**PyDi**

ARCHITECTURE

**Project:** Distributed File System  
**Language:** Python  
**Operating System:** Windows, Linux, Mac OS

# File System

**Type:**

**Immutable** - Files are editable by the client, but they are saved as a **new** file on the server side while the old file is removed.

**Structure**

The directory structure is completely virtual - i.e. there are no physical folders. The directory structure is maintained by a **Tree** structure, in which there are two types of nodes.

Each node has a name, type (True for Directory, False for File), and a Binary Name.

* A **Directory** node contains a list of child nodes and one parent node. Each child node can be a directory or a file.
* A **File** node also has a size attribute and a list of two servers it is stored on.

Each server contains a copy of this Tree structure and caches a copy when it is turned off. Whenever a server comes online, it checks if any other server is online and if there is, it receives the latest Tree structure from that server. If no other server is online, this server will load its old copy from the cache and this copy will now be given to any other server that comes online.

# Server - Side

***LIMITATION: T****he Server that is the* ***last to be turned off*** *should be the* ***first to turn on*** *in order to preserve Server state since only the latest server would have the latest state of files and the directory structure. Otherwise, there is a chance of* ***data loss****.*

*The system has been tested with up to 3 servers, but with a replication factor of* ***2****, it’ll maintain access to 100% of the files on the system even with 2 servers up and running. The system also works perfectly with one server but will lose access to around 20-40% of the files.*

Each Server has its own name, which is used to find where a file is stored. For the first time you start a server, it will ask you for a server number which it will then save through Serialization in the **code** folder.

## STARTUP

When a Server starts up, it first checks if any other server is online. If there isn’t, it will load its copy of the directory structure it had stored. If there was no directory structure copy (starting for the first time), it will create a new structure from scratch with only the **Home** folder.

This new structure will act as the new state and will be given off to every client or server that comes online.

Whenever a server comes online, it will check if any other server is online. If there is, it will initiate a TCP connection and receive the latest directory structure. Then, both the servers will engage in a **file-exchange-handshake**, where both of them will send over the files the other server should have but doesn’t.

This *ensures* that all of the file replications that failed when a server is off, are sent over now that it has come online. Moreover, it also ensures that if a file (or even all the files) that are stored on a server are corrupted or deleted, the server will still repair and restore them automatically as long as there is one copy of those files on another server.

After the handshake is done, both the servers will start waiting on the TCP socket for messages from each other.

The server that just came online will start two threads,

* Multicast - UDP Socket
* Client-Server Connections - TCP

## MULTICASTING

One thread will be responsible for multicasting the server’s name and IP on a pre-set multicast group and port. This multicast group is the only way all the servers and clients can interact with each other.



Each time a server comes online, it first checks this multicast group to check for other online servers, receives their name and IP and connects with them via a TCP socket.

## ACCEPTING CONNECTIONS

One thread is dedicated to creating a server-side TCP socket, accepting any connection that comes through and handing the connection to another thread, then waiting for more connections.

My system is using this one TCP thread for both, connecting to clients as well as servers. Whenever it receives a connection, the first message it receives determines if it is a server or a client. Depending on the message, it passes off the connection to one of these two threads,

* **clientComm**
* **serverComm**

Both of these threads work in tandem on different servers to make the system work.

When clientComm thread receives a message from the client, it executes the instruction and returns the answer along with the new directory path. The server is **stateless** in the sense that the client has to send the complete instruction every time. However, the TCP connection is maintained throughout the client’s session so the client doesn’t have to go through the same authentication procedure each time, so the server is partially stateful.

However, being partially stateless allows the client program to connect to any other server if the server it is currently connected to fails/turns off/exits. The transition is completely seamless and the user has no idea that a server switch took place, or even that multiple servers exist.

Moreover, if a server doesn’t have a particular file, it can pass the client a message to exit the connection and start a new connection to the server that stores the file without the user knowing. If all the other servers that store the file are down, the client program will continue with the existing connection and show a ‘File not accessible error’.

FILE DOWNLOAD

If a client wants to download a file, the TCP socket will be used to send the file in chunks of 1024 bytes and the client program will handle that (more detail on that in the client-side architecture section).

FILE UPLOAD, EDIT, CREATE

File uploads, edits, and creation are done in a different way than downloads. File creation and upload are basically the same, since the text editor I used first creates a local file and then uploads it. The system uses an immutable architecture, so the only difference with Edits is that the when a client downloads a file, edits it and then saves it, the older copy is deleted a new version is created with a completely new Binary name.

*Note: Each server starts with own range of Binary names to avoid confusion and overlap.*

If a client wants to upload a file or create a new file, the client program will start a new thread that will connect to any available server through a new socket connection through the serverComm thread and will upload the file in the background. This option allows the client to continue to interact with the system while an upload takes place in the background.

Once the serverComm thread receives the file, it will do two things,

* **Choose a replication server** - it will check the file structure and find the server with the least number of files stored, and choose it for replication. Then, it will run a thread to pass on the file to that server (the replication server will receive the file and also update its structure).
* **Send a message to every other server -** it will send a message to every other structure with the details of the newly added file so they can update their directory structures

Similarly, if a file makes any other modification to the directory structure like,

* Add/Remove a Directory
* Remove a File

The exact message the client sends to the server it is connected to is passed on to the other servers. Since the servers are basically stateless, they need no information about the previous instructions and all the servers behave in exactly the same way to every instruction, updating their directory structures.

SERVER TO SERVER CONNECTION

Servers are connected to each other via two methods,

* A TCP socket initiated by the server that comes online later  
  Example. If Server A is the first one to be turned on, Server B will come online and initiate a connection with Server A. Once initiated, the socket will be waited on by a thread in each server for little updates and messages like adding/removing the directory and/or removing a file.
* File Transfers for replication are done through a new TCP socket created just when a file comes and closed after that file is transferred.  
  Example: If ***client1*** uploads a file on Server A, Server A will choose a replication server(based on least number of files stored), create a new TCP socket with that Server and send the file. If that Server is down, when it will come online, it will check the directory structure and automatically request that file from Server A.

# Client - Side

When a client program is turned on, it checks the multicast group for online servers and connects to the first one it can find.

After that, it uses the IP and connects to the server using a TCP socket. It takes an input from the console, does some error checking, and sends it to the server, waits for a reply from the server, does some processing to check if a server is sending an instruction to the client program or just a response, and prints the response after the processing.

In case the socket closes, the client checks for any online servers and connects to them without the user knowing. If the server the client is connected to doesn’t host a file, it will send an instruction to the client program who will try to connect to the server names sent by the first server and, if successfully connected, will receive the file from that server without the client knowing.

THREADING IN CLIENT

All file uploads are multi-threaded, so if a user uploads a file, it will start and connect to another socket to send the file, allowing the user to freely interact with the file system while the upload carries on in the background.

FILE EDITING

For file editing, I have used code from the internet for Tkinter. Once a file is downloaded, my program checks if it is a text file and gives you the option of opening it in the text editor. The text is copied into the editor and can be saved by clicking on the **SAVE** and then the **OK** button. Once the OK button is pressed, it will initiate a new connection with any available server and send the file.