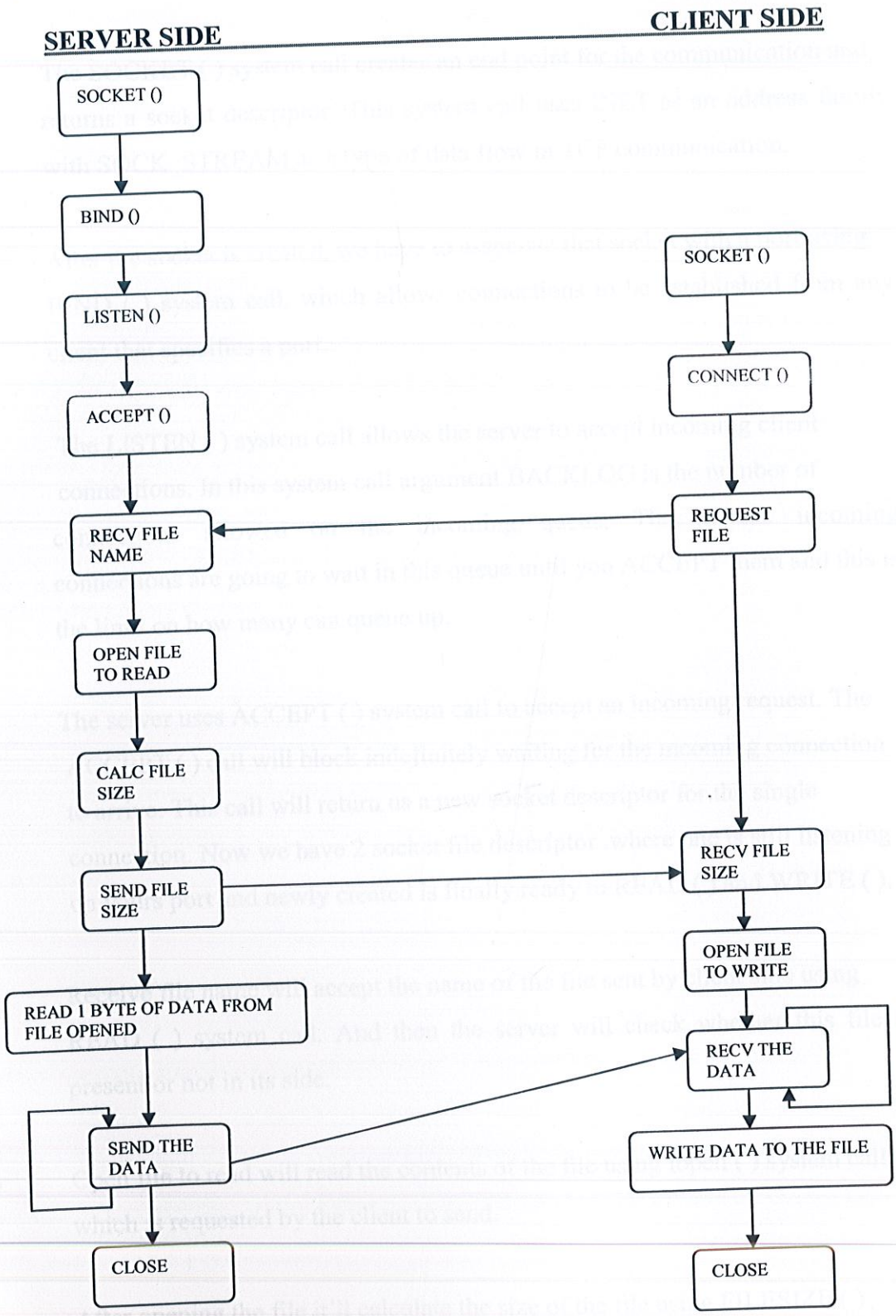
4.14 fwrite ( ) Function

size_t fwrite (const void *ptr , size_t size , size_t n , FILE*stream) ;

- fwrite appends a specified number of equal-sized data items to an output file.

Argument	What It Is/Does
ptr	Pointer to any object; the data written begins at ptr
size	Length of each item of data
n	Number of data items to be appended
stream	Specifies output file

- The total number of bytes written is (n * size).
- **Return Value :**
  - On success, returns the number of items (not bytes) actually written.
  - On error, returns a short count.



The previous flowchart illustrates the client/server relationship of the sockets API for a connection-oriented protocol.

Now we'll discuss each and every step of flowchart one by one as given in last figure.

Socket flow of events on SERVER side: Connection-oriented SERVER.

- The SOCKET ( ) system call creates an end point for the communication and returns a socket descriptor. This system call uses INET as an address family with SOCK_STREAM as a type of data flow in TCP communication.

- After the socket is created, we have to associate that socket with a port using BIND ( ) system call, which allows connections to be established from any client that specifies a port.

- The LISTEN ( ) system call allows the server to accept incoming client connections. In this system call argument BACKLOG is the number of connections allowed on the incoming queue. That means incoming connections are going to wait in this queue until you ACCEPT them and this is the limit on how many can queue up.

- The server uses ACCEPT ( ) system call to accept an incoming request. The ACCEPT ( ) call will block indefinitely waiting for the incoming connection to arrive. This call will return us a new socket descriptor for the single connection. Now we have 2 socket file descriptor .where one is still listening on yours port and newly created is finally ready to READ ( ) and WRITE ( ).

- Receive file name will accept the name of the file sent by client side using READ ( ) system call. And then the server will check whether this file is present or not in its side.

- Open file to read will read the contents of the file using fopen ( ) system call, which is requested by the client to send.

- After opening the file it'll calculate the size of the file using FILESIZE ( ) system call.

- After calculating the size of the file it'll send the size of the file to client using WRITE ( ) system call.

- Fread ( ) system call is used by the server to read the each byte of the file.

- After this the file is sent using SEND ( ) system call and a loop is also made on the server as well as on the client side so that we can transfer whole data of the file without any interruption.

- After sending all the data on server side we close socket file descriptor using CLOSE ( ) system call.

Socket flow of events on CLIENT side: Connection-oriented CLIENT.

- The SOCKET ( ) system call creates an end point for the communication and returns a socket descriptor. This system call uses INET as an address family with SOCK_STREAM as a type of data flow in TCP communication. Rest is same as server side.

- After we received the socket descriptor, we use CONNECT ( ) system call to establish a connection to the server.
- After establishing the connection with the server we send the name of the file with extension to the server using WRITE ( ) system call.
- After sending the request of the file, we receive the size of the file, sent from the server's end using READ ( ) system call.
- After this we use fopen ( ) system call to open the file on the client side so that the size of the file that we received from the server side.
- We use RECV ( ) system call to get 1 byte of the data send by the server.
- After this we use fwrite ( ) system call to write all the data on the file and we also employ a loop so that we get all the data from the server side.
- After receiving all the data on client side we close socket file descriptor using CLOSE ( ) system call.

CHAPTER 6: BIBLIOGRAPHY

[1] Advanced UNIX Network Programming by W. Richard Stevens.

[2] Http://www.wikipedia.com

[3] Beej's Guide to Network Programming.