

CHAPTER 1

Introduction

1.1 Objective

In computing, a distributed file system or network file system is any file system that allows access to files from multiple hosts sharing via a computer network. This makes it possible for multiple users on multiple machines to share files and storage resources. The client nodes do not have direct access to the underlying storage block but interact over the network using a protocol. This makes it possible to restrict access to the file system depending on access to the file system depending on access lists or capabilities on both the servers and the clients, depending on how the protocol is designed.

In contrast, in a shared disk file system all nodes have equal access to the block storage where the file system is located. On these systems the access control must reside on the client. In this project, we will create a Distributed File System which works on Client - Server-Slaves Architecture.

As DFS is a file system, it's a huge project and requires enormous efforts and man hours, therefore due to time constraint of the project we will provide the basic functionalities in the project and implement them. And the basic objectives of the projects are following :-

- login to the file system
- logout from the file system
- opening existing files in the file system
- closing already opened files in the file system
- writing data to file system
- reading data from file system
- creating directory in file system
- removing directory from file system
- creating regular file in file system
- unlinking regular file from file system
- creating symbolic link to other file in file system

- changing ownership of the files in the file system
- changing the permissions of the files in the file system

Our objective is to provide a reusable interface of the file system for other developers as library so that it can be used by others in their applications, we will implement the file system as a set of functions which can be treated as file system calls of UNIX like OS. We will also provide a shell kind of environment for accessing the file system which uses the core library created in this project. Once the core functionalities are implemented, the project would be made as a community developed product by making it open source. It will be released under GNU GPL (GNU General Public License) version 3. So that it will respect the core freedoms of it's users.

The details of GNU GPL License version 3 are attached in APPENDIX I. It can also be found online at www.fsf.org (Free Software Foundation) website.

1.2 Scope of the project

Distributed File System Project can be used as a stand alone application for storing very large files of size in Tera-bytes over different small storage devices connected through the network. Or it can be used as a OS Module for File Management in Grid Computing OS. With the help of this project we don't need to restrict ourselves to the maximum size available of a storage device for storing an enormously large file. We can combine many small storage devices through DFS and can have feel of a single big chunk of storage space of single address space logically. Therefore, we can make use of the old hardware which we consider obsolete in today's high demanding computational world. As the project will be released under GNU GPL (GNU General Public License). Its scope becomes enormously large. The source code would be available online for the users and developers, so that it can be molded according to the users' needs and new features would be added through communal development.

1.3 About the Company

NXIT is a team of experienced professionals working in IT Infrastructure,

Server Management, Linux Kernel Internals, Application Development and Embedded System. NXIT attached with different companies and our performance has been so far very good. NXIT have been maintaining the servers, cluster and hardware of big corporate, small organizations. NXIT have blended their qualitative service with the latest technology to offer our customers a competitive edge.

NXIT provides training & solutions on Open Source to different clients. It help them to reduce their running system cost. To full fill the requirement of the market for the open source professional, They have started their own training modules. NXIT provides training in Linux Kernel, Device Driver, System Programming, Linux Application Development, Embedded system and Computer Networking.

1.4 Theoretical Background

In order to fully implement the DFS as a product, the knowledge and expertise of various tools and technologies is needed. 1) Basic UNIX Operating System Concepts, 2) Linux System Programming, 3) C Language Programming, 4) (Network Programming) Socket Programming, 5) Core Level File System Concepts, 6) Free (Open Source) Software Development Principles and Development Approach.

1.4.1 Basic UNIX Operating System Concepts -

UNIX is a powerful computer operating system originally developed at AT&T Bell Laboratories. It is very popular among the scientific, engineering, and academic communities due to its multi-user and multi-tasking environment, flexibility and portability, electronic mail and networking capabilities, and the numerous programming, text processing and scientific utilities available. It has also gained widespread acceptance in government and business. Over the years, two major forms (with several vendor's variants of each) of UNIX have evolved: AT&T UNIX System V and the University of California at Berkeley's Berkeley Software Distribution (BSD). This document will be based on the Sun OS 4.1.3_U1, Sun's combination of BSD UNIX (BSD versions 4.2 and 4.3) and System V because it is the primary version of UNIX available at Rice. Also available are Solaris, a System V- based version, and IRIX, used by Silicon Graphics machines.

UNIX in itself a vast topic to discuss, many books have been written to

explain its concepts and functionalities. Here we mention its six basic elements. They are: commands, files, directories, your environment, processes, and jobs. Commands are the instructions you give the system to tell it what to do. Files are collections of data that have been given filenames. A file is analogous to a container in which you can store documents, raw data, or programs. A single file might contain the text of a research project, statistical data, or an equation processing formula. Files are stored in directories. A directory is similar to a file cabinet drawer that contains many files. A directory can also contain other directories. Every directory has a name, like files. Your environment is a collection of items that describe or modify how your computing session will be carried out. It contains things such as where the commands are located and which printer to send your output to. A command or application running on the computer is called a process. The sequence of instructions given to the computer from the time you initiate a particular task until it ends it is called a job. A job may have one or more processes in it. We will explore each of these elements in a little greater detail later on, but first you need to learn how to initiate a session on a Unix machine.

1.4.2 GNU/Linux System Programming

Linux is a freely distributed implementation of a UNIX-like kernel, the low-level core of an operating system. Because Linux takes the UNIX system as its inspiration, Linux and UNIX programs are very similar. In fact, almost all programs written for UNIX can be compiled and run on Linux. Also, some commercial applications sold for commercial versions of UNIX can run unchanged in binary form on Linux systems. Linux was developed by Linus Torvalds at the University of Helsinki, with the help of UNIX programmers from across the Internet. It began as a hobby inspired by Andy Tanenbaum's Minix, a small UNIX-like system, but has grown to become a complete system in its own right. The intention is that the Linux kernel will not incorporate proprietary code but will contain nothing but freely distributable code. Versions of Linux are now available for a wide variety of computer systems using many different types of CPUs, including PCs based on Intel x86 and compatible processors; workstations and servers using Sun SPARC, IBM PowerPC, and Intel Itanium; and even some handheld PDAs and the Sony Playstation 2. If it's

got a processor, someone somewhere is trying to get Linux running on it!

Linux owes its existence to the cooperative efforts of a large number of people. The operating system kernel itself forms only a small part of a usable development system. Commercial UNIX systems traditionally come bundled with applications that provide system services and tools. For Linux systems, these additional programs have been written by many different programmers and have been freely contributed.

The Linux community (together with others) supports the concept of free software, that is, software that is free from restrictions, subject to the GNU General Public License. Although there may be a cost involved in obtaining the software, it can thereafter be used in any way desired and is usually distributed in source form.

The Free Software Foundation was set up by Richard Stallman, the author of GNU Emacs, one of the best-known text editors for UNIX and other systems. Stallman is a pioneer of the free software concept and started the GNU Project (the name GNU stands for GNU's Not Unix), an attempt to create an operating system and development environment that would be compatible with UNIX but not suffer the restrictions of the proprietary UNIX name and source code. GNU may turn out to be very different from UNIX at the lowest level but still supports UNIX applications.

The GNU Project has already provided the software community with many applications that closely mimic those found on UNIX systems. All these programs, so-called GNU software, are distributed under the terms of the GNU General Public License (GPL), a copy of that may be found at <http://www.gnu.org> or <http://www.fsf.org>. This license embodies the concept of copyleft (a takeoff on "copyright"). Copyleft is intended to prevent others from placing restrictions on the use of free software.

A few major examples of software from the GNU Project distributed under the GPL follow:

- GCC: The GNU Compiler Collection, containing the GNU C compiler
- G++: A C++ compiler, included as part of GCC
- GDB: A source code-level debugger
- GNU make: A version of UNIX make
- Bison: A parser generator compatible with UNIX yacc

- bash: A command shell
- GNU Emacs: A text editor and environment

There is now so much free software available that with the addition of the Linux kernel it could be said that the goal of creating GNU, a free UNIX-like system, has been achieved with Linux. To recognize the contribution made by GNU software, many people now refer to Linux systems in general as GNU/Linux.

1.4.3 C Language Programming

C was invented and first implemented by Dennis Ritchie on a DEC PDP-11 that used the UNIX operating system. C is the result of a development process that started with an older language called BCPL. BCPL was developed by Martin Richards, and it influenced a language called B, which was invented by Ken Thompson. B led to the development of C in the 1970s.

C is often called a middle-level computer language. This does not mean that C is less powerful, harder to use, or less developed than a high-level language such as BASIC or Pascal, nor does it imply that C has the cumbersome nature of assembly language (and its associated troubles). Rather, C is thought of as a middle-level language because it combines the best elements of high-level languages with the control and flexibility of assembly language. Table 1-1 shows how C fits into the spectrum of computer languages.

Surprisingly, not all computer programming languages were for programmers. Consider the classic examples of nonprogrammer languages, COBOL and BASIC. COBOL was designed not to better the programmer's lot, not to improve the reliability of the code produced, and not even to improve the speed with which code can be written. Rather, COBOL was designed, in part, to enable non-programmers to read and presumably (however unlikely) to understand the program. BASIC was created essentially to allow non-programmers to program a computer to solve relatively simple problems.

In contrast, C was created, influenced, and field-tested by working programmers. The end result is that C gives the programmer what the programmer wants: few restrictions, few complaints, block structure, stand-alone functions, and a compact set of keywords. By using C, you can nearly achieve the efficiency of

assembly code combined with the structure of Pascal or Modula-2. It is no wonder that C has become the universal language of programmers around the world.

The fact that C can often be used in place of assembly language was a major factor in its initial success. Assembly language uses a symbolic representation of the actual binary code that the computer executes directly. Each assembly-language operation maps into a single task for the computer to perform. Although assembly language gives programmers the potential to accomplish tasks with maximum flexibility and efficiency, it is notoriously difficult to work with when developing and debugging a program. Furthermore, since assembly language is unstructured, the final program tends to be spaghetti code— a tangled mess of jumps, calls, and indexes. This lack of structure makes assembly-language programs difficult to read, enhance, and maintain. Perhaps more important, assembly-language routines are not portable between machines with different CPUs.

Initially, C was used for systems programming. A systems program forms a portion of the operating system of the computer or its support utilities, such as editors, compilers, linkers, and the like. As C grew in popularity, many programmers began to use it to program all tasks because of its portability and efficiency— and because they liked it! At the time of its creation, C was a much longed-for, dramatic improvement in programming languages. In the years that have since elapsed, C has proven that it is up to any task.

1.4.4 Socket Programming (Network Programming)

Socket Programming is also termed as Network Programming is way of programming in which two or more machines are made to work together by sharing data between them over the network. Sockets are main building blocks of Network Programming. A socket is a communication mechanism that allows client/server systems to be developed either locally, on a single machine, or across networks. Linux functions such as printing and network utilities such as rlogin and ftp usually use sockets to communicate.

Sockets are created and used differently from pipes because they make a clear distinction between client and server. The socket mechanism can implement multiple clients attached to a single server.

You can think of socket connections as telephone calls into a busy building. A call comes into an organization and is answered by a receptionist who puts the call through to the correct department (the server process) and from there to the right person (the server socket). Each incoming call (client) is routed to an appropriate end point and the intervening operators are free to deal with further calls. Before you look at the way socket connections are established in Linux systems, you need to understand how they operate for socket applications that maintain a connection.

First of all, a server application creates a socket, which like a file descriptor is a resource assigned to the server process and that process alone. The server creates it using the system call `socket`, and it can't be shared with other processes.

Next, the server process gives the socket a name. Local sockets are given a filename in the Linux file system, often to be found in `/tmp` or `/usr/tmp`. For network sockets, the filename will be a service identifier (port number/access point) relevant to the particular network to which the clients can connect. This identifier allows Linux to route incoming connections specifying a particular port number to the correct server process. A socket is named using the system call `bind`. The server process then waits for a client to connect to the named socket. The system call, `listen`, creates a queue for incoming connections. The server can accept them using the system call `accept`.

When the server calls `accept`, a new socket is created that is distinct from the named socket. This new socket is used solely for communication with this particular client. The named socket remains for further connections from other clients. If the server is written appropriately, it can take advantage of multiple connections. For a simple server, further clients wait on the `listen` queue until the server is ready again.

The client side of a socket-based system is more straightforward. The client creates an unnamed socket by calling `socket`. It then calls `connect` to establish a connection with the server by using the server's named socket as an address. Once established, sockets can be used like low-level file descriptors, providing two-way data communications.

1.4.5 Core Level File System Concepts

Basic File I/O is the core of Linux Operating System. GNU/Linux treats everything in the system as a file. Therefore, In order to access any data or process

info, or any device etc. , they all be treated as files and accessed through their inode entries. Before a file can be read from or written to, it must be opened. The kernel maintains a per- process list of open files, called the file table. This table is indexed via nonnegative integers known as file descriptors (often abbreviated `fds`). Each entry in the list contains information about an open file, including a pointer to an in-memory copy of the file's backing inode and associated metadata, such as the file position and access modes. Both user space and kernel space use file descriptors as unique per-process cookies. Opening a file returns a file descriptor, while subsequent operations (reading, writing, and so on) take the file descriptor as their primary argument.

By default, a child process receives a copy of its parent's file table. The list of open files and their access modes, current file positions, and so on, are the same, but a change in one process—say, the child closing a file—does not affect the other process' file table.

File descriptors are represented by the C `int` type. Not using a special type `fd_t`, say is often considered odd, but is, historically, the Unix way. Each Linux process has a maximum number of files that it may open. File descriptors start at 0, and go up to one less than this maximum value. By default, the maximum is 1,024, but it can be configured as high as 1,048,576. Because negative values are not legal file descriptors, -1 is often used to indicate an error from a function that would otherwise return a valid file descriptor.

Note that file descriptors can reference more than just regular files. They are used for accessing device files and pipes, directories and `futexes`, `FIFOs`, and sockets—following the everything-is-a-file philosophy, just about anything you can read or write is accessible via a file descriptor.

We can access and control files and devices using a small number of functions. These functions, known as system calls, are provided by UNIX (and Linux) directly, and are the interface to the operating system itself. At the heart of the operating system, the kernel, are a number of device drivers. These are a collection of low-level interfaces for controlling system hardware. Similarly, a low-level hard disk device driver will only write whole numbers of disk sectors at a time, but will be able to access any desired disk block directly, because the disk is a random access device.

To provide a similar interface, device drivers encapsulate all of the hardware-dependent features. Idiosyncratic features of the hardware are usually available through `ioctl`.

Device files in `/dev` are used in the same way; they can be opened, read, written, and closed. For example, the same open call used to access a regular file is used to access a user terminal, a printer, or a tape drive.

The low-level functions used to access the device drivers, the system calls, include:

- `open`: Open a file or device
- `read`: Read from an open file or device
- `write`: Write to a file or device
- `close`: Close the file or device
- `ioctl`: Pass control information to a device driver

The `ioctl` system call is used to provide some necessary hardware-specific control (as opposed to regular input and output), so its use varies from device to device. For example, a call to `ioctl` can be used to rewind a tape drive or set the flow control characteristics of a serial port. For this reason, `ioctl` isn't necessarily portable from machine to machine. In addition, each driver defines its own set of `ioctl` commands.

These and other system calls are usually documented in section 2 of the manual pages. Prototypes providing the parameter lists and function return types for system calls, and associated `#defines` of constants, are provided in include files. The particular ones required for each system call will be included with the descriptions of individual calls.

1.4.6 Free (Open Source) Software Development Principles and Methods

Free software means software that respects users' freedom and community. Roughly, the users have the freedom to run, copy, distribute, study, change and improve the software. With these freedoms, the users (both individually and collectively) control the program and what it does for them.

When users don't control the program, the program controls the users. The developer controls the program, and through it controls the users. This nonfree or "proprietary" program is therefore an instrument of unjust power.

Thus, “free software” is a matter of liberty, not price. To understand the concept, you should think of “free” as in “free speech,” not as in “free beer”.

A program is free software if the program's users have the four essential freedoms:

- The freedom to run the program, for any purpose (freedom 0).
- The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

A program is free software if users have all of these freedoms. Thus, you should be free to redistribute copies, either with or without modifications, either gratis or charging a fee for distribution, to anyone anywhere. Being free to do these things means (among other things) that you do not have to ask or pay for permission to do so.

You should also have the freedom to make modifications and use them privately in your own work or play, without even mentioning that they exist. If you do publish your changes, you should not be required to notify anyone in particular, or in any particular way. More information can be found at www.fsf.org website. The free software movement is one of the most successful social movements to emerge in the past 25 years, driven by a worldwide community of ethical programmers dedicated to the cause of freedom and sharing.

1.5 Problem Definition

In today's world for engineering and scientific tasks we daily come across very high computations and need to deal with a very large data sets. For which we require enormous amount of computing units and storage units. For scientific (DNA related, Quantum Physics and etc.) calculations and representations we require storage spaces

in Tera-bytes. For which we require very costly Hardware to provide that much space. As a result problem arises to get that much space. In this project we are going to deal with this situation and create a kind of storage by joining many different small storage devices to form a single big chunk of storage which can be used as a single storage space. Here we deal with the problem of getting the enormous data storage by utilizing the contribution small systems to form a single big storage space.

Therefore, We are going to design a kind of File System in which we will combine the address space of different machines to form a single big chunk of address space to facilitate the storage for such large data chunks. We name that file system as DFS which stands for Distributed File System because it is distributed over the network between different machines.

CHAPTER 2

Requirement Analysis

2.1 Existing System

The Existing Distributed File system which is available is NFS(Network File System), which is reasonably slow because of its implementation in java which runs on a java virtual machine, moreover it doesn't support the files bigger than the available single hard drive because it doesn't show all the available networked space in a single address space. NFS is mostly used for sharing and storing data over a network. And the project we proposed will provide all the available networked devices in a single address space, which can provide enough space to store enormously large files.

2.2 Feasibility Study

Feasibility is the determination of whether or not a project is worth doing. A feasibility study is an analysis of a problem to determine if it can be solved effectively in the given budgetary, operational, technical and schedule constraints in place. The results of the feasibility study determine which, if any, of a number of feasible solutions to be developed in the design phase. The aim of the feasibility study is to identify the best solution under the circumstances by identifying the effects of this solution on the organization. The contents and recommendations of such a study will be used as a sound basis for deciding whether to proceed, postpone or cancel the project.

2.3 Technical Feasibility

This is concerned with specifying details and software that will successfully satisfy the user requirements. The technical feasibility addresses the questions. Is the technology that is required to develop and use the project is commercially available?

As the company already using other project of the similar type there is no requirement to purchase new equipment or work stations for this project exclusively. If it is not so also the project is technically feasible, as all the hardware that is required to implement the project is commercially available.

Software requirements for this project at various abstractions are as follows:

- Linux Kernel (Core of GNU/Linux Distribution OS).
- GCC: The GNU Compiler Collection, containing the GNU C compiler
- GDB: A source code-level debugger
- GNU make: A version of UNIX make
- bash: A command shell
- Gedit: A text editor with X-windows capabilities
- Vim: A text editor supported by text mode

As all the requirements are already available it is not required to invest anything in addition for the software.

2.4 Economic Feasibility

Economic feasibility is the most frequently used technique for evaluating the effectiveness of the proposed system. More commonly known as Cost/Benefit analysis. Since cost plays quite an important role in deciding the new system, it must be identified and estimated properly. Costs vary by type and consist of various distinct elements. Benefits are also of different type and can be grouped on the basis of advantages they provide to the management.

The proposed system will have its unique functionality is based on distributed computing. It has promising functionality which is desired by the heavy work of today's scientific works to store and access very large chunks of data sets. It won't require costly HW and can be implemented on the easily available resources, which improves the performance of the organization as a whole. The proposed system can be easily operable with the existing human force; only they need to be familiarized with the present system. The proposed system will provide the same kind of interfaces which are available for the present file systems, therefore it won't desire any kind of specific training for its users, they can make use of this project in the same way.

Also with the existing hardware and software are all already available, nothing more is required to be invested for developing the proposed system. Hence it is economically feasible for taking up the project.

2.5 Operational Feasibility

Proposed project is beneficial only if it can be turned into information system that will meet the operating requirements of the organization. As already client machines are available in the organization for other purpose, switching over to the proposed system from the existing system will be possible without any resistance from the users. User interface screens are also made such that the layouts are familiar with the users, and at the end of the project implementation user training is provided, so that they need not hire another employee for the operating of this system. Hence it is operationally feasible for taking up this project. Operational feasibility mainly addressed the following questions:

What new skills will be required? Do the existing staff members have these skills?

As the existing staff of this organization has the knowledge of computers, no need to make them to learn new skills. The project is developed to allow the user to interact with through it's specific shell which has the same kind of interface as other Linux shells. And also most of the commands are similar to the already available commands. So, the end user can quickly learn how to use this distributed file system.

2.6 Requirement Specification Document

2.6.1 Introduction to SRS

Software requirement specification (SRS) is the starting point of the software development activity. Little importance was given to this phase in the early days of software development. The emphasis was first on coding and then shifted to design.

As systems grew more complex, it became evident that the goals of the entire system cannot be easily comprehended. Hence the need for the requirement analysis phase arose. Now, for large software systems, requirements analysis is perhaps the most difficult activity and also the most error prone.

Some of the difficulty is due to the scope of this phase. The software project is initiated by the client's needs. In the beginning these needs are in the minds of various people in the client organization. The requirement analyst has to identify the requirements by talking to these people and understanding their needs. In situations

where the software is to automate a currently manual process, most of the needs can be understood by observing the current practice.

The SRS is a means of translating the ideas in the minds of the clients (the input), into formal document (the output of the requirements phase). Thus, the output of the phase is a set of formally specified requirements, which hopefully are complete and consistent, while the input has none of these properties.

2.6.2 General Discription of Project

Functions - The purposed Distributed File System will provide the basic function calls in the same way as other file system calls are provided by the GNU/Linux OS. The end use just have to get the libraries created in this project and can make use the following functions for their own purposes:-

1. dfs_open(path, flags, mode) -

This function can be used to open an existing file on the DFS. It will take three arguments.

- First, the path to the file which the user want to open.
- Seconds, the flags to be provided while opening the file (Eg. Read only, Write only etc)
- Third, the mode of the file to be opened. (Eg. The permissions to be given to file)
- This function will return the file descriptor as integer value which can be used to access the data of the file for writing or reading.

2. dfs_write(fd, data_buffer, no of bytes)-

This function will be used to write data to the file which has been opened for writing. It will take three arguments.

- First argument is fd which is an integer value returned by dfs_open() for the given file.
- Second is the data_buffer, which is a data stream consists of various bytes.
- Third is the no of bytes to be written to file from the data_buffer.
- It will return the no of successfully written bytes to the file.

3. dfs_read(fd, data_buffer, no of bytes)-

This function will be used to read data from the file which has been opened for reading. It will take three arguments.

- First argument is fd which is an integer value returned by dfs_open() for the given file.
- Second is the data_buffer, which is a bytes array to collect the data.
- Third is the no of bytes to be copied from file to the data_buffer.
- It will return the no of successfully read bytes to the file.

4. dfs_close(fd)-

This function will close the already opened file and make sure all the data of the file has been updated and written. It will take only one argument.

- The fd, an integer value of the opened file.
- It will return 0 on success and -1 on failure and set the ERROR value accordingly.

5. dfs_creat(path, mode)-

This function will create a new file on the path given as argument. It requires two arguments.

- First, the path of the file to be created.
- Second, the mode set of the file to be created.
- It will return fd as integer value which is used for accessing that newly created file.

6. dfs_lseek(fd, offset, reference)-

This function will change the offset position of the opened file, so that the next read or write will be performed on the given offset. It requires three arguments.

- First, fd is the integer value of the opened file.
- Second, the offset, the position in terms of bytes with reference to the third argument..

- Third, reference, It can be the current position, start of file or the end of file.
- It will return the newly set offset of the file with reference to the start of the file.

7. dfs_chdir(path)-

This function will change the current working directory of the environment. It will take only one argument.

- The path of the new directory on which we want to move.
- It will return 0 on success or -1 on error and set the ERROR value accordingly.

8. dfs_chmod(path, mode)-

This function will set the permissions of the given file on the given path. It requires only two arguments.

- First, the path of the file of which the permissions needed to be changed.
- Second, the mode value which states the permissions of the file.
- It will return 0 on success or -1 on error and set the ERROR value accordingly.

9. dfs_chown(path, userid, groupid)-

This function is used to change the ownership of the file to given user and group. It will take three arguments.

- First, the path of the file whose ownership needed to be changed.
- Second, is the userid of the new owner
- Third, is the groupid of the group of the new owner.
- It will return 0 on success or -1 on error and set the ERROR value accordingly.

10. dfs_link(old path, new path)-

This function will create a symbolic link of the file which exists on the old path and make entry for the new path. It will require only two arguments.

- First, the old path of the existing file.
- Second, the link's path of the created symbolic link to the existing file.
- It will return 0 on success or -1 on error and set the ERROR value accordingly.

11. dfs_unlink(path)-

This function will be used to remove the entry of the file from the directory and reduce the link count of the respective inode of the content. It will take only one argument.

- The path of the file which needed to be unlinked.
- It will return 0 on success or -1 on error and set the ERROR value accordingly.

12. dfs_getcwd(data_buffer, size of the buffer)-

This function will return the current working directory of the environment and copy that path into the argument supplied. It will take two arguments.

- The character array which is used to store the path of the current working directory.
- The size of the character array, so that it won't write on the location which is not the part of the given character array.
- It will return a pointer to the character array which contains the current working path of the environment.

13. dfs_rmdir(path)-

This function will be used to delete the empty directory. If directory contains any element, it will return failure otherwise success. It will take only one argument.

- The path of the directory to be deleted.
- It will return 0 on success or -1 on error and set the ERROR value accordingly.

2.6.3 Performance Constraints

Throughput and response time of our system is relatively dependent on the speed of network because the whole system is distributed over the network. Therefore the speed of the network creates a limit for the access speed of the DFS.

2.7 Certain Specific Requirements

The certain Requirements for the projects are of type Hardware-dependent as well as Software-dependent which are following:

2.7.1 Software Requirements

The DFS will make use of the system calls provided by the GNU/Linux Operating System. It's a system software, as a result it won't require any kind of API. All it needs is just the system calls which are the core functionalities provided by the Operating System. It will have basic three modules 1) Client 2) Server 3) Slaves. All the three modules needed to be installed on GNU/Linux OS.

Development Time Requirements -

- GCC – GNU Compiler Collection (specific for C)
- Gmake – GNU Make Utility
- Vim – Shell based Text Editor
- Virtual Box – Tool to run various guest machines on a single host machine

Implementation Time Requirements -

- Client System Requirements - A GNU/Linux 64 bit OS. (because in this system we have specified the offset is a type of unsigned long which 64 bit in 64 bit system and 32 bit on a 32 bit system) Therefore to support a very large size files we need 64 bit system.
- Server System Requirements - A GNU/Linux 64 bit OS. Because the server will also make use of the offset of the file, hence it needs to be big enough.
- Slave System Requirements - Any GNU/Linux OS. It doesn't make use of offset of the file. All it does the storage of the blocks of data of the file.

2.7.2 Hardware Requirements -

- Client and Server - We would prefer 64 bit architecture which is compatible with GNU/Linux OS. Any Modern CPU will work.
- Slave System - For slaves any of the normal system will work which supports GNU/Linux OS with specified secondary storage.

Core HW Requirements

- Processor : x_86 based 64 bit Intel Processors, amd 64 bit based Processors
- Ram : minimum 512 Mb, recommended 4Gb
- HDD : 80Gb
- Additional : Completed LAN Setup of (minimum 10Mb/s)

CHAPTER 3

Software Design

3.1 System Design

Analysts collect a great deal of unstructured data through interviews, questionnaires, on-site observations, procedural manuals and like. It is required to organize and convert the data through system flowcharts, data flow diagrams, structured English, decision tables and the like which support future developments of the system.

The Data flow diagrams and various processing logic techniques show how, where, and when data are used or changed in an information system, but these techniques do not show the definition, structure, and relationships within the data.

It is a way to focus on functions rather than the physical implementation. This is analogous to the architect's blueprint as a starting point for system design. The design is a solution, a "how to" approach, compared to analysis, a "what is" orientation.

System design is a highly creative process. This System design process is also referred as data modeling. The design concepts provide the software designer with a foundation from which more sophisticated methods can be applied. A set of fundamental design concepts has evolved. They are:

- **Abstraction** - Abstraction is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically in order to retain only information which is relevant for a particular purpose.
- **Refinement** - It is the process of elaboration. A hierarchy is developed by decomposing a macroscopic statement of function in a stepwise fashion until programming language statements are reached. In each step, one or several instructions of a given program are decomposed into more detailed instructions. Abstraction and Refinement are complementary concepts.
- **Modularity** - Software architecture is divided into components called modules.
- **Software Architecture** - It refers to the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. A software architecture is the development work product that gives the highest return on investment with respect to quality, schedule and cost.

- Control Hierarchy - A program structure that represent the organization of a program components and implies a hierarchy of control.
- Structural Partitioning - The program structure can be divided both horizontally and vertically. Horizontal partitions define separate branches of modular hierarchy for each major program function. Vertical partitioning suggests that control and work should be distributed top down in the program structure.
- Data Structure - It is a representation of the logical relationship among individual elements of data.
- Software Procedure - It focuses on the processing of each modules individually
- Information Hiding - Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

There are many aspects to consider in the design of a piece of software. The importance of each should reflect the goals the software is trying to achieve. Some of these aspects are:

- Compatibility - The software is able to operate with other products that are designed for interoperability with another product. For example, a piece of software may be backward-compatible with an older version of itself.
- Extensibility - New capabilities can be added to the software without major changes to the underlying architecture.
- Fault-tolerance - The software is resistant to and able to recover from component failure.
- Maintainability - The software can be restored to a specified condition within a specified period of time. For example, antivirus software may include the ability to periodically receive virus definition updates in order to maintain the software's effectiveness.
- Modularity - the resulting software comprises well defined, independent components. That leads to better maintainability. The components could be

then implemented and tested in isolation before being integrated to form a desired software system. This allows division of work in a software development project.

- Packaging - Printed material such as the box and manuals should match the style designated for the target market and should enhance usability. All compatibility information should be visible on the outside of the package. All components required for use should be included in the package or specified as a requirement on the outside of the package.
- Reliability - The software is able to perform a required function under stated conditions for a specified period of time.
- Re-usability - the modular components designed should capture the essence of the functionality expected out of them and no more or less. This single-minded purpose renders the components reusable wherever there are similar needs in other designs.
- Robustness - The software is able to operate under stress or tolerate unpredictable or invalid input. For example, it can be designed with a resilience to low memory conditions.
- Security - The software is able to withstand hostile acts and influences.
- Usability - The software user interface must be usable for its target user/audience. Default values for the parameters must be chosen so that they are a good choice for the majority of the users. In many cases, online help should be included and also carefully designed.

3.2 Distributed File System Overview

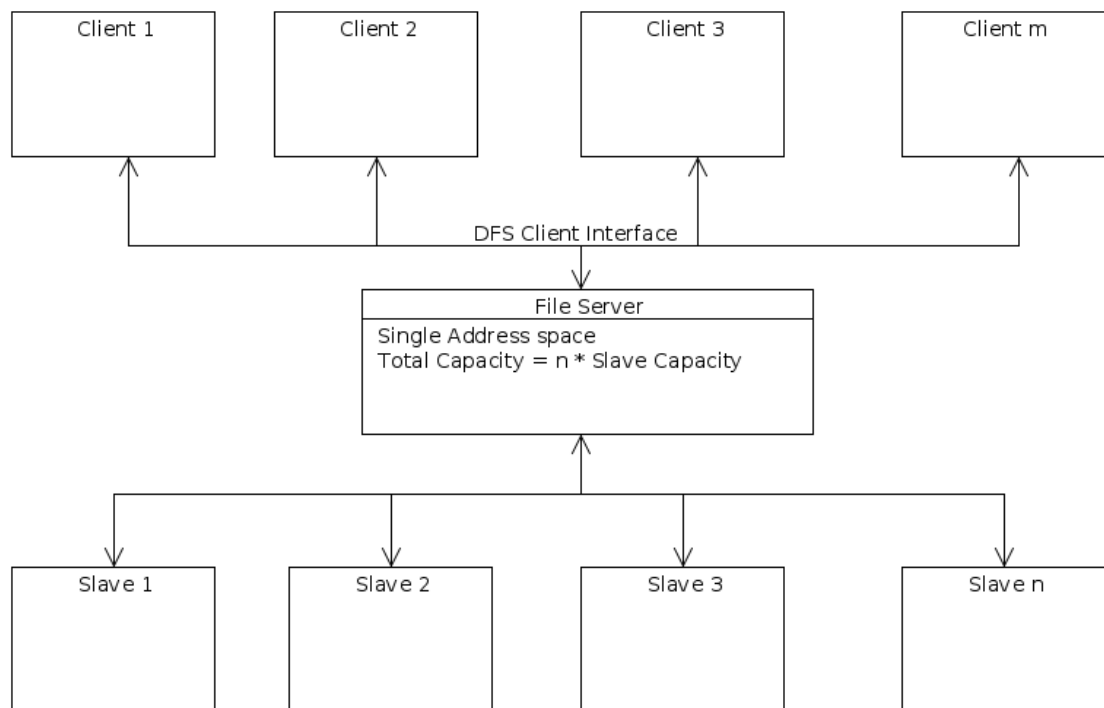


Fig 3.1 – Distributed File System Block Diagram

DFS is Distributed File System. There is a File Server which runs on a dedicated machine. The File Server will keep all the meta information about the whole File System. Client will access the File System through client modules. Clients can reside on any machine which is connected to our File System over the network. Slaves are the actual storage devices where the data of the files is stored, they are connected to the File Server over the network. Slaves do not support direct connection with the clients. The above figure explain the whole Distributed File System.

3.3 Architecture Design

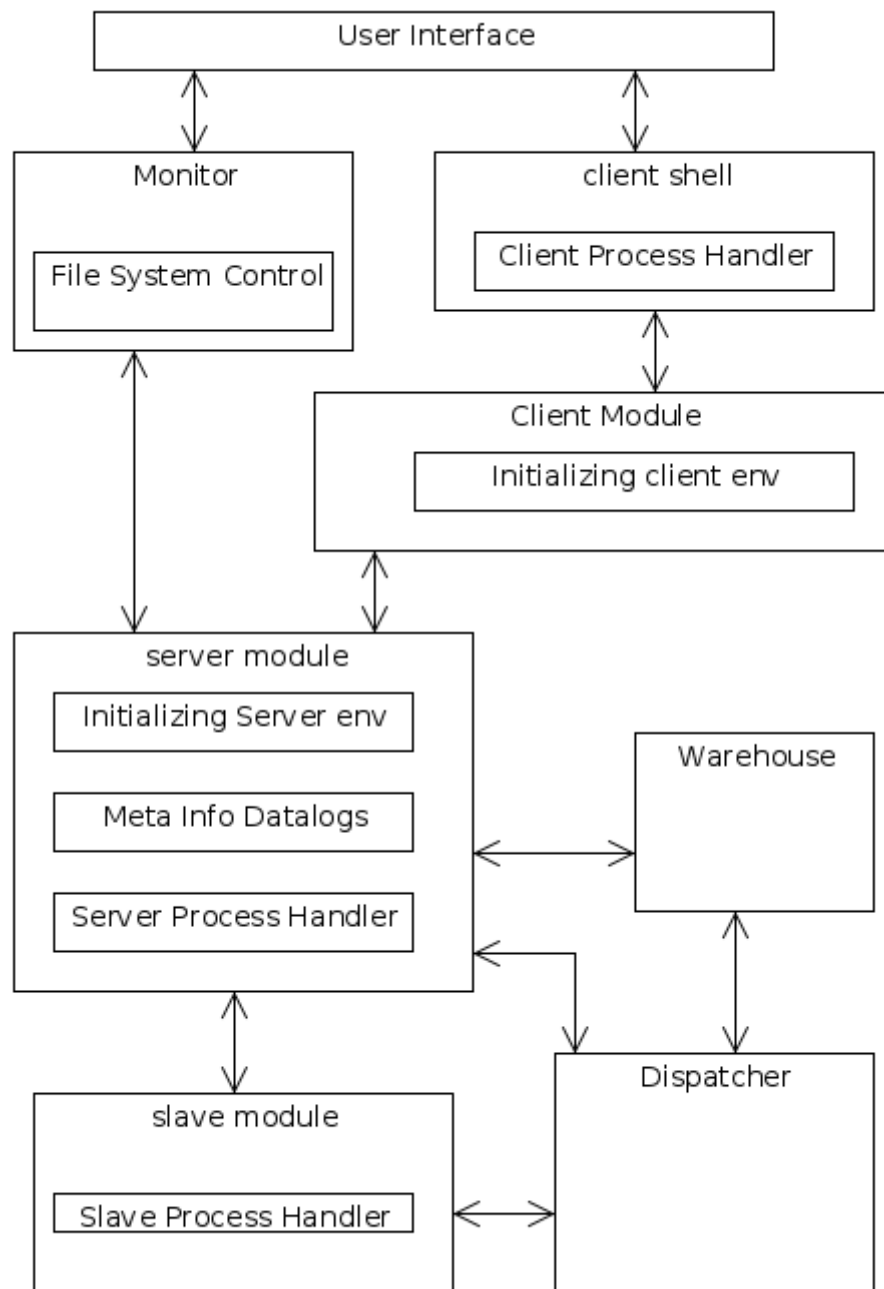
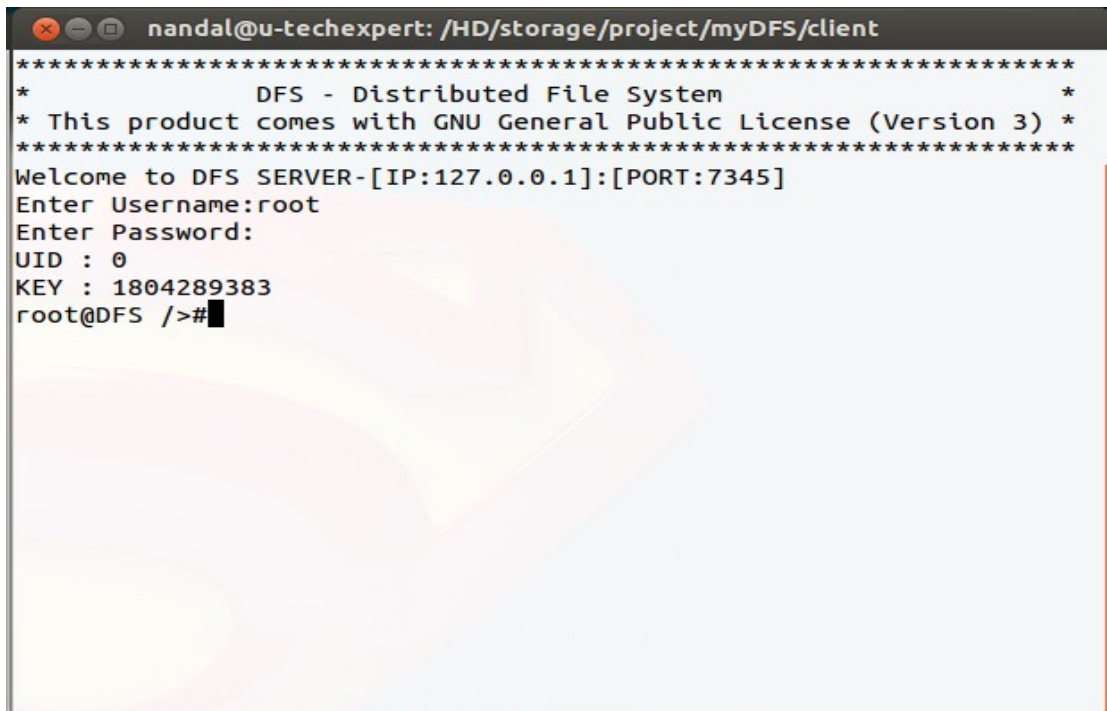


Fig 3.2 – Core Modules of DFS

The above figure displays all the Core Modules of the DFS. Here it shows which module can communicate with the other module in DFS. The User Interface will facilitate the user with the core functions desired from a file system. The User Interface will use the Client Module to connect with DFS.

3.4 User Interface Design

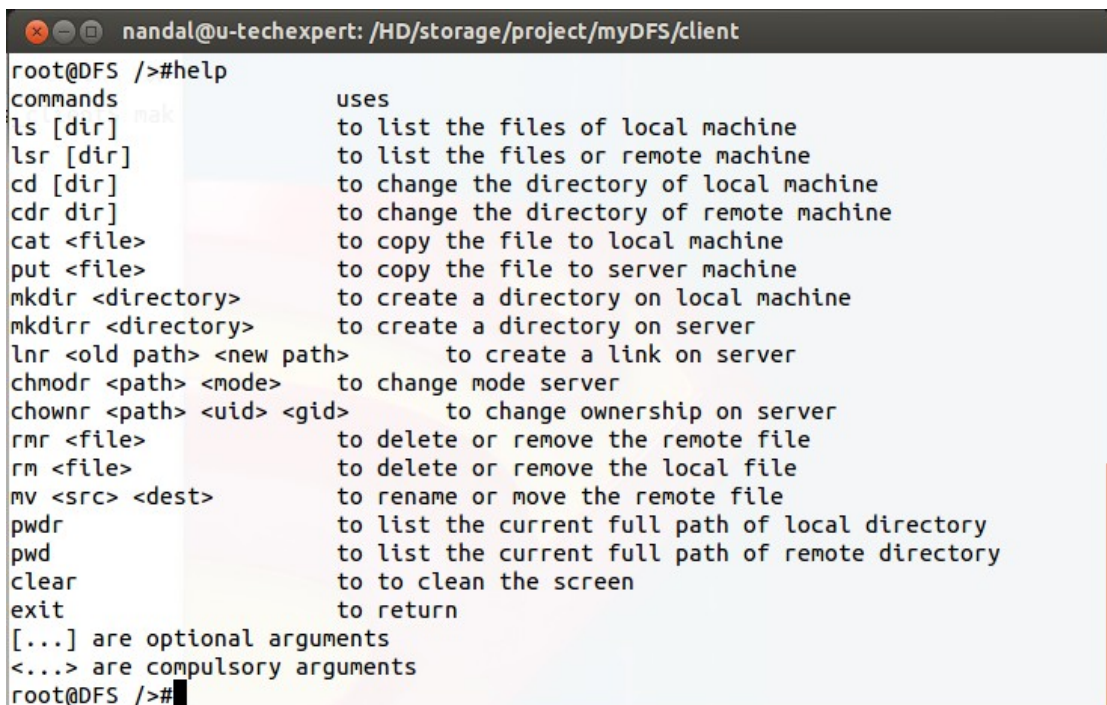
3.4.1 Client Shell Login Screen



```
nandal@u-techexpert: /HD/storage/project/myDFS/client
*****
*                               DFS - Distributed File System                               *
* This product comes with GNU General Public License (Version 3) *
*****
Welcome to DFS SERVER-[IP:127.0.0.1]:[PORT:7345]
Enter Username:root
Enter Password:
UID : 0
KEY : 1804289383
root@DFS />#
```

Fig 3.3 – Client Shell Login Screen

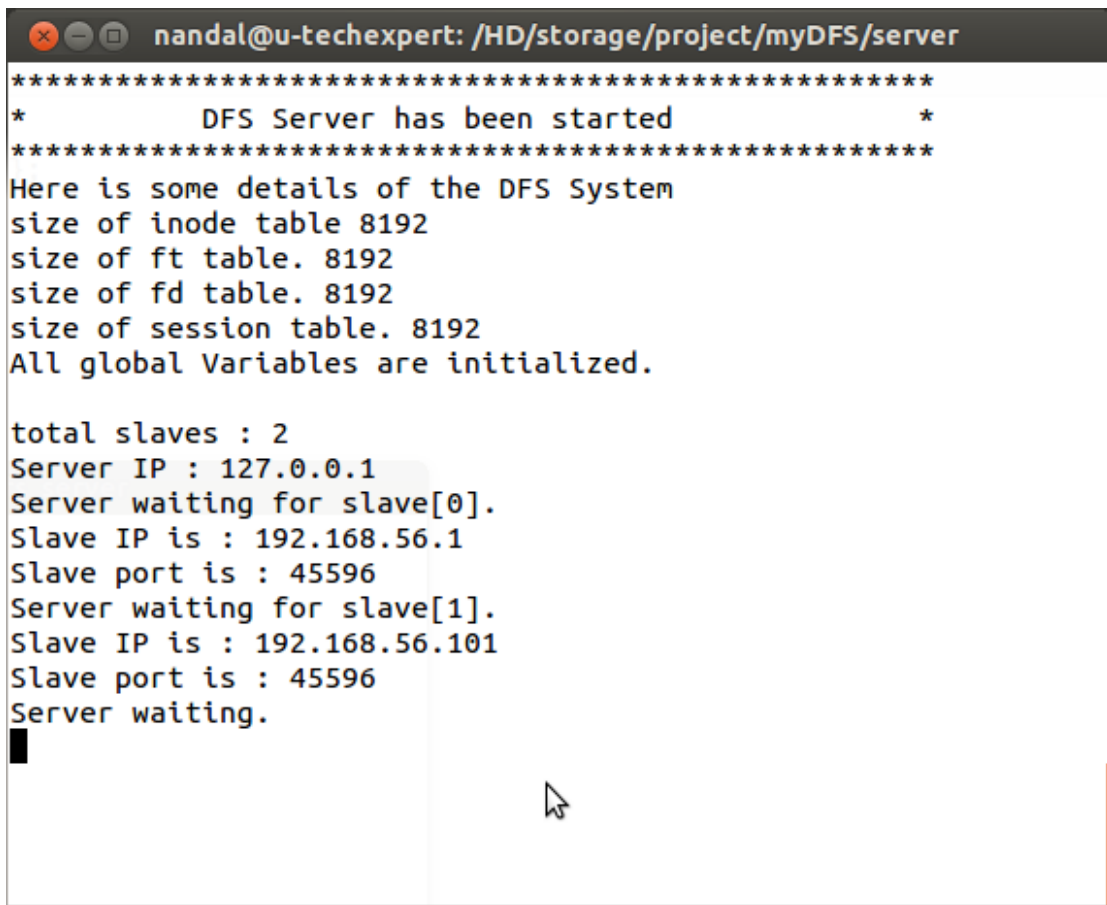
3.4.2 Client Shell Help Screen



```
nandal@u-techexpert: /HD/storage/project/myDFS/client
root@DFS />#help
commands      uses
ls [dir]      to list the files of local machine
lsr [dir]     to list the files or remote machine
cd [dir]      to change the directory of local machine
cdr dir]     to change the directory of remote machine
cat <file>    to copy the file to local machine
put <file>    to copy the file to server machine
mkdir <directory> to create a directory on local machine
mkdirr <directory> to create a directory on server
lnr <old path> <new path> to create a link on server
chmodr <path> <mode> to change mode server
chownr <path> <uid> <gid> to change ownership on server
rmr <file>    to delete or remove the remote file
rm <file>     to delete or remove the local file
mv <src> <dest> to rename or move the remote file
pwdr          to list the current full path of local directory
pwd          to list the current full path of remote directory
clear        to to clean the screen
exit         to return
[...] are optional arguments
<...> are compulsory arguments
root@DFS />#
```

Fig 3.4 – Client Shell Help Screen

3.4.3 DFS Server Start Up Screen

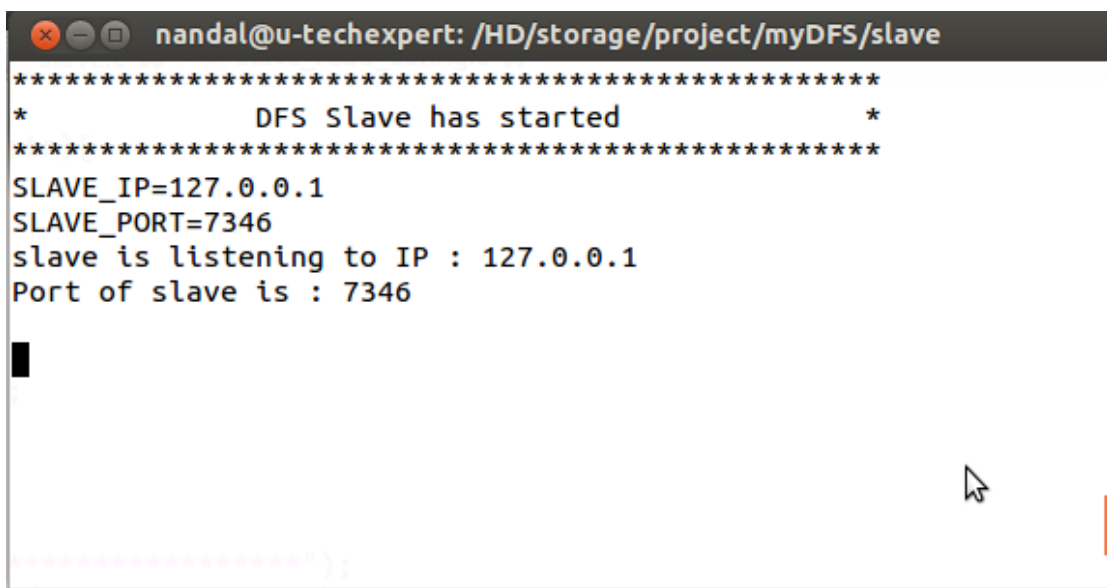
A terminal window titled 'nandal@u-techexpert: /HD/storage/project/myDFS/server' showing the DFS Server start-up process. The output includes a star separator, a confirmation message, details of the DFS system (inode, ft, fd, and session tables, all 8192), initialization of global variables, and slave configuration (2 slaves, server IP 127.0.0.1, slave IPs 192.168.56.1 and 192.168.56.101, slave port 45596). The server is currently waiting for the slaves to connect.

```
nandal@u-techexpert: /HD/storage/project/myDFS/server
*****
*          DFS Server has been started          *
*****
Here is some details of the DFS System
size of inode table 8192
size of ft table. 8192
size of fd table. 8192
size of session table. 8192
All global Variables are initialized.

total slaves : 2
Server IP : 127.0.0.1
Server waiting for slave[0].
Slave IP is : 192.168.56.1
Slave port is : 45596
Server waiting for slave[1].
Slave IP is : 192.168.56.101
Slave port is : 45596
Server waiting.
█
```

Fig 3.5 – DFS Server Start Up Screen

3.4.4 DFS Slave Start Up Screen

A terminal window titled 'nandal@u-techexpert: /HD/storage/project/myDFS/slave' showing the DFS Slave start-up process. The output includes a star separator, a confirmation message, and slave configuration (SLAVE_IP=127.0.0.1, SLAVE_PORT=7346). The slave is listening to the server IP and port.

```
nandal@u-techexpert: /HD/storage/project/myDFS/slave
*****
*          DFS Slave has started          *
*****
SLAVE_IP=127.0.0.1
SLAVE_PORT=7346
slave is listening to IP : 127.0.0.1
Port of slave is : 7346
█
```

Fig 3.6 – DFS Slave Start Up Screen

3.5 Functional Design(Component Level)

3.5.1 Client Module's Components

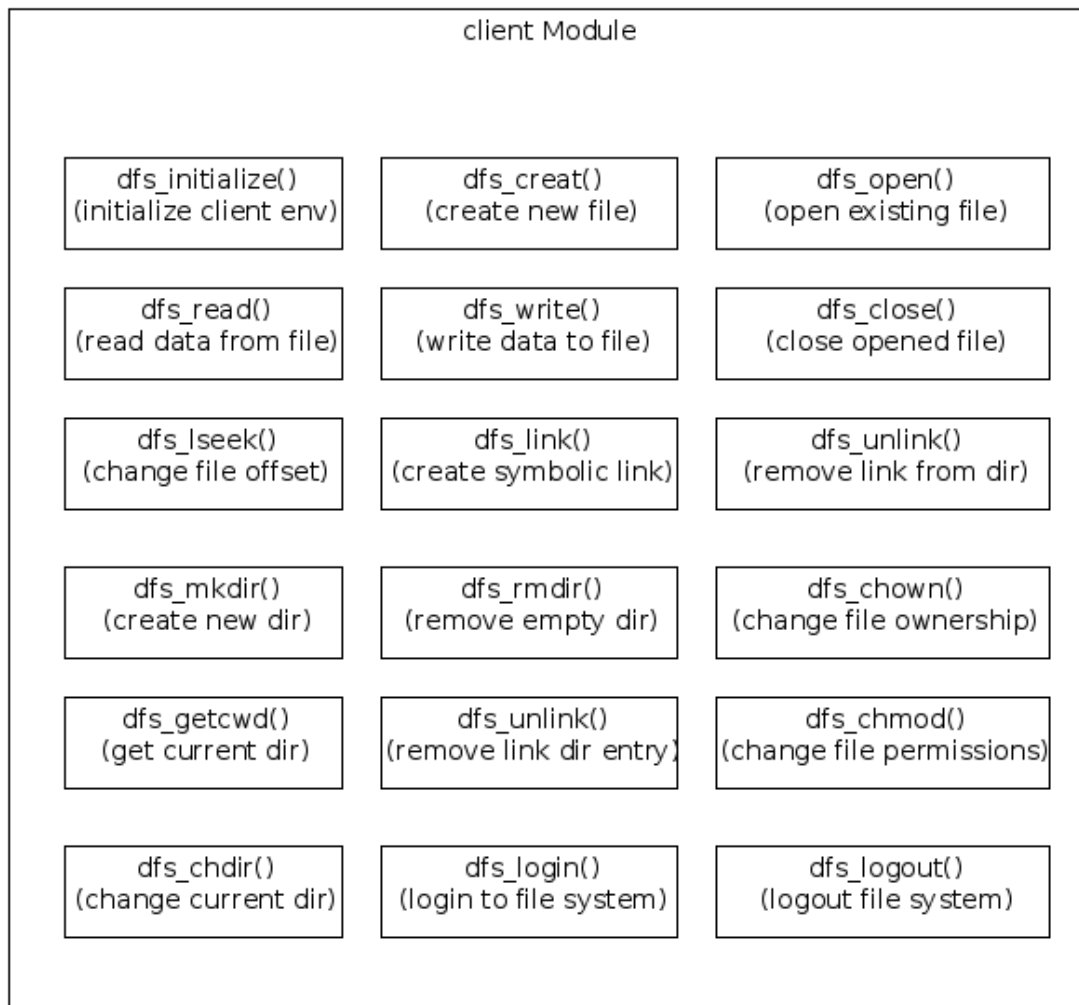


Fig 3.7 – Components of Client Module

The above diagram shows all the core components of client module. They all are implemented as distinct features of client module, any application developer can simply use them as functions in their work by simply using the DFS library. All of them are declared in the `dfs_client.h` header file in the source code. They are quite similar to the system calls provided by GNU/Linux for file handling. These can be used in the same manner.

3.5.2 Server Module Components

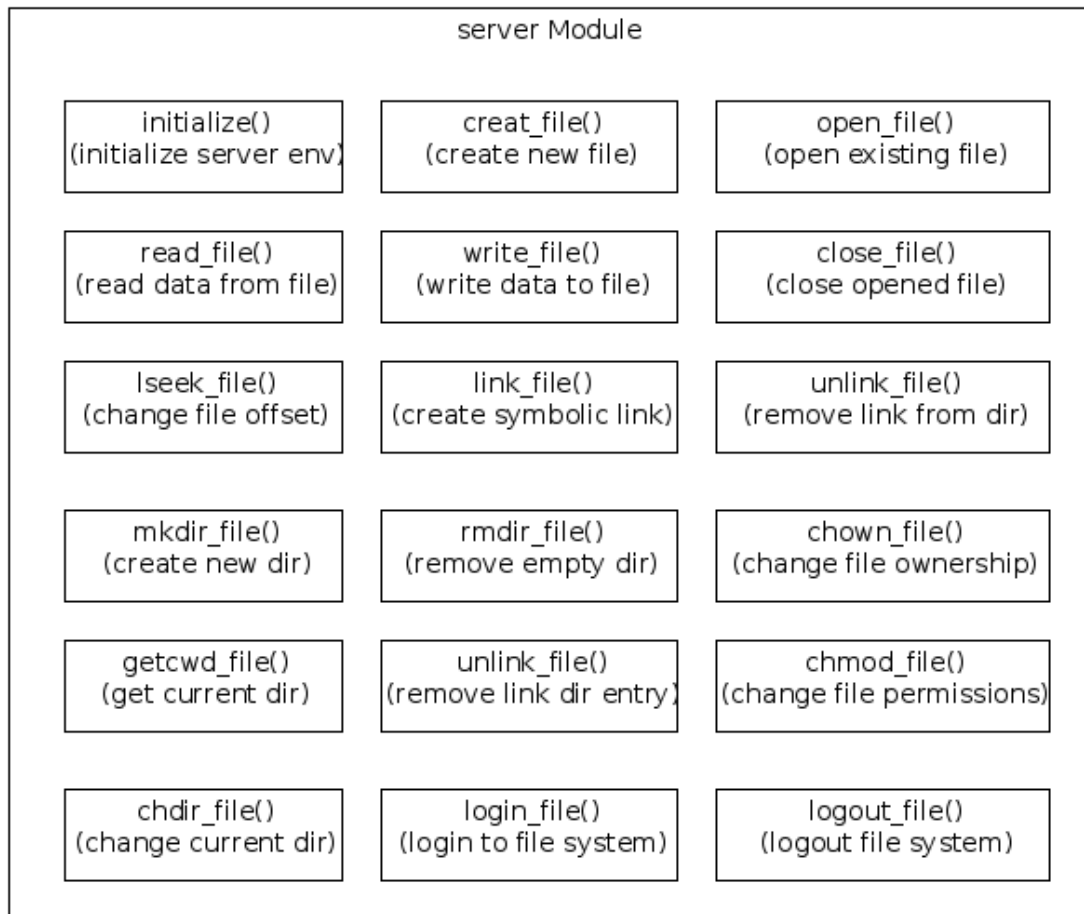


Fig 3.8 – Server Module Components

The above figure explains the Components of Server Modules of DFS. These are the used by DFS only. They shouldn't be accessed from outside. The Client Module will send the request to the Server Module over the network and then according to the request, the Server Module will invoke one of these functions. These functions will serve the request of the clients.

3.5.3 Slave Module Components

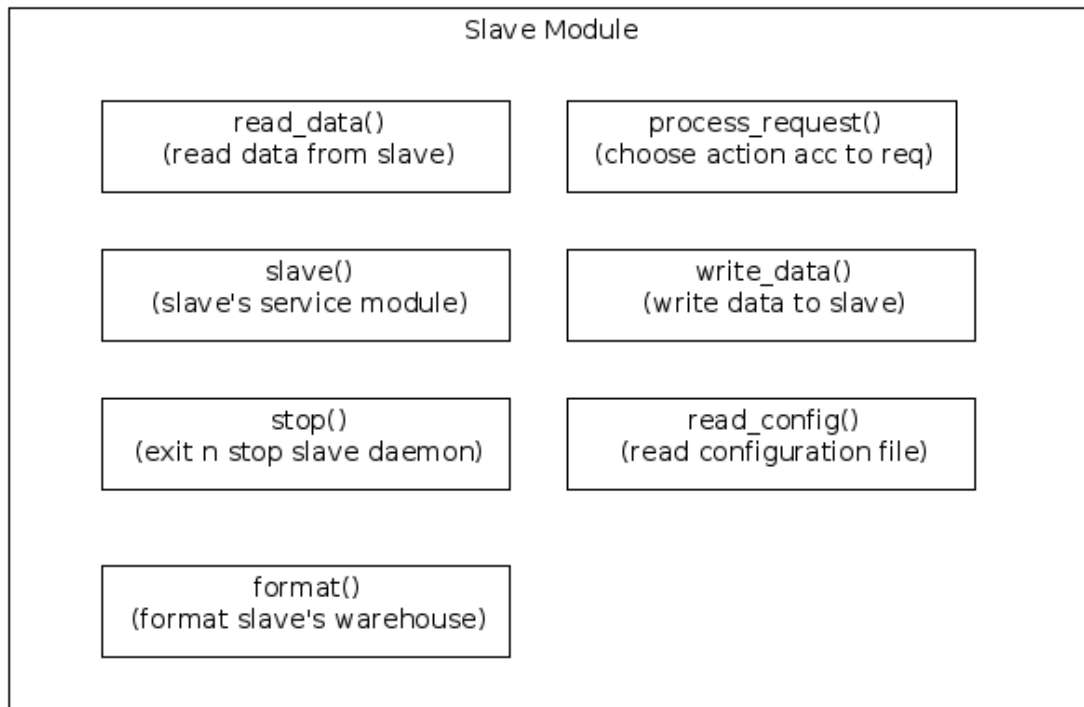


Fig 3.9 - Slave Module Components

The above figure explains the core Components of Slave Module. These functions will be invoked on the request from the DFS Server. The Slave Module will entertain only the DFS Server Requests. The Slave module will be installed on the machines which plays the role of storage devices. They must be connected with the DFS Server Over the network. The total of all the Slave's machine's storage capacity is combined and formed in a single address space for the whole DFS.

3.5.4 Client Shell Module Components

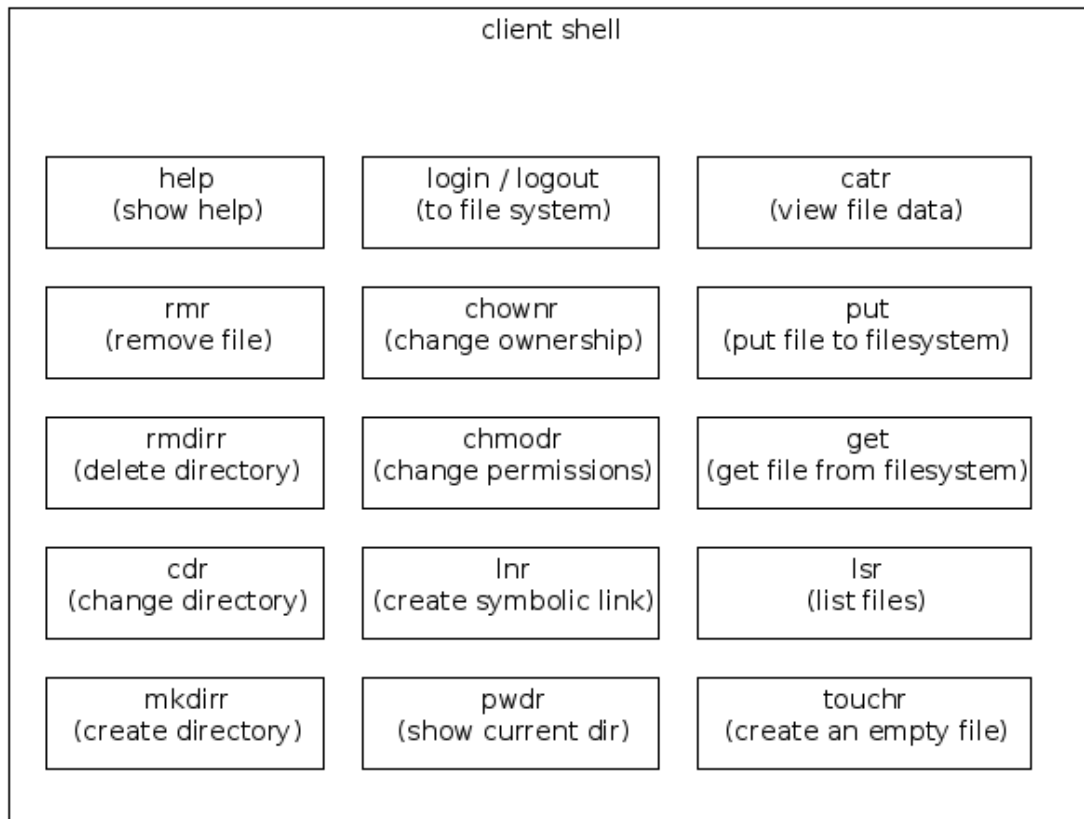


Fig 3.10 – Client Shell Module Components

The Client Shell Module contains all the basic functions desired from a file system. They will make use of the functions provided by the DFS Client Module and serve the user's requests. The end user will login into the shell of DFS and then can make use of the above commands.

3.5.5 Dispatcher Module Components

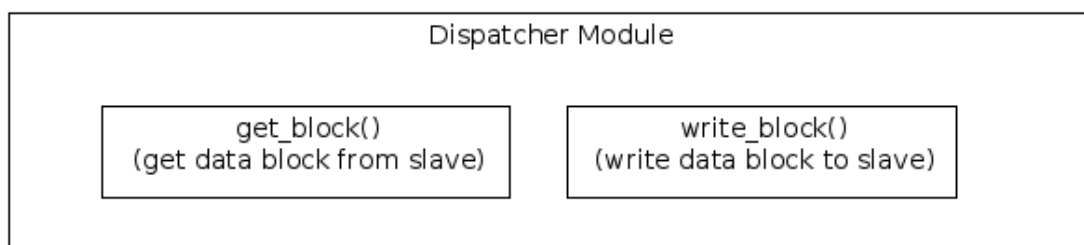


Fig 3.11 - Dispatcher Module Components

Dispatcher Module has two main components, `get_block()` and `write_block()`. The `get_block()` will be used by the DFS Server Module to collect the data block from the underlined DFS Slave and copy that block in the DFS Server to serve the request of the DFS Client. On the other hand `write_block()` will write the data block from the DFS Server to the DFS Slave machine for storage.

3.5.6 Monitor Module Components

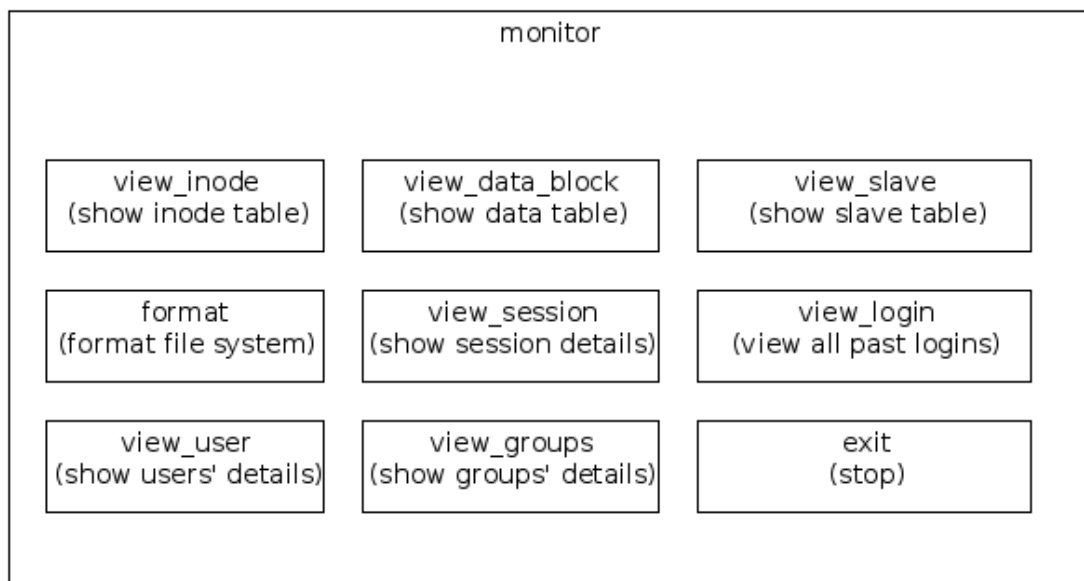


Fig 3.12 – Monitor Module Components

The Monitor Module contains the functionalities, which will be used view the DFS Server stats and with the help of this module we can manipulate the DFS Server Meta Information. We can also reset the whole DFS System with the help of this module. It plays a vital role in understanding the internals of the DFS.

3.5.7 Core Tables Used in DFS Server

<table><tr><th>user.tbl</th></tr><tr><td>uhead</td></tr><tr><td>--free (uid)</td></tr><tr><td>unode</td></tr><tr><td>--id (uid)</td></tr><tr><td>--name[] (char)</td></tr><tr><td>--pass[] (char)</td></tr></table>	user.tbl	uhead	--free (uid)	unode	--id (uid)	--name[] (char)	--pass[] (char)	<table><tr><th>inode.tbl</th></tr><tr><td>ihead</td></tr><tr><td>--free (inode)</td></tr><tr><td>inode</td></tr><tr><td>--id (inode)</td></tr><tr><td>--mode (mode)</td></tr><tr><td>--size (size)</td></tr><tr><td>--nlink (nlink)</td></tr><tr><td>--uid (uid)</td></tr><tr><td>--gid (gid)</td></tr><tr><td>--blocks (blk)</td></tr><tr><td>--data[] (blk)</td></tr><tr><td>--atime (time)</td></tr><tr><td>--ctime (time)</td></tr><tr><td>--mtime (time)</td></tr><tr><td>--lock (char)</td></tr></table>	inode.tbl	ihead	--free (inode)	inode	--id (inode)	--mode (mode)	--size (size)	--nlink (nlink)	--uid (uid)	--gid (gid)	--blocks (blk)	--data[] (blk)	--atime (time)	--ctime (time)	--mtime (time)	--lock (char)	<table><tr><th>name_to_inode.tbl</th></tr><tr><td>nhead</td></tr><tr><td>--free (iid)</td></tr><tr><td>--self (inode)</td></tr><tr><td>--parent (inode)</td></tr><tr><td>nnode</td></tr><tr><td>--inode (inode)</td></tr><tr><td>--name[] (char)</td></tr></table>	name_to_inode.tbl	nhead	--free (iid)	--self (inode)	--parent (inode)	nnode	--inode (inode)	--name[] (char)
user.tbl																																	
uhead																																	
--free (uid)																																	
unode																																	
--id (uid)																																	
--name[] (char)																																	
--pass[] (char)																																	
inode.tbl																																	
ihead																																	
--free (inode)																																	
inode																																	
--id (inode)																																	
--mode (mode)																																	
--size (size)																																	
--nlink (nlink)																																	
--uid (uid)																																	
--gid (gid)																																	
--blocks (blk)																																	
--data[] (blk)																																	
--atime (time)																																	
--ctime (time)																																	
--mtime (time)																																	
--lock (char)																																	
name_to_inode.tbl																																	
nhead																																	
--free (iid)																																	
--self (inode)																																	
--parent (inode)																																	
nnode																																	
--inode (inode)																																	
--name[] (char)																																	
<table><tr><th>group.tbl</th></tr><tr><td>ghead</td></tr><tr><td>--free (gid)</td></tr><tr><td>gnode</td></tr><tr><td>--id (gid)</td></tr><tr><td>--name[] (char)</td></tr></table>	group.tbl	ghead	--free (gid)	gnode	--id (gid)	--name[] (char)																											
group.tbl																																	
ghead																																	
--free (gid)																																	
gnode																																	
--id (gid)																																	
--name[] (char)																																	
<table><tr><th>slave.tbl</th></tr><tr><td>shead</td></tr><tr><td>--free (dev)</td></tr><tr><td>snode</td></tr><tr><td>--id (dev)</td></tr><tr><td>--ip[] (char)</td></tr><tr><td>--port (short)</td></tr></table>	slave.tbl	shead	--free (dev)	snode	--id (dev)	--ip[] (char)	--port (short)		<table><tr><th>data_block.tbl</th></tr><tr><td>dhead</td></tr><tr><td>--free (blk)</td></tr><tr><td>dnode</td></tr><tr><td>--id (blk)</td></tr><tr><td>--slave_id (dev)</td></tr><tr><td>--status (status)</td></tr><tr><td>--sub_id (uint)</td></tr></table>	data_block.tbl	dhead	--free (blk)	dnode	--id (blk)	--slave_id (dev)	--status (status)	--sub_id (uint)																
slave.tbl																																	
shead																																	
--free (dev)																																	
snode																																	
--id (dev)																																	
--ip[] (char)																																	
--port (short)																																	
data_block.tbl																																	
dhead																																	
--free (blk)																																	
dnode																																	
--id (blk)																																	
--slave_id (dev)																																	
--status (status)																																	
--sub_id (uint)																																	

Fig 3.13 – Core Tables Used in DFS Server

The above figure shows the core Tables used in the DFS Server. User.tbl contains the data of all the registered user of DFS. The main table is inode.tbl which contains the inode entries of the whole DFS. The details of the user groups is stored in group.tbl. The data_block.tbl table contains the information about the actual data of the files. The slave.tbl table contains the list of all the dedicated slaves of DFS for the storage of data blocks. The name_to_inode.tbl is the table which is created for each directory of the DFS.

3.6 Protocols Used in DFS

Now here is a list of all the protocols used for each core functions provided by the DFS Client for the User. These protocols explain the working of the core functions diagrammatically. All the Modules are represented as Blocks at top of the diagrams. And the sequence of their communication is from top to bottom. The messages are represented with arrows which points to the receiver from the sender of the message.

3.6.1 Login Request Protocol

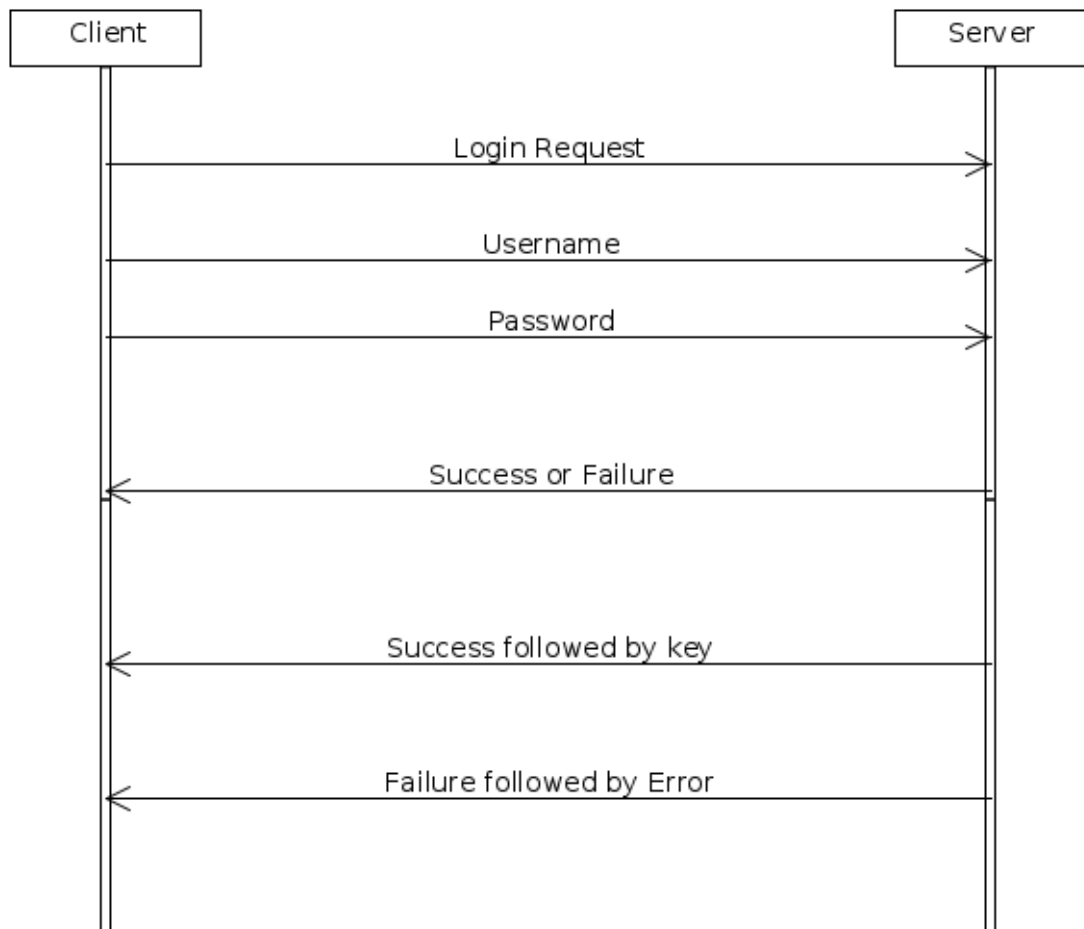


Fig 3.14– `dfs_login` Protocol

3.6.2 Logout Request Protocol

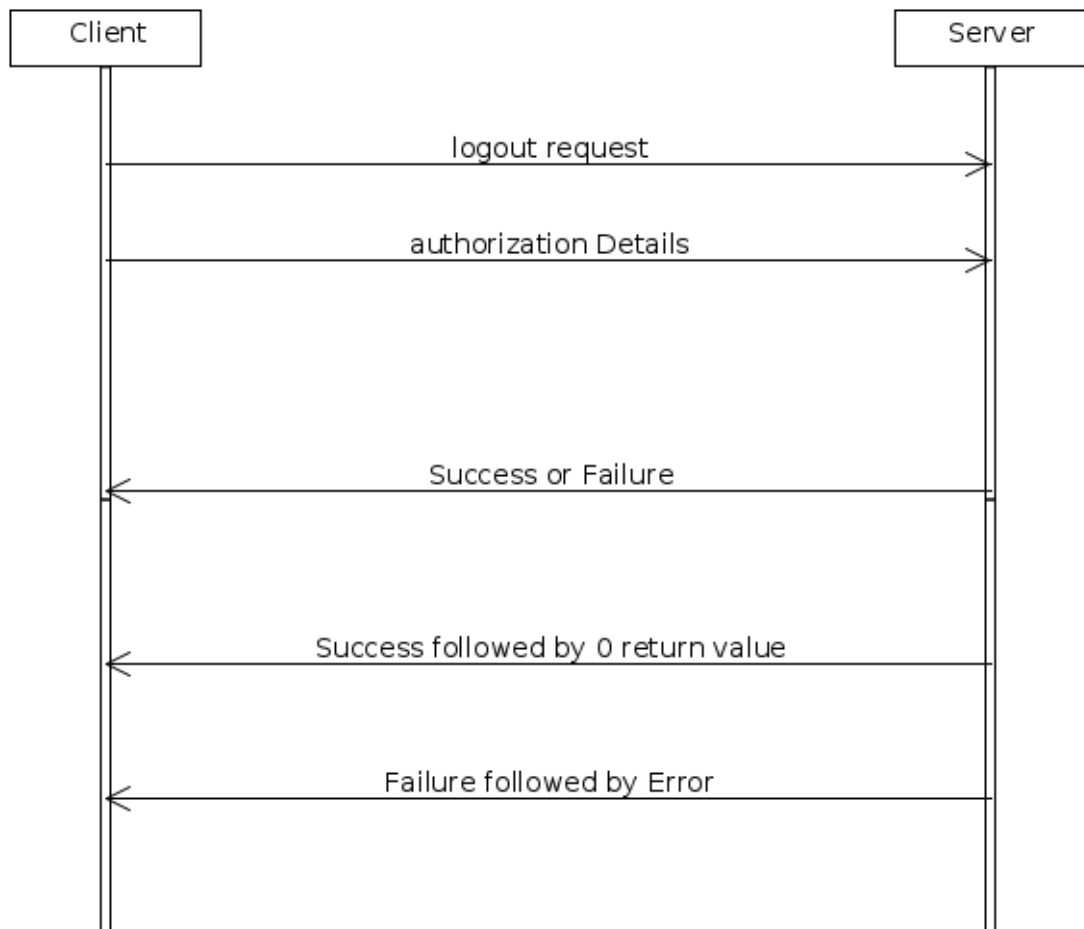


Fig 3.15– `dfs_logout` Protocol

3.6.3 Open Request Protocol

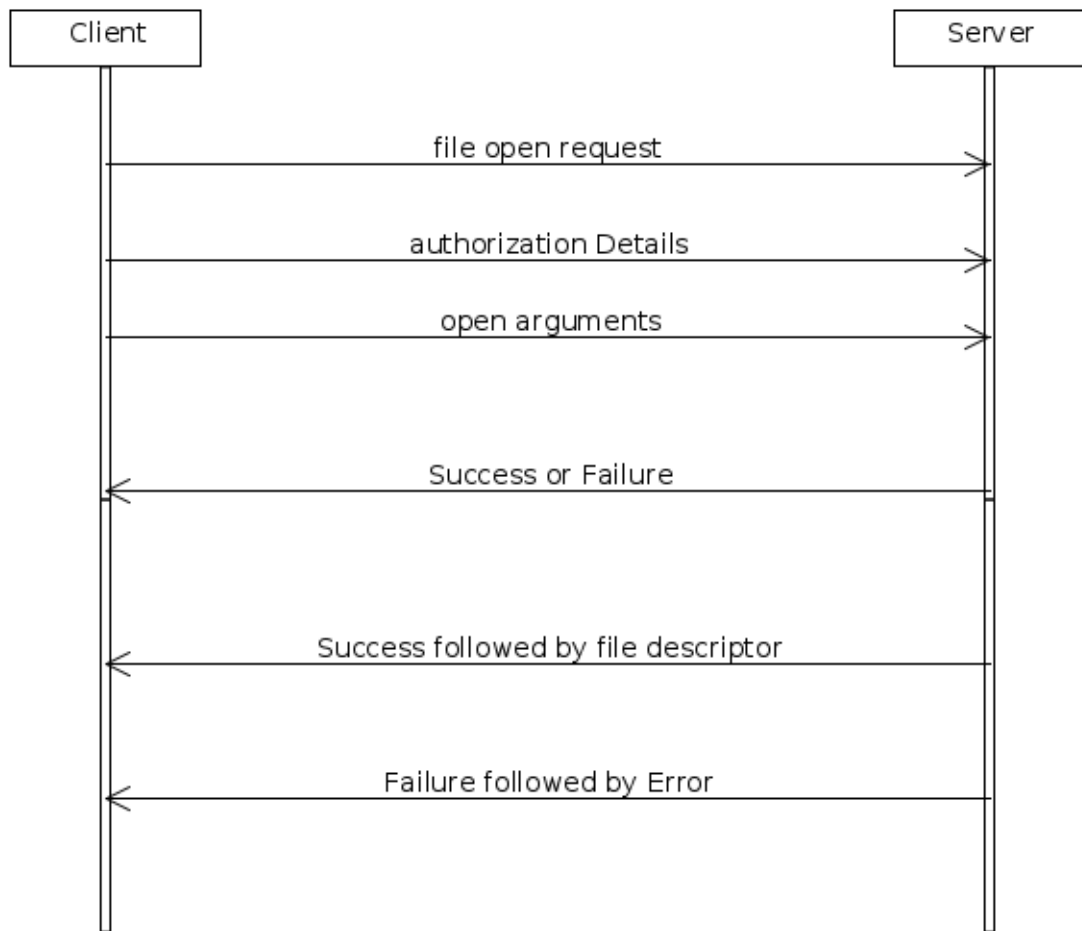


Fig 3.16 – `dfs_open()` Protocol

3.6.4 Write Request Protocol

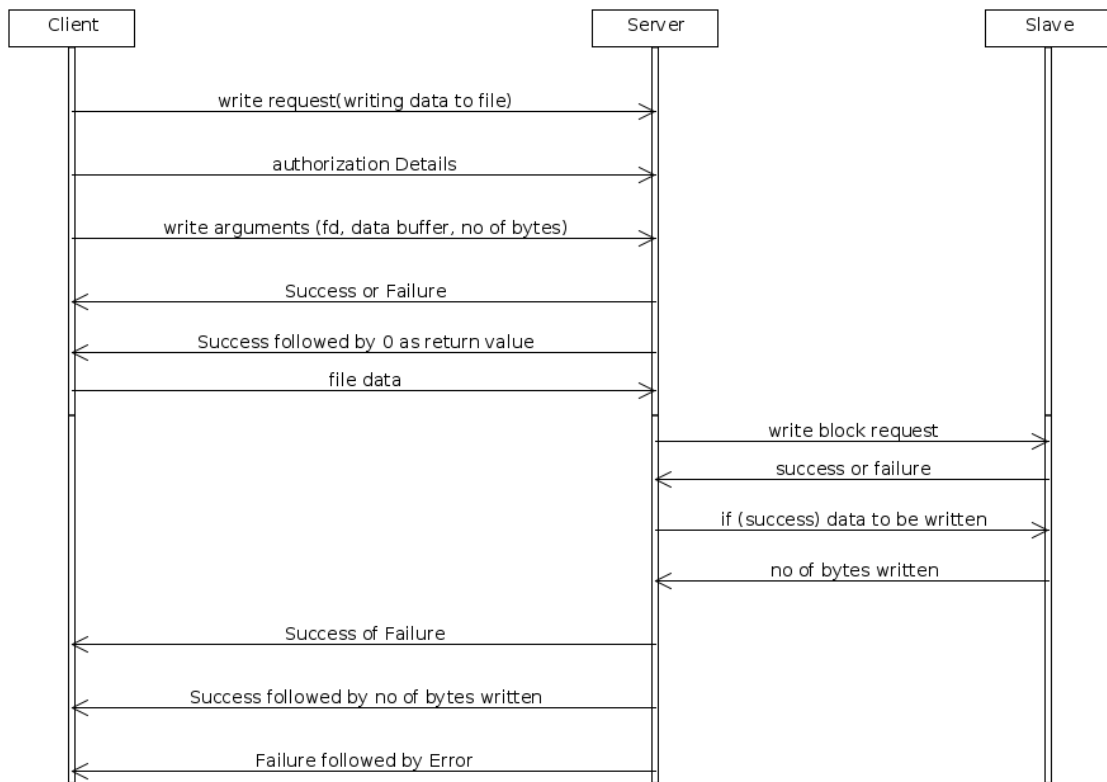


Fig 3.17 – `dfs_write()` Protocol

3.6.5 Read Request Protocol

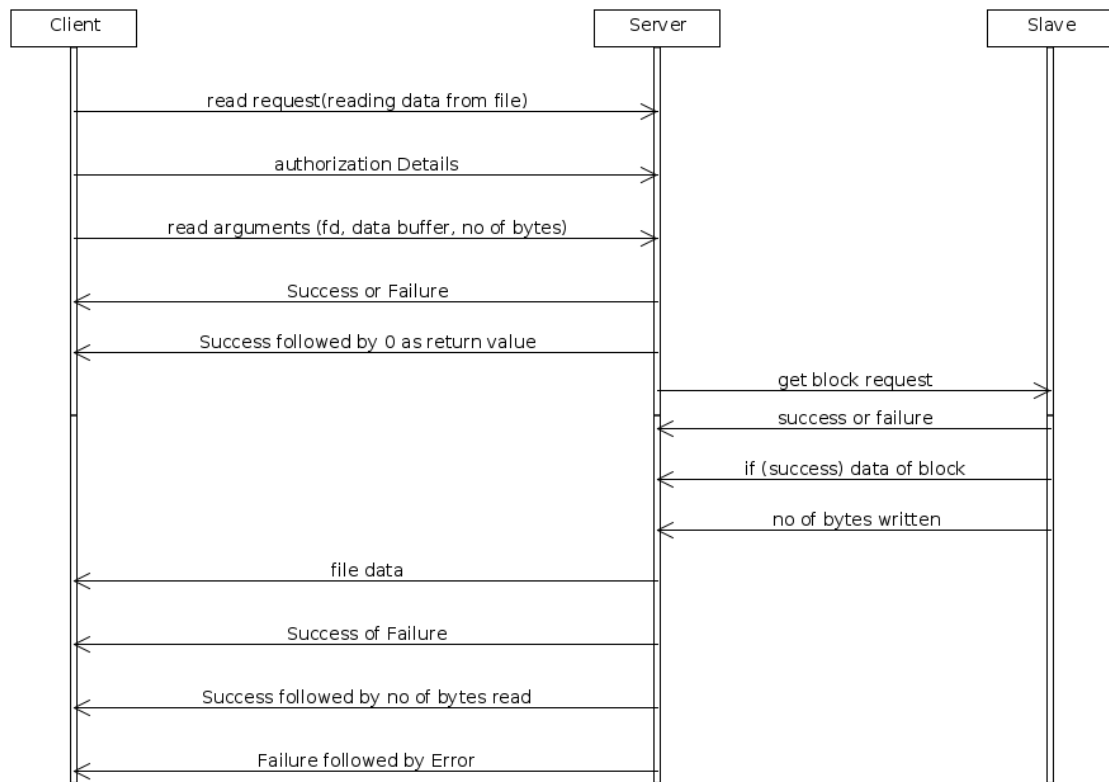


Fig 3.18– `dfs_read()` Protocol

3.6.6 Close Request Protocol

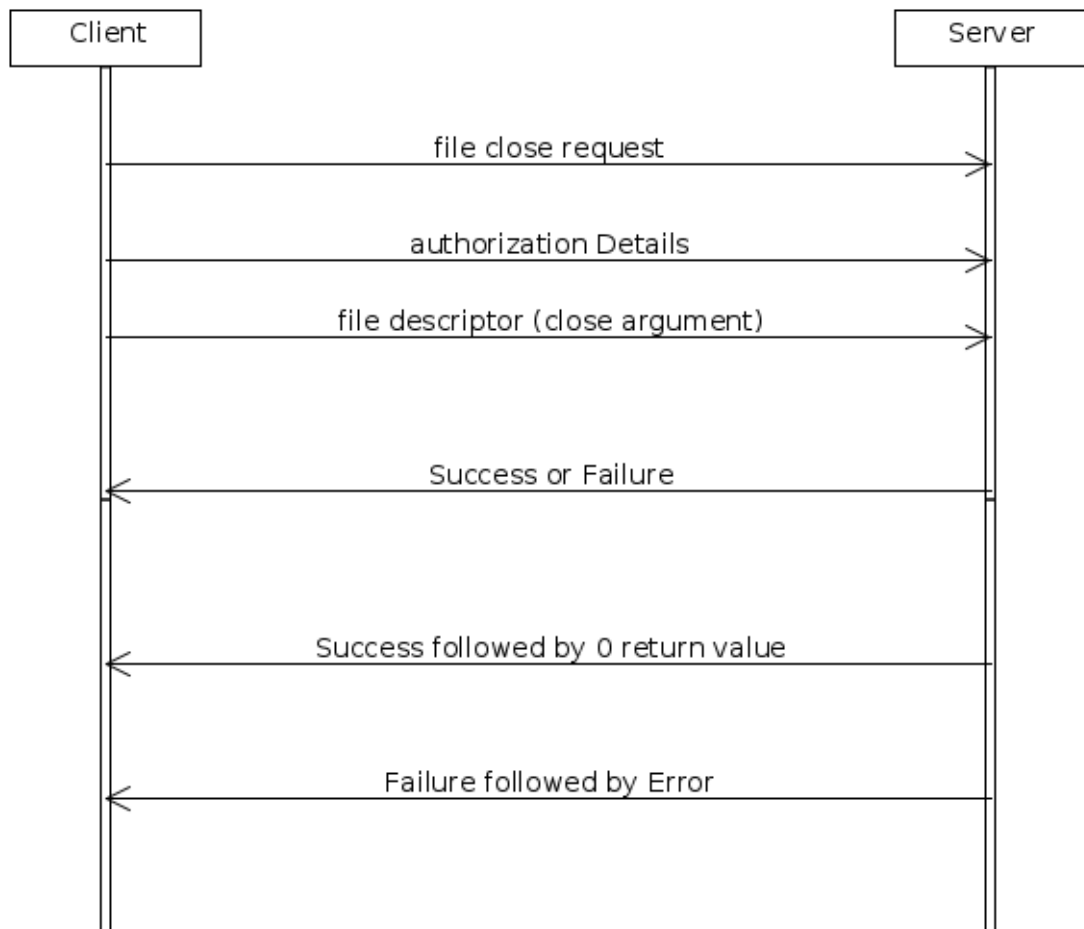


Fig 3.19 – `dfs_close()` Protocol

3.6.7 Create Request Protocol

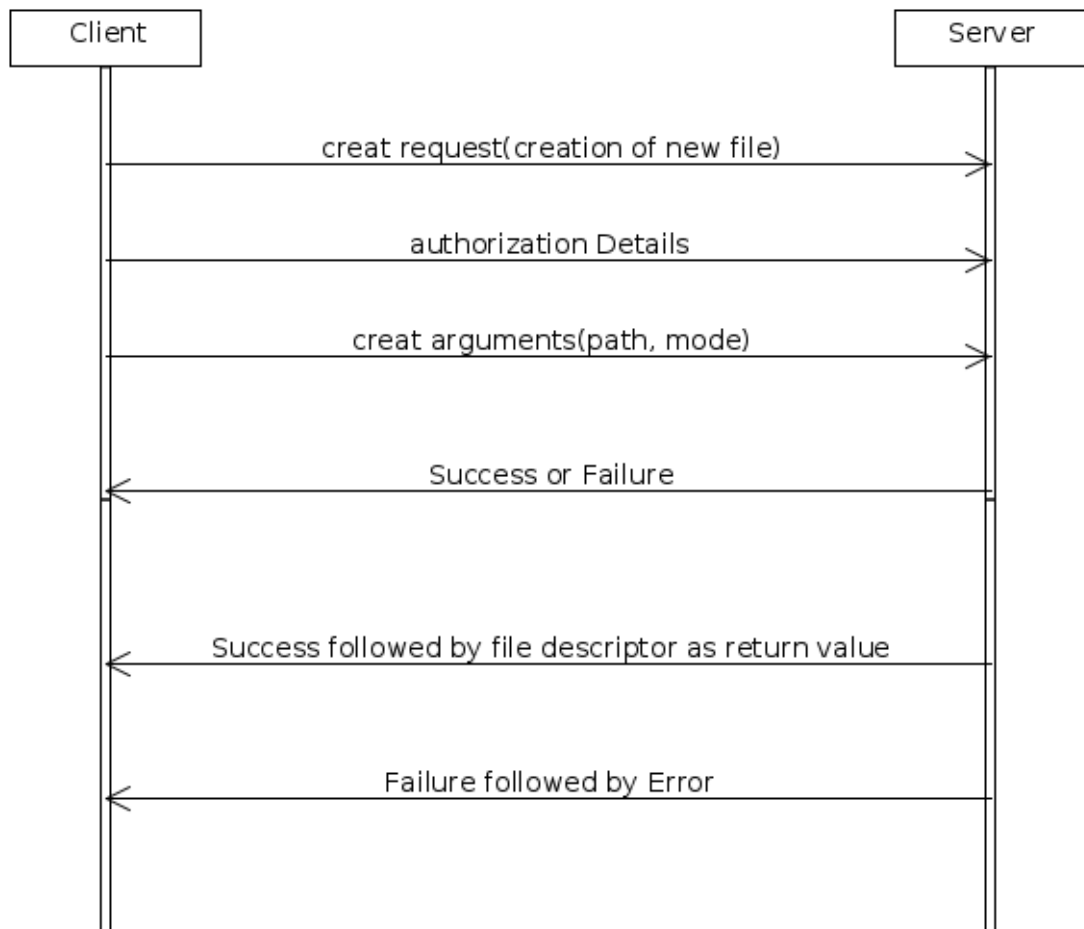


Fig 3.20 – dfs_creat() Protocol

3.6.8 Lseek Request Protocol

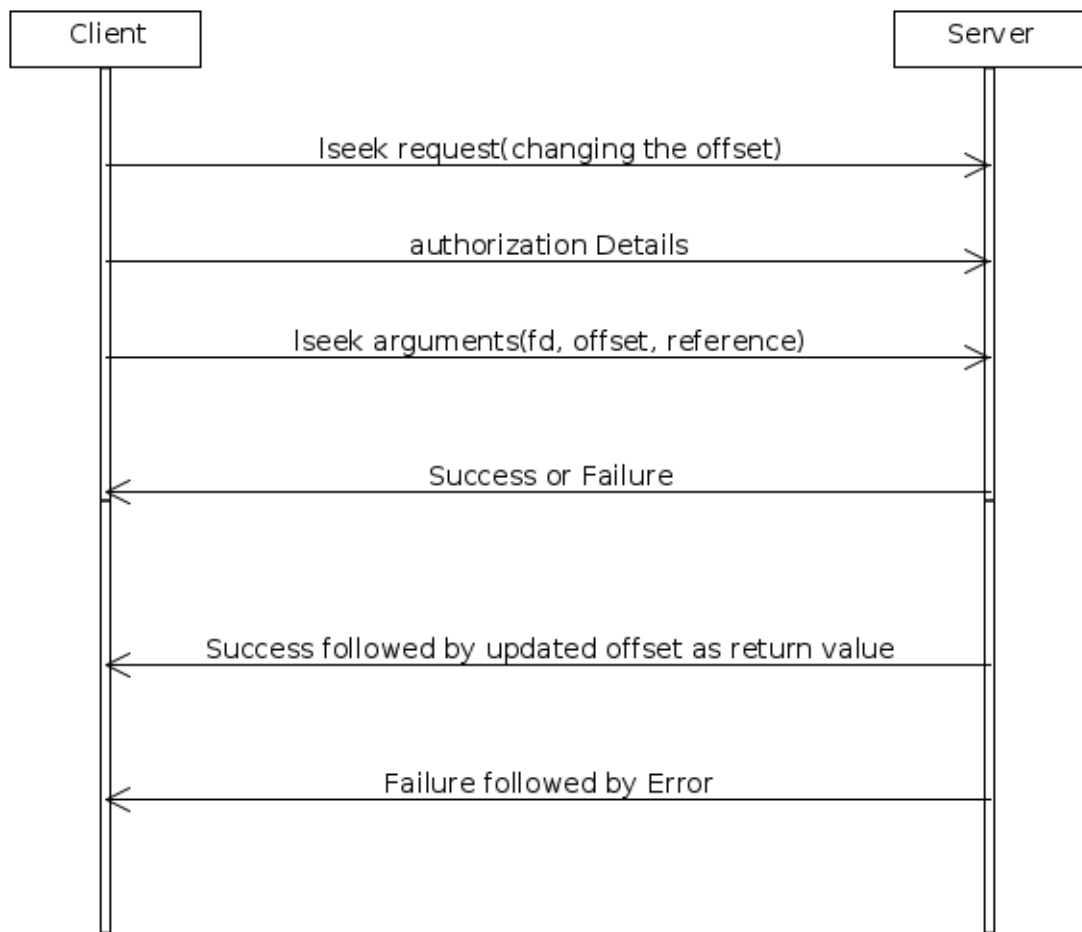


Fig 3.21 – `dfs_lseek()` Protocol

3.6.9 Link Request Protocol

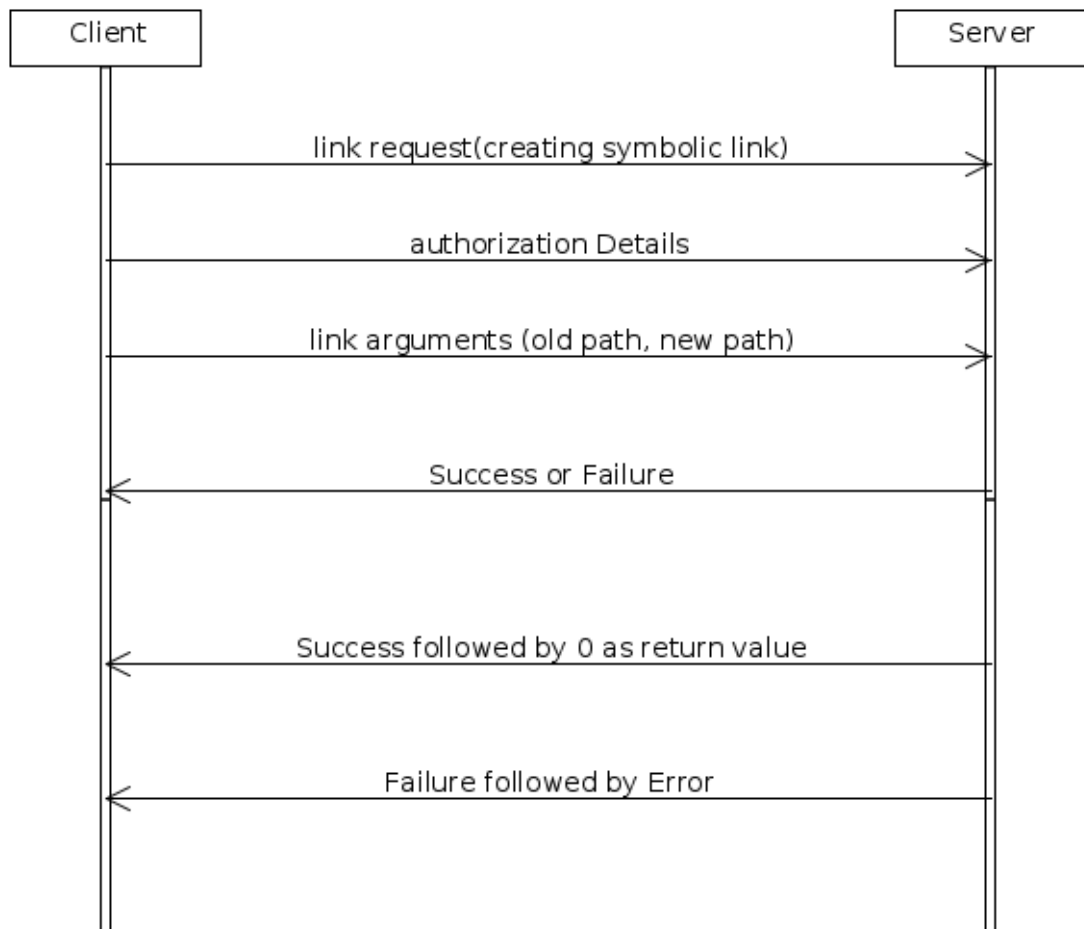


Fig 3.22 – `dfs_link()` Protocol

3.6.10 Make Directory Request Protocol

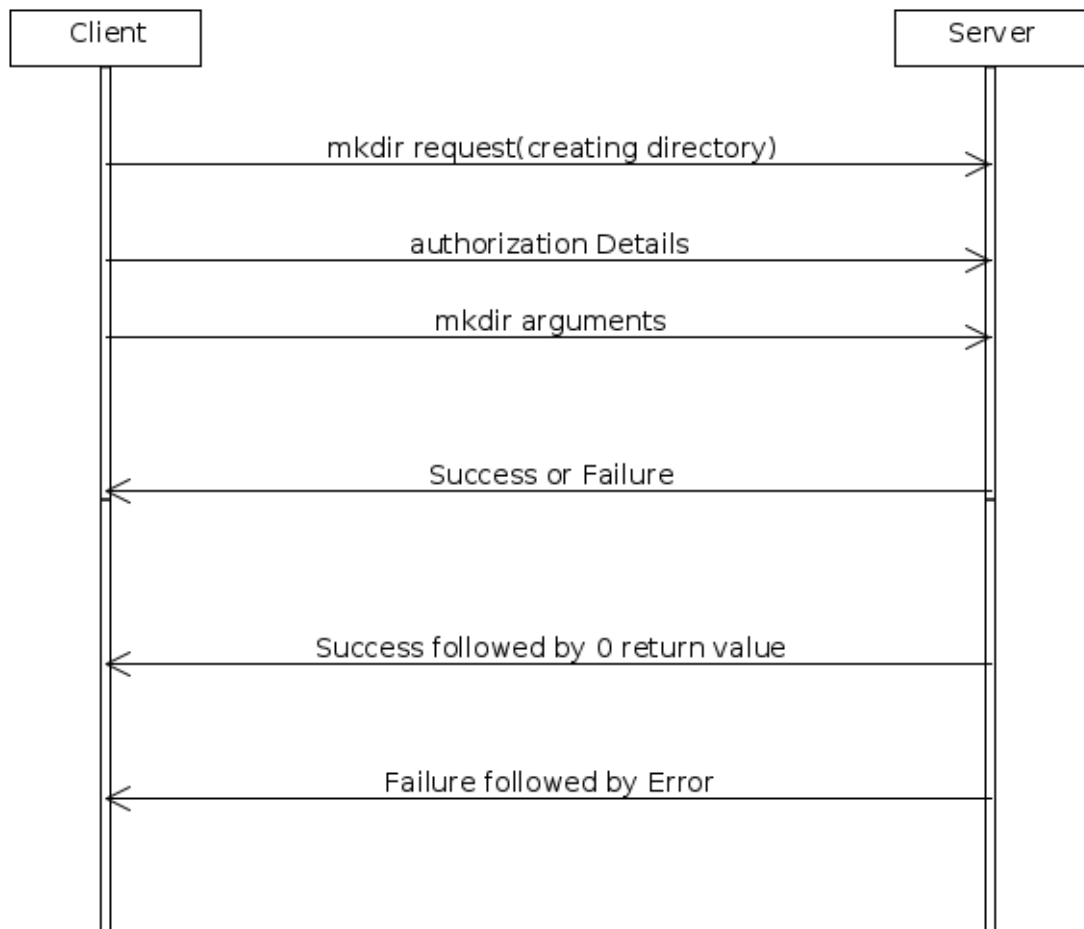


Fig 3.23 – `dfs_mkdir()` Protocol

3.6.11 Remove Directory Request Protocol

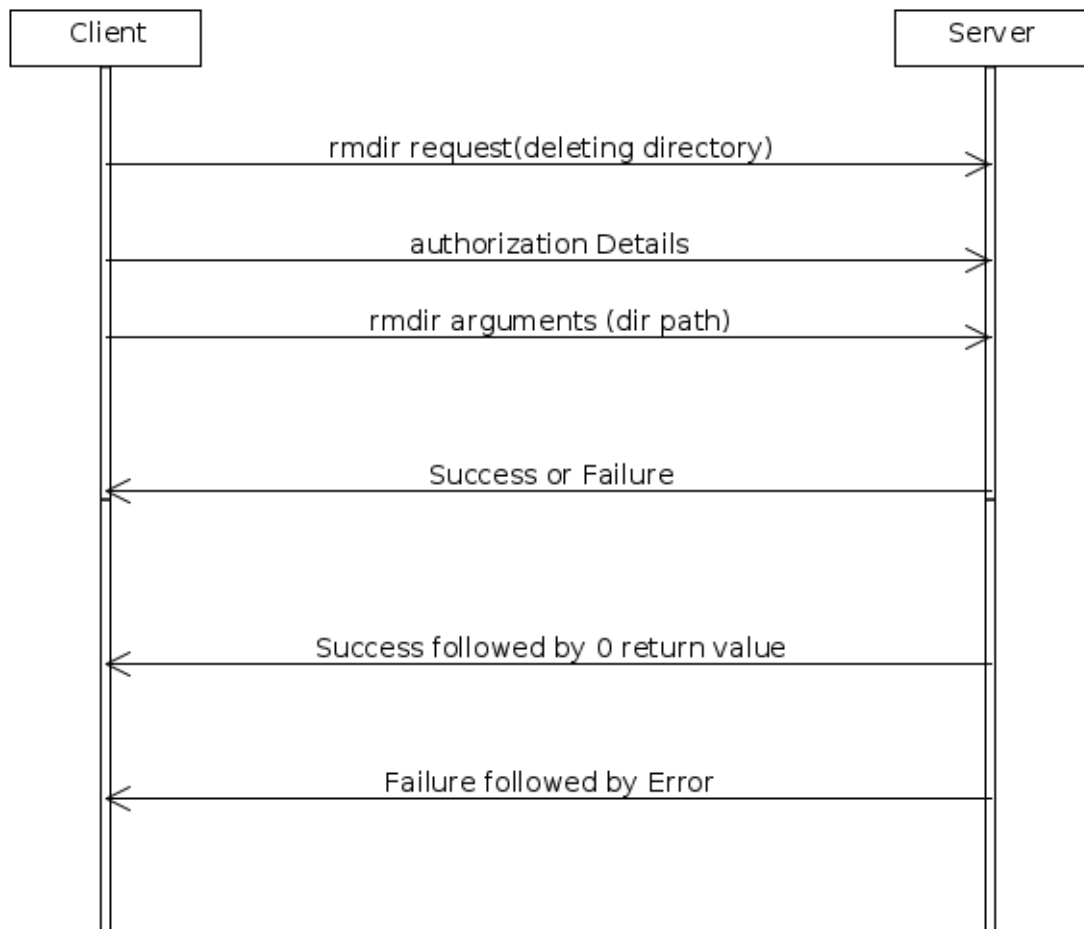


Fig 3.24 – dfs_rmdir() Protocol

3.6.12 Change Permissions Request Protocol

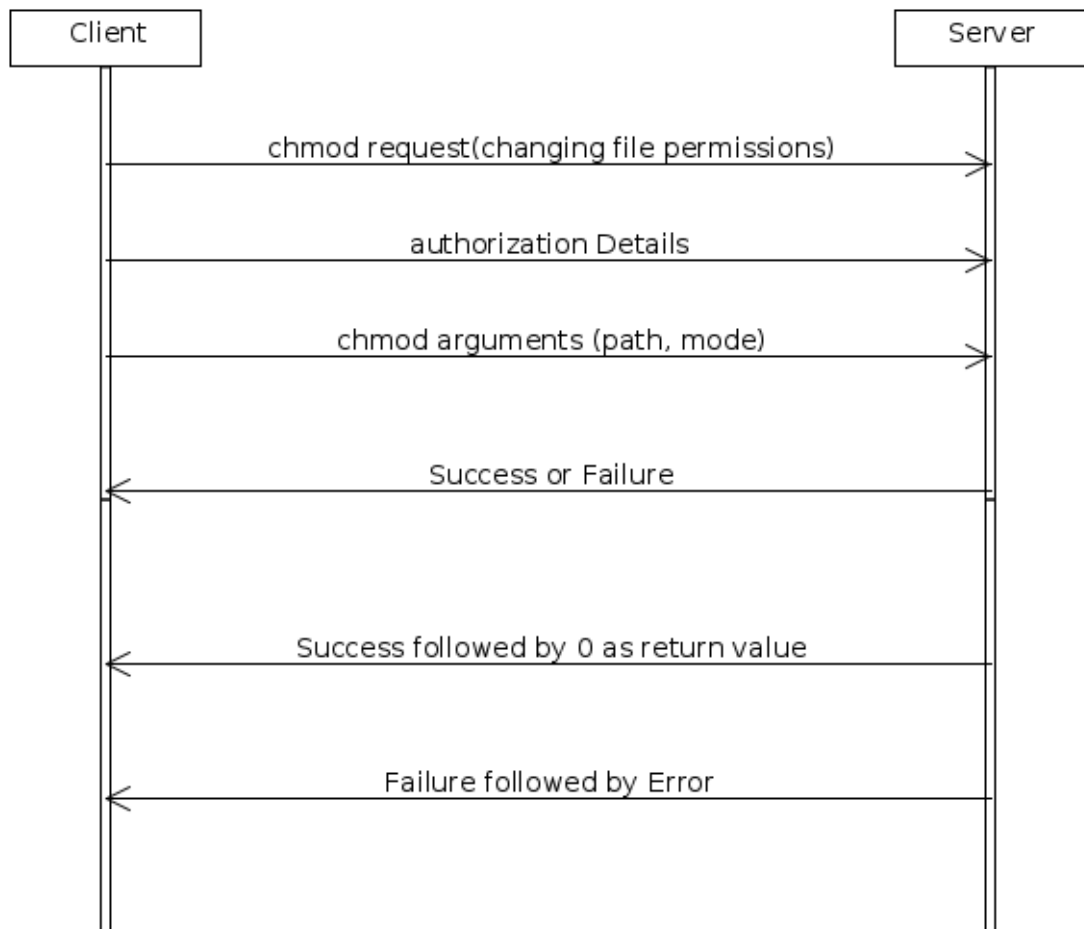


Fig 3.25 – `dfs_chmod()` Protocol

3.6.13 Change Ownership Request Protocol

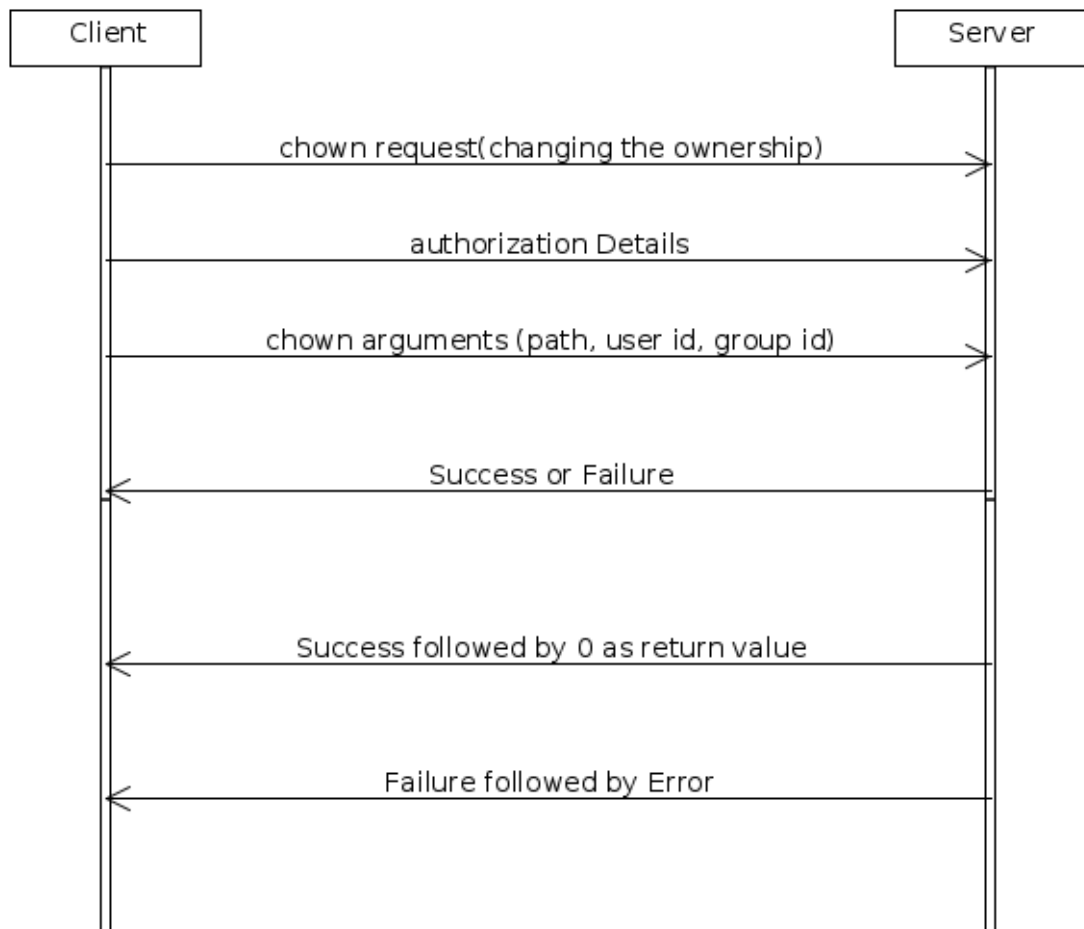


Fig 3.26 – `dfs_chown()` Protocol

3.6.14 Get Current Working Directory Request Protocol

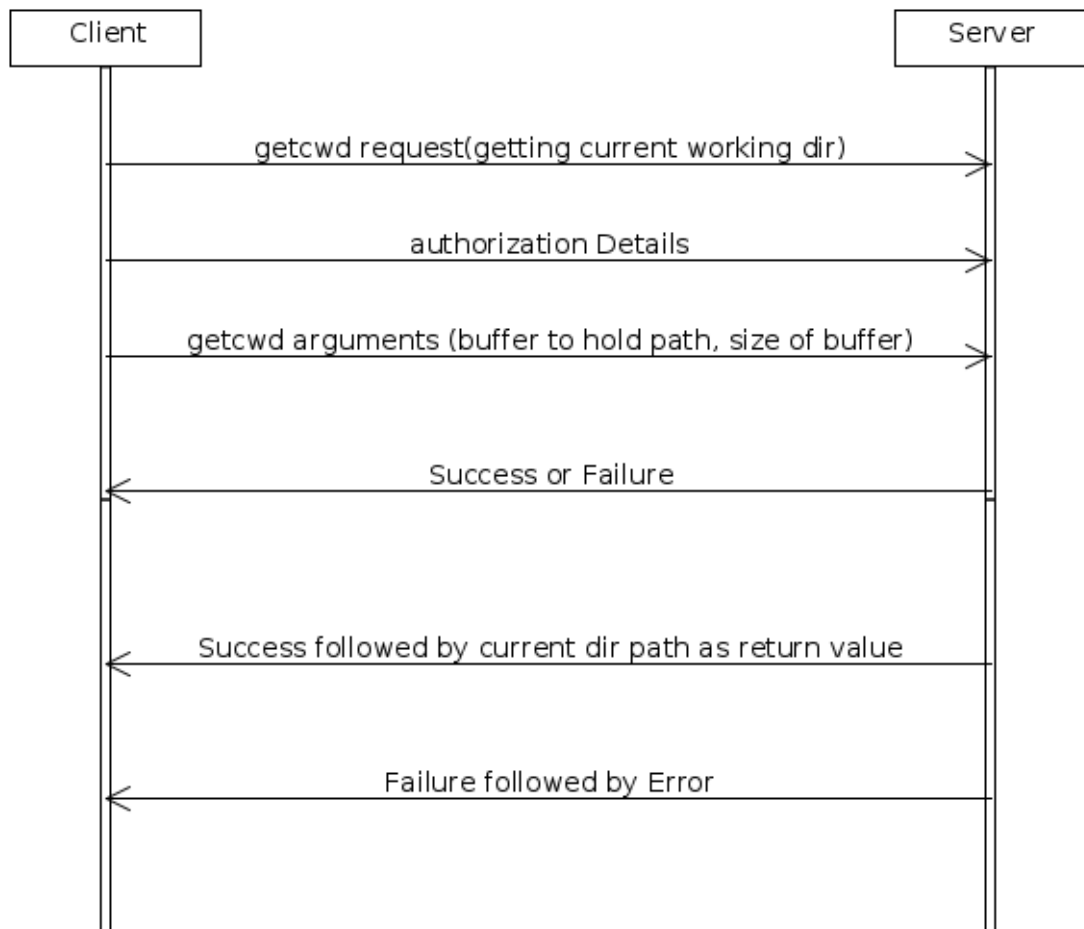


Fig 3.27 – `dfs_getcwd()` Protocol

3.6.15 Change the Current Working Directory Protocol

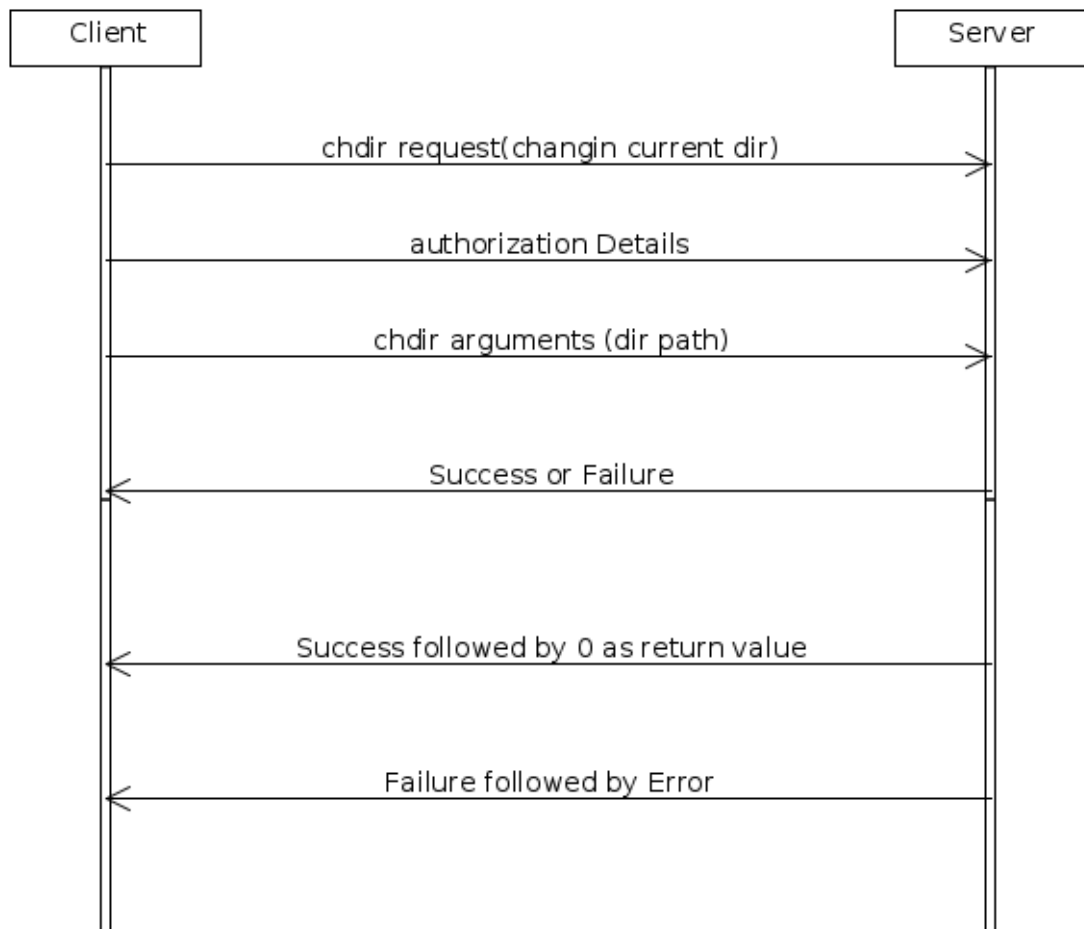


Fig 3.28 – dfs_chdir() Protocol

CHAPTER 4

Coding

This means program construction with procedural specifications has finished and the coding for the program begins:

- Once the design phase was over, coding commenced.
- Coding is natural consequence of design.
- Coding step translate a detailed design representation of software into a programming language realization.
- Main emphasis while coding was on style so that the end result was an optimized code.

The following points were kept into consideration while coding:

4.1 Coding Style

The Structured programming method was used in all the modules of the project. It incorporates the following methodologies:

- The code has been written so that the definition and implementation of each function is contained in one file.
- A group of related function was clubbed together in one file to include it when needed and save us from the labour of writing it again and again.

4.2 Naming Convention

- As the project size grows, so does the complexity of recognizing the purpose of the variables. Thus the variables were given meaningful names, which would help in understanding the context and the purpose of the variable.
- The function names are also given meaningful names that can be easily understood by the user.

4.3 Indentation

Judicious use of indentation can make the task of reading and understanding a program much simpler. Indentation is an essential part of a good program. If code is intended without thought it will seriously affect the readability of the program.

- The higher-level statements like the definition of the variables, constants and the function are indented, with each nested block indented, stating their purposes in the code.
- Blank line is also left between each function definition to make the code look neat.
- Indentation for each source file stating the purpose of the file is also done.

4.4 Source Code Core Modules

4.4.1 Code for Client Shell

```
/*
Copyright (C) <2012> <author sandeep nandal>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

** The author's original notes follow :-
**
** AUTHOR: Sandeep Nandal (sandeep@nandal.in)
** DATE: May 18, 2012
** WARRANTY: None, expressed, impressed, implied
**           or other
** STATUS: Public domain
*/
```

```

/*
    mydfs.c - it's the main file of DFS Client Shell containing the main
function
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "include/client.h"

myDFS_char  prompt[MAX_PATH_SIZE];

int main(int argc, char *argv[]){
    myDFS_int  i, x, fd;
    myDFS_char  pass[50];
    myDFS_char  choice[50], data[myDFS_BUFSIZ]="";
    myDFS_login_s  loginDetail;

    system("clear");
    dfs_initialize();

    create_path(&current_dir, "/");

    for(i=1; i<argc; i++){
        if(strcmp(argv[i], "-ip") == 0){
            strcpy(server_ip, argv[i+1]);
        }else if(strcmp(argv[i], "-p") == 0){
            server_port = atoi(argv[i+1]);
        }
    }

    printf("Welcome to DFS SERVER-[IP:%s]:[PORT:%i]\n", server_ip,
server_port);
go:
    printf("Enter Username:");
    fgets(user, MAX_USERNAME, stdin);
    user[strlen(user)-1] = '\0';
    strcpy(loginDetail.name, user);

```

```

printf("Enter Password:");
system("stty -echo");
fgets(loginDetail.pass, MAX_PASSWORD, stdin);
loginDetail.pass[strlen(loginDetail.pass)-1] = '\0';
system("stty echo");

if(dfs_login(&loginDetail)!=0){
    printf("Login Attempt Failed\n");
    goto go;
}
bzero(&loginDetail,sizeof(loginDetail));

if(strcmp(user, "root") == 0 && auth.uid == 0){
    strcpy(current_dir.path, "/");
    sprintf(prompt, "%s@DFS />#", user);
}else{
    sprintf(current_dir.path, "/home/%s/", user);
    sprintf(prompt, "%s@DFS ~/>$", user);
}
printf("\n");
while(1){
    printf("%s", prompt);
    fgets(choice, 50, stdin);
    choice[strlen(choice)-1] = '\0';
    if(strncmp(choice, "exit", 4) == 0)break;
    x = process(choice);
}

return 0;

}

```

4.4.2 Code for Request Processing Module of DFS Client Shell

```
/*
Copyright (C) <2012> <author sandeep nandal>
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

** The author's original notes follow :-
**
** AUTHOR: Sandeep Nandal (sandeep@nandal.in)
** DATE: May 18, 2012
** WARRANTY: None, expressed, impressed, implied
**           or other
** STATUS: Public domain
*/
/*
process.c - it's the processing module file of DFS Client Shell
containing the
functions to be invoked accordding to the request.
*/
#include "include/client.h"
#include <string.h>
#include <stdio.h>

int process(const myDFS_char *choice){
    int i;
    if(strncmp(choice, "logout", 6)==0){
        puts("Request to logout is sent.");
        dfs_logout();
    }else if(strncmp(choice, "status", 6)==0){
        puts("Request to show status is sent.");
    }
}
```

```

        dfs_status();
    }else if(strncmp(choice, "lsr", 3)==0){
        puts("Here is a list of remote files.");
        list_file(choice);
    }else if(strncmp(choice, "ls", 2)==0){
        puts("Here is a list of local files.");
        system(choice);
    }else if(strncmp(choice, "cat ", 4)==0){
        puts("The file is being download.");
        read_file(choice);
    }else if(strncmp(choice, "put ", 4)==0){
        puts("The file is being upload.");
        write_file(choice);
    }else if(strncmp(choice, "rmr ", 4)==0){
        puts("The remote file has been deleted.");
        delete_file(choice);
    }else if(strncmp(choice, "rm ", 3)==0){
        puts("The local file has been deleted.");
        system(choice);
    }else if(strncmp(choice, "mkdirr ", 7)==0){
        puts("The remote dir is created.");
        make_dir(choice);
    }else if(strncmp(choice, "mkdir ", 6)==0){
        puts("local dir is created.");
        system(choice);
    }else if(strncmp(choice, "rmdirr ", 7)==0){
        puts("removing remote dir request is sent.");
        remove_dir(choice);
    }else if(strncmp(choice, "rmdir ", 6)==0){
        puts("The removing local dir request is sent.");
        system(choice);
    }else if(strncmp(choice, "stop", 4)==0){
        puts("The server is asked to stop.");
        ask_server_stop();
    }else if(strncmp(choice, "mvr ", 4)==0){
        puts("The remote file is moved.");
        move_file(choice);
    }else if(strncmp(choice, "mv ", 3)==0){

```

```

        puts("The locat file is moved.");
        system(choice);
    }else if(strncmp(choice, "lnr ", 4)==0){
        puts("New link is being created on remote.");
        link_file(choice);
    }else if(strncmp(choice, "ln ", 3)==0){
        puts("New link is being created on local.");
        system(choice);
    }else if(strncmp(choice, "chmodr ", 7)==0){
        puts("Mode of file is beign chnged on remote");
        chmod_file(choice);
    }else if(strncmp(choice, "chmod ", 6)==0){
        puts("Mode of file is beign chnged on local");
        system(choice);
    }else if(strncmp(choice, "chownr ", 7)==0){
        puts("Ownership of file is being changed on remote.");
        chown_file(choice);
    }else if(strncmp(choice, "chown ", 6)==0){
        puts("Ownership of file is being changed locally.");
        system(choice);
    }else if(strncmp(choice, "pwdr", 4)==0){
        getcwd_dir();
        printf("%s\n", current_dir.path);
    }else if(strncmp(choice, "pwd", 3)==0){
        puts("Local Directory's full path.");
        system(choice);
    }else if(strncmp(choice, "cdr", 3)==0){
        puts("Remote Directory is changed.");
        change_dir(choice);
    }else if(strncmp(choice, "cd", 2)==0){
        puts("Local Directory is changed.");
        system(choice);
    }else if(strncmp(choice, "help", 4)==0){
        help();
    }else if(strncmp(choice, "clear", 5)==0){
        system("clear");
    }else{
        puts("Command not recognized.");
    }

```



```
        puts("Enter help for options.");  
    }  
    return 0;  
}
```

4.4.3 Code for DFS Server main Module

```
/*
Copyright (C) <2012> <author sandeep nandal>
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

** The author's original notes follow :-
**
** AUTHOR: Sandeep Nandal (sandeep@nandal.in)
** DATE: May 18, 2012
** WARRANTY: None, expressed, impressed, implied
**           or other
** STATUS: Public domain
*/
/*
server.c - it's the main file of DFS Server containing the main
function
*/
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include "include/server.h"

int main(int argc, char *argv[]){
    myDFS_int server_sockfd, client_sockfd;
```

```

myDFS_int    server_len, client_len;
myDFS_int    i=0, fd, x;
myDFS_msg    msg;
myDFS_node_s    node;

struct sockaddr_in server_address;
struct sockaddr_in client_address;

initialize_globals();

printf("Server IP : %s\n", SERVER_IP);
server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
if(server_sockfd == -1){
    chk("socket creation");
    return -1;
}

server_address.sin_family = AF_INET;
server_address.sin_addr.s_addr = inet_addr(SERVER_IP);
server_address.sin_port = htons(SERVER_PORT);

server_len = sizeof(server_address);
x = bind(server_sockfd, (struct sockaddr *)&server_address,
server_len);
if(x == -1){
    chk("binding of socket");
    return -1;
}

x = listen(server_sockfd, 5);
if(x == -1){
    chk("listening by socket");
    return -1;
}

fd = open("dataloggs/slave.tbl", O_RDONLY);
if(fd == -1){
    chk("opening slave.tbl");

```

```

        return -1;
    }
    for(i=0 ; i<TOTAL_SLAVES; i++){
        printf("Server waiting for slave[%d].\n", i);
        x = read(fd, &slaves[i], sizeof(myDFS_node_s));
        if(x == -1){
            chk("reading slave info from slave.tbl");
        }
        printf("Slave IP is : %s\n", slaves[i].ip);
        printf("Slave port is : %i\n", ntohs(slaves[i].port));
    }

    close(fd);

    /* Now server is ready with its slaves waiting for clients */
    while(1){
        printf("Server waiting.\n");
        client_len = sizeof(client_address);
        client_sockfd = accept(server_sockfd, (struct sockaddr
*)&client_address, &client_len);
        if(client_sockfd == -1){
            chk("accepting of client socket");
            return -1;
        }
        printf("Client IP is : %s\n",
inet_ntoa(client_address.sin_addr));
        printf("Client port is : %i\n",
ntohs(client_address.sin_port));
        strcpy(node.ip, inet_ntoa(client_address.sin_addr));
        node.port = ntohs(client_address.sin_port);
        x = read(client_sockfd, &msg, sizeof(myDFS_msg));
        if(x == -1){
            chk("reading from client socket");
            return -1;
        }
        if(msg == myDFS_MSG_LOGIN){
            printf("login request\n");
            login_file(client_sockfd, &node);
        }else{

```

```
        process_request(client_sockfd, msg);
    }
    close(client_sockfd);
}
return 0;
}
```

4.4.4 Code for DFS Server's Request Processing Module

```
/*
Copyright (C) <2012> <author sandeep nandal>
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

** The author's original notes follow :-
**
** AUTHOR: Sandeep Nandal (sandeep@nandal.in)
** DATE: May 18, 2012
** WARRANTY: None, expressed, impressed, implied
**           or other
** STATUS: Public domain
*/
/*
process_requests.c - it's the processing module file of DFS Server
containing the
functions to be invoked accordding to the request.
*/
#include <stdio.h>
#include <stdlib.h>
#include "../include/dfs_server.h"

int process_request(myDFS_int sockfd, const myDFS_msg chkmsg){

    myDFS_int    x;
    myDFS_msg    msg;
    myDFS_auth_s  auth;

    msg = myDFS_MSG_FAILURE;
```

```

printf("myDFS_msg = %d\n", chkmsg);

/* reading accessing details of user */

x = read(sockfd, &auth, sizeof(auth));
if(x == -1){
    msg = myDFS_MSG_FAILURE;
    chk("reading authorization from socket");
    write(sockfd, &msg, sizeof(msg));
    return -1;
}
x = access_permission(&auth, msg);
if(x != 0){
    printf("Access Denied\n");
    msg = myDFS_MSG_FAILURE;
    write(sockfd, &msg, sizeof(msg));
    return -1;
}

/* PROCESS REQUEST */
switch(chkmsg){

/* OPEN REQUEST */
case myDFS_MSG_OPEN:{
    puts("request of open file function.");
    x = open_file(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}
break;

/* CLOSE REQUEST */
case myDFS_MSG_CLOSE:{
    puts("request to close file discriptor.");
    x = close_file(sockfd, &auth);
    if(x == -1){

```

```

        chk("reading of file failed");
    }
}
break;

/* READ REQUEST */
case myDFS_MSG_READ:{
    x = read_file(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}
break;

/* WRITE REQUEST */
case myDFS_MSG_WRITE:{
    x = write_file(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}
break;

/* LSEEK REQUEST */
case myDFS_MSG_LSEEK:{
    x = lseek_file(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}
break;

/* CREAT REQUEST */
case myDFS_MSG_CREAT:{
    x = creat_file(sockfd, &auth);
    if(x == -1){

```



```

        chk("reading of file failed");
    }
}
break;

/* CHDIR REQUEST */
case myDFS_MSG_CHDIR:{
    x = chdir_file(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}
break;

/* GETCWD REQUEST */
case myDFS_MSG_GETCWD:{
    x = getcwd_file(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}
break;

/* CHMOD REQUEST */
case myDFS_MSG_CHMOD:{
    x = chmod_file(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}
break;

/* CHOWN REQUEST */
case myDFS_MSG_CHOWN:{
    x = chown_file(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}

```

```

        }
    }
    break;

/* LINK REQUEST */
case myDFS_MSG_LINK:{
    x = link_file(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}
break;

/* MKDIR REQUEST */
case myDFS_MSG_MKDIR:{
    x = mkdir_file(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}
break;

/* RMDIR REQUEST */
case myDFS_MSG_RMDIR:{
    x = rmdir_file(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}
break;

/* UNLINK REQUEST */
case myDFS_MSG_UNLINK:{
    x = unlink_file(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}
}

```

```

        break;

/* STOP REQUEST */
case myDFS_MSG_STOP:{
    x = stop_slaves(&auth);
    if(x == -1){
        printf("An error occurred.\n");
        perror("process_request.c:52");
        msg = myDFS_MSG_FAILURE;
        x = write(sockfd, &msg, sizeof(msg));
        if(x == -1){
            chk("writing to socket");
            return -1;
        }
    }
    msg = myDFS_MSG_SUCCESS;
    x = write(sockfd, &msg, sizeof(msg));
    if(x == -1){
        chk("writing to socket");
        return -1;
    }
    close(sockfd);
    exit(0);
}
break;

/* FORMAT REQUEST */
case myDFS_MSG_FORMAT:{
    msg = myDFS_MSG_SUCCESS;
    x = write(sockfd, &msg, sizeof(msg));
    if(x == -1){
        chk("writing to socket");
        return -1;
    }
    x = format(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}

```

```

        }
        break;

/* LOGOUT REQUEST */
case myDFS_MSG_LOGOUT:{
    if(x == -1){
        chk("writing to socket");
        return -1;
    }
    x = logout_file(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}
break;

/* STATUS REQUEST */
case myDFS_MSG_STATUS:{
    printf("Request for showing status\n");
    if(x == -1){
        chk("writing to socket");
        return -1;
    }
    x = status_file(sockfd, &auth);
    if(x == -1){
        chk("reading of file failed");
    }
}
break;
}
return 0;
}

```

4.4.5 Code for DFS Slave main Module

```
/*
Copyright (C) <2012> <author sandeep nandal>
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

** The author's original notes follow :-
**
** AUTHOR: Sandeep Nandal (sandeep@nandal.in)
** DATE: May 18, 2012
** WARRANTY: None, expressed, impressed, implied
**           or other
** STATUS: Public domain
*/
/*
  slave.c - it's the main module file of DFS Slave containing the
  main function of slave Module.
*/
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include "header/slave.h"
```

```

int main(int argc, char *argv[]){
    myDFS_int    server_sockfd, client_sockfd;
    myDFS_int    server_len, client_len;
    myDFS_node_s    slave_node;
    myDFS_int    x;
    myDFS_msg    msg;

    struct sockaddr_in client_address;
    struct sockaddr_in server_address;

    slave_read_config(&slave_node);
    slave_make_entry_slave_list(&slave_node);
    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(server_sockfd == -1){
        chk("server socket creationg");
        return -1;
    }

    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = inet_addr(slave_node.ip);
    server_address.sin_port = htons(slave_node.port);
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address,
server_len);
    listen(server_sockfd, 5);
    while(1){
        printf("slave is listening to IP : %s\n", slave_node.ip);
        printf("Port of slave is : %i\n", slave_node.port);

        client_len = sizeof(client_address);
        client_sockfd = accept(server_sockfd, (struct sockaddr
*)&client_address, &client_len);
        printf("Client IP is : %s\n",
inet_ntoa(client_address.sin_addr));
        printf("Client port is : %i\n",
ntohs(client_address.sin_port));
        read(client_sockfd, &msg, sizeof(myDFS_msg));
        perror("reading message req");
    }
}

```

```
        slave_process_request(client_sockfd, msg);
        close(client_sockfd);
    }
    return 0;
}
```

4.4.6 Code for DFS Slave's Request Processing Module

```
/*
Copyright (C) <2012> <author sandeep nandal>
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

** The author's original notes follow :-
**
** AUTHOR: Sandeep Nandal (sandeep@nandal.in)
** DATE: May 18, 2012
** WARRANTY: None, expressed, impressed, implied
**           or other
** STATUS: Public domain
*/
/*
  slave_process_request.c - it's the processing module file of DFS
  Slave containing the
  functions to be invoked accordding to the request.
*/
#include <stdio.h>
#include <stdlib.h>
#include "header/slave.h"

int slave_process_request(myDFS_int sockfd, const myDFS_msg msg_ptr){
    myDFS_int x = 0;
    myDFS_msg msg;

    msg = myDFS_MSG_FAILURE;
    printf("myDFS_msg %d\n", msg);
```



```

switch(msg_ptr){
case myDFS_MSG_READ:{
    perror("process_request:12:read");
    slave_read_data(sockfd);
    msg = myDFS_MSG_SUCCESS;
    }
    break;
case myDFS_MSG_WRITE:{
    slave_write_data(sockfd);
    msg = myDFS_MSG_SUCCESS;
    write(sockfd, &msg, sizeof(myDFS_msg));
    }
    break;
case myDFS_MSG_STOP:{
    printf("Asked to stop by server");
    msg = myDFS_MSG_SUCCESS;
    write(sockfd, &msg, sizeof(myDFS_msg));
    exit(0);
    }
    break;
case myDFS_MSG_FORMAT:{
    printf("Server asked to format the system.\n");
    msg = myDFS_MSG_SUCCESS;
    }
    break;
}
return 0;
}

```

4.4.7 Header File Containing all the Data Types Used for DFS

```
/*
Copyright (C) <2012> <author sandeep nandal>
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

** The author's original notes follow :-
**
** AUTHOR: Sandeep Nandal (sandeep@nandal.in)
** DATE: May 18, 2012
** WARRANTY: None, expressed, impressed, implied
** or other
** STATUS: Public domain
*/

/* myDFS Data Types to be used through out the whole system */
#include <time.h>
#ifndef myDFS_DATA_TYPES_
#define myDFS_DATA_TYPES_
    typedef unsigned char    myDFS_uchar;
    typedef signed char      myDFS_char;
    typedef unsigned short   myDFS_ushort;
    typedef signed short     myDFS_short;
    typedef unsigned int     myDFS_uint;
    typedef signed int       myDFS_int;
    typedef unsigned long    myDFS_ulong;
    typedef signed long      myDFS_long;
    typedef double           myDFS_double;
    typedef float            myDFS_float;

    typedef myDFS_uint       myDFS_offset;

```

```

typedef myDFS_uint      myDFS_size;
typedef myDFS_int myDFS_ssize;
typedef myDFS_ushort    myDFS_dev;
typedef myDFS_uchar     myDFS_nlink;
typedef myDFS_ushort    myDFS_gid;
typedef myDFS_ushort    myDFS_uid;
typedef myDFS_ushort    myDFS_mode;
typedef myDFS_uint      myDFS_inode;
typedef myDFS_uchar     myDFS_status;
typedef myDFS_uint      myDFS_blk;
typedef myDFS_ushort    myDFS_id;
typedef myDFS_uint      myDFS_iid;
typedef myDFS_ulong     myDFS_lid;
typedef time_t          myDFS_time;
typedef myDFS_int myDFS_key;
typedef myDFS_ushort    myDFS_msg;
typedef myDFS_int myDFS_ERROR;
typedef myDFS_int myDFS_flag;

/* COMMON #defines n Constants */
#define MAX_IP          30
#define SERVER_IP       "127.0.0.1"
#define SERVER_PORT     7345
#define SLAVE_PORT      7346

#define MAX_USERNAME    50
#define MAX_PASSWORD    50
#define MAX_GROUPNAME   50
#define MAX_FILE_NAME   256
#define MAX_PATH_SIZE   1024 /* no of characters a path contain
*/

#define MAX_FILE_SIZE   1024 /* no of BLKSIZ (1GB) */
#define MAX_NO_FILES    1024 /* 32 bit inode identifier */
#define MAX_DIR_ENTRY   1024 /* must be same to MAX_FILE_SIZE
no of blks*/

#define MAX_FILE_OPEN   1024 /* MAX FILE CAN BE OPENED IN THE
SYSTEM AT ONE TIME */

```

```

#define MAX_FD_PER_USER 256
#define MAX_FD          1024
#define MAX_SESSIONS    1024


#define myDFS_BUFSIZ     256
#define myDFS_BLKSIIZ    1024
#define myDFS_CYCLES     myDFS_BLKSIIZ / myDFS_BUFSIZ


#define myDFS_REG_FILE   1
#define myDFS_DIR_FILE   0
#define myDFS_LNK_FILE   2


#define myDFS_EOF_ "myDFS_EOF_"


typedef struct{
    myDFS_char ip[MAX_IP];
    myDFS_short port;
}myDFS_node_s;


typedef struct{
    myDFS_char name[MAX_FILE_NAME];
    myDFS_size size;
    myDFS_time atime;
    myDFS_time ctime;
    myDFS_time mtime;
    myDFS_mode mode;
    myDFS_uid uid;
    myDFS_gid gid;
}myDFS_fhead_s;
typedef struct{
    myDFS_char path[MAX_PATH_SIZE];
    myDFS_char type;
    myDFS_char depth;
}myDFS_path_s;
#define myDFS_FULL_PATH 0
#define myDFS_REL_PATH 1
#endif

```

4.4.8 Header File Containing all the prototypes of the core functions of DFS

```
/*
Copyright (C) <2012> <author sandeep nandal>
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

** The author's original notes follow :-
**
** AUTHOR: Sandeep Nandal (sandeep@nandal.in)
** DATE: May 18, 2012
** WARRANTY: None, expressed, impressed, implied
**           or other
** STATUS: Public domain
*/
/* dfs_client.h - this is the file containing all the functions
prototypes
which can be used in any applicationg by the user through DFS File
System */

#include "/HD/storage/project/myDFS/common/include/common.h"
#include "/HD/storage/project/myDFS/common/include/myDFS_msg.h"
#include "/HD/storage/project/myDFS/common/include/myDFS_error.h"

#ifndef _GLOBAL_CLIENT_H_
#define _GLOBAL_CLIENT_H_
extern myDFS_char user[MAX_USERNAME];
extern myDFS_auth_s auth; /* used as
authorization certificate of the user */
extern myDFS_char server_ip[MAX_IP];
extern myDFS_short server_port;
```

```

extern      myDFS_path_s      current_dir;
extern      myDFS_ERROR ERROR;
#endif

myDFS_int dfs_open(const myDFS_char *path, myDFS_int flags,
myDFS_mode mode);
myDFS_int dfs_close(myDFS_int fd);
myDFS_ssize dfs_write(myDFS_int fd, const myDFS_char *buf, myDFS_int
len);
myDFS_ssize dfs_read(myDFS_int fd, myDFS_char *buf, myDFS_int len);
myDFS_int dfs_creat(const myDFS_char *path, myDFS_mode mode);
myDFS_offset dfs_lseek(myDFS_int fd, myDFS_offset offset, myDFS_int
ref);

myDFS_int dfs_chdir(const myDFS_char *path);
myDFS_int dfs_chmod(const myDFS_char *path, myDFS_mode mode);
myDFS_int dfs_chown(const myDFS_char *path, myDFS_uid owner,
myDFS_gid group);
myDFS_int dfs_link(const myDFS_char *oldpath, const myDFS_char
*newpath);
myDFS_int dfs_mkdir(const myDFS_char *path, myDFS_mode mode);
myDFS_int dfs_rmdir(const myDFS_char *path);
myDFS_int dfs_unlink(const myDFS_char *path);
myDFS_char *dfs_getcwd(myDFS_char *buf, myDFS_size size);

myDFS_int dfs_login(const myDFS_login_s *login);
myDFS_int dfs_logout();

```

Note – As stated earlier the project comes with GNU General Public License Version 3. Therefore the complete source code is open for every user. The complete source code of the project can be downloaded from our website www.dfs.osslabs.org from download section. The details of GNU General Public License can be obtained from www.fsf.org or www.gnu.org.

CHAPTER 5

Testing

Software testing is a critical element of software quality assurance and represents the Ultimate review of specification, design, and coding .The purpose of product testing is to verify and validate the various work products viz. units, integrated unit, final product to ensure that they meet their respective requirements.

Testing Objectives: -

- Testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an as – yet undiscovered error.
- A successful test is one that uncover an as – yet undiscovered error.

Our objective is to design tests that systematically uncover different classes of errors and do so with a minimum amount of time and effort.

This process has two parts :

- Planning : This involves writing and reviewing unit, integration, functional, validation and acceptance test plans.
- Execution : This involves executing these test plans, measuring, collecting data and verifying if it meets the quality criteria set in the Quality Plan chapter of PMP. Data collected is used to make appropriate changes in the plans related to development and testing.

5.1 History of Testing

The separation of debugging from testing was initially introduced by Glenford J. Myers in 1979. Although his attention was on breakage testing ("a successful test is one that finds a bug") it illustrated the desire of the software engineering community to separate fundamental development activities, such as debugging, from that of verification. Dave Gelperin and William C. Hetzel classified in 1988 the phases and goals in software testing in the following stages:

- Until 1956 - Debugging oriented
- 1957–1978 - Demonstration oriented

- 1979–1982 - Destruction oriented
- 1983–1987 - Evaluation oriented
- 1988–2000 - Prevention oriented

5.2 Scope of Testing

A primary purpose for testing is to detect software failures so that defects may be uncovered and corrected. This is a non-trivial pursuit. Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions.

The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do. In the current culture of software development, a testing organization may be separate from the development team. There are various roles for testing team members. Information derived from software testing may be used to correct the process by which software is developed.

5.3 Functional vs. Non-Functional Testing

Functional testing refers to tests that verify a specific action or function of the code. These are usually found in the code requirements documentation, although some development methodologies work from use cases or user stories. Functional tests tend to answer the question of "can the user do this" or "does this particular feature work".

Non-functional testing refers to aspects of the software that may not be related to a specific function or user action, such as scalability or security. Non-functional testing tends to answer such questions as "how many people can log in at once", or "how easy is it to hack this software".

5.4 Defects and Failures

Not all software defects are caused by coding errors. One common source of expensive defects is caused by requirement gaps, e.g., unrecognized requirements,

that result in errors of omission by the program designer. A common source of requirements gaps is non-functional requirements such as testability, scalability, maintainability, usability, performance, and security.

Software faults occur through the following processes. A programmer makes an error (mistake), which results in a defect (fault, bug) in the software source code. If this defect is executed, in certain situations the system will produce wrong results, causing a failure. Not all defects will necessarily result in failures. For example, defects in dead code will never result in failures. A defect can turn into a failure when the environment is changed. Examples of these changes in environment include the software being run on a new hardware platform, alterations in source data or interacting with different software. A single defect may result in a wide range of failure symptoms.

5.5 Static vs. Dynamic Testing

There are many approaches to software testing. Reviews, walkthroughs, or inspections are considered as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing. Static testing can be (and unfortunately in practice often is) omitted. Dynamic testing takes place when the program itself is used for the first time (which is generally considered the beginning of the testing stage). Dynamic testing may begin before the program is 100% complete in order to test particular sections of code (modules or discrete functions). Typical techniques for this are either using stubs/drivers or execution from a debugger environment. For example, Spreadsheet programs are, by their very nature, tested to a large extent interactively ("on the fly"), with results displayed immediately after each calculation or text manipulation.

5.6 Software Verification and Validation

Software testing is used in association with verification and validation: Verification: Have we built the software right? (i.e., does it match the specification). Validation: Have we built the right software? (i.e., is this what the customer wants). The terms verification and validation are commonly used interchangeably in the industry; it is also common to see these two terms incorrectly defined. According to

the IEEE Standard Glossary of Software Engineering Terminology:

- Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
- Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

5.7 The Software Testing Team

Software testing can be done by software testers. Until the 1980s the term "software tester" was used generally, but later it was also seen as a separate profession. Regarding the periods and the different goals in software testing, different roles have been established: manager, test lead, test designer, tester, automation developer, and test administrator.

5.8 Software Quality Assurance (SQA)

Though controversial, software testing may be viewed as an important part of the software quality assurance (SQA) process. In SQA, software process specialists and auditors take a broader view on software and its development. They examine and change the software engineering process itself to reduce the amount of faults that end up in the delivered software: the so-called defect rate.

Software Testing is a task intended to detect defects in software by contrasting a computer program's expected results with its actual results for a given set of inputs. By contrast, QA (Quality Assurance) is the implementation of policies and procedures intended to prevent defects from occurring in the first place.

5.9 Testing methods

5.9.1 Black Box Testing

Black box testing treats the software as a "black box"—without any knowledge of internal implementation. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix, exploratory testing and specification-based

testing.

Specification-based testing: Specification-based testing aims to test the functionality of software according to the applicable requirements. Thus, the tester inputs data into, and only sees the output from, the test object. This level of testing usually requires thorough test cases to be provided to the tester, who then can simply verify that for a given input, the output value (or behavior), either "is" or "is not" the same as the expected value specified in the test case.

Specification-based testing is necessary, but it is insufficient to guard against certain risks.

Advantages and disadvantages: The black box tester has no "bonds" with the code, and a tester's perception is very simple: a code must have bugs. Using the principle, "Ask and you shall receive," black box testers find bugs where programmers do not. But, on the other hand, black box testing has been said to be "like a walk in a dark labyrinth without a flashlight," because the tester doesn't know how the software being tested was actually constructed. As a result, there are situations when a tester writes many test cases to check something that could have been tested by only one test case, and/or some parts of the back-end are not tested at all.

Therefore, black box testing has the advantage of "an unaffiliated opinion," on the one hand, and the disadvantage of "blind exploring," on the other.

5.9.2 White Box Testing

White box testing is when the tester has access to the internal data structures and algorithms including the code that implement these.

The following types of white box testing exist:

- API testing (application programming interface) - Testing of the application using Public and Private APIs
- Code coverage - creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)
- Fault injection methods - improving the coverage of a test by introducing faults to test code paths

- Mutation testing methods
- Static testing - White box testing includes all static testing

5.9.3 Test Coverage

White box testing methods can also be used to evaluate the completeness of a test suite that was created with black box testing methods. This allows the software team to examine parts of a system that are rarely tested and ensures that the most important function points have been tested.

Two common forms of code coverage are:

- Function coverage, which reports on functions executed
- Statement coverage, which reports on the number of lines executed to complete the test

They both return a code coverage metric, measured as a percentage.

5.9.4 Grey Box Testing

Grey box testing (American spelling: Gray box testing) involves having access to internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level. Manipulating input data and formatting output do not qualify as grey box, because the input and output are clearly outside of the "black-box" that we are calling the system under test. This distinction is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test. However, modifying a data repository does qualify as grey box, as the user would not normally be able to change the data outside of the system under test. Grey box testing may also include reverse engineering to determine, for instance, boundary values or error messages.

5.9.5 Unit Testing

Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.

These type of tests are usually written by developers as they work on code

(white-box style), to ensure that the specific function is working as expected. One function might have multiple tests, to catch corner cases or other branches in the code. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to assure that the building blocks the software uses work independently of each other. Unit testing is also called Component Testing.

5.9.6 Integration Testing

Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design. Software components may be integrated in an iterative way or all together ("big bang"). Normally the former is considered a better practice since it allows interface issues to be localised more quickly and fixed.

Integration testing works to expose defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.

5.9.7 Regression Testing

Regression testing focuses on finding defects after a major code change has occurred. Specifically, it seeks to uncover software regressions, or old bugs that have come back. Such regressions occur whenever software functionality that was previously working correctly stops working as intended. Typically, regressions occur as an unintended consequence of program changes, when the newly developed part of the software collides with the previously existing code. Common methods of regression testing include re-running previously run tests and checking whether previously fixed faults have re-emerged. The depth of testing depends on the phase in the release process and the risk of the added features. They can either be complete, for changes added late in the release or deemed to be risky, to very shallow, consisting of positive tests on each feature, if the changes are early in the release or deemed to be of low risk.

5.9.8 Acceptance Testing

Acceptance testing can mean one of two things:

- A smoke test is used as an acceptance test prior to introducing a new build to the main testing process, i.e. before integration or regression.
- Acceptance testing performed by the customer, often in their lab environment on their own HW, is known as user acceptance testing (UAT). Acceptance testing may be performed as part of the hand-off process between any two phases of development.

5.9.9 Alpha Testing

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

5.9.10 Beta Testing

Beta testing comes after alpha testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

Special methods exist to test non-functional aspects of software. In contrast to functional testing, which establishes the correct operation of the software (correct in that it matches the expected behavior defined in the design requirements), non-functional testing verifies that the software functions properly even when it receives invalid or unexpected inputs. Software fault injection, in the form of fuzzing, is an example of non-functional testing. Non-functional testing, especially for software, is designed to establish whether the device under test can tolerate invalid or unexpected inputs, thereby establishing the robustness of input validation routines as well as error-handling routines. Various commercial non-functional testing tools are linked from the Software fault injection page; there are also numerous open-source and free software tools available that perform non-functional testing.

5.9.11 Software Performance Testing and Load Testing

Performance testing is executed to determine how fast a system or sub-system performs under a particular workload. It can also serve to validate and verify other quality attributes of the system, such as scalability, reliability and resource usage. Load testing is primarily concerned with testing that can continue to operate under a specific load, whether that be large quantities of data or a large number of users. This is generally referred to as software scalability. The related load testing activity of when performed as a non-functional activity is often referred to as Endurance Testing.

5.9.12 Volume Testing

Volume testing is a way to test functionality. Stress testing is a way to test reliability. Load testing is a way to test performance. There is little agreement on what the specific goals of load testing are. The terms load testing, performance testing, reliability testing, and volume testing, are often used interchangeably.

5.9.13 Stability Testing

Stability testing checks to see if the software can continuously function well in or above an acceptable period. This activity of Non Functional Software Testing is oftentimes referred to as load (or endurance) testing.

5.9.14 Usability Testing

Usability testing is needed to check if the user interface is easy to use and understand.

5.9.15 Security Testing

Security testing is essential for software that processes confidential data to prevent system intrusion by hackers.

5.10 The Testing Process

Traditional CMMI or waterfall development model, A common practice of software testing is that testing is performed by an independent group of testers after the functionality is developed, before it is shipped to the customer. This practice often

results in the testing phase being used as a project buffer to compensate for project delays, thereby compromising the time devoted to testing.

While automation cannot reproduce everything that a human can do (and all the strange ways they think of doing it), it can be very useful for regression testing. However, it does require a well-developed test suite of testing scripts in order to be truly useful.

5.11 Testing Tools

Program testing and fault detection can be aided significantly by testing tools and debuggers. Testing/debug tools include features such as: Program monitors, permitting full or partial monitoring of program code including: Instruction Set Simulator, emitting complete instruction level monitoring and trace facilities. Program animation, permitting step-by-step execution and conditional breakpoint at source level or in machine code.

CHAPTER 6

Implementation & Evaluation of Project

At last the programmer does the Maintenance of the project. During implementation of the project some errors or mistake can be left in the project so removing these or adding some extra things to the project is called Maintenance. Mainly there are following types of the Maintenance:-

- Corrective Maintenance
- Adaptive Maintenance

It basically includes the following activity:

- Correcting design errors
- Correcting errors
- Updating, documentation and test data
- Adding modifying or redeveloping the code
- Regular acceptance and validation testing

6.1 Login Screen of DFS Client



```
nandal@u-techexpert: /HD/storage/project/myDFS/client
File Edit View Search Terminal Help

*****
*           DFS - Distributed File System           *
* This product comes with GNU General Public License (Version 3) *
*****
Welcome to DFS SERVER-[IP:127.0.0.1]:[PORT:7345]
Enter Username:

□
```

Fig 6.1 - DFS Client Login Screen

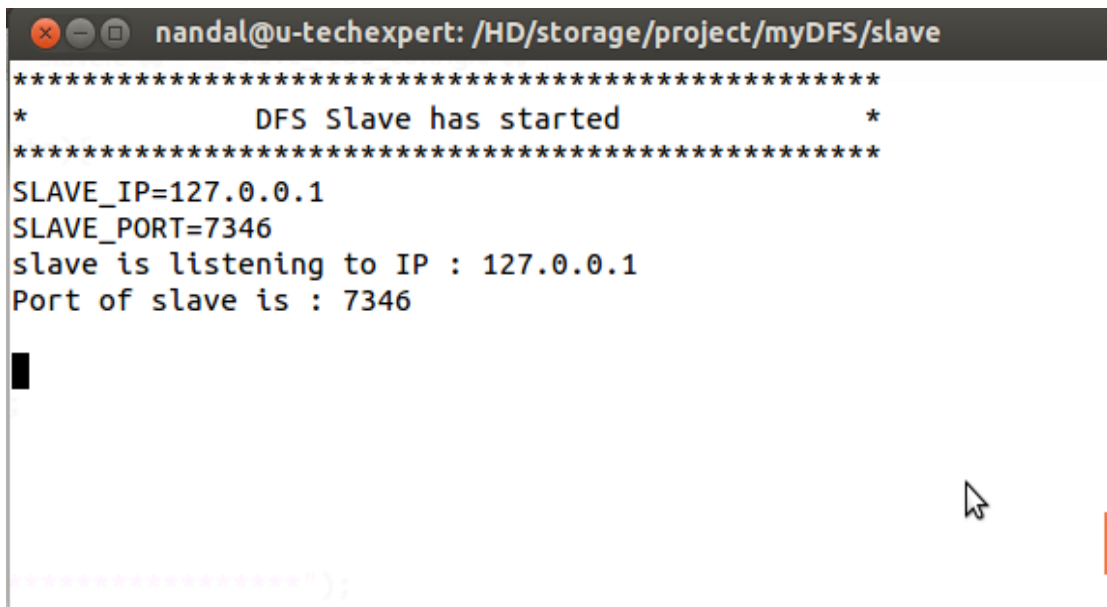
6.2 Welcome Screen of DFS Client

A terminal window titled 'nandal@u-techexpert: /HD/storage/project/myDFS/client'. The output shows a welcome message for the DFS - Distributed File System, including the GNU General Public License (Version 3) notice. It prompts for a username (root) and password, then displays the UID (0) and KEY (1804289383). The prompt 'root@DFS />#' is shown at the end.

```
nandal@u-techexpert: /HD/storage/project/myDFS/client
*****
*                DFS - Distributed File System                *
* This product comes with GNU General Public License (Version 3) *
*****
Welcome to DFS SERVER-[IP:127.0.0.1]:[PORT:7345]
Enter Username:root
Enter Password:
UID : 0
KEY : 1804289383
root@DFS />#
```

Fig. 6.2 – DFS Client Welcome Screen

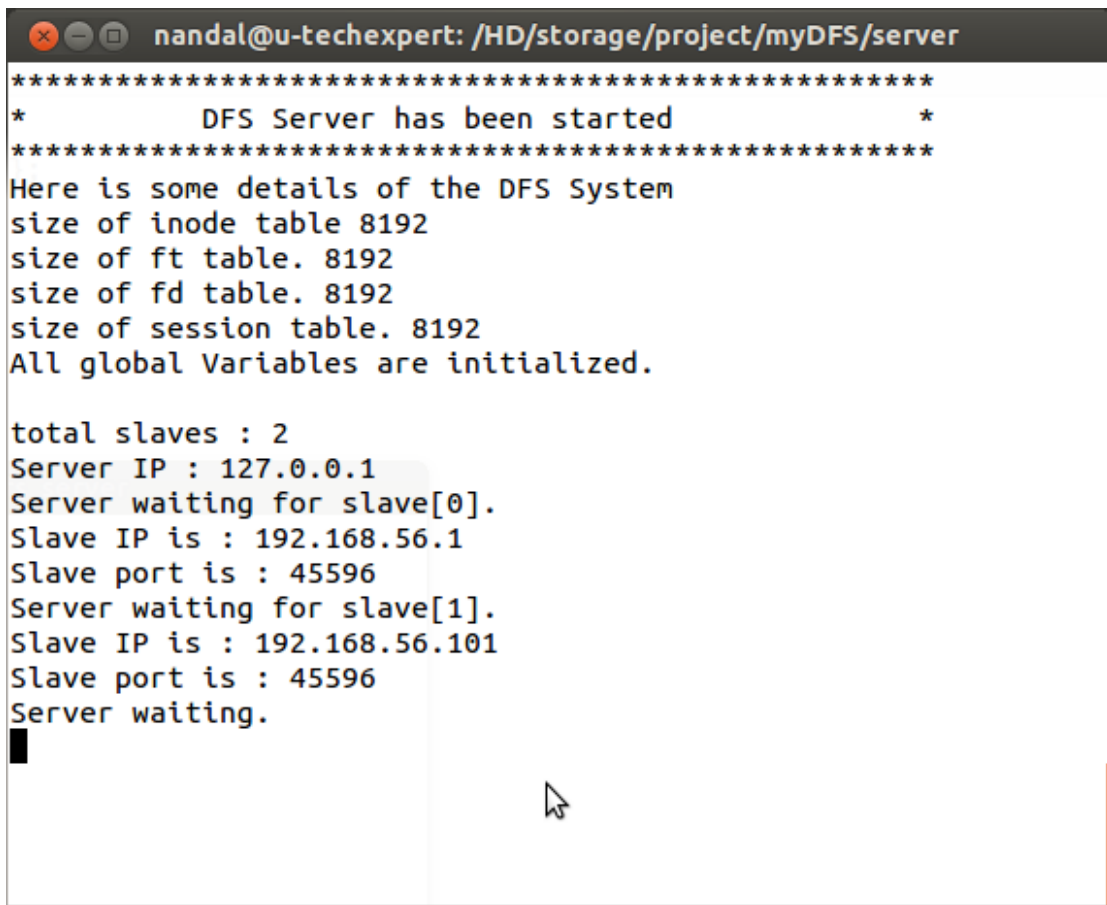
6.3 Welcome Screen of DFS Slave

A terminal window titled 'nandal@u-techexpert: /HD/storage/project/myDFS/slave'. The output shows a message 'DFS Slave has started' followed by configuration details: SLAVE_IP=127.0.0.1, SLAVE_PORT=7346, and a confirmation that the slave is listening to IP 127.0.0.1 on port 7346. A cursor is visible at the bottom left.

```
nandal@u-techexpert: /HD/storage/project/myDFS/slave
*****
*                DFS Slave has started                *
*****
SLAVE_IP=127.0.0.1
SLAVE_PORT=7346
slave is listening to IP : 127.0.0.1
Port of slave is : 7346
█
```

Fig. 6.4 – DFS Slave Welcome Screen

6.4 Welcome Screen of DFS Server

A terminal window with a dark title bar containing the text 'nandal@u-techexpert: /HD/storage/project/myDFS/server'. The terminal output displays the DFS Server welcome screen, which includes a star separator, a confirmation message, system details, slave information, and a cursor at the bottom.

```
nandal@u-techexpert: /HD/storage/project/myDFS/server
*****
*           DFS Server has been started           *
*****
Here is some details of the DFS System
size of inode table 8192
size of ft table. 8192
size of fd table. 8192
size of session table. 8192
All global Variables are initialized.

total slaves : 2
Server IP : 127.0.0.1
Server waiting for slave[0].
Slave IP is : 192.168.56.1
Slave port is : 45596
Server waiting for slave[1].
Slave IP is : 192.168.56.101
Slave port is : 45596
Server waiting.
█
```

Fig. 6.3 – DFS Server Welcome Screen

CHAPTER 7

Conclusion & Future Scope of the project

7.1 Future Scope of the project

Scope of this project is in all types of computing environments. With help of the project, problem of manual work is overcome. Enormously large data chunks can be handled easily. We have implemented all the core file handling functions in DFS and hence it can easily be integrated in any project or application by the users.

Distributed File System Project can be used as a stand alone application for storing very large files of size in Terabytes over different small storage devices connected through the network. Or it can be used as an OS Module for File Management in Grid Computing OS. With the help of this project we don't need to be restricted ourselves to the maximum size available of a storage device for storing an enormously large file. We can combine many small storage devices through DFS and can have the feel of a single big chunk of storage space of single address space logically. Therefore, we can make use of the old hardware which we consider obsolete in today's high demanding computational world.

7.2 Conclusion

The system has been developed for the given condition and is found working effectively. The developed system is flexible and changes whenever can be made easy. Using the facilities and functionalities of C language, the software has been developed in a neat and simple manner, thereby reducing the operators work.

The speed and accuracy are maintained in proper way. The user friendly nature of this software developed in C language is very easy to work with both for the higher management as well as other employees with little knowledge of computer. The results obtained were fully satisfactory from the user point of view.

The system was verified with valid as well as invalid data in each manner. The system is run with an insight into the necessary modifications that may require in the future. Hence the system can be maintained successfully without much network.

Any Application Developer who wishes to deal with enormously large data

chunks can use the DFS for the easy handling of such issues related with the large data chunks and can easily create the effective applications.

There have been 5KLOC of code has been written and tested in C Language for the project for providing the basic functionality of Distributed File System. For making the project a Product, many KLOC code has to written for error checkings and exceptions handling when the product goes in use.

As the Distributed File System has been built using the C language, as a result we get the compiled code of DFS to execute which need not to be interpreted by an Interpreter. The code of DFS is executed by the machine instead of being interpreted, hence we get high performance.

The core bottle neck for the Distributed File System is the speed of the LAN connection we use, because all the data is transferred over the network from one machine to the other. Hence, DFS will desire High Speed LAN for high efficiency.

DFS – The Distributed File System has is free software: it can be redistributed and/or modified by anyone under the terms of GNU General Public License. GNU General Public License's details can be found on the website of the Free Software Foundation – www.fsf.com. Making the DFS a free software will increase the chances of it's wide usability. And developing DFS through a community development platform will expedite it's development pace and additions of new functionalities.

Bibliography

8.1 Books

- The ANSI C Programming Language
by Brian W. Kernighan, Dennis M. Ritchie
- The Complete Reference C 4th Edition
by Herbert Schildt
- Linux for Programmers and Users
by Graham Glass, King Ables
- Beginning Linux Programming 3rd Edition
by Neil Matthew, Richard Stones
- Linux Socket Programming by Example
by Warren W. Gay
- The Design of the UNIX Operating System
by Maurice J. Bach

8.2 Websites & URLs

- Wikipedia (www.wikipedia.org) for various concepts.
- The LDP – The Linux Documentation Development Project (www.tldp.org)
for various Linux related documentations regarding : -
 - The Linux Programmer's Guide - <http://tldp.org/LDP/lpg/index.html>
 - Linux Command Line Tools Summary - <http://tldp.org/LDP/GNU-Linux-Tools-Summary/html/index.html>