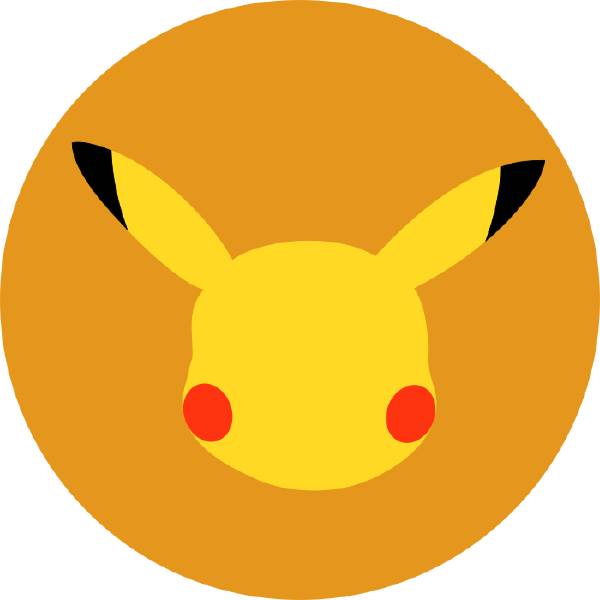
|  |
| --- |
| FCS -IBA- Karachi |
| Peek-achu Distributed FS |
| Distributed Systems Project Report |
| **Syed Zain Zaidi - 11476** |
| **BSCS-VII** |
| **01-Dec-18** |



|  |
| --- |
| Peek-achu DFS is a simulation of a Distributed File System with a high degree of transparency. Users may connect to the DFS and perform file creation, manipulation and deletion without having knowledge of the physical attributes of the files. The system provides a degree of fault tolerance with replication. |

# Introduction

The goal of this project was to design a software simulation of a Distributed File System. When a user connects to the system, he gets the impression of a single machine that is hosting all the files and directories. The user is capable of creating, manipulating and deleting files and directories on the system. Multiple users can be connected to the system at a given moment. Files in the system have replicas to provide fault tolerance incase a server goes offline. Users in the system will have the illusion of a single file system image at all given times.

# Features

1. Directory listing is displayed in a tree structure with proper nesting for sub directories.
2. File editing can be done on a separate Text editor while main program remains functional.
3. Client auto reconnects to the system.
4. Only files with readable extensions can be read or edited.
5. Folders and subfolders can be created and removed.
6. Server is able to respond to commands.
7. Edited files are automatically updated throughout the system once saved.
8. Message transmission to and from server is tracked.
9. No limit on size of files that can be read or edited.

# Architecture

## Inter Server Communication:

When a server comes online, the first thing it does is establishing a TCP socket which will be used to accept connections from fellow servers. The accept () call remains in a block inside an infinite loop in a thread. After this, the server sends a multicast message declaring its identity. Servers respond to the multicast message by sending a connection request to the server’s TCP socket. After a connection is established, initial handshake takes place in which both servers exchange their file list and synchronize replicas of files. Then the server starts another thread that remains in an infinite loop while receiving messages from the newly connected server on its socket.   
  
While this thread is blocked, the main thread starts another thread that continuously listens on the multicast IP group for newly awakened servers or clients. Thread safety mechanisms are kept in place to reduce race conditions. Writes on sockets are atomic operations.

## Server-Client Communication:

When a client comes online, it prepares a TCP socket which will be used to accept connection request from a responding server. Then the client sends a multicast message declaring its identity. All online servers receive this multicast message and attempt to make a connection with the client. The first server’s connection is accepted by the client. In case of no responses, the client waits for 5 seconds and resends the multicast message iteratively until it is connected to a server.

After connection establishment, the client starts a thread that is responsible for communicating with the server. A polling mechanism is placed in the thread which responds to the read events of the keyboard input or the server’s connected socket. If the connection with the server is unexpectedly terminated, the client attempts to connect to another server. In certain situations, the server requests the client to implicitly connect to another server for a performance gain. This is done without knowledge of the user.

## Summary of Threads in the System:

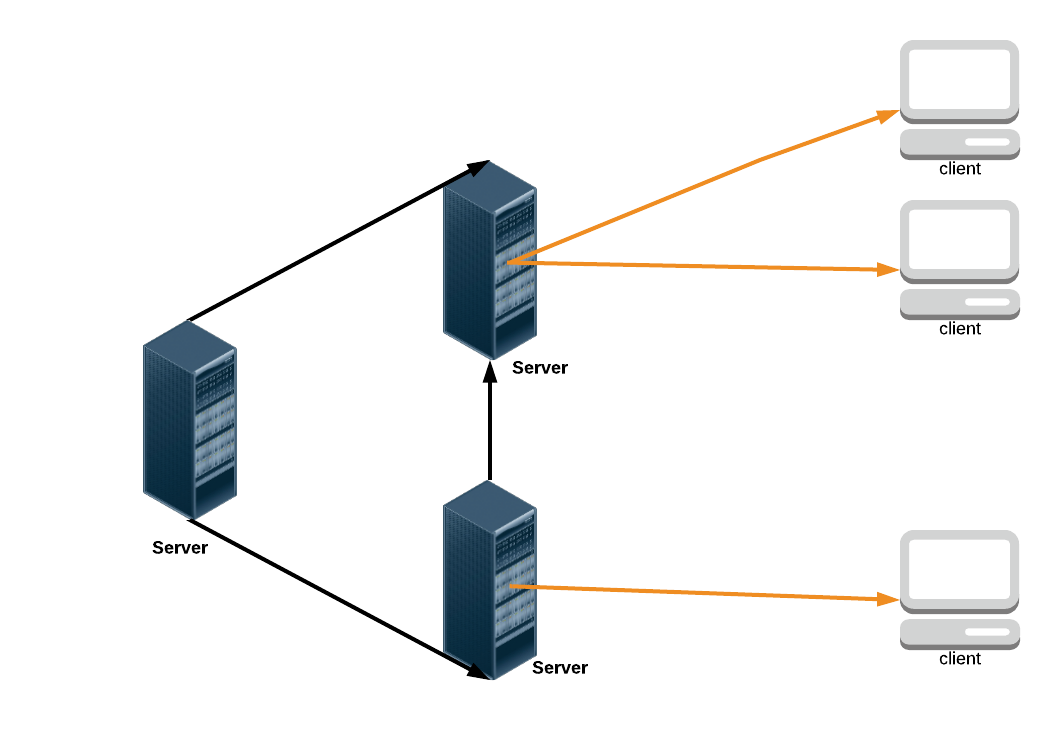
1. TCP Accept Thread: Accept incoming server connections.
2. Multicast Listen Thread: Listen to messages on multicast group and respond accordingly
3. Server Listen Thread: Listen to the socket connected to a server. The number of these threads is equal to the number of servers in the system.
4. Client Listen Thread: Listen to the socket connected to a client. The number of these threads is equal to the number of clients connected to a single server.
5. Main Thread: This takes commands from the system administrator, allowing server smartness.
6. (Client) Main Thread: Responsible for establishing and reestablishing connections with the server and exiting the client program.
7. (Client) Communication Thread: Responsible for communicating with the server. A polling mechanism here removes the serial aspect of taking commands from user and waiting for server responses.

Figure 1. DFS Network Diagram

## File Sharing Semantics:

My DFS architecture is based on **Session Semantics** for file sharing. When a client writes to a file, the changes made to the file are not visible to other users on the system until the client chooses to save the file.

## Client-Server Communication Model:

The architecture that I have implemented is the **upload download model** for file editing. If a client wishes to only read the contents of the file, the server displays the contents of the file to the user within the terminal. If the client wishes to alter the file, a temporary copy of the file is made locally on the client’s system and is opened in an in-software editor. After the editing is complete, the client may choose to keep the changes made to the file. In this scenario the client program uploads the file to the server which updates its current copy.

## Replication Strategy:

At the time of a file’s creation, a copy of the file is made on another server in the system. The selection of the server is based on a cost function that looks at other server’s file count. When a file is edited or deleted, the global file directory is scanned to check for replicas and the change is propagated through to the replicas. If a server comes online with a stale copy of the file, it will have a version number associated with it in the file’s meta attributes (extended attribute). By comparing the version with other file copies on the network, it will update the file’s contents with the latest file.

# Help

## Client Side Commands:

|  |  |
| --- | --- |
| 1. peek | View Directory Listing |
| 1. read [filename] | Read a File’s contents |
| 1. writ [filename] | Edit a File in Text Editor |
| 1. remv [file / folder] | Remove a file or a folder |
| 1. rmdir [Directory] | Remove a non-empty folder |
| 1. updt [filename] | Manually update a file |
| 1. make [file / folder] | Make a file or folder |
| 1. exit | Close Program |
| 1. cmds | Display all available commands |

## Server Side Commands (server shares most client commands):

|  |  |
| --- | --- |
| 1. cons | View all Connections |
| 1. repl | View Local replicas |
| 1. dir | View Local File List |

# Dependencies

* The current system only supports a UNIX environment.
* Before running the program, it is necessary to ensure that the xattr library for python3 has been installed. This library allows for editing extended attributes of a file.
* The network that the system is running on must allow for multicast of messages.
* A folder called ‘root’ must be present at the location of the server program file.

# Limitations

The current network architecture ensures that all servers on the system are interconnected with each other directly. This may introduce problems as the DFS begins to scale to a higher range. An alternate inter-server connection model would have all the servers connected in a ring topology. In this scenario, one server may only be connected to two other servers and all servers must be synchronized with each other at a given moment in time.