



Jaypee University of Information Technology
Solan (H.P.)
LEARNING RESOURCE CENTER

Acc. Num **SP05111** Call Num:

General Guidelines:

- ◆ Library books should be used with great care.
- ◆ Tearing, folding, cutting of library books or making any marks on them is not permitted and shall lead to disciplinary action.
- ◆ Any defect noticed at the time of borrowing books must be brought to the library staff immediately. Otherwise the borrower may be required to replace the book by a new copy.
- ◆ The loss of LRC book(s) must be immediately brought to the notice of the Librarian in writing.

Learning Resource Centre-JUIT



SP05111

DISTRIBUTED FILE STORAGE AND SHARING SYSTEM

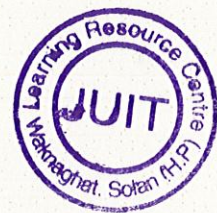
By

ABHISHEK RAI– 051216

AKANSHA MADAN – 051273

LAXMI LATA KESHOTE – 051288

PARUL GUPTA – 051419



MAY-2009

**Submitted in partial fulfillment of the Degree of
Bachelor of Technology**

**DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
AND INFORMATION TECHNOLOGY**

**JAYPEE UNIVERSITY OF INFORMATION
TECHNOLOGY-WAKNAGHAT**

CERTIFICATE

This is to certify that the work entitled, "Distributed File Storage System" submitted by Abhishek Rai, Akansha Madan, Laxmi Lata Keshote and Parul Gupta in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science and Engineering of Jaypee University of Information Technology has been carried out 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.

Nitin
28th May 2009
Dr. Nitin

Assistant Professor

Jaypee University of Information Technology

Waknaghat

ACKNOWLEDGEMENT

We wish to express our earnest gratitude to **Dr. Nitin**, our Project Guide, for providing us invaluable guidance and suggestions, which inspired us to submit this thesis report on time.

We would also like to thank all the staff members of Computer Science and Engineering Department of Jaypee University of Information Technology, Wakanaghat, especially **Mr. Satya Prakash Ghrera** (Head Of Department), and **Mr. Satish Chandra** (Project Co-coordinator), for providing us all the facilities required for completion of this thesis.

Last but not the least, we wish to thank all our classmates and friends for their timely suggestions and co-operation during the period of our project work and thesis



Abhishek Rai

051216



Akansha Madan

051273



Laxmi Lata Keshote

051288



Parul Gupta

051419

Btech(CSE and IT)

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY

CONTENTS

LIST OF FIGURES	1
ABSTRACT	5
CHAPTER 1	
INTRODUCTION	6
1.1 Description of Distributed File Storage System	6
1.2 Description of Our Project	7
1.3 Motivation	7
1.4 Problem Specification	8
1.5 Organization of Thesis	8
CHAPTER 2	
TECHNICAL COMPOSITION	10
2.1 Platform	10
2.2 Compiler	10
2.3 The Server-Client Model	12
2.4 Sockets	15
2.5 POSIX Threads	18
2.6 JAVA 'Swings'	21
2.7 Communication Protocol	24
2.8 Commands and Functions Used	26
CHAPTER 3	
FILE TRANSFER PROTOCOL	31
3.1 Software Architecture	31
3.2 Description	33
3.3 Server Description	33
3.4 Client Description	34
3.5 Tracker Description	35
3.6 Processing Details	36
3.7 Implementation Details	42
3.8 Installation Steps	61
CHAPTER 4	
TRACKER	62
4.1 Description	62

4.2 Server-Tracker Relation	62
4.3 Client-Tracker Relation	66
4.4 Fault Recovery Feature	70
 CHAPTER 5	
GRAPHICAL USER INTERFACE	71
5.1 Introduction to the GUI	71
5.2 Software Bridge	74
5.3 GUI Description	75
5.4 Installation Steps for the java file	98
 CHAPTER 6	
FAULT TOLERANCE	99
6.1 Introduction	99
6.2 Description	99
6.3 Working and Implementation Details	101
 CHAPTER 7	
CONCLUSION	111
 BIBLIOGRAPHY	112

LIST OF FIGURES

Figure 2.1	Client-Server Interaction	13
Figure 2.2	Connection Establishment Between a Client and Server	14
Figure 2.3	Layers of TCP/IP protocol	25
Figure 3.1	Three tier architecture of client, tracker and server	32
Figure 3.2	Modes of file transfer	34
Figure 3.3	Client connecting itself to tracker	35
Figure 3.4	Server registering itself to tracker	35
Figure 3.5	Client connected to tracker	35
Figure 3.6	Sharedfolder on Server Side	41
Figure 3.7	'Copiedfiles' folder after the chunks are received	41
Figure 3.8	Flow Of Application On 'REQ' Command	42
Figure 3.9	List of files on the server	44
Figure 3.10	Client sends 'GET' command and waits for file transmission	47
Figure 3.11	Server writing the desired chunks on socket one by one	47
Figure 3.12	Transmission of a file in byte mode and its md5 being checked	48
Figure 3.13	Server receives command for transfer of file in byte mode	48
Figure 3.14	A file being transferred in chunk mode	49
Figure 3.15	Each chunk being copied on the client side (in chunk mode)	49
Figure 3.16	Server writing each chunk on socket one by one	49
Figure 3.17	A file being transferred in large chunk (supr) mode	50
Figure 3.18	A large chunk is written on socket (bytes read in chunk form are concatenated 30 times to form a large chunk)	50
Figure 3,19	Chunks are created on the server side as the file is transmitted for the first time.	51
Figure 3.20	Large chunk is written on socket	51

Figure 3.21	MD5 of each chunk is sent to the client	51
Figure 3.22	Client Server interaction for 'GET' command	52
Figure 3.23	All the chunks are not available and cannot be merged	59
Figure 3.24	Client checks the infofiles for merging condition and calls the merger function	59
Figure 3.25	Chunks are opened one by one and merged on client side	60
Figure 3.26	MD5 of merged and original file are being cross checked for authentication	60
Figure 3.27	'Copiedfiles' folder after the chunks are received.	60
Figure 3.28	'Copiedfiles' folder after all the chunks are received and merged	61
Figure 4.1	Tracker awaiting connection from Server	63
Figure 4.2	Tracker on receipt of new Server address adds it to the list	64
Figure 4.3	Tracker on receipt of exiting Server address, resets it	64
Figure 4.4	The List of Servers as displayed	65
Figure 4.5	Server awaiting connection from Clients	66
Figure 4.6	Tracker on request for a new Server address	67
Figure 4.7	Client request a change in Server	67
Figure 4.8	Tracker on Request of Change Of Server	67
Figure 5.1	Screenshot of GUI showing the request tab when the connection with the client is not established	71
Figure 5.2	Screenshot of GUI showing the request tab after connection is established with the server when "Request Files On Server" button is activated	72
Figure 5.3	Screenshot of GUI showing the get tab	73
Figure 5.4	Screenshot GUI showing the setting tab	73
Figure 5.5	Software bridge depicting synchronization between Java and C.	74
Figure 5.6	Screenshot displaying when there is an error in connection	77

	of client with the server	
Figure 5.7	Screenshot displaying after the connection of client with the server is established	78
Figure 5.8	Screenshot displaying the list of files to the client, present on the server side. This is in response to request files on server	82
Figure 5.9	Snapshot showing the 'GET FILE' where file and its start and end chunks are selected	85
Figure 5.10	Snapshot that highlights the download progress bar that appears until download takes place	85
Figure 5.11	Snapshot of 'copiedfiles' folder, after the byte transmission of the file 'abc.jpg'.	86
Figure 5.12	Snapshot displaying message after download has been complete	86
Figure 5.13	Snapshot showing the message, when the md5 of copied file is validated	87
Figure 5.14	Snapshot of 'copiedfiles' folder when file has been merged successfully	90
Figure 5.15	Snapshot of GUI displaying message when all chunks are not available for merging	91
Figure 5.16	Snapshot of GUI awaiting user response when the file to be downloaded is already present	91
Figure 5.17	Snapshot of GUI highlighting the view File/Chunk Status button	93
Figure 5.18	Snapshot of the GUI showing the files that have been downloaded.	94
Figure 5.19	Snapshot of GUI showing incomplete files	94
Figure 5.20	Screenshot displaying when the client is disconnected from the server	97
Figure 6.1	Steps involved in fault tolerance mechanism	101
Figure 6.2	GUI Screenshot notifying the user that fault has occurred	104
Figure 6.3	Snapshot of server when fault occurs and the server is	104

restated automatically in case of chunk transmission

Figure 6.4	Snapshot of server side when fault occurs and the server is restarted automatically in case of large chunk transmission	105
Figure 6.5	Snapshot of the new client process started, reconnect to the server after error in connecting to server once	106
Figure 6.6	Snapshot of original command received from client to transmit 6 chunks of the file 'Client.c'	108
Figure 6.7	Snapshot of 'copiedfiles' folder showing only 4 chunks downloaded of file 'client.c' and the '.fault' file being made in the folder	108
Figure 6.8	Snapshot of contents of 'infoclient' file showing that only 1-4 chunks have been copied	108
Figure 6.9	Snapshot of the command received from client to transmit the remaining chunks from 5 to 6 of the file 'Client.c' after error occurred at chunk number 5.	109
Figure 6.10	GUI Snapshot showing the message to the user that reconnection has been established and remaining chunks are being downloaded	109
Figure 6.11	Snapshot showing transmission of remaining chunk of file 'client.c' from server side	109
Figure 6.12	Snapshot of 'copiedfiles' folder showing all requested chunks (from 1-6) downloaded in the folder	110

ABSTRACT

With the advancement in technology, building efficient networked system with hundreds of nodes is feasible. A vital component of these systems is the file transfer (or data transfer) that enables the nodes to communicate among themselves. File transfer being an important aspect of a network has many ways of completion, most common being the FTP protocol. The problem of connectivity or fault occurrence at the time of file transfer can hinder a peer's file transferring ability and can result in an incomplete or a faulty file. Therefore, the software developed caters to the ever growing need for distributed file storage and transfer along with the capability to handle faults, that may occur at the time of file transmission.

The goal of this project is to develop an application tool for Linux based platforms which would provide for a user friendly GUI than more commonly available command based software, thereby making it more preferable over them. The basic functionality involves partial storage of files on different peers and retrieval of original file when required. The storage and transfer of files has been implemented in a unique way which would support the fault tolerance feature thus ensuring interrupt free download. The application aims to utilize the best features of languages, Java and C where GNU C makes the core processing of data transfer faster while the user interacts with a Java (swing) graphics based interface.

CHAPTER 1

INTRODUCTION

Today, computers have entered every field of our day-to-day life. With computerization comes the need of networking. Its almost impossible to imagine a full use of computer without networking today, be it communication, e-mailing or anything else. All this requires the most important aspect of networking: File, Data and Information Sharing.

Data sharing has become so dominant amongst all uses of Networking, that the efficiency of a network is generally seen in terms of efficiency of data transfer. Thus, the efficiency of methods used for data sharing greatly effects that of networked environment. Amongst all methods, one of the most used and efficient one is Distributed File Storage System.

1.1 Description of Distributed File Storage System

Distributed File Storage System is the name given to a system where instead of storing files to be shared on a single system, files are stored at various places simultaneously. A very good example is the World Wide Web or Internet where the information is stored at different servers at different geographical locations and can be accessed from anywhere. Going more deeply, Web-Giants like Google keep redundant copies of their data at different locations and clients on request gets connected to the nearest server with least load.

In this type of communication architecture, the server is not pre-determined for the client but is decided by a third level system. It decides the server to which the client should be connected at any time, on the basis of criteria like geographical location, load on servers, files and services needed, and many more. This not just makes it easier for the client, but also creates a transparency between the server and the client and eradicates the need of the clients to know the whereabouts of the server. This adds another abstraction level to the complete communication system.

1.2 Description of our project

Our project has been implemented on the same line of the above stated, with the difference it has been designed to be used on a Local Area Network. The System has three levels or tiers of system, namely:

1. Server
2. Tracker
3. Client

This three tiers work in unison and synchronization to make our LAN-based Distributed file Storage System work efficiently.

- The Servers are responsible for distributing files to clients by providing file transfer services using File Transfer Protocol(FTP).
- The Clients are those programs that can avail the services of Servers by connecting to them on specific port number and address.
- Tracker is responsible for making this communication happen, by keeping a track of available servers and providing Server address to Clients as and when required.

The connections are established using Transmission Certificate Protocol(TCP) connection APIs and the communications occurs using the Socket communication APIs of TCP.

1.3 Motivation

The Motivation behind developing this application as project comes from an earlier project on "Information Sharing Over Multicast Router" submitted in partial fulfillment by Abhishek Bhatia, Kunal Krishna and Nakul Sharma.

Although our project is motivated from it, but the implementation techniques we have used for the development are very different in terms of

Changes like,

- Concept of Chunks
- Merging of chunks

And additional features like,

- Java based GUI
- Software Bridge
- Fault Tolerance

which and more we will come across, and discuss in details as we proceed further.

1.4 Problem Specifications

The designers face huge amount of hardships while designing file transfer systems for multiprocessor computers because its very hard to design a 100% efficient network i.e. all permutation passable in event of failures, have high reliability during failures, fault tolerance, possess high bandwidth, large throughput, nice and easy interface, high probability of acceptance and above all fulfilling these criteria, it should be cost effective too. Though many protocols have been designed, like the one discussed earlier, but no one stands up to the mark when all performance metrics applied simultaneously. So in order to design a protocol which is all permutation passable in the event of failure, have high reliability thereby maintaining file integrity and is easy to use through a Graphical User Interface using a three-tier architecture.

1.5 Organization Of Thesis

The Chapter 1: INTRODUCTION describes the basics of a Distributed File Storage System on a networked environment and introduces our motivation and approach to implementing it on a LAN-based environment.

The Chapter 2: TECHNICAL COMPOSITION describes the constructional features of the various technical functions that were used in the creation of the project.

The Chapter 3: FILE TRANSFER PROTOCOL describes the Server-Tracker-Client architecture developed and explains how the file transfer takes place between the server and the client.

The Chapter 4: TRACKER describes the implementation and in-depth analysis of a multithreaded Tracker server responsible for establishing communication between different components of the system.

The Chapter 5: GRAPHICAL USER INTERFACE describes implementation and creation of a Java based GUI for the otherwise command based system.

The Chapter 6: FAULT TOLERANCE describes the implantation and introduction of fault tolerance and recovery in the system.

The Chapter 7: CONCLUSION concludes the thesis followed by highlighting the scope for future work.

CHAPTER 2

TECHNICAL COMPOSITION

2.1 Platform

The application has been developed for LINUX platform, Open Suse Version 10.2 or higher. Linux is predominantly known for its use in servers, although it is installed on a wide variety of computer hardware, ranging from embedded devices and mobile phones to supercomputers.

The basic motive to develop the application was to build an efficient File Transfer Protocol for LINUX platform using GUI, thus making it user-friendly.

2.2 Compiler

The application has been built on 'GNU 'C' compiler.

2.2.1 Introduction

The GNU Compiler Collection (usually shortened to GCC) is a compiler system produced by the GNU Project supporting various programming language. As well as being the official compiler of the GNU system, GCC has been adopted as the standard compiler by most other modern Unix-like computer operating systems, including GNU/Linux and Mac OS X. GCC has been ported to a wide variety of processor architectures, and is widely deployed as a tool in commercial, proprietary and closed source software development.

The C language provides no built-in facilities for performing such common operations as input/output, memory management, string manipulation, and the like. Instead, these facilities are defined in a standard library, which you compile and link with your programs. The GNU C library, described in this document, defines all of the library functions that are specified by the ISO C standard, as well as additional features specific to POSIX and other derivatives of the Unix operating system, and extensions specific to the GNU system.

2.2.2 The 'C' Language

The application has been built with the 'C' language because it directly gets converted into Assembly Language and then machine code. This would help make the communication faster.

GCC supports the version of the C standard, although support for the most recent version is not yet complete.

The original ANSI C standard (X3.159-1989) was ratified in 1989 and published in 1990. This standard was ratified as an ISO standard (ISO/IEC 9899:1990). There were no technical differences between these publications, although the sections of the ANSI standard were renumbered and became clauses in the ISO standard. This standard, in both its forms, is commonly known as C89, or occasionally as C90, from the dates of ratification. The ANSI standard, but not the ISO standard, also came with a Rationale document. To select this standard in GCC, use one of the options `-ansi`, `-std=c89` or `-std=iso9899:1990`; to obtain all the diagnostics required by the standard, you should also specify `-pedantic-errors` if you want them to be errors rather than warnings).

GCC aims towards being usable as a conforming freestanding implementation, or as the compiler for a conforming hosted implementation. By default, it will act as the compiler for a hosted implementation, defining `_STDC_HOSTED_` as 1 and presuming that when the names of ISO C functions are used, they have the semantics defined in the standard. To make it act as a conforming freestanding implementation for a freestanding environment, use the option `-ffreestanding`; it will then define `_STDC_HOSTED_` to 0 and not make assumptions about the meanings of function names from the standard library, with exceptions noted below. To build an OS kernel, you may well still need to make your own arrangements for linking and startup.

GCC does not provide the library facilities required only of hosted implementations, nor yet did all the facilities require by C99 of freestanding implementations; to use the facilities of a hosted environment, you will need to find them elsewhere (for example, in the GNU C library).

Most of the compiler support routines used by GCC are present in `libgcc`, but there are a few exceptions. GCC requires the freestanding environment provide `memcpy`,

memmove, memset and memcmp. Finally, if `__builtin_trap` is used, and the target does not implement the trap pattern, then GCC will emit a call to abort.

2.3 The Server Client Model

Most interprocess communication uses the client server model. These terms refer to the two processes which will be communicating with each other. One of the two processes, the client, connects to the other process, the server, typically to make a request for information.

Notice that the client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established. Notice also that once a connection is established, both sides can send and receive information.

2.3.1 Interaction

A server is a network node that makes compute or data resources available. A client is a network node that utilizes server resources. For clients and servers to interact, they must be interacted.

A typical client server interaction goes like this:

1. The user runs client software to create a query.
2. The client connects to the server.
3. The client sends the query to the server.
4. The server analyzes the query.
5. The server computes the results of the query.
6. The server sends the results to the client.
7. The client presents the results to the user.
8. Repeat as necessary.

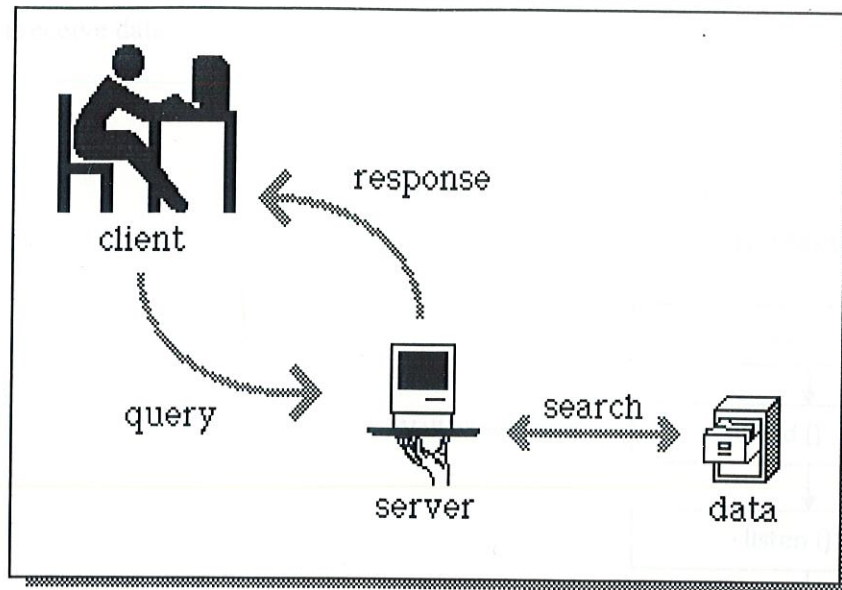


Figure 2.1: Client-Server Interaction

2.3.2 Connection

The system calls for establishing a connection are somewhat different for the client and server, but both involve the basic construct of socket. The two processes each establish their own socket.

The steps involved in establishing a socket on the client side are as follows:

- Create a socket with the `socket()` system call
- Connect the socket to the address of the server using the `connect()` system call
- Send and receive data. There are a number of ways to do this, but the simplest is to use the `read()` and `write()` system calls.

The steps involved in establishing a socket on the server side are as follows:

- Create a socket with the `socket()` system call
- Bind the socket to an address using the `bind()` system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the `listen()` system call
- Accept a connection with the `accept()` system call. This call typically blocks until a client connects with the server.

- Send and receive data.

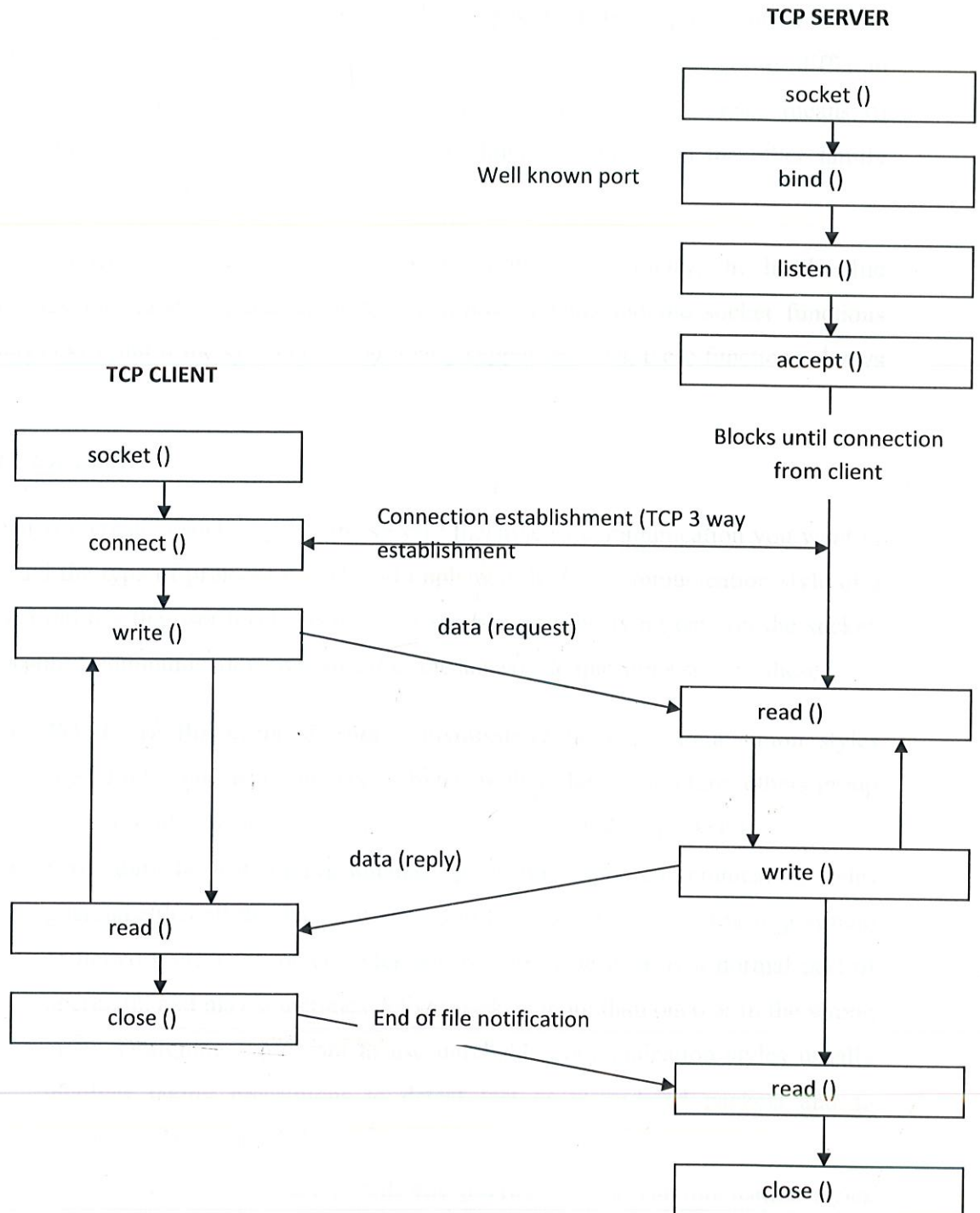


Figure 2.2 Connection Establishment Between a Client and Server

2.4 Sockets

2.4.1 Introduction

A socket is a generalized interprocess communication channel. Like a pipe, a socket is represented as a file descriptor. But, unlike pipes, sockets support communication between unrelated processes, and even between processes running on different machines that communicate over a network. Sockets are the primary means of communicating with other machines; telnet, rlogin, ftp, talk, and the other family programs use sockets.

Not all operating systems support sockets. In the GNU library, the header file 'sys/socket.h' exists regardless of the operating system, and the socket functions always exist, but if the system does not really support sockets, these functions always fail.

2.4.2 Creation

When you create a socket, you must specify the style of communication you want to use and the type of protocol that should implement it. The communication style of a socket defines the user-level semantics of sending and receiving data on the socket. Choosing a communication style specifies the answers to questions such as these:

- **What are the units of data transmission?** Some communication styles regard the data as a sequence of bytes, with no larger structure; others group the bytes into records (which are known in this context as packets).
- **Can data be lost during normal operation?** Some communication styles guarantee that all the data sent arrives in the order it was sent (barring system or network crashes); other styles occasionally lose data as a normal part of operation, and may sometimes deliver packets more than once or in the wrong order. Designing a program to use unreliable communication styles usually involves taking precautions to detect lost or misordered packets and to retransmit data as needed.
- **Is communication entirely with one partner?** Some communication styles are like a telephone call—you make a connection with one remote socket, and then exchange data freely. Other styles are like mailing letters—you specify a destination address for each message you send.

You must also choose a namespace for naming the socket. A socket name (“address”) is meaningful only in the context of a particular namespace. In fact, even the data type to use for a socket name may depend on the namespace. Namespaces are also called “domains”, but we avoid that word as it can be confused with other usage of the same term. Each namespace has a symbolic name that starting with ‘AF_’ designates the address format for that namespace.

Finally you must choose the protocol to carry out communication. The protocol determines what low-level mechanism is used to transmit and receive data. Each protocol is valid for a particular namespace and communication style; a namespace is sometimes called a protocol family because of this, which is why the namespace names start with ‘PF_’. The rules of a protocol apply to the data passing between two programs, perhaps on different computers; most of these rules are handled by the operating system, and you need not know about them. What you need to know about is as follows:

- In order to have communication between two sockets, they must specify the same protocol.
- Each protocol is meaningful with particular style/namespace combinations and cannot be used with inappropriate combinations. For example, the TCP protocol fits only the byte stream style of communication and the Internet namespace.
- For each combination of style and namespace, there is a default protocol which you can request by specifying 0 as the protocol number. And that’s what you should normally do—use the default.

Throughout the following description at various places variables/parameters have been used to denote sizes as required. And here the trouble starts. In the first implementation the type of these variables was simply `int`. This type was on all machines of this time 32 bits wide and so a de-factor standard required 32 bit variables. This is important since references to variables of this type are passed to the kernel.

But now the POSIX people came and unified the interface with their words “all size values are of type `size_t`”. But on 64 bit machines `size_t` is 64 bits wide and so variable references are not anymore possible.

A solution provides the Unix98 specification which finally introduces a type `socklen_t`. This type is used in all of the cases in previously changed to use `size_t`. The only requirement of this type is that it is an unsigned type of at least 32 bits. Therefore, implementation which requires references to 32 bit values be passed from the start of 64 bit values.

2.4.3 Communication Styles

The GNU library includes support for several different kinds of sockets, each with different characteristics. This section describes the supported socket types. The symbolic constants listed here are defined in '`sys/socket.h`'.

Macro: `int SOCK_STREAM`

The `SOCK_STREAM` style is like a pipe, it operates over a connection with a particular remote socket, and transmits data reliably as a stream of bytes.

The above style has been used in our application.

Macro: `int SOCK_DGRAM`

The `SOCK_DGRAM` style is used for sending individually addressed packets, unreliably. It is the diametrical opposite of `SOCK_STREAM`.

Each time you write data to a socket of this kind, that data becomes one packet. Since `SOCK_DGRAM` sockets do not have communications, you must specify the receipt address with each packet.

The only guarantee that the system makes about your requests to transmit data is that it will try its best to deliver each packet you sent. It may succeed with the sixth packet after failing with the fourth and fifth packet; the seventh packet may arrive before the sixth, and may arrive a second time after the sixth.

The typical use for `SOCK_DGRAM` is in situations where it is acceptable to simply resend a packet if no response is been in a reasonable amount of time.

Macro: `int SOCK_RAW`

This style provides access to low-level network protocols and interfaces. Ordinary user programs usually have no need to use this style.

2.5 POSIX Threads

2.5.1 Introduction

A thread is a semi-process that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc. All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel).

The advantage of using a thread group instead of a normal serial program is that several operations may be carried out in parallel, and thus events can be handled immediately as they arrive (for example, if we have one thread handling a user interface, and another thread handling database queries, we can execute a heavy query requested by the user, and still respond to user input while the query is executed).

The advantage of using a thread group over using a process group is that context switching between threads is much faster than context switching between processes (context switching means that the system switches from running one thread or process, to running another thread or process). Also, communications between two threads is usually faster and easier to implement than communications between two processes.

On the other hand, because threads in a group all use the same memory space, if one of them corrupts the contents of its memory, other threads might suffer as well. With processes, the operating system normally protects processes from one another, and thus if one corrupts its own memory space, other processes won't suffer. Another advantage of using processes is that they can run on different machines, while all the threads have to run on the same machine (at least normally).

2.5.2 Creating and destroying threads

When a multi-threaded program starts executing, it has one thread running, which executes the main () function of the program. This is already a full-fledged thread,

with its own thread ID. In order to create a new thread, the program should use the `pthread_create()` function. Here is how to use it:

```
id=pthread_create(&pt, NULL, (void*)serThrd, (void*)&nSock);
```

Notes about the above program:

1. Note that the main program is also a thread, so it executes the `do_loop()` function in parallel to the thread it creates.
2. `pthread_create()` takes 4 parameters. The first parameter is used by the function to supply the program with information about the thread. The second parameter is used to set some attributes for the new thread. In our application, this parameter is supplied as `NULL` pointer to tell `pthread_create()` to use the default values. The third parameter is the name of the function that the thread will start executing. The forth parameter is an argument to pass to this function. Note the cast to a `'void*'`. It is not required by ANSI-C syntax, but is mentioned here for clarification.
3. The delay loop inside the function is used only to demonstrate that the threads are executing in parallel.
4. The call to `pthread_exit()` causes the current thread to exit and free any thread-specific resources it is taking. There is no need to use this call at the end of the thread's top function, since when it returns, the thread would exit automatically anyway. This function is useful if we want to exit a thread in the middle of its execution.

In order to compile a multi-threaded program using `gcc`, we need to link it with the `pthread` library. Assuming you have this library already installed on your system, here is how to compile the first program :

```
gcc pthread_create.c -o pthread_cate -lpthread
```

2.5.3 MUTEX (Mutual Exclusion)

A basic mechanism supplied by the `pthread` library to solve this problem, is called a `mutex`. A `mutex` is a lock that gaurantees three things :

1. **Atomicity** – Locking a mutex is an atomic operation, meaning that the operating system (or threads library) assures you that if you locked a mutex, no other thread would succeed in locking this mutex at the same time.
2. **Singularity** – If a thread managed to lock a mutex, it is assured that no other thread will be able to lock the thread until the original thread releases the lock.
3. **Non-Busy Wait** – If a thread attempts to lock a thread that was locked by a second thread, the first thread will be suspended (and will not consume any CPU resources) until the lock is freed by the second thread. At this time, the first thread will wake up and continue execution, having the mutex locked by it.

From the above points we can see a mutex can be used to assure exclusive access to variables (or in general critical code sections). Here is some pseudo-code that updates the two variables we were talking about in the previous section, and can be used by the first thread :

lock mutex 'X1' .

set first variable to '0' .

set second variable to '0' .

unlock mutex 'X1' .

Meanwhile, the second thread will do something like this :

Lock mutex 'X1' .

set first variable to '1' .

set second variable to '1' .

Unlock mutex 'X1' .

Assuming both threads use the same mutex, we are assured that after they both ran through this code, either both variables are set to '0', or both are set to '1'. You would note this requires some work from the programmer – If a third thread was to access these variables via some code that does not use mutex, it still might mess up the variable's contents. Thus, it is important to enclose all the code that accesses these

variables in a small set of functions, and always use only these functions to access these variables.

2.5.4 Locking/Unlocking MUTEX:

In order to lock a mutex, we may use the function `pthread_mutex_lock()`. This function attempts to lock the mutex, or block the thread if the mutex is already locked by another thread. In this case, when the mutex is unlocked by the first process, the function will return with the mutex locked by our process. Here is how to lock a mutex (assuming it was initialized earlier) :

```
pthread_mutex_lock(&mutVar1);
```

After the thread did what it had to (change variables or data structures, handle file, or whatever it intended to do), it should free the mutex, using the `pthread_mutex_unlock()` function, like this

```
pthread_mutex_unlock(&mutVar1);
```

2.6 JAVA 'Swings'

2.6.1 JAVA Language

Java is a programming language originally developed by James Gosling at Sun Microsystems and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture.

2.6.1.2 Significant Language Features

- **Platform Independence** - Java compilers do not produce native object code for a particular platform but rather 'byte code' instructions for the Java Virtual Machine (JVM). Making Java code work on a particular platform is then simply a matter of writing a byte code interpreter to simulate a JVM. What this all means is that the same compiled byte code will run unmodified on any platform that supports Java.



- **Object Orientated** - Java is a pure object-oriented language. This means that everything in a Java program is an object and everything is descended from a root object class.
- **Rich Standard Library** - One of Java's most attractive features is its standard library. The Java environment includes hundreds of classes and methods in six major functional areas.
 - Language Support classes for advanced language features such as strings, arrays, threads, and exception handling.
 - Utility classes like a random number generator, date and time functions, and container classes.
 - Input/output classes to read and write data of many types to and from a variety of sources.
 - Networking classes to allow inter-computer communications over a local network or the Internet.
 - Abstract Window Toolkit for creating platform-independent GUI applications.
 - Applet is a class that lets you create Java programs that can be downloaded and run on a client browser.
- **Applet Interface** - in addition to being able to create stand-alone applications, Java developers can create programs that can download from a web page and run on a client browser.
- **Familiar C++-like Syntax** - One of the factors enabling the rapid adoption of Java is the similarity of the Java syntax to that of the popular C++ programming language.
- **Garbage Collection** - Java does not require programmers to explicitly free dynamically allocated memory. This makes Java programs easier to write and less prone to memory errors.

2.6.1.3 Areas Of Application

- World Wide Web Applets
- Cross-Platform Application Development
- Other Network Applications

2.6.2 Swings

Swing is a widget toolkit for Java. It is part of Sun Microsystems' Java Foundation Classes (JFC) — an API for providing a graphical user interface (GUI) for Java programs. Swing was developed to provide a more sophisticated set of GUI components than the earlier Abstract Window Toolkit. Swing provides a native look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform

Originally distributed as a separately downloadable library, Swing has been included as part of the Java Standard Edition since release 1.2. The Swing classes and components are contained in the `javax.swing` package hierarchy.

2.6.2.1 Architecture

Swing is a platform-independent, Model-View-Controller GUI framework for Java. It follows a single-threaded programming model.

2.6.2.2 Significant Features

- **Platform independence**

Swing is platform independent both in terms of its expression (Java) and its implementation (non-native universal rendering of widgets).

- **Extensibility**

Swing is a highly partitioned architecture, which allows for the "plugging" of various custom implementations of specified framework interfaces. Users can provide their own implementation(s) of these components to override the default implementations

- **Component-Oriented**

Swing is a component-based framework. The distinction between objects and components is a fairly subtle point: concisely, a component is a well-behaved object with a known/specified characteristic pattern of behaviour.

- **Customizable**

Given the programmatic rendering model of the Swing framework, fine control over the details of rendering of a component is possible in Swing. As a general pattern, the visual representation of a Swing component is a composition of a standard set of elements, such as a "border", "inset", decorations, etc.

- **Configurable**

Swing's heavy reliance on runtime mechanisms and indirect composition patterns allows it to respond at runtime to fundamental changes in its settings.

- **Look and feel**

Swing allows one to specialize the look and feel of widgets, by modifying the default (via runtime parameters), deriving from an existing one, by creating one from scratch, or, beginning with **J2SE 5.0**, by using the skinnable synth Look and Feel (see Synth Look and Feel). The look and feel can be changed at runtime, and early demonstrations of Swing frequently provided a way to do this.

2.7 Communication Protocols

The most common protocols used for communication in networks are :

- TCP/IP
- UDP

2.7.1 TCP/IP

The Internet Protocol Suite (commonly known as TCP/IP) is the set of communications protocols used for the Internet and other similar networks. The Transmission Control Protocol (TCP) is one of the core protocols of the Internet Protocol Suite. TCP was one of the two original components, with Internet

Protocol (IP), of the suite, so that the entire suite is commonly referred to as TCP/IP. Whereas IP handles lower-level transmissions from computer to computer as a message makes its way across the Internet, TCP operates at a higher level, concerned only with the two end systems, for example, a Web browser and a Web server. In particular, TCP provides reliable, ordered delivery of a stream of bytes from a program on one computer to another program on another computer. Besides the Web, other common applications of TCP include e-mail and file transfer.

This protocol has been used for our application.

The different layers in TCP/IP are as follows:

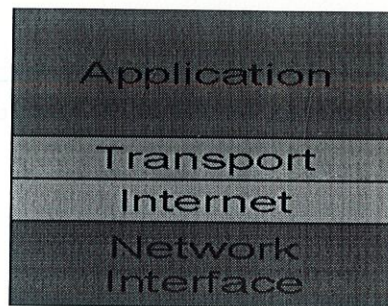


Figure 2.3 : Layers of TCP/IP protocol

2.7.2 UDP

The User Datagram Protocol (UDP) is one of the core members of the Internet Protocol Suite, the set of network protocols used for the Internet.

With UDP, computer applications can send messages, in this case referred to as datagrams, to other hosts on an Internet Protocol (IP) network without requiring prior communications to set up special transmission channels or data paths.

The service provided by UDP is an unreliable service that provides no guarantees for delivery and no protection from duplication. UDP provides a minimal, unreliable, best-effort, message-passing transport to applications and upper-layer protocols. Compared to other transport protocols, UDP and its UDP-Lite variant are unique in that they do not establish end-to-end connections between communicating end systems. UDP communication consequently does not incur connection establishment and teardown overheads and there is minimal associated end system state. Because of

these characteristics, UDP can offer a very efficient communication transport to some applications, but has no inherent congestion control or reliability.

2.8 Commands and Functions Used

2.8.1 'C' Based Functions

Socket Function

It specifies the type of communication protocol desired (TCP, UDP, Unix domain stream protocol, etc).

```
if( (SerSock = socket(AF_INET, SOCK_STREAM, 0) ) < 0)
{
    printf("!!!Socket Creation Error!!!\n");
    return -1;
}
```

Connect Function

It is used by a TCP client to establish a connection with TCP server. It returns 0 when executes successfully and -1 on error.

```
if(connect(s, (struct sockaddr *)
&hostInfo1, sizeof(hostInfo1)) < 0)
{
    printf("\nError connecting...");
}
```

Bind Function

This function assigns a local protocol address to a socket. A process can bind a specific IP address to its socket through this. The IP address must belong to an interface on the host.

```
{
```

```

    bzero((char *) &Ser, sizeof(Ser)); //Socket Address
Structure
    being initialised 0

    Ser.sin_family = AF_INET; //The Socket Address
Structure
    Ser.sin_port = htons(prt); //The Socket Address
being
assigned values
    Ser.sin_addr.s_addr =
    htons(INADDR_ANY);

    printf("\n!!!Binding Socket with the Port!!!");

    bind(SerSock, (struct sockaddr *) &Ser, sizeof(Ser));
}

```

Listen Function

It is called only by a TCP server and it performs the following actions:

- When a socket is created by the socket function, it is assumed to be an active socket, that is, a client socket that will issue a connect. The listen function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.
- The function specifies the maximum number of connections the kernel should queue for this socket.

```

if( listen(SerSock, MAXCONNECTIONS) < 0 )
{
    printf("\n!!!Error in Listening!!!");

    return -1;
}

```

MAXCONNECTIONS specify the maximum connections a server can have at a particular time.

Accept Function

It is called by a TCP server to return the next completed connection from the front of the completed queue. If the completed queue is empty, the queue is put to sleep.

```
if ((ClSock = accept(SerSock, (struct sockaddr *) &Clnt, (socklen_t *) &ClntLen)) < 0)
{
    printf("\n!!!Error creating client socket!!!");
    return -1;
}
```

Gethostbyname Function

This function looks up a hostname. If successful it returns a pointer to a hostent structure that contains all the Ipv4 addresses for the host.

```
if((hp1=gethostbyname(tracker))==NULL) //Tracker Address
{
    printf("\nError resolving host name...");
    return -1;
}
```

Pthread_create Function

The pthread_create() function is used to create a new thread, with attributes specified by attr, within a process. If attr is NULL, the default attributes are used. If the attributes specified by attr are modified later, the thread's attributes are not affected. Upon successful completion, pthread_create() stores the ID of the created thread in the location referenced by thread.

```
id=pthread_create(&pt,NULL,(void*)serThrd,(void*)&nSock);
```


MD5

It is an algorithm used in software to provide some assurance that a transferred file has arrived intact. File servers often provide a pre-computed MD5 checksum for the files, so that a user can compare the checksum of the downloaded file to it.

Netstat -inet

The netstat command is used to show the network status. The inet parameter used along with netstat tells the ip address of the machine.

System Commands used in the code

- **du -b** : This is a system command which gives the size of file in bytes.

Syntax : du -b filename

- **openssl md5** : This command gives the md5 of the specified file.

Syntax : openssl md5 -out md5temp.dat

The md5 is taken in a file named md5temp.dat

- **scandir** : This command scans the specified directory and stores the information of all the files present in the directory in its structure.

Syntax : scandir(loc, &namelist, 0, alphasort)

Loc : The path of directory.

Namelist : Information about the files is collected in this array.

0 : To select all directory entries.

Alphasort : It is a comparison function.

2.7.2 'JAVA' Based Functions

- **Runtime.getRuntime().exec("./checkcli");**
The function runs a command on linux through java.
- **FileOutputStream outfile1 = null;**
outfile1=new FileOutputStream(path);

The above lines of code declare and create an output stream file.

if(f1.exists())

The function checks for a file at a particular path 'f1'.

- **br.readLine()**

The function reads a string till the end of line.

- **s.substring(0, s.indexOf('\t'));**

The substring function selects a part of string whose starting and ending index are mentioned in parantheses.

The indexOf function returns the index of a particular character.

- **String s[]=f1.list();**

The list() function returns the list of files present at the desired path.

- **timer = new Timer(2000,actionListener);**
timer.start();

The above lines of code define and start a scheduler(timer). For Eg: The code sets the timer for 2 seconds.

- **t=new Thread(new thread1());**
t.start();

The code declares and starts a thread.

CHAPTER 3

FILE TRANSFER PROTOCOL

File Transfer Protocol (FTP) is a network protocol used to exchange and manipulate files over a TCP computer network, such as the Internet. FTP is built on a client-server architecture and utilizes separate control and data connections between the client and server applications. FTP is commonly used to transfer Web page files from their creator to the server for everyone on the Internet. It's also commonly used to download programs and other files to your computer from other servers.

As a user, you can use FTP with a simple command line interface (for example, from the Windows MS-DOS Prompt window) or with a commercial program that offers a graphical user interface. Your Web browser can also make FTP requests to download programs you select from a Web page. Using FTP, you can also update (delete, rename, move and copy) files at a server. You need to logon to an FTP server. However, publicly available files are available using anonymous FTP.

3.1 Software Architecture

The backbone of the core "Distributed File Storage System" consists of 3 kinds of nodes: the servers, the tracker and the clients, thus resulting in three-tier architecture.

SERVERS:

These are the hosts which are responsible for providing the services to the client nodes i.e. these provide the file sharing services over network.

CLIENTS:

These are the nodes which make use of the file sharing over the network and get connected to the server, with the help of tracker.

TRACKER:

3.2 Description:

Most inter-process communication use client server model. A separate client server application is written in this module. The client connects to the other process, the server, typically to make a request for information. The server reads parameters from a separate configuration file. Client program connects to the server via a tracker, on a specified IP address and port number. The client contacts the tracker which in return sends it the IP address of server. Once a connection is established, both sides can send and receive information. A TCP/IP based socket programming has been used to establish connection and thereby appropriate communication and data transfer between the two processes, i.e. the client and the server. The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. A socket is one end of an inter-process communication channel. The two processes each establish their own socket.

Client program is now able to request directory listing and also download any file kept on server either partially or completely. Server program can to send over to the client the complete directory list of the shared folder along with the related details such as file size in bytes and files' MD5 checksum.

3.3 Server Description:

The server creates chunks of files and sends those chunks to the clients. The server creates the number of chunks from the original file and sends them when the file is requested.

The server is able to read parameters from a separate configuration file. The purpose of using a configuration file is to help keep our code clean and versatile. A typical configuration file will contain details like 'shared directory path', 'port number' and 'tracker IP address'. The server when started begins listening on the specified port number for incoming client connections; once a connection is established, further behavior of the server depends on client requests.

The server is able to support multiple client connections. It implements client handling as a separate thread. Once the server receives a new connection, the new connection details are passed on to the client handler thread which handles all the client requests and also the server responses to that client.

3.4 Client Description:

It is able to request the list of files present in the shared folder (explained later in the chapter) on the server. The mode of transfer of file from the server depends on the user request and the size of the file. The server provides three modes of transfer: Byte, Chunk and Large Chunk mode.

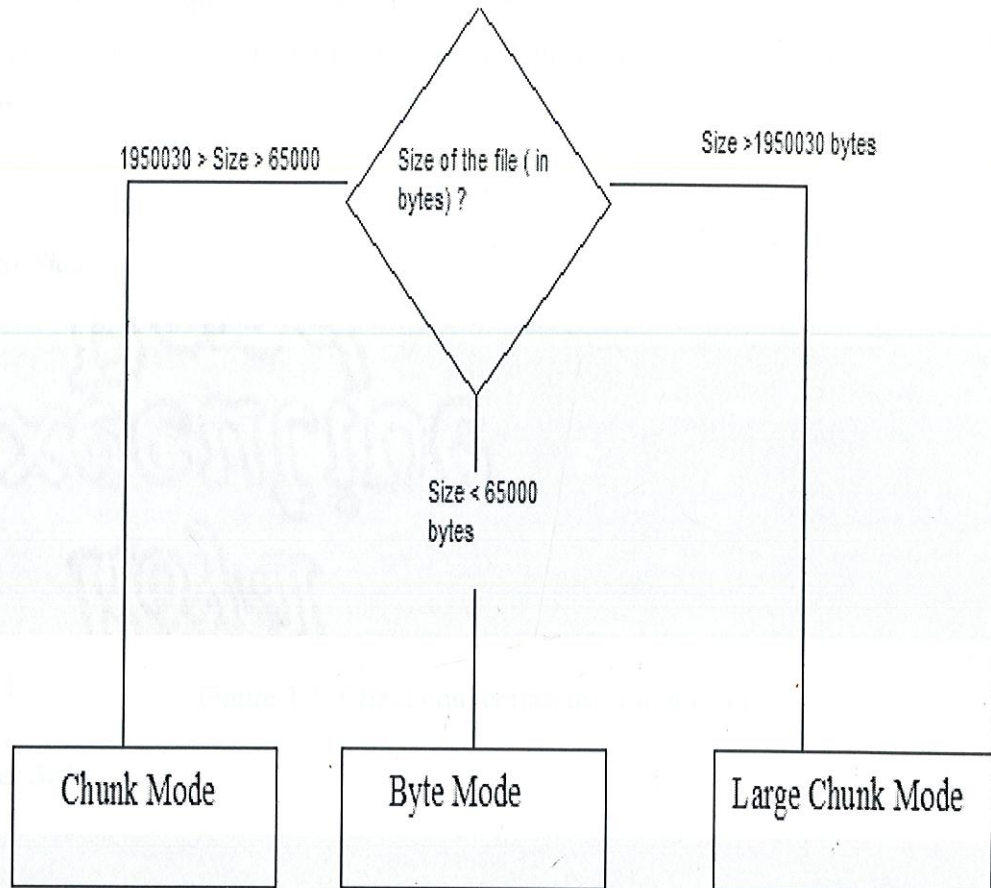


Figure 3.2: Modes of file transfer

The files are transferred according to the modes of transmission. If the file size is less than 65000 bytes, mode of transmission is Byte. If the file size is greater than 65000 bytes but less than 1950030, file mode transmission is Chunk and if file is greater than 1950030 bytes, the mode of transmission is Large Chunk. The description of these modes has been explained in the later part of chapter.

3.5 Tracker Description:

The client has address of the tracker and initially connects with it. The tracker then sends the IP address of a server to the client. The client thereafter contacts the server and establishes connection with it. Therefore, it acts as a link between the communicating server and client. In case of an error that occurs, like server is not reachable or the request by server is not responded, the tracker redirects the client to another server address. This functionality has been discussed in the later part of the thesis.

Client Side

```
CONNECTING TO TRACKER...
!!!Tracker Host Name Resolved!!!
!!!Connection To Tracker Socket 14 Successful!!!

SERVER Address Received : 172.16.9.136Initial Socket : 3SERVER>
!!!Host Resolved Successfully!!!
!!!Connection to Server Socket 3 Successful!!!
connection status connectedG
```

Figure 3.3: Client connecting itself to tracker

Server Side

```
CONNECTING TO TRACKER...
!!!Tracker Host Name Resolved!!!
!!!Connection To Tracker Socket 3 Successful!!!
!!!Server Address Sent!!! Listening at PORT : 4005
Shared Directory Path : /home/Parul/sharedfolder
Socket Created Successfully as 3
!!!Binding Socket with the Port!!!
!!!Listening for Client!!!
```

Figure 3.4: Server registering itself to tracker

```
!!!Listening Successful!!!
shdir value msg sent /home/Parul/sharedfolder 4
```

Figure 3.5: Client connected to server

3.6 Processing Details

These include all the intermediate processing required for the working of the application.

3.6.1 Chunk Structure and Name

Structure of chunk

Chunks are small files (data portions) that are created from a shared file at the server side. These chunks are stored as hidden files on the server and the client side (transmitted chunks).

The chunk structure used in the code is shown below:

```
struct chunk
{
    char header[50];
    char content[CHUNK_SIZE+1];
    char md5[80];
}
```

A chunk has three parts:

Header: SIZE (50 bytes).Identifies the chunk and stores the basic information, i.e. size of the original file md5 value.

File Content: SIZE (5000 bytes | 65001 bytes). Optimal size depends on the size of files transmitted and the network environment chosen. Empirically, the suitable value found for the project is 5000 B for a chunk and 70000 bytes for a large chunk.

MD5: SIZE (80 bytes).This is MD5 checksum/hash of the file content part of the chunk, can be used to verify data integrity of file chunks transmitted.

Structure of a super chunk

```
struct superchunk  
{  
  
    char header[50];  
  
    char content[SUPERCHUNK_SIZE+1];  
  
    char md5[80];  
  
}
```

The super chunk content is 65001 bytes. The header and md5 are same as that of chunk structure.

The file is divided into chunks when the request for transmission of the file is received on the server side and an infofile keeps track of the chunks created on the server side and thus file is not repeatedly divided into chunks if already done so.

The file is copied to the client, chunk by chunk and again an infofile keeps track of the chunks copied to the client. This helps to merge the various chunks into the original file, once all the chunks are available.

The file can be transmitted in the form of chunks or byte stream. This decision would be taken by the application itself depending on the file size and thus there would be a level of abstraction between the user and the application. While transmitting small files, byte stream transfer would be useful as there will no need to divide the file into chunks. The complete file will be transmitted in the form of a single chunk.

Example: Assume we have file name lion.avi of size 11308 kb.

There would be six chunks created on server side of size 1904 kilobytes each, having the format:

Chunk1

<Header:ZZ || size of the file|| md5 of original file> <content (1904 kb)> <md5 of chunk content>

Chunk2

<Header:ZZ || size of the file|| md5 of original file> <content (1904 kb)> <md5 of chunk content>

..

..

..

Chunk6

<Header:ZZ || size of the file|| md5 of original file> <content (1904 kb)> <md5 of chunk content>

NOTE: the chunk structure defined both on the server and client side should be the same.

Chunk Name

Format of chunk name:

.<filename>_CH<chunk number>_<start byte>- <end byte>

. indicating it is a hidden file.

<filename> is the name of the original file.

"CH" indicates it is a chunk.

<chunk number> is the chunk sequence number.

<start byte> indicates byte sequence number of the starting byte written in this chunk content

<end byte> indicates byte sequence number of the end byte written in this chunk

Example: Assume we have file name *Chrome.exe* of size 486128 bytes. Then the chunk name will be as follows:

Chunk1

.pic_CH0_0-64999

Chunk2

.pic_CH2_65000-129999

...

...

...

Chunk8

.pic_CH7_455000-486128

3.6.2 Info files

These are temporary files that help in background hidden processing in the software. These files are created both on the server and client side for each file that is transmitted. They contain useful information about the chunks that are created or copied on the corresponding sides.

.On Server Side

The hidden file infofile at the server side is formed in the shared folder. It contains information about the number of chunks of the file. It is helpful in the case when the file has already been divided into chunks by the server and there is a request for that file. The server would then just check the infofile to see the availability of chunks.

Name format: .info||<filename>

On Client Side

The infofile at the client side is formed in the copiedfiles folder. It stores information about the total number of chunks. The infofile stores the 0s for all the chunks. As soon as the client receives a chunk, the 0 at the position corresponding to the chunk number is changed to 1. With the help of this the client would be able to know the number of

chunks received. If all the chunks are available, the client would call the merger function.

Name format: .info||<filename>

3.6.3 Configuration Files

These make the code portable, clean and easy to modify. They help configure the corresponding client and server settings. The client and the server each have their separate configuration files, which are explained as under :

Server Side

- **Config_ini** : The file is required by the server. It contains the following information

Path : Where the server files are stored.

Port Number : It represents a channel for network communications. Port numbers allow different applications on the same computer to utilize network resources without interfering with each other.

Client Side

- **Config_client.ini** : The file is used by the client. It contains the following information :

Path : It is the path of the default folder where the client saves the copied files folder.

Port Number : It allows different applications on the same computer to utilize network resources.

IP Address : The IP address of tracker. The client connects itself to the tracker with this address.

3.6.4 Shared folder :

It is a folder in the root directory of server where all the files that are to be shared with the clients are stored. The files are divided in the form of chunks and then transferred. These chunks are stored as hidden files.

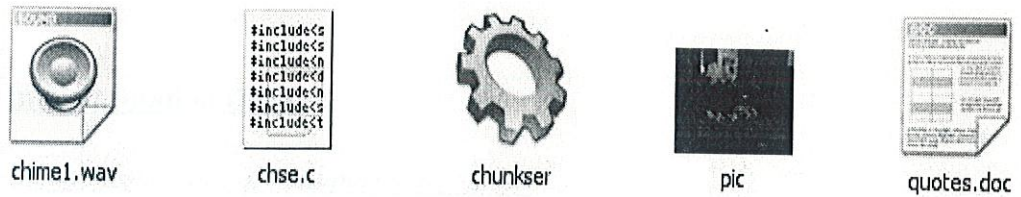


Fig 3.6: Sharedfolder on Server Side

3.6.5 Copiedfiles :

It is a folder in the root directory of client where all the files or chunks to be transferred are stored along with the infofile. The chunks that are transferred are stored as hidden files and are not visible to the user. The chunks that are transmitted are saved as hidden files. Once all the chunks are received and file merged, the chunks and the infofile of the respective file are deleted.

Copiedfiles Folder

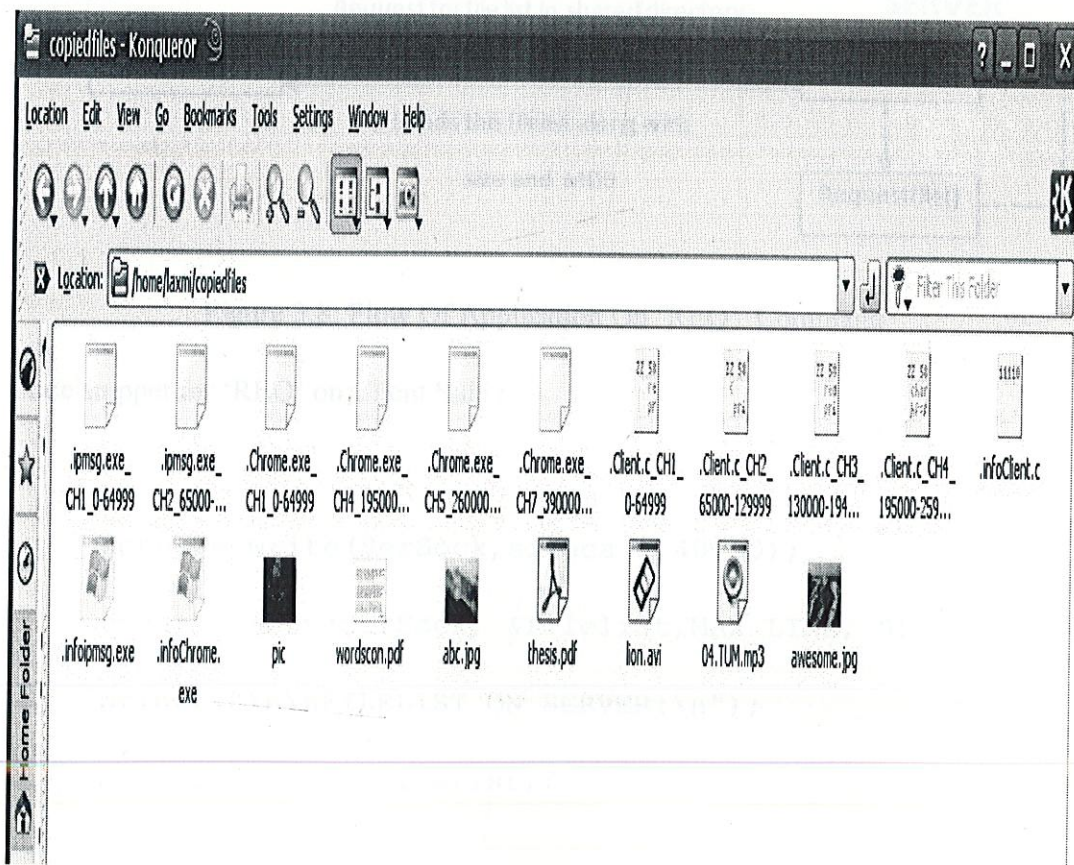


Figure 3.7: 'Copiedfiles' folder after the chunks are received.

3.7 Implementation Details

3.7.1 Commands that are supported by server

REQ - This command sent by the connected client instructs the server to send over to the requesting client the list of files in the shared folder at the server. The format of the incoming message from the connected client will be <REQ>.

In reply to the REQ command, the server reply message structure is as follows:

```
<1 filename>      <file1size>   <file1MD5>  <number of chunks>\n
<2 file2name>      <file2size>   <file2MD5>  <number of chunks>\n
```

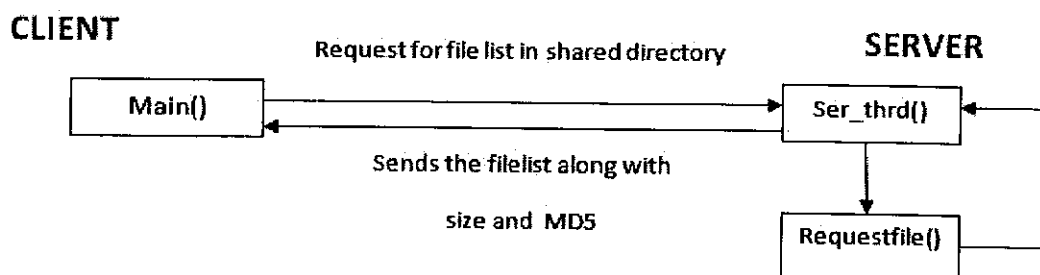


Figure 3.8: Flow Of Application On 'REQ' Command

Code snippet for 'REQ' on Client Side :

```
{
    count = write(SerSock, sizecalc, 40, 0);

    count = recv(SerSock, &filelist, MAX_LINE, 0);

    printf("\n\nFILELIST ON SERVER:\n");

    printf("%s\n", filelist);
}
```

The write command writes 'REQ' command on the socket and requests for file list on server.

The file list is read in filelist array and received by the client with the help of command 'recv'.

Code snippet for 'REQ' on Server side.

```
if(!strcmp(prc, "REQ"))
{
    requestFile(filelist, shdir);

    printf("\n\nFILELIST:\n%s", filelist);

    dataCount = send(ClSock, &filelist, LENGTH, 0);
}

void requestFile(char *filelist, char shdir[])
{
    n = scandir(loc, &namelist, 0, alphasort);
    ..

    for(i=0; i<n; i++)
    {
        strcpy(md5com, "openssl md5 -out md5temp.dat
        "); //md5 func written in file

        strcat(md5com, namelist[i]->d_name);

        ..

        system(md5com); //command being given to system

        ..

        strcpy(sizecom, "du -b "); //Command for file
size
```



```

        strcat(sizecom, " > sizetemp.dat"); //Command
        written in file

        ..

        system(sizecom); //Command executed by system

        ..

        ntc=(atol(sizecalc)/CHUNK_SIZE)+1; //No of
        chunks of file

    }

}

```

The command sent by client is 'REQ', a 'requestFile' function is called. The scandir function returns the information of all the files in the 'shared folder'. The md5 of each file is calculated by the function 'openssl md5' which is written in file 'md5temp.dat' and given as a system command. The file size is calculated with 'du -b', and given as a system command. ntc calculates the chunks of s file. All this information is stored in filelist and written on socket.

Server Side

```

CONNECTING TO TRACKER...
!!!Tracker Host Name Resolved!!!
!!!Connection to Tracker Server : Successful!!!
!!!Server address : 192.168.1.100!!! Listening at PORT : 4005
Shared Folder Path : sharedFolder/sharedFolder
Server Name : Server-110-05
!!!Binding socket with the Port!!!
!!!Listening for Clients!!!
!!!Listening Port : 4005!!!
Client Name : 192.168.1.100, Name : sharedFolder
!!!Server is created successfully!!!
bytes read 50, value Req
bytes read 10, value Req
Command received: Req

File List:
File Name      MD5      Size      File Path      File Type
Chrome.exe     MD5: 3e7de009d993c4f11f1c0073004e9467  51274161361  C:\TEMP\3-  00000000
Client.c       MD5: C971141b2c7b15a353c9d0aa35d0b0c  51274161361  C:\TEMP\3-  00000000
Song111.mp3    MD5: 34b7531e9903170b0b0b413f7c0e003  51274161361  C:\TEMP\3-  00000000

```

Figure 3.9: List of files on the server

Client Side

The same is shown on the client side.

GET – This command sent by the connected client requests the server a file (in the shared directory) or particular chunks of the file. The format is as follows :

<GET fileName startChunk endChunk>

If only one chunk is requested, the startChunk and endChunk will be the same. Also, when the file is completely downloaded by the client, it must verify the file correctness by recalculating the MD5 checksum and comparing it with the value received via the REQ reply from the server or from the last message sent by the server, which is MD5.

Code Snippet for 'GET' on Client side

```
{  
  
    count = write(SerSock, &sizecalc, 100, 0);  
  
    client_transfer(SerSock, shdir);  
  
}
```

The Client writes the 'GET' command on the server and calls the client_transfer() function which reads the chunks written on socket by the server.

Code Snippet for 'GET' on Server side

```
{  
  
    comcheck( );  
  
    if(id=2)  
  
        byte_transfer();  
  
    if(id=1 || id=3)  
  
        chunk_transfer();  
  
}
```

The comcheck() function checks for the mode of transmission (explained in later part) and accordingly calls the byte_transfer() or chunk_transfer() function.

```
chunk_transfer()
{
    ..

    write(sockfd,nam,strlen(nam)+1); //filename written
                                     on socket

    ..

    gcvt(m,10,send);

    write(sockfd,send,strlen(send)+1); //sending      total
                                     num      of
                                     chunks to be
                                     received

    ..

    for(i=startc;i<=endc;i++)
    {

        write(sockfd,chname,strlen(chname)+1);

                                     //writi
                                     ng chunk
                                     name on
                                     socket

        ..

        p=write(sockfd,&sendchunk,sizeof(struct
        chunk)); //chunk written on socket

    }

}
```


In `chunk_transfer()` function, if the mode of transmission is 'chk', first the file name, the no. of chunks then the chunk name is written on socket and sent to the client. Then the entire chunk is sent to the client. If the transmission mode is 'supr' (large chunk), first the filename, number of chunks and chunk name are written on socket. The chunk is not sent altogether but first the header, and then content followed by the md5 of chunk are sent to the client.

Command : 'GET Chrome.exe 5 8'

Client Side

```

SYS : />
Waiting for file transfer from server

THE FILE TRANSFER MODE:   chk
server sent filename: Chrome.exe
  
```

Figure 3.10: Client sends 'GET' command and waits for file transmission

Server Side

```

client sent:ch
WRITING CHUNK ON SOCKET

No. of bytes being written on socket : 65131
opening file to read chunk /home/Parul/sharedfolder/.Chrome.exe_CH5_260000-324999
sending filename to server Chrome.exe_CH5_260000-324999
recv=
No. of bytes being read from socket : 3
  
```

Figure 3.11: Server writing the desired chunks on socket one by one

3.7.2 Modes Of Transmission

- **Byte mode** – The file size of less than 65000 bytes is not divided in chunks on the server side. The complete file is written on socket by the server at one time and the client reads it and copies the file. The server also sends the md5

checksum to client side which it cross checks with the calculated md5 of the downloaded file on the client side, validating the integrity of the downloaded file. While giving the command for file transfer, the value of startChunk and endChunk has to be '1'.

Command : GET abc.jpg 1 1

Client Side

```
THE FILE TRANSFER MODE: byte
server sent filename: abc.jpg
size read 47812
file size in long 47812
opening file path:/home/laxmi/copiedfiles/abc.jpgfile opened
bytes read from the socket:47812
md5 on server side: 85b437530f7a373af95f66eb13587f67
md5 on client side: 85b437530f7a373af95f66eb13587f67
```

Figure 3.12: Transmission of a file in byte mode and its md5 being checked

Server Side

```
Command Received: GET abc.jpg 1 1
calling filetransfer /home/Parul/sharedfolder
n = 3
the name extracted= abc.jpg
Found abc.jpg (null)
startchunk= 1
endchunk= 1
The total chunks of file are 1
The requested file is less than 65000 B, so BYTE mode transfer of complete File
```

Figure 3.13: Server receives command for transfer of file in byte mode

- **Chunk mode** – The file size of greater than 65000 bytes but less than 1950030 bytes(kb) is divided in chunks on server side. The desired numbers of chunks are requested by the client, which are then sent by the server. Each chunk size is of 65131 bytes (header(50 bytes) + content(65001 bytes) + md5(80 bytes)).

A particular file again, will not be divided in chunks (the chunks are already present on the server side).If all the chunks of the file are being transferred, they are merged to form the original file. The md5 of the merged file is calculated and cross checked with the md5 of the original file (extracted from the header of the first chunk), thus validating the integrity of the downloaded file.

Command : GET Chrome.exe 5 8

Client Side

```
THE FILE TRANSFER MODE:  chunk
server sent filename: Chrome.exe
TRANSFER OF CHUNKS begins
```

Figure 3.14: A file being transferred in chunk mode

```
name read Chrome.exe_CH5_260000-324999
bytes read from socket 65131
opening file /home/laxmi/copiedfiles/.Chrome.exe_CH5_260000-324999 to write on
client side
```

Figure 3.15: Each chunk being copied on the client side (in chunk mode)

Server Side

```
client sent:ch
WRITING CHUNK ON SOCKET

No. of bytes being written on socket : 65131
opening file to read chunk /home/Parul/sharedfolder/.Chrome.exe_CH5_260000-324999
sending filename to server Chrome.exe_CH5_260000-324999
recv=
No. of bytes being read from socket : 3
```

Figure 3.16: Server writing each chunk on socket one by one

- **Large Chunk mode** – The file size greater than 1950030 bytes is divided in the form of large chunks on the server side. Desired number of chunks are requested by the the client and then sent by the server. If a particular file is requested again, the file will not be divided in chunks (the chunks are already

present on the server side). Each large chunk size is of 1904 kb (header(50 bytes) + content (1950030 bytes) + md5(80 bytes)). Each time, 65001 bytes are read from the socket in a chunk. This is repeated for 30 times and the bytes are concatenated to form a large. If all the chunks are available, the md5 of the merged file is calculated and checked with the md5 of original file (extracted from the header of first chunk), checking for the integrity of downloaded file. When client requests the filelist on server, the 'number of chunks' has a 'L' concatenated with it to depict that it is a large file.

Command : GET lion.avi 1 4

Client side

```
THE FILE TRANSFER MODE:  supr
server sent filename: lion.avi

TRANSFER OF CHUNKS begins

RECEIVING CHUNK

FILE TO BE TRANSMITTED IN CHUNKS lion.avi
Total no. of chunks of file 6
file to be checked for chunks received previously  lion.avi
NO Previous records found
Creating and Initialising info filefile to be opened /home/laxmi/copiedfiles/.i
nfolion.avi
```

Figure 3.17: A file being transferred in large chunk (supr) mode.

```
name read lion.avi_CH1_0-1950028
50
1 65001
2 65001
3 65001
4 65001
5 65001
6 65001
7 65001
8 65001
9 65001
```

Figure 3.18: A large chunk is written on socket (bytes read in chunk form are concatenated 30 times to form a large chunk)

Server Side *How Of Application on GET command*

```
No. of Chunks(65000 B) 6
creating chunk /home/Parul/sharedfolder/.lion.avi_CH1_0-1950028
creating chunk /home/Parul/sharedfolder/.lion.avi_CH2_1950029-3900057
creating chunk /home/Parul/sharedfolder/.lion.avi_CH3_3900058-5850086
creating chunk /home/Parul/sharedfolder/.lion.avi_CH4_5850087-7800115
creating chunk /home/Parul/sharedfolder/.lion.avi_CH5_7800116-9750144
creating chunk /home/Parul/sharedfolder/.lion.avi_CH6_9750145-11580174

CHUNKS CREATED ON SERVER SIDE
```

Figure 3.19: Chunks are created on the server side as the file is transmitted for the first time.

```
WRITING CHUNK ON SOCKET
```

```
1-65001
2-65001
3-65001
4-65001
5-65001
6-65001
7-65001
8-65001
```

Figure 3.20: Large chunk is written on socket

```
MD5 of chunk 6a645a5f0e8892942d931c394269d060
md5 bytes written on socket 80
opening file to read chunk /home/Parul/sharedfolder/.lion.avi_CH2_1950029-3900057
sending filename to server lion.avi_CH2_1950029-3900057
```

Figure 3.21: MD5 of each chunk is sent to the client

3.7.3 Flow Of Application on GET command

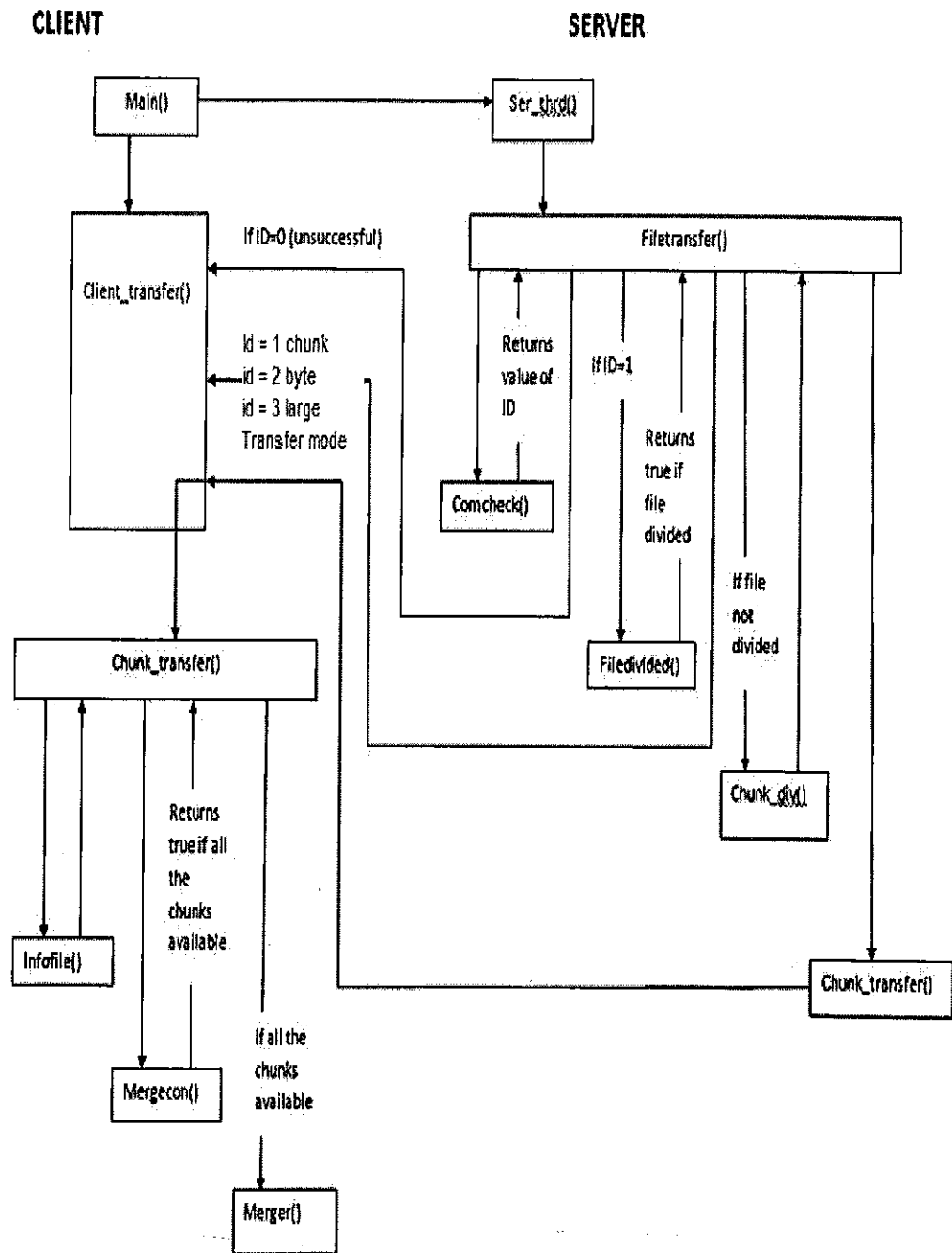


Figure 3.22: Client Server interaction for 'GET' command

Server Side

The application calls a file transfer function for the 'GET' command. A Comcheck() function is called which returns the value of ID. This ID decides if the file has to be transmitted in 'byte mode', 'chunk mode' or 'large chunk mode'. If value of ID is 2, means file will be transmitted in byte mode. If ID returns a value 1, Filedivided() function is called. It returns true if the file has already been divided (i.e. infofile is present). If it returns false, first the file is divided and then Chunk_transfer function is called. The Chunk_transfer function again checks for the value of ID. If ID is 1, chunk structure is used and the mode of transmission is 'chnk'. If ID returned is 3, the mode is 'supr' and superchunk structure (large chunk) is used. The formation of a large chunk has already been explained above.

Client Side

After the mode of transmission has been decided, the server sends the mode to the client and subsequently the kind of transmission takes place. The Chunk_transfer function checks for the infofile and updates it accordingly. The Mergecon() function checks for the merging condition (by checking the infofile) and accordingly calls for the Merger() function.

3.7.4 Merging

When all the chunks of the requested file are available on the client side, these are merged automatically to form the original file again. The merger function opens the corresponding 'infofile' to check for the availability of all chunks. If they are available i.e. the infofile contains all 1s, each chunk is opened one by one and the content is concatenated in a new file. The new file is renamed with the original file name. The md5 of downloaded file is calculated and checked with the md5 of the original file. This md5 is extracted from the header of the first chunk. This would authenticate the integrity of the received file.

In case the file has only one chunk (in byte mode transmission), the file is not divided into chunks and is written at once on the socket and thus transmitted to the client side. To check for the authenticity, the server sends the md5 to the client, after which it is checked with the md5 of the received file, which is calculated at the client side. Therefore merging is not required in this case.

```
(mergecon(nc,hfile,shdir))//mergecondition true function  
definition
```

```
{  
  
    while(!feof(hf))  
  
        {  
  
            ch=fgetc(hf); //reading every character  
            of 'infofile'  
  
            if(ch=='0')  
  
                return 0; // returning 0 if '0' found  
  
        }  
  
    unlink(loc); //delete the infofile  
  
    return 1; //Returning 1 no 0s found in the  
    file  
}  
  
if(mergecon(nc,hfile,shdir))  
  
    merger(nmfile,shdir,tnc,tid); //If mergecon()  
returns 1, merger  
function called  
  
else  
  
    {  
  
        printf("\n\n ALL chunks NOT available to merge  
them..");  
  
        ....  
  
    }  
  
return;
```

```

}

void merger(char fname[],char shdir[],int nc,int tid)
//File merging function

{
    ..

    file=fopen(fname,"w");//File opened to write chunks

    for(i=1;i<=nc;i++)
    {
        gcvt(i,10,num);

        if(tid==1)      //Checking tid for mode of
        transmission

        gcvt(((i-1)*CHUNK_SIZE),10,num); //Chunk mode

        else

        gcvt(((i-1)*SUPERCHUNK_SIZE),10,num); //Large
        Chunk mode

        ..

        if(nc==1) //If only '1' chunk to merge

        {

            n=scandir(loc, &namelist, 0, alphasort);
            //Check for chunks

            ..

            strcat(first,namelist[i]->d_name);

            ..

```



```

    }

else
{
    ..

    if(tid==1)

        gcvt(CHUNK_SIZE-1,10,num);

    else

        gcvt(SUPERCHUNK_SIZE-1,10,num);

    ..
}

fp=fopen(first,"r"); //Opening chunk

fread(header,50,1,fp); //Reading header

from file

unlink(first);

..

fclose(fp);

}

else
{
    ..

    f[i-1]=fopen(fpath,"r");

    if(tid==1)

```

```

        fread(&chnk,sizeof(struct
chunk),1,f[i-1]); //Reading chunk

        else

            fread(&schk,sizeof(struct
superchunk),1,f[i-1]);

                                // Reading large
                                chunk

        if(i!=nc)

        { ..

            if(tid==1)

                fwrite(chnk.content,CHUNK_SIZE,1,file);

                                else                                //Writing chunk
in file

                fwrite(schk.con,SUPERCHUNK_SIZE,1,file);

                                //Writing large chunk
in file

        else

        {

            if(tid==1)

                fwrite(chnk.content,(fsize-(i-
1)*CHUNK_SIZE),1,file);

                                //Last chunk

        else

```

```

        fwrite(schk.con, (fsize-(i-
1)*SUPERCHUNK_SIZE), 1, file);

//Last      super
chunk

    }

    fclose(f[i-1]);

    if(i!=1)

        unlink(fpath);

}

fclose(file);

..

calmd5(filemd5,md5); //MD5 of new file

md5[33]='\0';

printf("THE MD5 OF MERGED FILE    %s",md5);

printf("THE MD5 OF ORIGINAL FILE  %s\n\n",original);
//MD5 of original file

if(strcmp(md5,original)==-1)

{

    printf("making merged file ");

    *fptros=fopen(dpath,"w"); //MD5
checked, 'merged' file formed

}

else

{

```



```

        FILE      *fptros=fopen(dpath,"w");    //MD5    not
matching, 'mergeerror'
        file formed

        unlink(filemd5);    //Deleting file with errors
    }

    return;
}

```

If all chunks are available to merge, the file will be merged by calling `merger()` function. If the MD5 of merged file and original file are same, 'merged' file formed in `copiedfiles` folder else 'mergeerror' file is formed and the new merged file is deleted.

All chunks not available to merge

Client Side

```

ALL CHUNKS COPIED SUCCESSFULLY

opening file /home/laxmi/copiedfiles/.infoChrome.exe to check merging condition

ALL chunks NOT available to merge them..

```

Figure 3.23: All the chunks are not available and cannot be merged

All chunks available to merge

Client Side

```

opening file /home/laxmi/copiedfiles/.infoChrome.exe to check merging condition

CALLING MERGER FUNCTION

filename Chrome.exe

```

Figure 3.24: Client checks the infofile for merging condition and calls the merger function

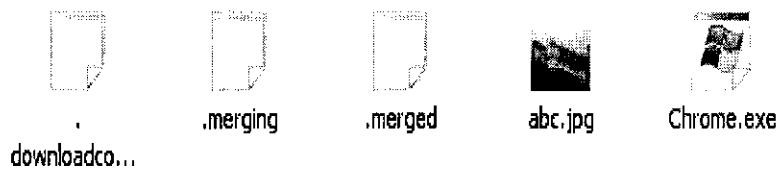


Figure 3.28: 'Copiedfiles' folder after all the chunks are received and merged

3.8 Installation Steps

- Copy all the files for client, tracker and server in the corresponding "root" folders of the specified LINUX operating system that is being used.
- Correct entries in the corresponding configuration files of the server (config.ini) and client (config_client.ini) of client should be ensured.
- Steps for compiling

```
$ gcc Client.c -o Client
```

```
$ gcc Tracker.c -o Tracker -lpthread
```

```
$ gcc Server.c -o Server -lpthread
```
- Steps for running

```
$ ./Client
```

```
$ ./Tracker
```

```
$ ./Server
```

CHAPTER 4

TRACKER

A general Server-Client model is a 2-tier model where Client connects to the desired Server, knowing which Server provides what services. In this case, the Client is supposed to have pre-knowledge of the services available with the Server and the location of server, in terms of its address.

The introduction of one more level, named by us as Tracker, adds 1 more tier to the communication architecture previously discussed, thus adding an abstraction level between the Server and Client. We will discuss this as we proceed.

4.1 Description

The Tracker, in our implementation, is responsible for creating a transparency between the Server and Client, as the Client does not need to know the services available with the Server and its address on the network, thus creating an abstraction level.

The Tracker acts as a server for both, the Servers and the Clients. It communicates with the Servers and registers their addresses with its list, whereas provides these address to the Client, whenever needed or asked for.

The Tracker communicates with these entities by creating a separate thread for each process, that is for each Server and Client, and uses the technique of mutual exclusion to ensure synchronization and perseverance of shared variables, shared amongst these threads.

4.2 Server-Tracker Relation

The Tracker, as already mentioned, acts as a server for the Servers. The communication between these two takes place in the following steps.

Step 1: Server establishes a connection with Tracker.

Step 2: Server sends its address to the Tracker.

Step 3: Tracker adds the address to its list.

We will now discuss each step in more details.

4.2.1 Step 1:

The Server establishes the connection with the Tracker, which is ready and listening for any connection request from the Server on a pre-specified port address. This connection is established as a normal server-client connection, with the Server acting as a client i.e. an active entity initiating the communication. The Tracker, as already mentioned, acts as the server in this communication and accepts the connection request, and creates a separate thread for the Server using POSIX Thread techniques and functions/APIs, discussed earlier.

```
***CLIENT BEGIN***
!!!Socket 3 Created!!!
!!!Binding Socket with Port!!!
!!!Waiting for a Client to connect!!!
!!!Trying to listen!!!
Listening...

***SERVER BEGIN***
!!!Socket 4 Created!!!
!!!Binding Socket with Port!!!
!!!Waiting for a Server to connect!!!
!!!Trying to listen!!!
Listening...

!!!Ready to Accept Connection from CLIENT!!!

!!!Ready to Accept Connection from Server!!!
```

Fig. 4.1 Tracker awaiting connection from Servers

4.2.2 Step 2:

After the connection has been established, the Server sends its address to the Tracker using send() function. The server receives this using the recv() function.

4.2.3 Step 3:

On successful receipt of the address, the Tracker can do 1 of the following depending on whether the address is already present or not.

- If the address is not present, the Tracker adds it as a new node to the array of servers.
- If the address is already there, it just resets the connect parameter with the server, which shows the existing number of clients connected to it.

```

!!!Ready to Accept Connection from SERVER!!!
!!!Server Address Recieved : 172.16.9.136 !!!
!!!Connection Terminated With The Server!!!

*****
TOTAL SERVER REGISTERED : 2
SERVER      CONNECTIONS
172.16.9.135      0
172.16.9.136      0
*****

```

Fig. 4.2 Tracker on receipt of new Server address adds it to the list

```

!!!Server Address Recieved : 172.16.9.135 !!!
!!! Server Exist !!! Resetting !!!
!!!Connection Terminated With The Server!!!

*****
TOTAL SERVER REGISTERED : 5
SERVER      CONNECTIONS
172.16.9.135      0
172.16.9.136      1
172.16.9.137      0
172.16.9.138      0
172.16.9.139      0
*****

```

Fig. 4.3 Tracker on receipt of existing Server address, resets it

The list of servers is maintained in an array of a structure, which is an abstract data type in C. It has been defined as

struct server

```

{
    char pr[40];
    int connect;
};

```

The structure consists of 2 parameters.

- pr is used to store the address of the servers.
- connect is used to store the number of connections in terms of Clients currently established with the server.

```
*****
TOTAL SERVER REGISTERED : 5
SERVER      CONNECTIONS
172.16.9.135      1
172.16.9.136      0
172.16.9.137      0
172.16.9.138      0
172.16.9.139      0
*****
```

Fig. 4.4 The List of Servers as displayed

The code for this is:

```
for(i=0; i<avSerCount; i++)
{
    if( strcmp(serList[i].pr, pr) == 0 )
    {
        serList[i].connect = 0;
        exist = 1;
    }
}

if(exist == 0)
{
    strcpy(serList[avSerCount++].pr, pr);
}
```


serList here is an array of servers and avSerCount keeps a track of total number of Servers registered with the Traker, i.e. the length of the list.

After the address has been added, the Tracker breaks the connection with the Server and destroys its thread too.

4.3 Client-Tracker Relation

The Tracker, as already mentioned, acts as a server for the Clients too. The communication between these two takes place in the following steps.

Step 1: Client establishes a connection with Tracker.

Step 2: Client asks for an address of the Server. Tracker decides address to send.

Step 3: Tracker sends the address.

We will now discuss each step in more details.

4.3.1 Step 1:

Like in the case of Servers in previous section, the Client establishes the connection with the Tracker, which is ready and listening for any connection request from the Clients on a pre-specified port address. This connection is established as a normal server-client connection, with the Client acting as a client i.e. an active entity initiating the communication. The Tracker, as already mentioned, acts as the server in this communication and accepts the connection request, and creates a separate thread for the Client using POSIX Thread techniques and functions/APIs, discussed earlier.

```
***CLIENT BEGIN***
!!!Socket 3 Created!!!
!!!Binding Socket with Port!!!
!!!Waiting for a Client to connect!!!
!!!Trying to listen!!!
Listening...

***SERVER BEGIN***
!!!Socket 4 Created!!!
!!!Binding Socket with Port!!!
!!!Waiting for a Server to connect!!!
!!!Trying to listen!!!
Listening...

!!!Ready to Accept Connection from CLIENT!!!
!!!Ready to Accept Connection from Server!!!
```

Fig. 4.5 Server awaiting connection from Clients

4.3.2 Step 2:

After the connection has been established, the client asks for an address depending upon whether it needs a change of server or not.

- If the Client doesn't send a valid IP address, it is sent the address of server with least number of existing connections.
- If the Client sends a valid IP address i.e. needs a change in the address, the server sends it an alternate address of the server, other than the one whose address has been sent, with least number of existing connections.

```
!!!REQUEST FOR NEW SERVER!!!
!!!Server Address Sent : 172.16.9.135!!!
!!!Connection Terminated With The Client!!!

*****
TOTAL SERVER REGISTERED : 5
SERVER      CONNECTIONS
172.16.9.135      1
172.16.9.136      0
172.16.9.137      0
172.16.9.138      0
172.16.9.139      0
*****
```

Fig. 4.6 Tracker on request for a new Server address

```
Previous Server : 172.16.9.136
SERVER Address Received : 172.16.9.135
```

Fig. 4.7 Client requesting a change in Server

```
!!!REQUEST FOR CHANGE OF SERVER FROM 172.16.9.136!!!
!!!Server Address Sent : 172.16.9.135!!!
!!!Connection Terminated With The Client!!!

*****
TOTAL SERVER REGISTERED : 5
SERVER      CONNECTIONS
172.16.9.135      1
172.16.9.136      0
172.16.9.137      0
172.16.9.138      0
172.16.9.139      0
*****
```

Fig. 4.8 Tracker on Request of Change Of Server

The code for the selection of the server address is:

```
if(avSerCount == 1)

{

    pos == 0;

    sleep(5);

}

else if( pr[0] == '0' )

{

    for(i=0; i<avSerCount; i++)

    {

        if(serList[i].connect < temp)

        {

            temp = serList[i].connect;

            pos = i;

        }

    }

}

else

{

    int postemp=0;

    for(i=0; i<avSerCount; i++)

    {

        if( (strcmp((serList[i].pr),pr))==0 )
```



```

        {
            postemp=i;
            if(serList[i].connect > 0)
                serList[i].connect -= 1;
        }
    }

    if(postemp == 0)
    {
        temp = serList[1].connect;
        pos = 1;
    }

    for(i=0; i<avSerCount; i++)
    {
        if(i == postemp)
            continue;

        if( (serList[i].connect<temp) )
        {
            temp = serList[i].connect;
            pos = i;
        }
    }
}

```

Here, if only 1 node exists in the list of Servers, the very same address will be sent. Else, as already discussed, the address will be sent depending on whether the Client needs a change in address, due to some fault in connection or server etc., or just needs a fresh connection.

On receipt of a valid address, the Tracker deducts 1 from the number of connections with the server in the List, i.e. the connect parameter of the server node.

4.3.3 Step 3:

The Tracker sends the selected address to the Client using the same send()-recv() APIs. The Tracker then adds 1 to the connect parameter in the node of the sent server, thus showing an increase in number of connections on the Server. It then breaks the connections with the Client and also destroys the thread for the Client.

4.4 Fault Recovery Feature

The fault recovery feature in the Tracker is basically accounted for faults occurring in Server-Client communication.

When a Client encounters a fault, on restarting it sends the address of the previous server. The Tracker then searches for the server with least connections, other than the one whose address has been received and sends the new address. It then deducts 1 from the number of connections in the node of the received server, thus maintaining the integrity too.

When a Server encounters a fault and restarts, it resends its address. The Tracker first checks the address in the existing list. When it finds an existing node for the Server, it resets the number of connections with it to 0, thus maintaining integrity once again.

CHAPTER 5

GRAPHICAL USER INTERFACE

Introduction

The uniqueness of the proposed software inherits from the amalgamation of two different languages that serve specific purposes. On one hand, GNU C handles the core processing backend efficiently while Java (Swing) based GUI eliminates the need for command line operated software that required the user to have prior knowledge of the correct command usage. This organization requires the interaction between the logically different components. This has been done by a simple yet effective method known as 'pooling', which uses temporary files as a common medium to share information.

5.1 Introduction to the GUI

5.1.1 Request Tab

The connect button in this tab is used to connect to the tracker which directs it to the server. The disconnect button is used to disconnect the client from the server.

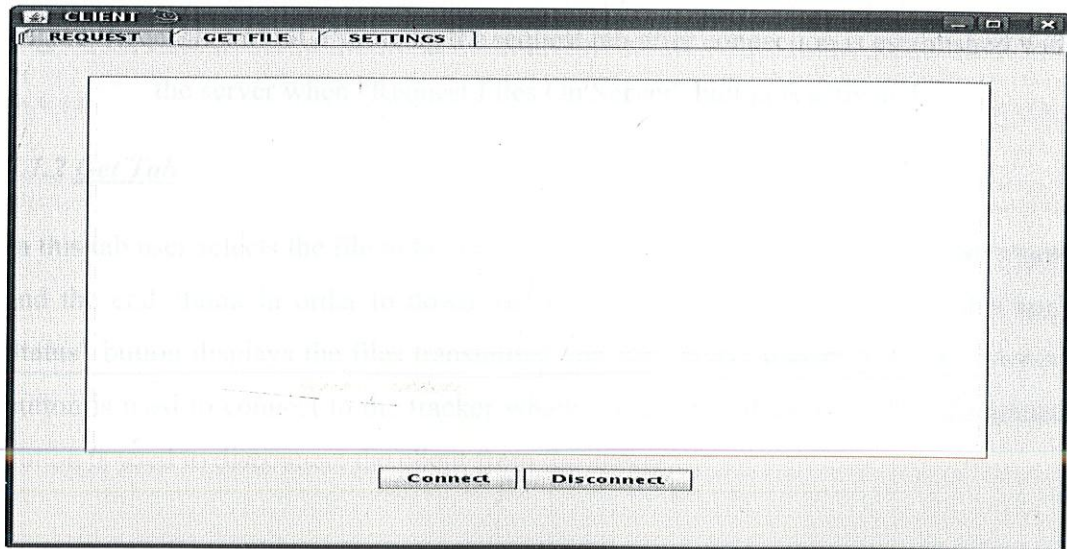


Fig5.1 Screenshot of GUI showing the request tab when the connection with the client is not established.

When the connection is established “Request File On Server” button is activated. This button instructs the server to send over to the requesting client the list of files in the shared folder at the server.

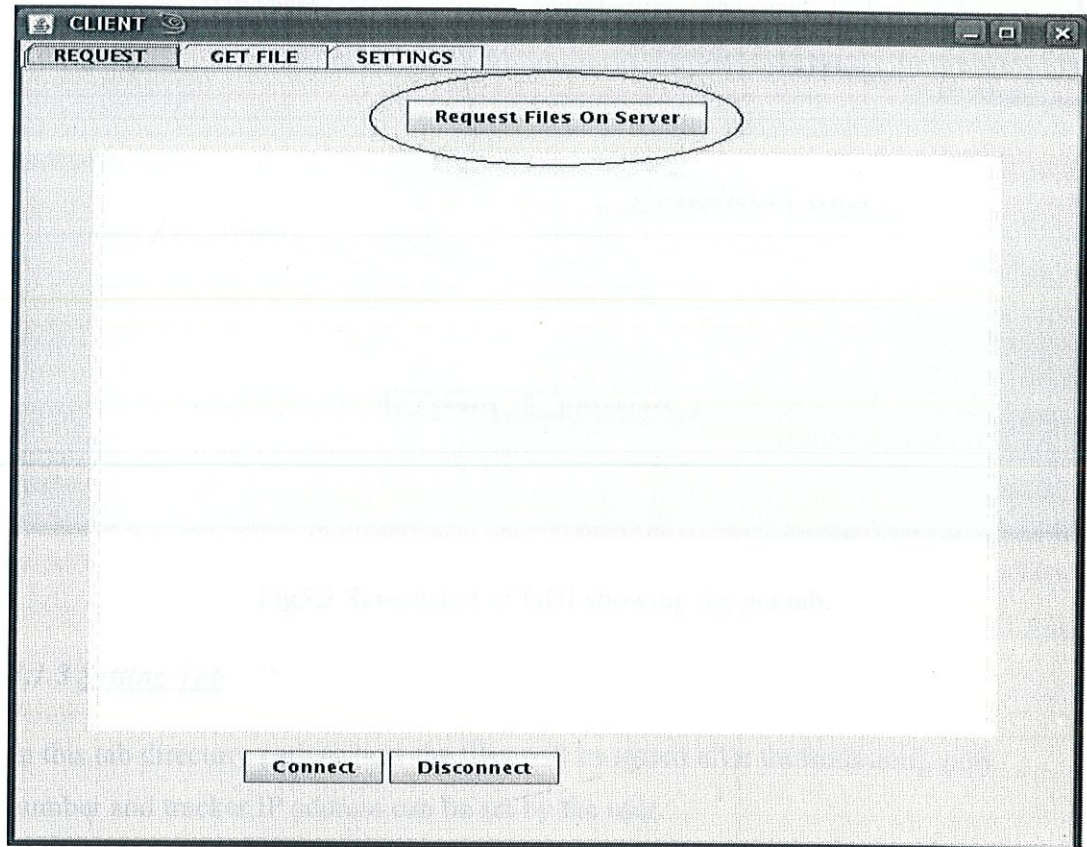


Fig5.2 Screenshot of GUI showing the request tab after connection is established with the server when “Request Files On Server” button is activated.

5.1.2 Get Tab

In this tab user selects the file to be transmitted. User selects the file name, start chunk and the end chunk in order to download a file from the server. “View File/Chunk Status” button displays the files transmitted and the chunks transmitted. The connect button is used to connect to the tracker which directs it to the server. The disconnect button is used to disconnect the client from the server.

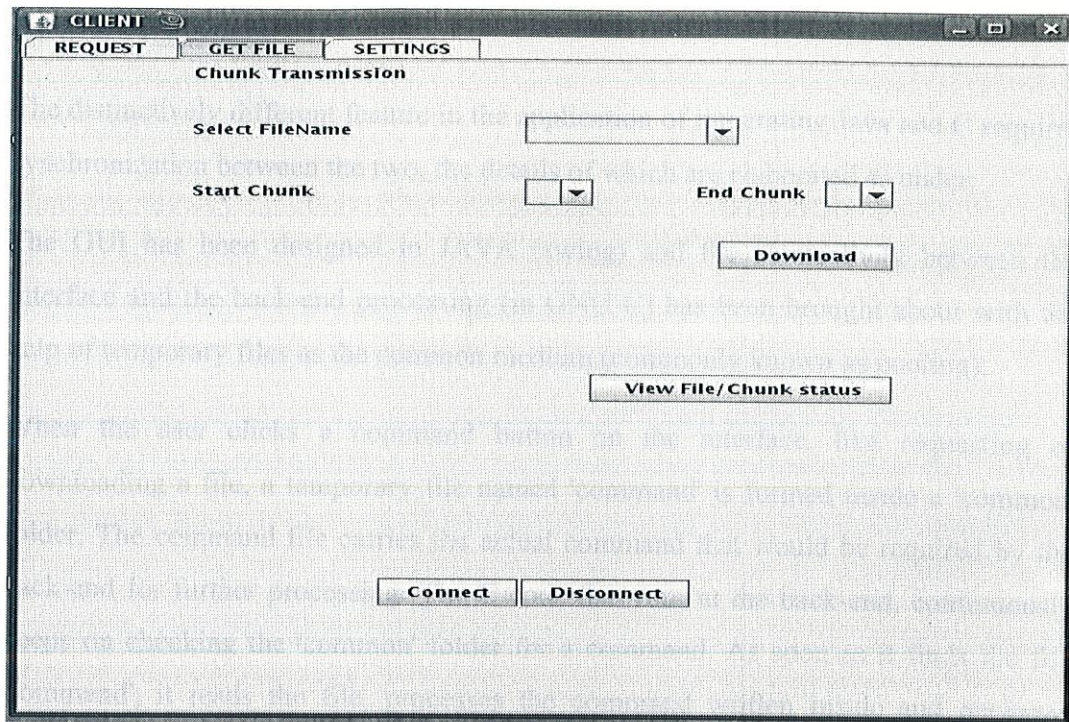


Fig5.3 Screenshot of GUI showing the get tab.

5.1.3 Setting Tab

In this tab directory path (where the files will be stored after transmission), port number and tracker IP address can be set by the user.

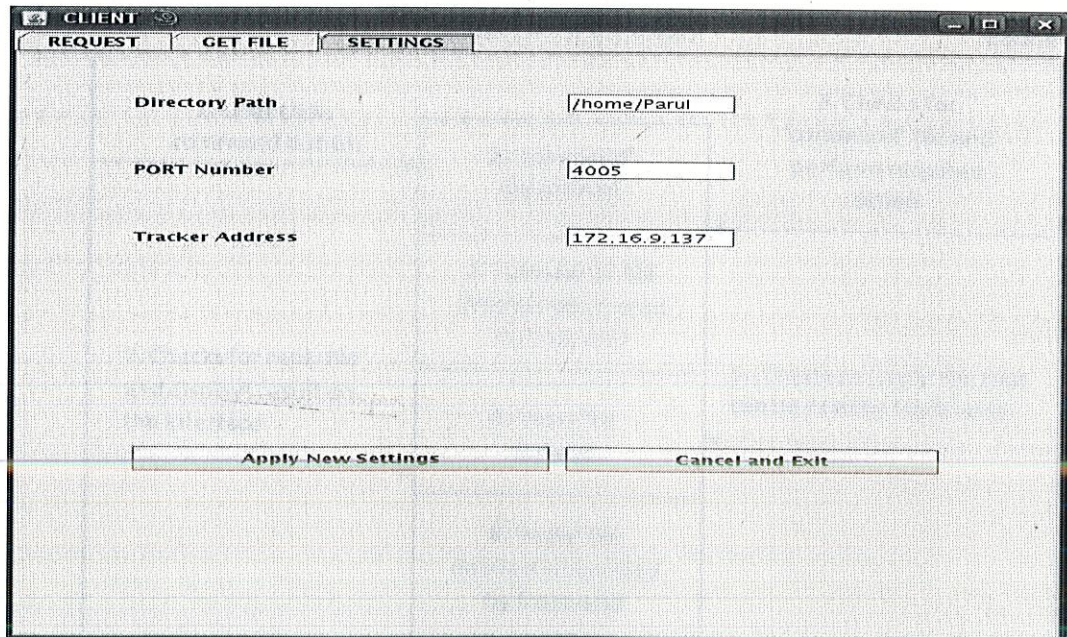


Fig5.4 Screenshot GUI showing the setting tab.

5.2 Software Bridge

The distinctively different feature in the application of integrating Java and C requires synchronization between the two, the details of which are elaborated as under:

The GUI has been designed in JAVA (swing) and the connectivity between the interface and the back-end processing (in GNU C) has been brought about with the help of temporary files as the common medium (commonly known as pooling).

When the user clicks a command button on the interface, like requesting or downloading a file, a temporary file named 'command' is formed inside a 'common' folder. The command file carries the actual command that would be required by the back-end for further processing. The C code that runs at the back-end, continuously keeps on checking the 'common' folder for a command. As soon as it finds the file 'command', it reads the file, processes the command written inside and performs required action. As soon as the command is processed by the C code, the temporary file is deleted. If the command is to list files on server, the code in the back-end puts the list of all the files present on the server in a file named 'reply', which again is formed in the 'common' folder. The Java code further accesses the 'reply' file and list the file on the interface.

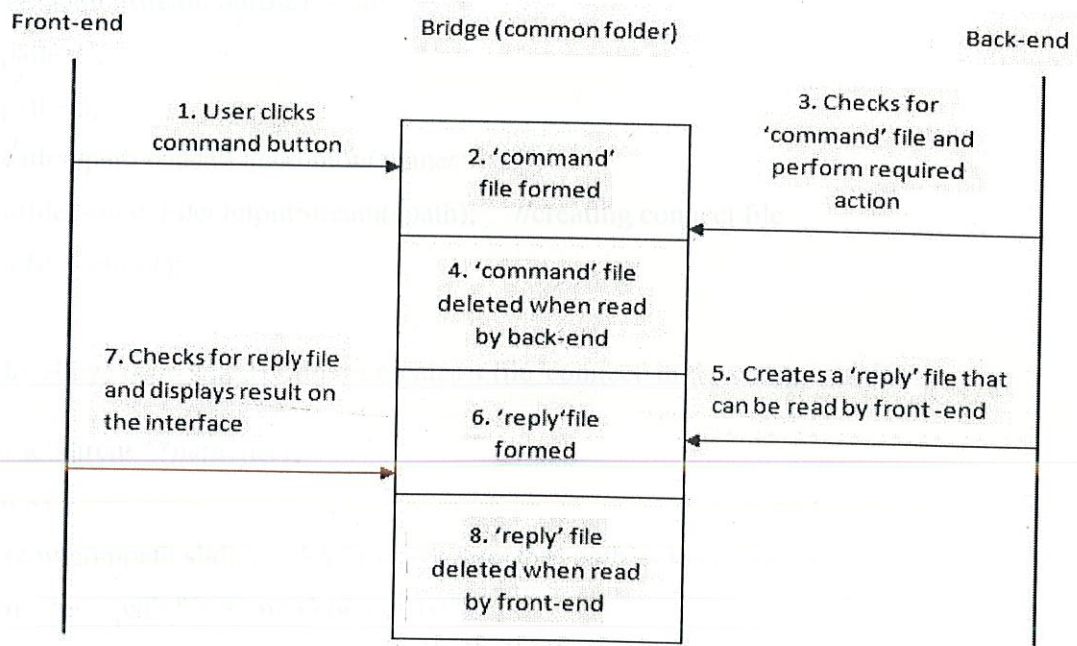


Fig5.5 Software bridge depicting synchronization between Java and C.

5.3 GUI Description

5.3.1 Connect

When the user clicks a connect button on the interface, a temporary file named 'connect' is formed inside a 'common' folder. The C code that runs at the back-end, continuously keeps on checking the 'common' folder for the 'connect' file. As soon as it finds the file 'connect', it reads the file, the client connect to the tracker which directs it to the server. If the connection is not established continuous attempts are made to connect to the server. As soon as the connection is established by the C code, the 'connect' file is deleted and a file named 'connected' is created in the common folder by back-end. The java code then checks for 'connected' file and displays in a dialog box "CONNECTED to server...."

```
public String st=" "; //st stores directory path as in config_client.ini
public String cpath="";
public FileReader f=null;
f = new FileReader("config_client.ini");
BufferedReader b=new BufferedReader(f);
st = b.readLine();
FileOutputStream outfile1 = null;
cpath="";
cpath=st;
cpath=cpath.concat("/common/connect.txt");
outfile1=new FileOutputStream(cpath); //creating connect file
outfile1.close();
```

The above code snippet in java creates a file 'connect' in the common folder.

```
struct dirent **namelist1;
int n1;
strcpy(compath,shdir); //shdir stores directory path as in config_client.ini
strcat(compath,"/common/connect.txt");
while(1)
{
```

```

        n1=scandir(loc1, &namelist1, 0, alphasort); // scanning the shared directory
for connect file
    if(n1>=3)
    {
        if(fopen(compath,"r")!=NULL) //reading connect file
        {
            unlink (compath); //deleting the connect file
            //code for connection with server
            strcpy(compath,shdir);
            strcat(compath,"/common/connected"); //creating connected
file
            FILE *fp=fopen(compath,"w");
            fclose(fp);
        }
    }
}

```

The above C code snippet reads 'connect' file and when connection established with the server creates a file 'connected' in the common folder.

```

public String st=" "; //st stores the directory path as in config_client.ini
public String cpath="";
cpath=st;
cpath=cpath.concat("/common");
File f1=new File(cpath);
cpath=cpath.concat("/connected");
File f2 = new File(cpath);
Thread.sleep(10000L);
if(f2.exists()) //checking for 'connected' file
{

```

```

    f2.delete(); //deleting the connected file
    String dialogtitle = "CLIENT";
    String dialogmessage = "CONNECTED to server.....";
    int dialogtype = JOptionPane.PLAIN_MESSAGE;

```



```
. JOptionPane.showMessageDialog((Component)null,dialogmessage,dialogtitle,  
dialogtype);  
}
```

The Java code snippet that checks for 'connected' file in the common folder and displays in a dialog box "CONNECTED to server...."

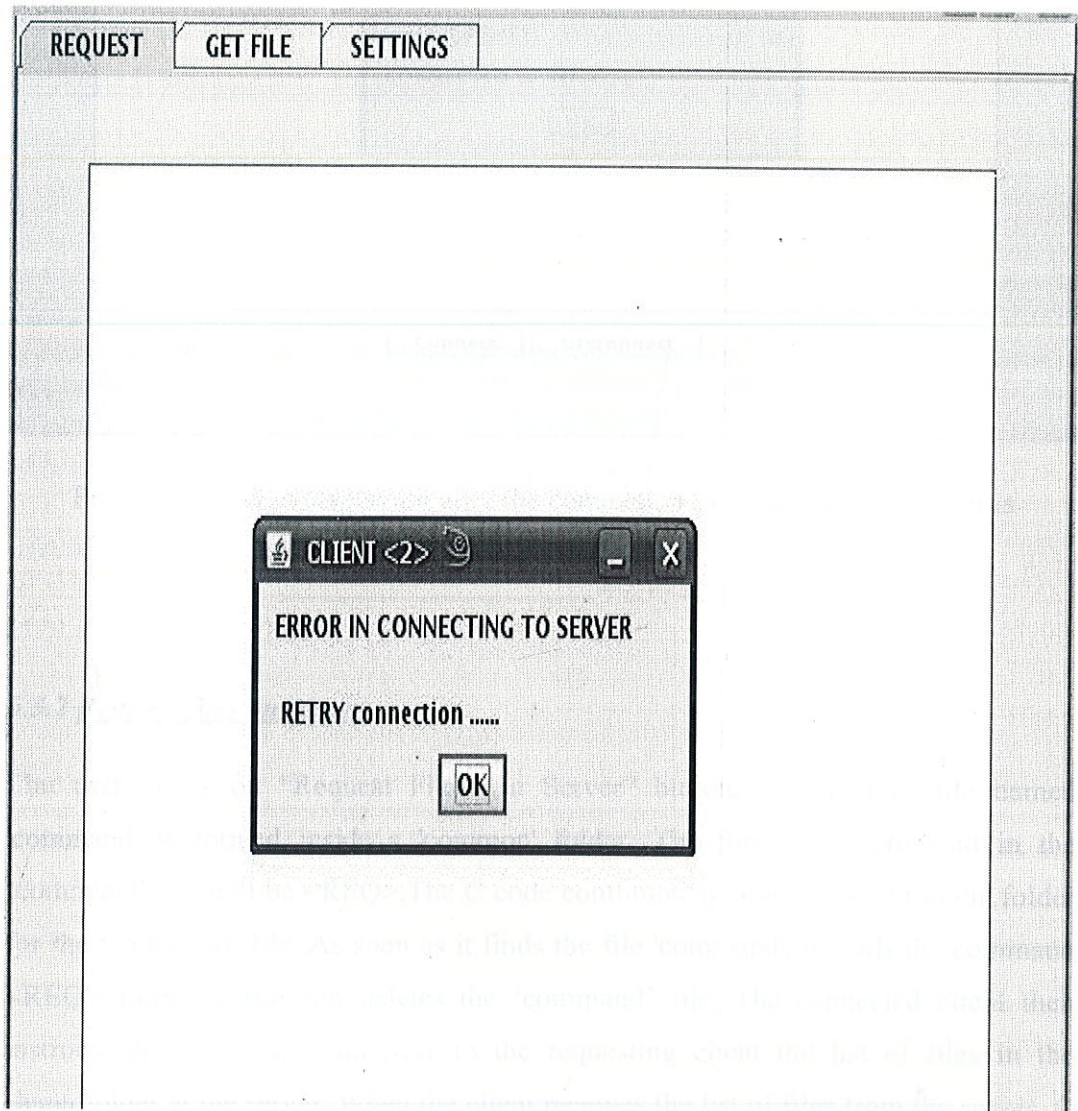


Fig5.6 Screenshot displaying when there is an error in connection of client with the server.

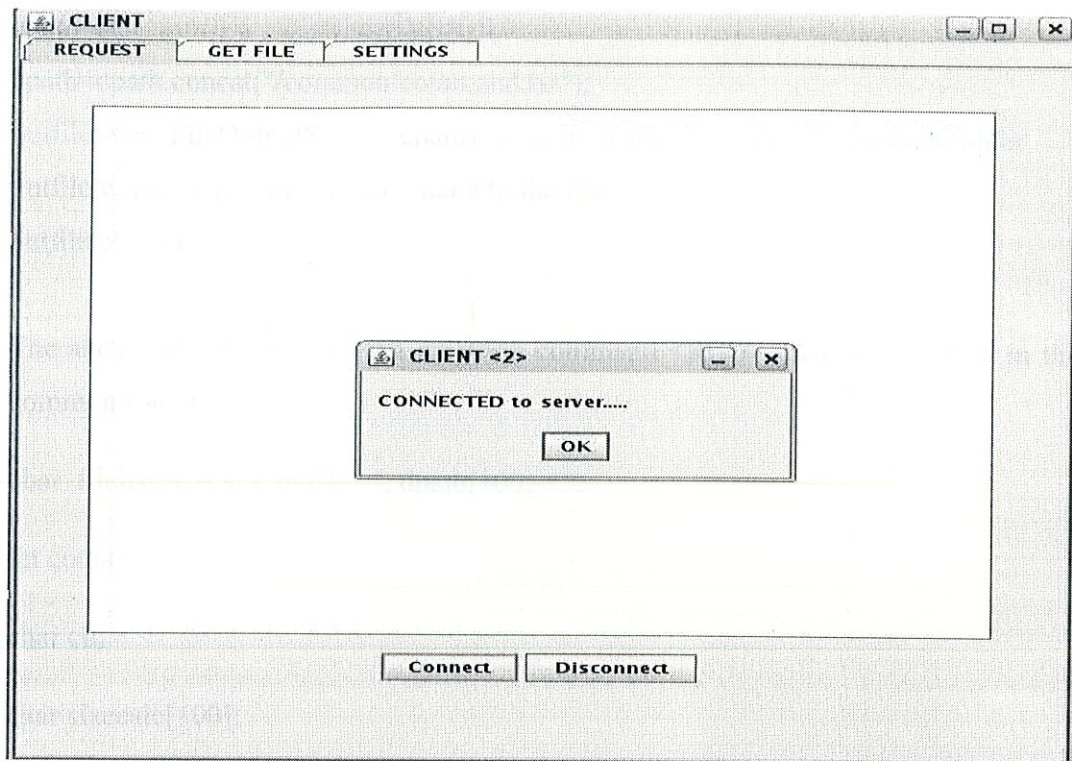


Fig5.7 Screenshot displaying after the connection of client with the server is established.

5.3.2 Request files on server

The user clicks on “Request Files On Server” button, a temporary file named 'command' is formed inside a 'common' folder. The format of command in the 'command' file will be <REQ>. The C code continuously checks the 'common' folder for the 'command' file. As soon as it finds the file 'command', it reads the command <REQ> from the file and deletes the 'command' file. The connected client then instructs the server to send over to the requesting client the list of files in the shared folder at the server. When the client receives the list of files from the server, C code creates a file named 'reply' in the common folder which contains file name, MD5 of the file, size of file and the number of chunks. The java code then checks for 'reply' file and displays the contents of the 'reply' file in a text box.

```
byte list[]={'R','E','Q'}; //command
FileOutputStream outfile = null;
cpath="";
```

```

cpath=st;
cpath=cpath.concat("/common/command.txt");
outfile=new FileOutputStream(cpath); //creating file command in common folder
outfile.write(list); //writing command to the file
outfile.close();

```

The above code snippet in java creates a command <REQ> in file 'command' in the common folder.

```

char filelist[MAX_LINE]=" ", dpath[100]="";

int count;

char ch;

char sizecalc[100];

struct dirent **namelist;

FILE *fp1,*fp2;

int n;

while(1)
{
    n = scandir(loc, &namelist, 0, alphasort);

    if(n>=3)
    {
        strcpy(dpath,shdir);    //shdir stores directory path as in
config_client.ini

        strcat(dpath,"/common/command.txt");

        if(fopen(dpath,"r")!=NULL) // checking for command file
        {

```

```

        fp1=fopen(dpath,"r");

        sleep(1);

        ch = fgetc(fp1);

        while(ch!=EOF) //reading the command in the command file
        {

                sizecalc[j] = ch;

                j++;

                ch = fgetc(fp1);

        }

        fclose(fp1);
}

if(!strcmp(sizecalc,"REQ"))
{

        unlink (dpath); //deleting the command file

        count = write(SerSock,&sizecalc,40, 0); //writing the command
on the socket

        count = recv(SerSock, &filelist,MAX_LINE, 0); //receiving the
list of files

        printf("\n\nFILELIST ON SERVER:\n");

        printf("%s\n", filelist);

        strcpy(dpath,shdir);

        strcat(dpath,"/common/reply.txt");

        fp2=fopen(dpath,"w"); //checking for the reply file in common
folder

```



```

        fputs(filelist,fp2); //writing the list of files in the reply file
        fclose(fp2)
    }
}
}

```

The above C code snippet reads command <REQ> from 'command' file. The command is then executed and list of file from server is received by the client and written in file 'reply'.

```

cpath="";
cpath=st;
cpath=cpath.concat("/common/reply.txt");
File fl = new File(cpath);
while(true)
{
    if(fl.exists()) //checking for the reply file in common folder
    {
        cpath="";
        cpath=st;
        cpath=cpath.concat("/common/reply.txt");
        int size;
        infile=new FileInputStream(cpath);
        size=infile.available();
        for(int j=0;j<size;j++)
            filelist.insert(j,((char)infile.read())); //reading the contents of reply file
        infile.close();
        String pr=new String(filelist);
        ta.insert(pr,1); //displaying the contents of reply file in the text box
        fl.delete();
    }
}
}

```

The Java code snippet that checks for 'reply' file and displays file name, MD5 of the file, size of file and the number of chunks in a text box.

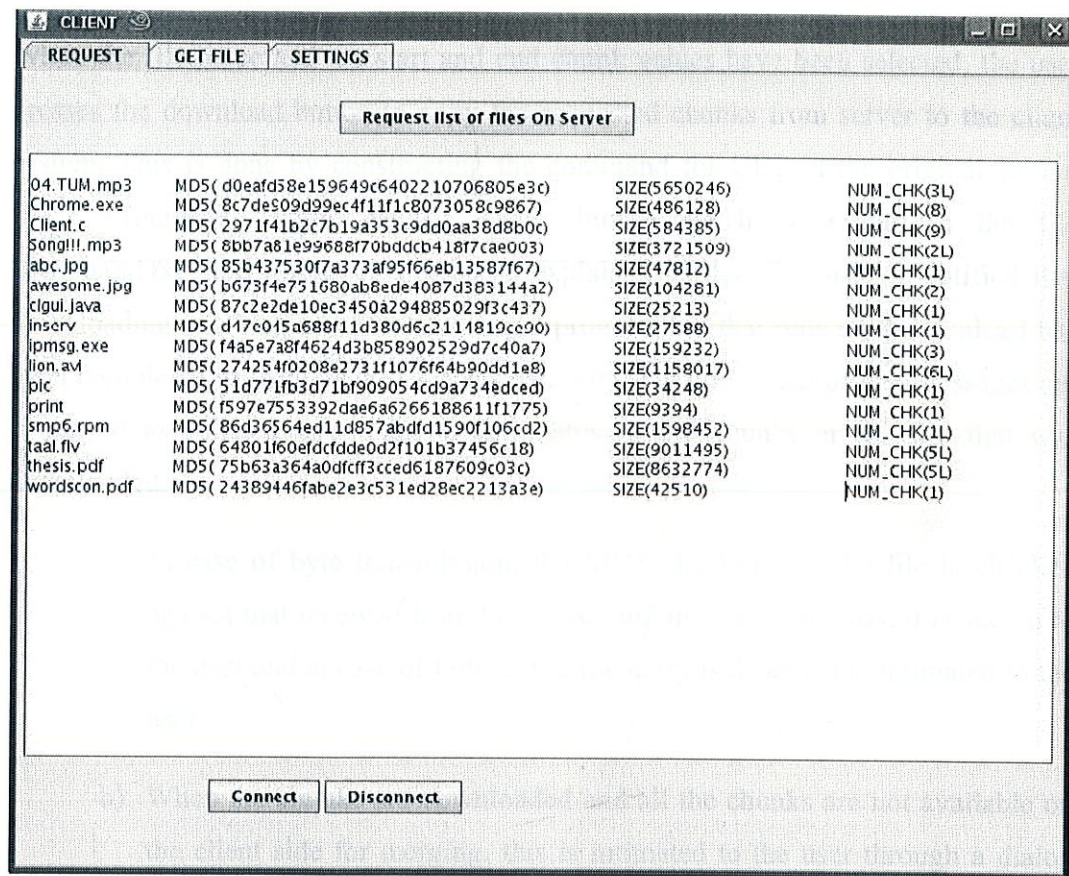


Fig.5.8 Screenshot displaying the list of files to the client, present on the server side.

This is in response to request files on server

5.3.3 Download files from server

The GUI aims at providing a simplified user-friendly interface for file download in the form of chunks and the GET FILE tab provides with these features and utilities. This tab lets the user select the file name that has to be downloaded and the start-chunk and the end-chunk numbers of the same that are to be downloaded, from simple dropdown menus.

The 'Select File Name' combo box contains the total list of files that are available with the server and along with the total chunk numbers of the corresponding file, in the bracket against their names. In case of large file chunks the chunks are concatenated with 'L' to denote the same. After selecting the name of the file, the user

can select the values of start and the end chunk from the dropdown menu. The values that can be selected are only the valid values applicable for that particular file, thereby avoiding any unintentional errors in requesting the server for unavailable chunks.

When the file name and the start and end chunk values have been selected, the user presses the download button to copy the requested chunks from server to the client system. This is done by constructing the command for GET in the original format 'GET <filename> <start chunk> <end chunk>' which is written in the file 'command.txt' and further processed as explained earlier. The user is notified that downloading is taking place by showing a progress bar that runs until download has been completed or some fault has occurred. After the download, proper messages are displayed as follows, according to the status of the chunks or the file that was downloaded.

- a) In case of byte transmission, the MD5 checksum of the file is checked against that received from the server and in case of success, it is shown to the user and in case of failure, the file entry is deleted and intimated to the user.
- b) When the chunks are downloaded and all the chunks are not available on the client side for merging, this is intimated to the user through a dialog box.
- c) In case all the chunks are present on the client side, merging takes place and merge status is made known to the user.
- d) When the user has requested a file that is already present on the client side, A warning message is shown to the user and if user accepts then only the file is overwritten from the beginning by deleting all previous records of it.

The following is the actionlistener of the the 'download' button, i.e. this is executed whenever the button is pressed.

....

```
String getcommand=new String("GET ");           //constructing  
name1=(String)jcb.getSelectedItemAt();           //command
```



```

name=name1.substring(0,name1.indexOf(" "));
....

start=jcb1.getSelectedItem().toString();
...
end=jcb2.getSelectedItem().toString();
getcommand=getcommand.concat(end);
FileOutputStream outfile;
cpath=cpath.concat("/common/command.txt");    //writing command constructed in
outfile2=new FileOutputStream(cpath);          //'command.txt' file.
outfile2.write(getfinal);
outfile2.close();

....
cpath=cpath.concat(".downloadcomplete");        //delete previous
'.downloadcomplete'
File f1 = new File(cpath);                      //file
if(f1.exists())
f1.delete();
barDo.setVisible(true);                        //show progressbar
timer = new Timer(2000,actionListener);         //starting timer that checks for
fault
timer.start();
t=new Thread(new thread1());                   //initiating thread that checks for
t.start();                                     //'downloadcomplete'

```

The above code snippet shows that how the command is constructed from the user selection in dropdown menus by extracting the necessary information and writing it in the format required to the 'command.txt' file in the 'common' folder. For example, if file name selected is 'song!!!.mp3' and start and end chunk values as 1 each, command written file- 'GET song!!!.mp3 1 1'.

Further the code initiates timer and a concurrent thread, checking for download state of the application which is known by the C code creating the appropriate files.

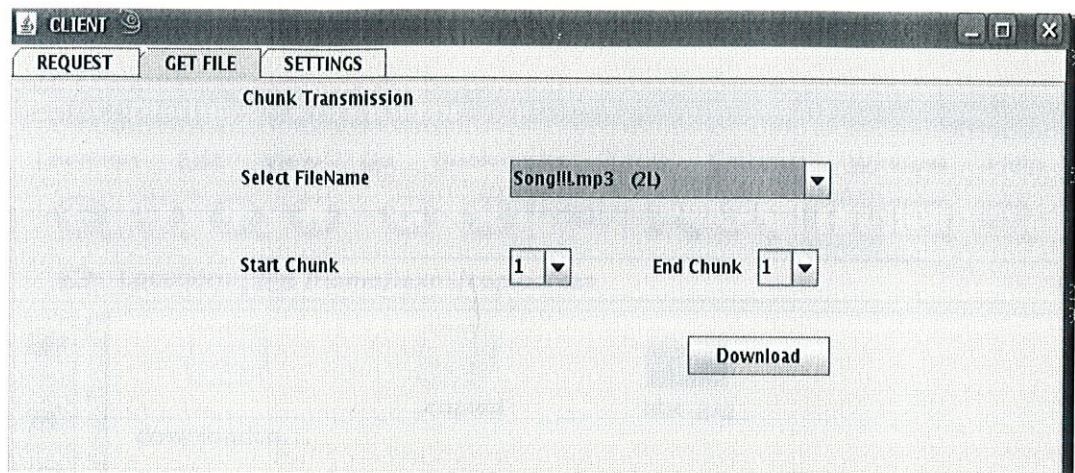


Fig.5.9 Snapshot showing the 'GET FILE' where file and its start and end chunks are selected.

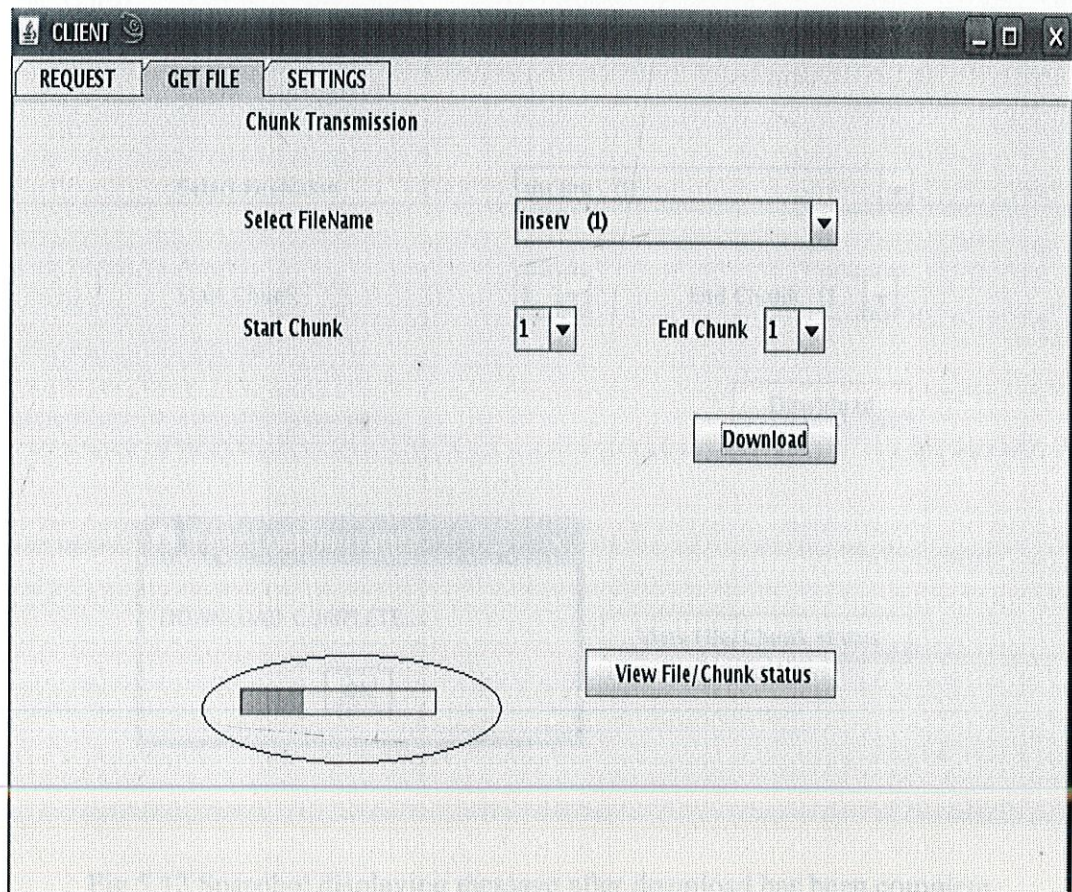


Fig.5.10 Snapshot that highlights the download progress bar that appears until download takes place.

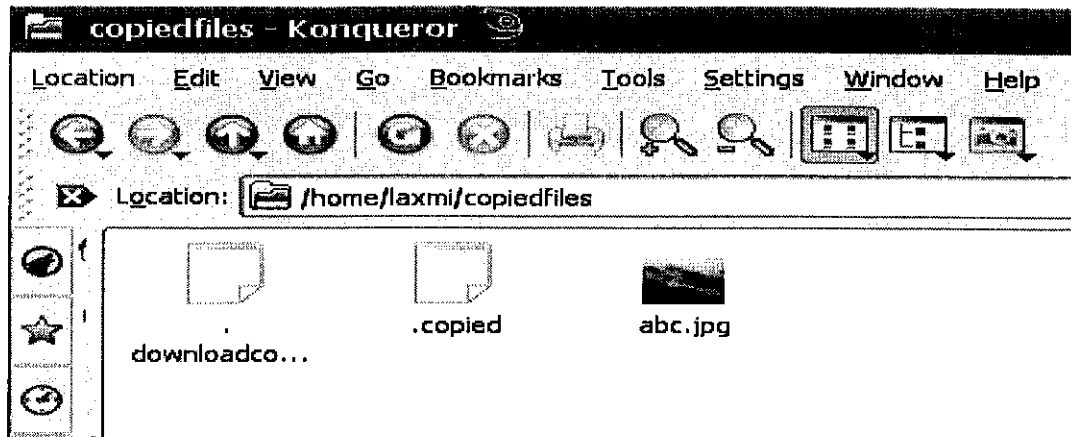


Fig.5.11 Snapshot of 'copiedfiles' folder, after the byte transmission of the file 'abc.jpg'.

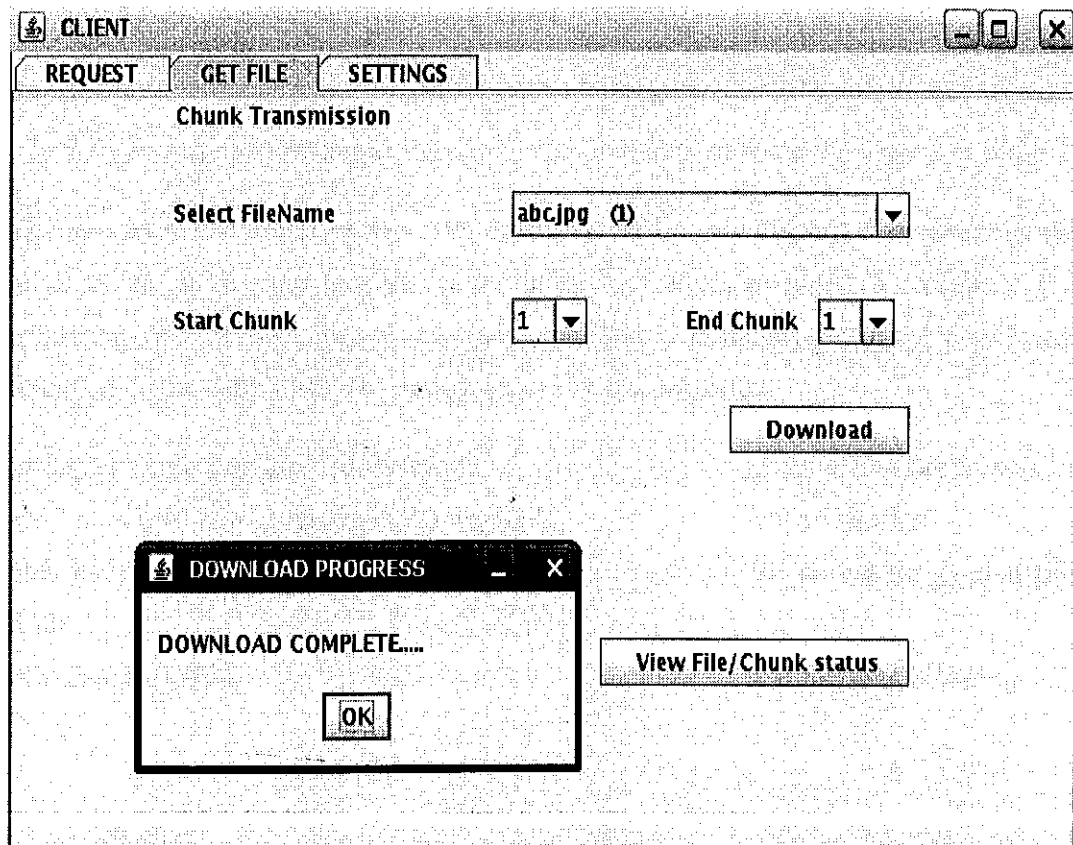


Fig.5.12 Snapshot displaying message after download has been complete.

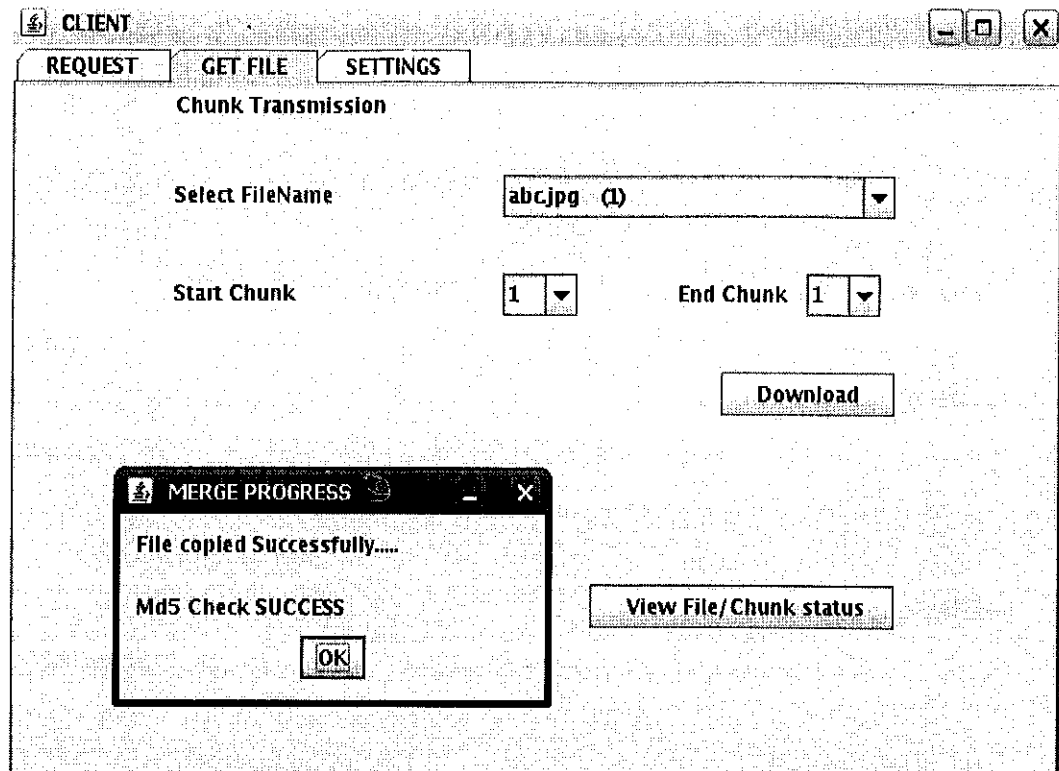


Fig 5.13 Snapshot showing the message, when the md5 of the copied file is validated.

The messages that are displayed after download of file, are displayed by scanning the 'copiedfiles' folder and looking for pertinent files after download is complete. The following code snippet implements the same. This is implemented in the concurrent thread after the code finds that download has been completed.

```
String a1=".merged",a2=".mergererror",           //initializing file names
a3=".ntmerge",a4=".copied",a5=".merging";         //to be checked.
String s3[]=f3.list();                          //scanning the 'copiedfiles
folder'
if(a5.equals(s3[i]))                             // if merging is taking
place.
{
    m=true;
    String p=cpath.concat("/") + a5;
    File f5=new File(p);
    f5.delete();                                //delete the file
}
```

```

        i=i+1;                                //adjust number of files,
        that are to be                        checked

        try
        {
            Thread.sleep(8000L);                //wait for 8 seconds

        before
        }                                        //further merge status is to
        checked.

        ...
        String sf[]=f3.list();
        for(i=0; i< sf.length;i++)                //scan the copiedfiles
        folder again
        {
            if(a1.equals(sf[i]))                //if successful merge
            {
                m=false;
                dialogmes = "File Merge SUCCESS.....\n \nMD5 Matched";

                p=cpath.concat("/")+"a1);        // show message
                File f4=new File(p);
                f4.delete();                        //delete file
                i=i+1;
                break;
            }
            if(a2.equals(sf[i]))                // if merge unsuccessful
            {
                m=false;
                dialogmes = "File Merge FAILURE.....\n \nFILE ENTRY
        REMOVED";

                p=cpath.concat("/")+"a2);
                File f4=new File(p);
                f4.delete();
                i=i+1;

```

```

        break;
    }
}
break;
}

else if(a3.equals(s3[i])) //if all chunks not available
{
    dialogmes = "File Cannot Be merged.....\n \nAll Chunks not available";
    String p=cpath.concat("/"+a3);
    File f4=new File(p);
    f4.delete();
    i=i+1;
    break;
}

else if(a4.equals(s3[i])) //if successful transmission of
small file.
{
    dialogmes = "File copied Successfully.....\n \nMd5 Check SUCCESS";
    String p=cpath.concat("/"+a4);
    File f4=new File(p);
    f4.delete();
    i=i+1;
    break;
}

if(!m)
    JOptionPane.showMessageDialog((Component)null,dialogmes,dititle,ditype;

//show appropriate message.

```

The five files that are created by the C client process at appropriate times, according to the status of chunks and download.

‘.merging’ – created when merging would take place on the client side.

‘.merged’ - created when chunks have been successfully merged to form original file.

‘.mergeerror’ – created when md5 of the merged file does not match with the original file, thereby indicating an error during merging.

‘.ntmerge’ – created when all chunks are not available of the file and thus merging cannot place.

‘.copied’ – formed in the case of byte transmission of file, after its md5 has been cross checked.

These files are checked to intimate the user through GUI, the status of the application and are then deleted at the same time. An appropriate dialog message is shown for each file made by the C client process.’

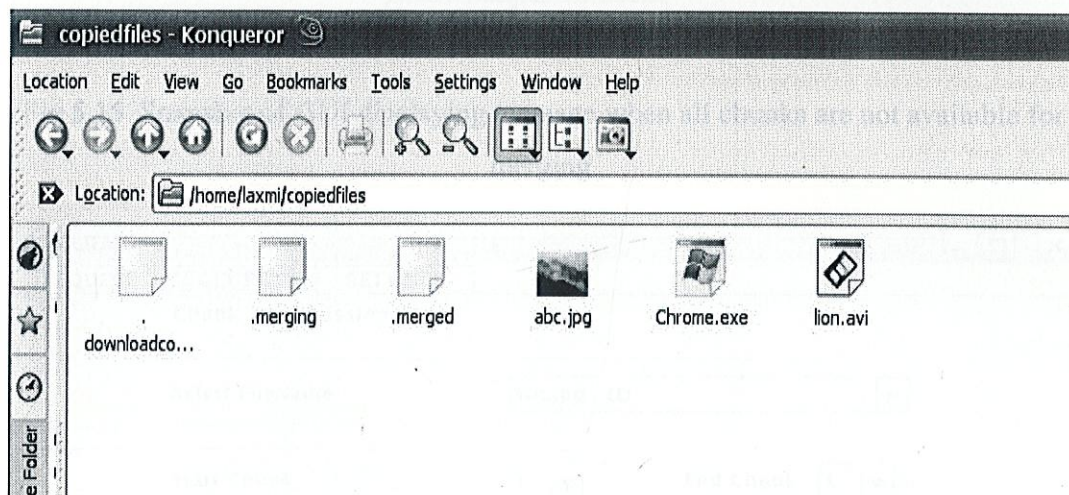


Fig 5.14 Snapshot of ‘copiedfiles’ folder when file has been merged successfully.

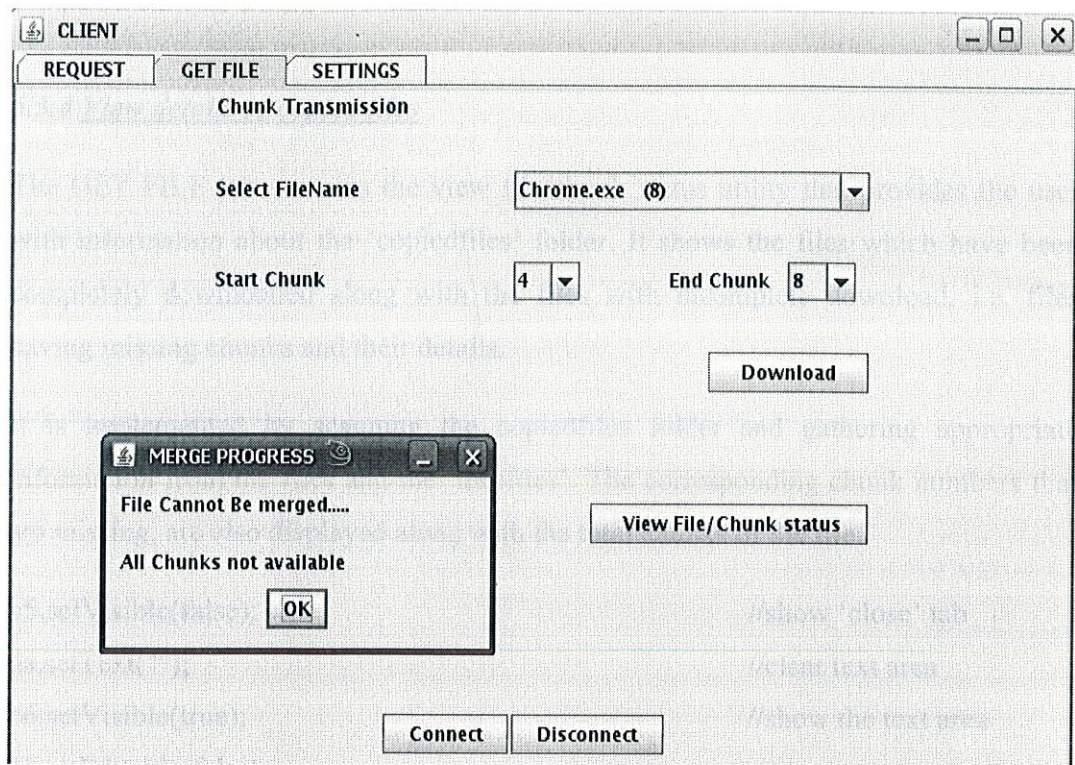


Fig 5.15 Snapshot of GUI displaying message when all chunks are not available for merging

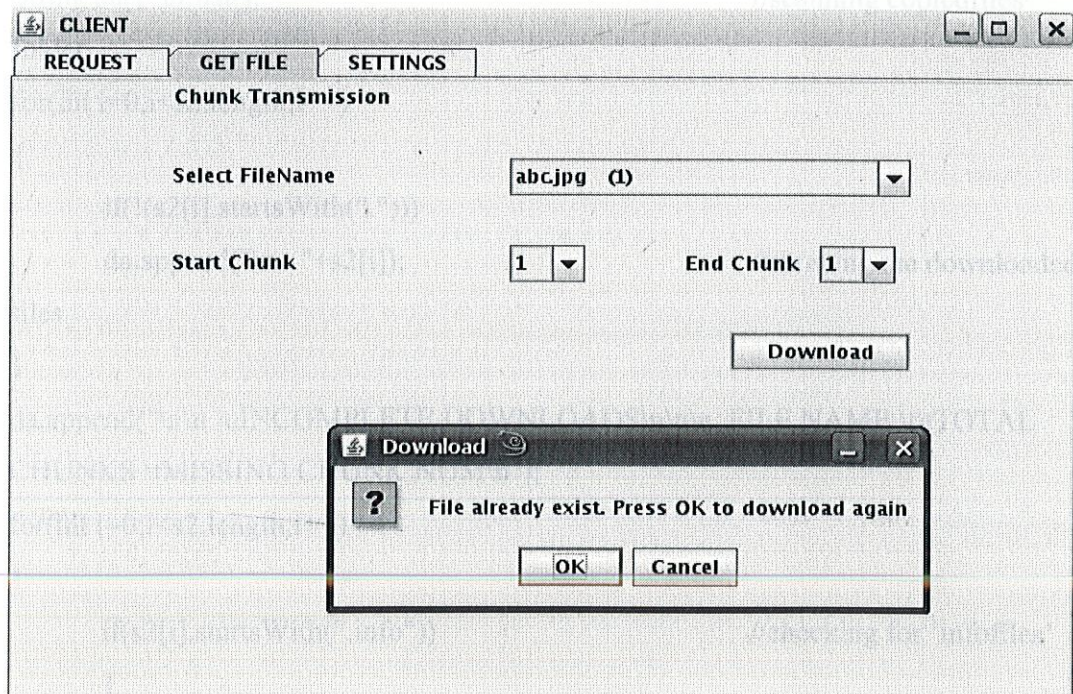


Fig 5.16 Snapshot of GUI awaiting user response when the file to be downloaded is already present.

5.3.4 View details of copied files

The GET FILE tab contains the view file/chunk status utility that provides the user with information about the 'copiedfiles' folder. It shows the files which have been completely downloaded along with the files with incomplete download, i.e. files having missing chunks and their details.

It is implemented by scanning the copiedfiles folder and gathering appropriate information from the files and the 'infofiles'. The corresponding chunk numbers that are missing, are also displayed along with the total chunks of the file.

```
b5.setVisible(false);                                //show 'close' tab
da.setText("");                                       //clear text area
b6.setVisible(true);                                  //show the text area
da.setEditable(false);
da.append("\n FILES DOWNLOADED");
File f3=new File(cipath);
String s2[]=f3.list();                               //scanning copiedfiles
folder
for(int i=0;i<s2.length;i++)
{
    if(!(s2[i].startsWith(".")))
        da.append("\n "+s2[i]);                     //showing the downloaded
files
}
da.append("\n\n \nINCOMPLETE DOWNLOADS\n\n\n FILE NAME \t\tTOTAL
CHUNKS \t\tMISSING CHUNK NUM\n");
for(int i=0;i<s2.length;i++)
{
    if(s2[i].startsWith(".info"))                   //checking for 'infofiles'
    {

        info=info.concat("/copiedfiles/");
        info=info.concat(s2[i]);
```



```

try{
    f = new FileReader(info);
    BufferedReader b=new BufferedReader(f);
    str = b.readLine();           //Reading info files
}
catch(Exception ex){}
da.append("\t\t "+str.length());
da.append("\t\t");
for(int j=0;j<str.length();j++)
{
    if(str.charAt(j)=='0';           //displaying missing
chunk numbers
    da.append((j+1)+" ");
}
}

```

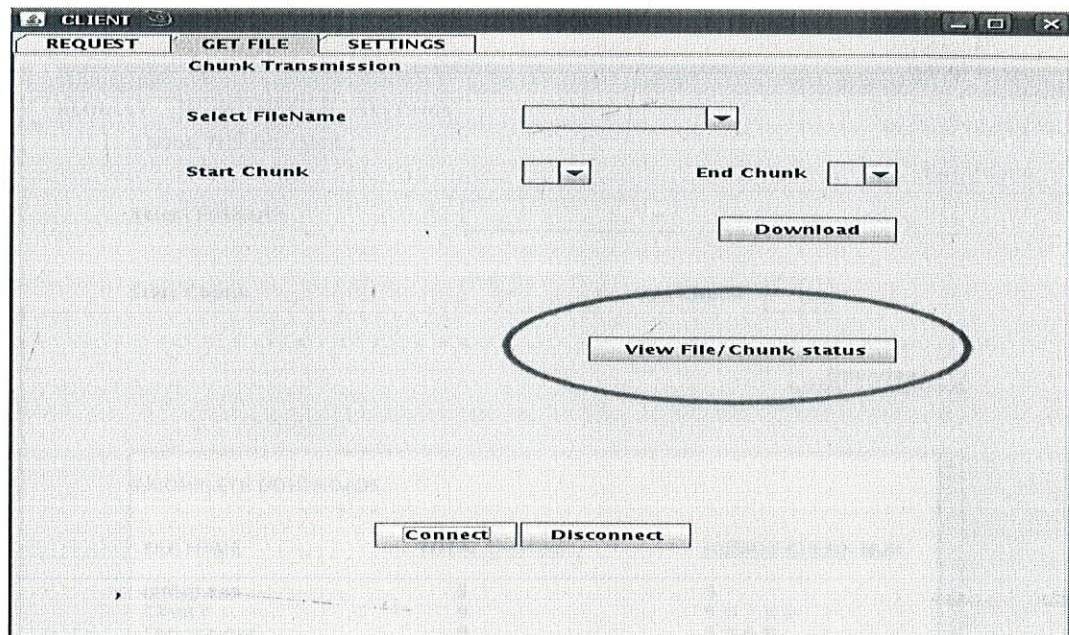


Fig.5.17 Snapshot of GUI highlighting the view File/Chunk Status button

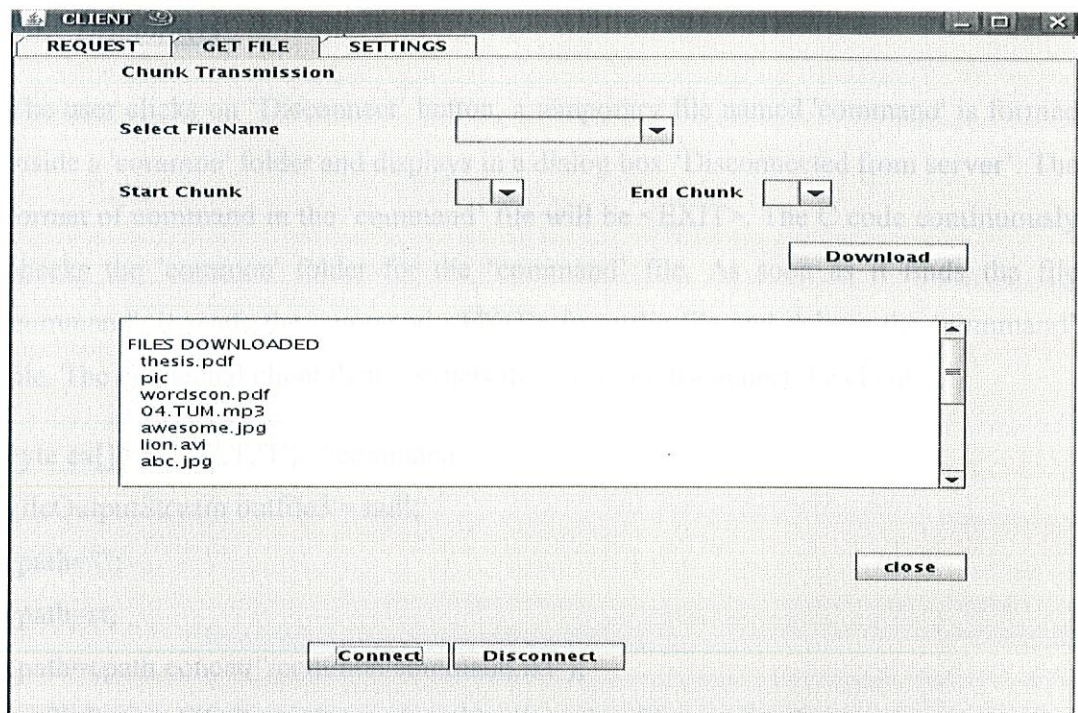


Fig.5.18 Snapshot of the GUI showing the files that have been downloaded.

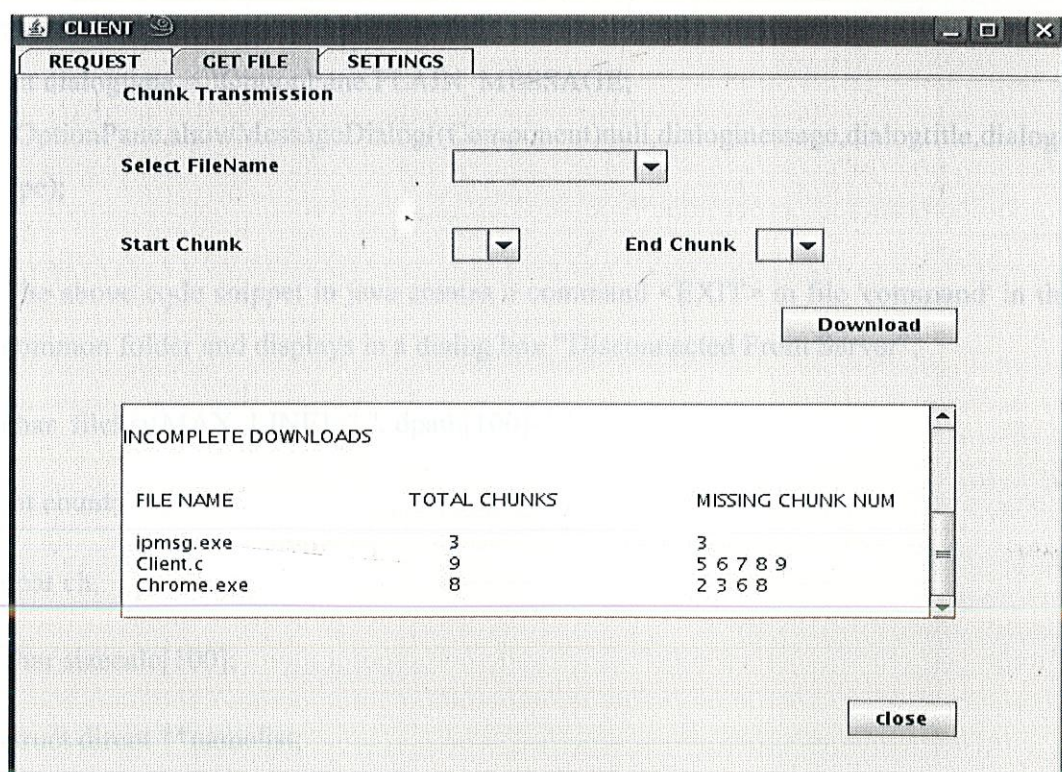


Fig.5.19 Snapshot of GUI showing incomplete files.

5.3.4 Disconnect

The user clicks on 'Disconnect' button, a temporary file named 'command' is formed inside a 'common' folder and displays in a dialog box 'Disconnected from server'. The format of command in the 'command' file will be <EXIT>. The C code continuously checks the 'common' folder for the 'command' file. As soon as it finds the file 'command', it reads the command <EXIT> from the file and deletes the 'command' file. The connected client then instructs the server to disconnect the client.

```
byte ex[]={'E','X','I','T'}; //command
FileOutputStream outfile3 = null;
cpath="";
cpath=st;
cpath=cpath.concat("/common/command.txt");
outfile3=new FileOutputStream(cpath); //creating file command
outfile3.write(ex); //writing the command to the file command
outfile3.close();
String dialogtitle = "CLIENT";
String dialogmessage = "DISCONNECTED FROM SERVER.....";
int dialogtype = JOptionPane.PLAIN_MESSAGE;
JOptionPane.showMessageDialog((Component)null,dialogmessage,dialogtitle,dialogt
ype);
```

The above code snippet in java creates a command <EXIT> in file 'command' in the common folder and displays in a dialog box "Disconnected From Server".

```
char filelist[MAX_LINE]=" ", dpath[100]="";

int count;

char ch;

char sizecalc[100];

struct dirent **namelist;

FILE *fp1,*fp2;
```



```

int n;

while(1)
{
    n = scandir(loc, &namelist, 0, alphasort);

    if(n>=3)
    {
        strcpy(dpath,shdir);    //shdir stores directory path as in
config_client.ini

        strcat(dpath,"/common/command.txt");

        if(fopen(dpath,"r")!=NULL) //checking for the command file
        {
            fp1=fopen(dpath,"r");

            sleep(1);

            ch = fgetc(fp1);

            while(ch!=EOF) //reading the contents of the command file
            {
                sizecalc[j] = ch;

                j++;

                ch = fgetc(fp1);
            }

            fclose(fp1);
        }

        if(!strcmp(sizecalc,"EXIT"))

```

```

    {
        count = write(SerSock,&sizecalc,100, 0); //writing command
on socket

        count = recv(SerSock, &filelist,MAX_LINE, 0);
        unlink (dpath); //deleting the command file
        break;
    }
}

```

The C code snippet reads command <EXIT> from 'command' file and directs the server to disconnect the client.

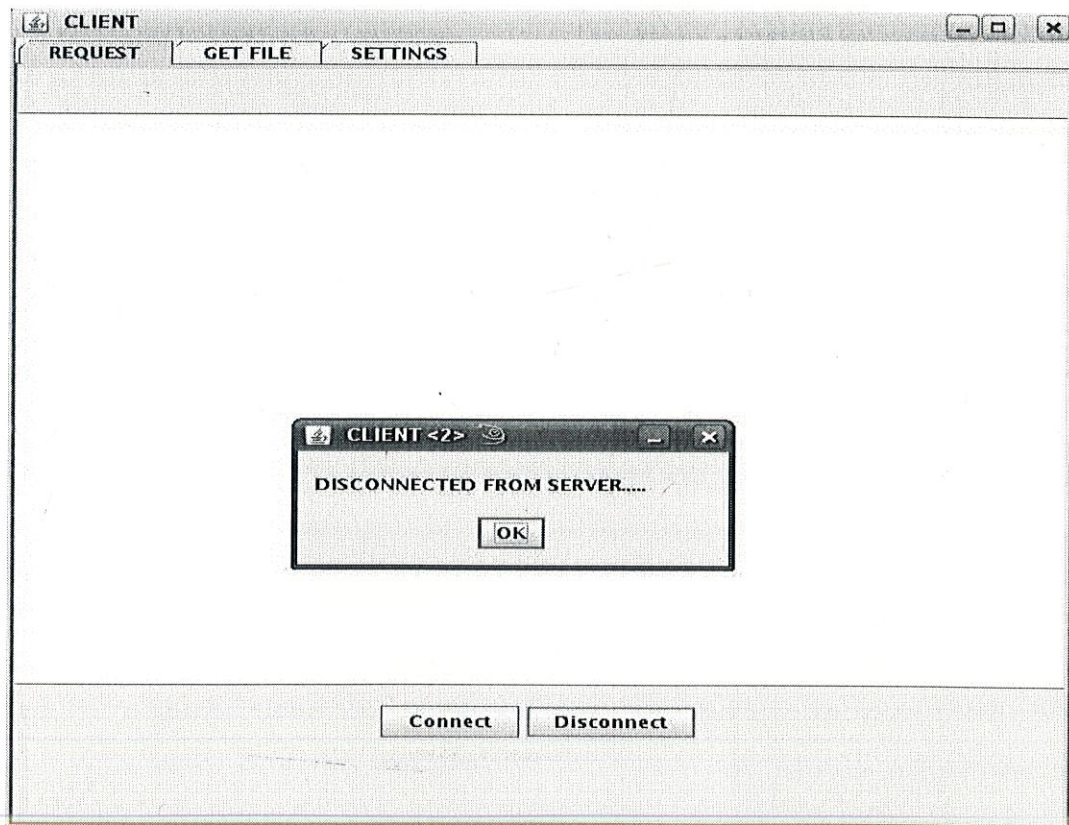


Fig.5.20 Screenshot displaying when the client is disconnected from the server.

5.4 Installation Steps for the java file

- Copy gui file in corresponding “root” folders of the specified LINUX operating system that is being used.
- Steps for compiling
 <Path to java bin>:\$PATH
 export PATH
 javac gui.java
- Steps for running
 java gui

CHAPTER 6

FAULT TOLERANCE

6.1 Introduction

During the client - server inter communication, the files being transferred over the network are susceptible to error in download due to loss in connectivity as files (chunks in this case) are being downloaded. As the transmission is taking place, a server may crash, server may become unresponsive or the connection may be lost with the server resulting in non-execution of user request or partial downloads leading to erroneous files being copied. Therefore, Fault Tolerance is a special feature that has been added to the application which takes care of the faults that may occur in the client server model. This feature has been added with the aim to ensure interrupt free download by continuing the download from the point where it was disrupted by establishing the connection with another server or the same server (depending on the availability of servers) via the tracker.

6.2 Description

In order to integrate fault tolerance features along with the file transfer system, the following three basic steps are imperative and have been intricately embedded into the basic transfer of chunks that takes place between the server and the client.

- a) *Fault detection*- It is the detection that a fault has occurred while file (or chunks) are being transmitted. The client needs to have knowledge that some error has occurred as the server has stopped responding or connection has been lost altogether. The client can take further measures to continue and complete the user request, notifying the user. This has been done by checking that expected amount and rate of data is being received from the server side.
- b) *Reconnection*- This step takes care that the client is reconnected to an available server (via the tracker), so that the further processing can take

place. This is done by exiting from the current client process and initiating a fresh client process that contacts the tracker first, establishing connection with the server whose address is received from the tracker.

- c) *Continue processing*- The main aim of fault tolerance is to continue processing from where it stopped and complete the user request. This step pertains to requesting the connected server to transmit chunks from where the fault occurred. This has been taken care by taking information the hidden processing files (explained earlier) and comparing them with the user request.

The implementation of the last two steps has been embedded in the JAVA(swing) code, while the detection of the fault is done by the current client process. The client program monitors the number of bytes received from the server and as soon as a mismatch is found between expected and received number of bytes, a file is made in the folder. Whenever the downloading process starts, a scheduler (timer) is initiated in the JAVA application. The fault tolerance protocol checks for the existence of the file. It collects information of the absent chunks and checks if the client is still active. If it is not active, the protocol restarts the client, reconnects to the server and sends the new command to the server for the downloading of remaining chunks. The reconnection with the appropriate available server is taken care by the tracker. The client is now reconnected to the server and the remaining commands executed. The application now behaves as in normal file transfer mode, i.e. the user can now request desired chunks and request catered by the new server connected and the user does not face any problems due to occurrence of fault.

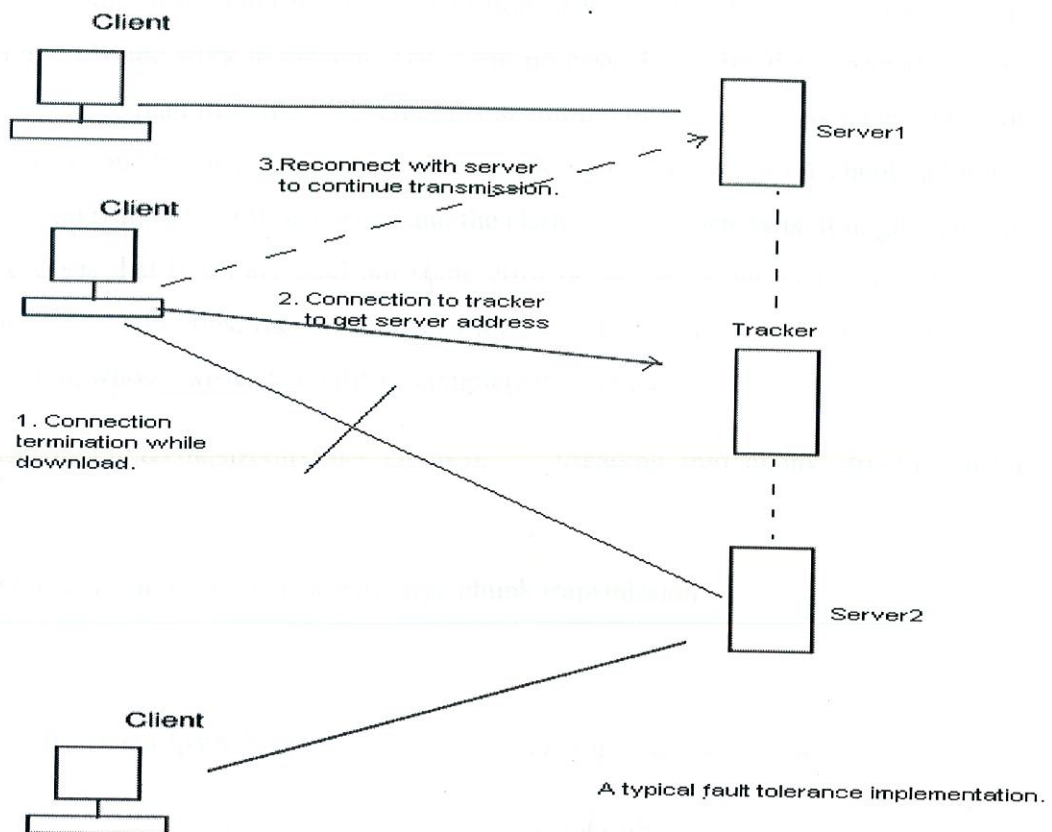


Fig.6.1 Steps involved in fault tolerance mechanism

The Fig.5.1 depicts the typical sequence of processing that occurs when there is an error in download due to connection problem or server unresponsiveness. In the example setup, the tracker has registered two servers-server1 and server2 and clients are connected to the servers. In the case when one of the clients loses connection with the server 2, it

establishes connection to the tracker to receive address of available server and thereby successfully connects to server 2 and continues its previous download by sending appropriate command to the connected server. The processing then proceeds normally, i.e. the user can further request downloading of chunks.

6.3 Working and Implementation details

6.3.1 Fault Detection

The first step in the fault tolerance, detection of the occurred fault is realized as the java and C code work in tandem. The client process checks for the expected number of bytes to be read from the socket that is transmitted by the server. As the client reads the chunks one by one, if the bytes written are less than the size of the chunk, a hidden file by the name of 'fault' is created and the client process then exits. It might occur in some cases that bytes are read but some error occurs when the last chunk is being created. To tackle this, the client checks for the existence of last chunk that the user requested, whose existence confirms complete download.

```
n=read(sockfd,&chk,sizeof(struct chunk));    //reading into chunk structure from
socket
```

```
if(n<65131)// (n<65001) in case of large chunk transmission
```

```
{
    ff=fopen(dpath,"w");                // creating file named 'fault' in
                                         copiedfiles folder.
    ...
    exit(-1);                          // exiting the client process
}
```

As implemented in the above client process code, the socket is being read and the number of bytes stored in the variable 'n' that is checked that it should be equal to size of chunk, which is 65131(header + content + md5) bytes in case of chunk transmission and 65001 bytes in case of large file transmission.

```
if(successtransfer(l,cmp,shdir))          //if request successfully
completed
{
    ...
    strcat(dpath,".downloadcomplete");    //creating 'downloadcomplete'
file.
```

```

        fp[12]=fopen(dpath,"w");
        ...
    }
else
{
    ...

    strcat(dpath,"/copiedfiles/.fault");    //creating 'fault' file.

    fp[12]=fopen(dpath,"w");

    exit(-1);

    ...
}

```

The code snippet implements that if request is successfully completed, which is checked by ensuring existence of last requested chunk being successfully copied, the '.downloadcomplete' file is created else 'fault' file indicating error in download.

In the java process, as soon as the download button is pressed, a timer is initiated that checks for the '.fault' file every 2 seconds. If the file is found, it means a fault has occurred during transmission and it proceeds to the next step. The timer can also be stopped in the case when the download successfully completes. This is known when the hidden file '.downloadcomplete' is found, which is being checked by a concurrent thread.

```

timer = new Timer(2000,actionListener);           //starting timer, interval 2
                                                    seconds.

timer.start();

```

The above code is implemented in the ActionListener of the 'Download' button that checks for '.fault' file every 2 seconds until it is found or the timer is stopped as the '.downloadcomplete' file is found. It is reinitiated every time the user requests for chunk download.

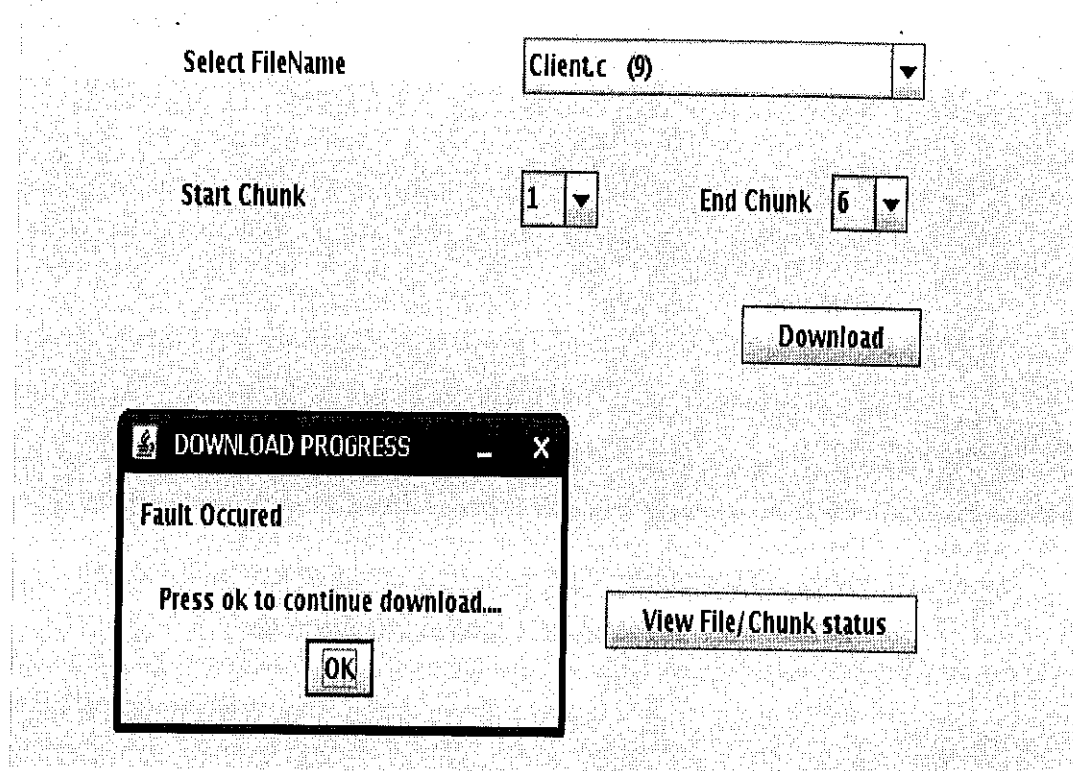


Fig.6.2 GUI Screenshot notifying the user that fault has occurred.

Server Side

```

WRITING CHUNK ON SOCKET
No. of bytes being written on socket : 35131
opening file to read chunk /home/Parul/sharedfolder/Client.c.H5 35000 374009

!!!DOWN OR FAULT OCCURED!!!

!!!Restarting Server!!!
CONNECTING TO TRACKER...
!!!Tracker Host Have Resolved!!!
!!!Connection to Tracker Socket is Successful!!!
!!!Server Address Sent!!! Listening at PORT : 4045
Shared Directory Path : /home/Parul/sharedfolder
Socket Created Successfully as 3
!!!Binding Socket with the Port!!!
!!!Listening for Client!!!

```

Fig.6.3 Snapshot of server side when fault occurs and the server is restarted automatically in case of chunk transmission.

Server Side

```
WRITING CHUNK ON SOCKET
1-65001
2-65001
3-65001
4-65001

!!!ERROR OR FAULT OCCURRED!!!

!!!RESTARTING SERVER!!!
CONNECTING TO TRACKER...
!!!Tracker Host Name Resolved!!!
!!!Connection To Tracker Socket 3 Successful!!!
!!!Server Address Sent!!! Listening at PORT : 4005
Shared Directory Path : /home/Parul/sharedfolder
Socket Created Successfully as 3
!!!Binding Socket with the Port!!!
!!!Listening for Client!!!
```

Fig.6.4 Snapshot of server side when fault occurs and the server is restarted automatically in case of large chunk transmission

6.3.2 Reconnection

This step establishes connection to the available server in the network that is registered with the tracker and the further requests are then handled by this connection. When the fault is detected, the previous client process is terminated and a fresh client process is started by executing the system command.

The java process now creates the file 'connect.txt' in the 'common' folder so that the C based client process connects to the tracker and finally to an available server.

The new server address that the client receives from the tracker depends on the server address it was previously connected to and on the servers registered with the tracker.

The client tries to establish connection, until it reconnects to an appropriate server and notifies the user.

```
for(i=0; i< s.length;i++)
{
    if(s[i].equals(".fault")) //The '.fault' file found
    {
        flag=true;
    }
}

if(flag)
```

```

{ ...

Process p = Runtime.getRuntime().exec("./checkcli");    //starting of new client
process

using the system command

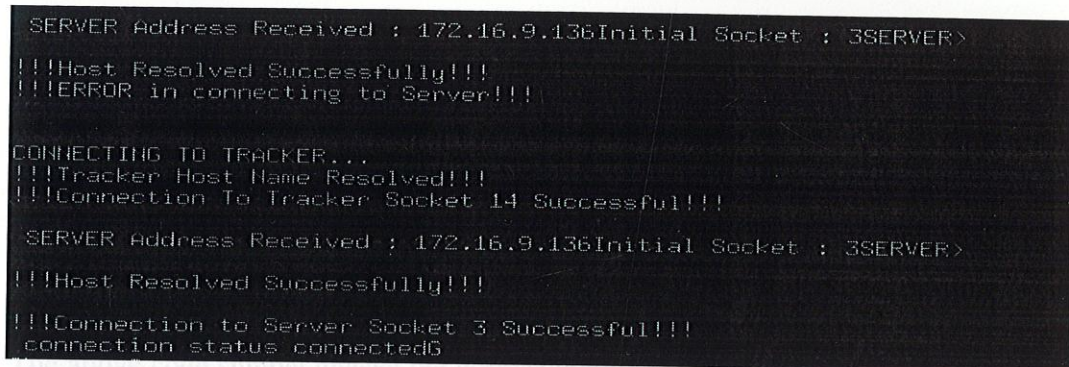
fopen("connect.txt");                                //Creating connect.txt file.

}

```

The above code snippet ensures that when fault has occurred, it starts a new client process and makes 'connect.txt' file in common folder which is checked for this file and then it automatically connects to the server via the tracker.

New Client Process



```

SERVER Address Received : 172.16.9.136Initial Socket : 3SERVER>
!!!Host Resolved Successfully!!!
!!!ERROR in connecting to Server!!!

CONNECTING TO TRACKER...
!!!Tracker Host Name Resolved!!!
!!!Connection To Tracker Socket 14 Successful!!!

SERVER Address Received : 172.16.9.136Initial Socket : 3SERVER>
!!!Host Resolved Successfully!!!
!!!Connection to Server Socket 3 Successful!!!
connection status connectedG

```

Fig.6.5 Snapshot of the new client process started, reconnect to the server after error in connecting to server once.

6.3.3 Continue processing

Fault tolerance feature involves that even after fault has occurred, the user request is not affected due to this and the download process continues from the point where it was stalled. This has been implemented through the help of hidden 'infofile' that is created for every file as it is being downloaded. This file is updated after every chunk has been copied to the client. Therefore, in order to find out where the download was interrupted, the corresponding 'infofile' is checked with the original request of chunks that user asked.

After collecting the required information, i.e. from which chunk number the file needs to be transmitted, the new command is formed and written in the 'command' file which is sent to the newly established connection and thus the request is completed.

```
for(int k=num1;k<=num2;k++,num1++)           // Checks the 'infofile' at chunk
                                              positions requested
{
    if(stri.charAt(k-1)=='0')                 //Checks for non-available chunk
                                              numbers
    {
        ff=false;
        break;
    }
}
```

The above code snippet checks the corresponding 'infofile' at the chunk numbers the user requested and the fault is found at the chunk where the chunk that was supposed to be copied is found missing. The command is then formed in the format as required:

GET <filename> <start_chunk> <end_chunk>

Here the start_chunk value is the chunk number where the fault is detected and end_chunk is the original end chunk number that the user requested before the fault occurred. This command thus formed is written in the file 'command.txt' in the 'common' folder by the java code from where the C client process sends it to the server thereby continuing the chunk download from the required point.

For example, if the original command given was: 'GET client.c 1 6', i.e. 1 to 6 chunks were requested for file client.c by the user and the fault occurred at chunk number 5. Therefore, at fifth position in the 'infoclient.c' file there would be a '0' instead of a

'1'. This indicates fault occurrence point and the chunks to be transmitted now would be from 5 to 6 and therefore command written in the 'command.txt' file is: 'GET client.c 5 6'.

Server side

```
Command Received: GET Client.c 1 6
calling filetransfer /home/Parul/sharedfolder
n = 3
the name extracted= Client.c
Found Client.c (null)
startchunk= 1
endchunk= 6
The total chunks of file are 9
COMMAND CHECK COMPLETE....
the file name=Client.c
preparing file Client.c (1,6)
command being sent to client
CHUNK TRANSFER OF FILE
```

Fig. 6.6 Snapshot of original command received from client to transmit 6 chunks of the file 'Client.c'

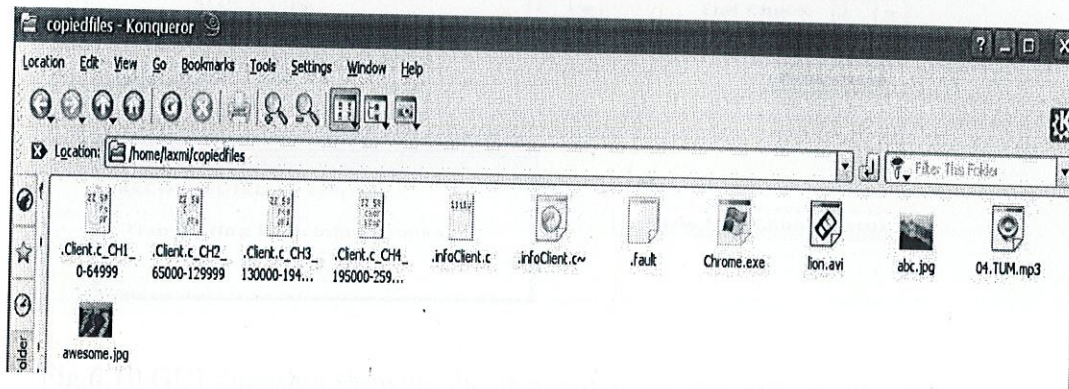


Fig.6.7 Snapshot of 'copiedfiles' folder showing only 4 chunks downloaded of file 'client.c' and the '.fault' file being made in the folder.

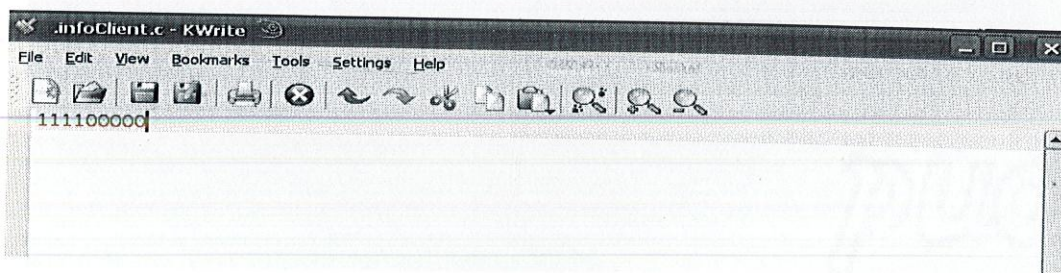


Fig.6.8 Snapshot of contents of 'infoclient' file showing that only 1-4 chunks have been copied.

Server Side

```
!!!Connection To Tracker Socket 3 Successful!!!  
!!!Server Address Sent!!! Listening at PORT : 4005  
Shared Directory Path : /home/Parul/sharedfolder  
Socket Created Successfully as 3  
!!!Binding Socket with the Port!!!  
!!!Listening for Client!!!  
!!!Listening Successful!!!  
chdir value msg sent /home/Parul/sharedfolder 4  
!!!Socket # 4, created Successfully!!!  
bytes read 40 value GET Client.c 5 6  
  
bytes read 40 value GET Client.c 5 6  
Command Received: GET Client.c 5 6
```

Fig.6.9 Snapshot of the command received from client to transmit the remaining chunks from 5 to 6 of the file 'Client.c' after error occurred at chunk number 5.

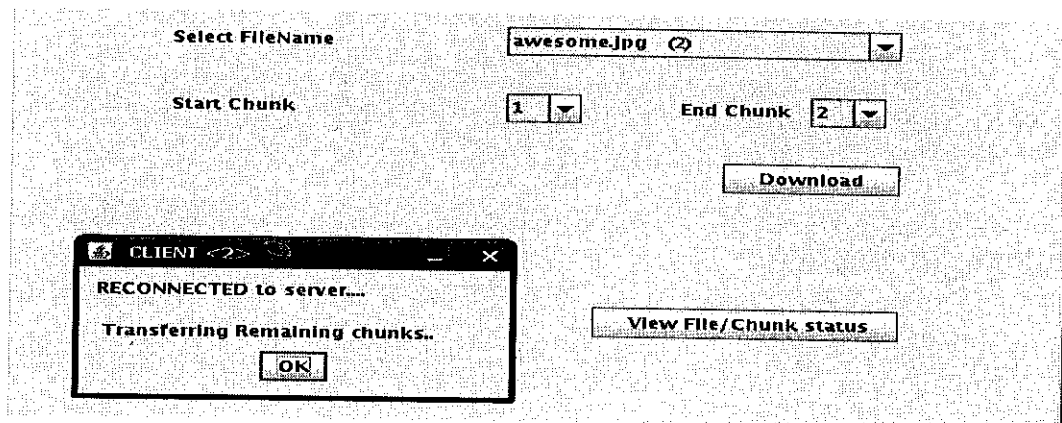


Fig.6.10 GUI Snapshot showing the message to the user that reconnection has been established and remaining chunks are being downloaded.

Server side

```
WRITING CHUNK ON SOCKET  
No. of bytes being written on socket : 65131  
opening file to read chunk /home/Parul/sharedfolder/.Client.c_043371007-319939  
sending filename to server Client.c_043371007-319939  
...  
No. of bytes being read from socket : 3  
client sent ch  
WRITING CHUNK ON SOCKET
```

Fig.6.11 Snapshot showing transmission of remaining chunks of file 'Client.c' from server side.

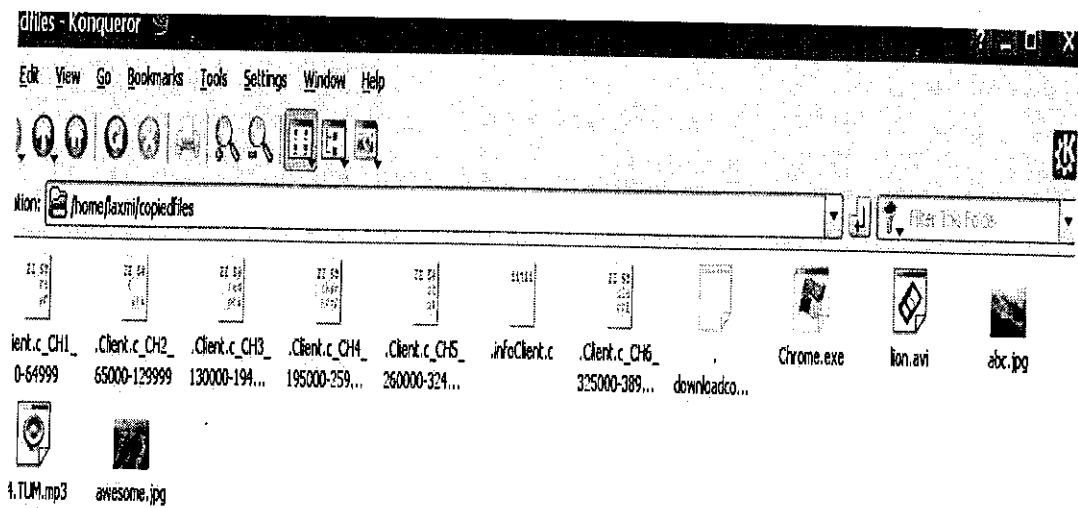


Fig.6.12 Snapshot of 'copiedfiles' folder showing all requested chunks (from 1-6) downloaded in the folder.

CHAPTER 7

CONCLUSION

Based on the results and discussions in presented in the thesis, the following major conclusions are drawn from the present Study:

1. The implementation presented in this work yield fairly good results, especially for downloading of files of any size in form of chunks and merging of these chunks on availability of all chunks of the file.
2. The fault tolerance and recovery feature yields good result especially for segmentation faults in servers, blocked file transfer in case of clients and connection to another server on occurrence of fault for completion of file download.
3. The GUI in java works well and is easy to use and understand. This eradicates the need of the user to be acquainted with the system and the commands and thus, can be used by anyone.
4. The system works for a single virtual LAN network i.e. between directly connected hosts.
5. The rechecking of files used in software bridge can create problem for longer durations of ideal time.
6. Better mutual exclusion is possible.

Scope for Future Work:

The file transfer speed can be improved further. The software bridge between GUI and C implementation can be created using JNI techniques, which is a better option. The system can be deployed on a shared server for increasing scalability by avoiding limitation of virtual LAN and can also be used on web using shared execution and storage space on servers providing this facility and a web interface.

BIBLIOGRAPHY

- Thesis on *Information Sharing Over Multicast Routers*. Abhishek Bhatiya, Kunal Krishna, Nakul Sharma. May 2007
- Online resources form GNU C LIBRARY
- Ralph Davis *Win32 Network Programming*. Addison Wesley Developer's Press. September 1996
- Bob Quinn and Dave Shute *Socket Network Programming*. Addison Wesley Publishing company 1995
- UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications, Prentice Hall, 1999.
- UNIX Network Programming, Volume 2, Second Edition: Networking APIs: Socket and XTI, Prentice Hall, 1998.
- TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols, Addison-Wesley, 1996.
- TCP/IP Illustrated, Volume 2: The Implementation, Addison-Wesley, 1995.
- TCP/IP Illustrated, Volume 1: The Protocols, Addison Wesley, 1994.
- Advanced Programming in the UNIX Enviroment, Addison-Wesley, 1992.
- UNIX Network Programming, Prentice Hall, 1990.