Matthew Palkimas
ECE 470

1.0 Server Design:

1.1 Server will decode messages sent in the design shown in **2.0**. If not sent correctly it will return an error message saying where it went wrong.
Note: this will be after the user has connected successfully.

1.2 The server architecture will be spread out through this document.

2.0 Message Design:

2.1 First three characters: Opcode
Current valid opcodes: STR, RTV, DEL, INF, LGO
In order store, retrieve, delete, information

2.2 Depending on the opcode the rest of the message will need to be encoded in a certain manner.

2.3 To end the message it will be two /r/n and to distinguish between two strings one /r/n will be needed.

2.4 STR:
Store
First string will be file name
Second string will be file path
Third string will be file size

Will take a file name, path and size and be ready to receive the exact file size and that's all.

2.5 RTV:
Retrieve
First string will be file name
Second string will be file path

Will retrieve the file from the database and transmit it after sending a message with the size.

Note: in regards to STR and RTV if at any time the server has waited to long to recv or the client took too long to start recv it will cancel the process and go back to listening.

2.6 DEL:
Delete
First string will be file name
Second string will be file path

Will delete the file if it is in the database.

2.7 INF:
Info on Files
Will return the entire database in a structured file, see *3.6* for more info.

2.8 LGN:
Login
Technically this opcode will not be needed to ever be sent to the server from the client as the only time the user can login will be when it first connects to the server, or logouts.

2.9 LGO:
Logout
Just LGO will need to be sent.
In the future maybe info can be sent with it for the server to save.
Disconnects the user out of the server.

3.0 Data Storage Protocol:

3.1 File system setup:
The file system will be stored locally where the file server is running. It will start off by saying the exact location of the file database which will be the current working directory joined with 'files_$FILE_SERVER_NUMBER'

3.2 When a ‘new’ user connects to the server their files/{username} directory will be created in all file servers, also the user database.

3.3 The client will not be able to create files directly in their 'files' directory but will instead have to give a path inside their ‘files’ directory.

3.4 The server will have the capability of creating directories and it should be able to delete directories also.

3.5 The server will not be storing the files exactly as they are sent, instead the files will be distributed to four different servers in pieces, although it will store their names in the User database.

3.6 As the server side database is built up, **5.0**, there will be a secondary storage for all the names and paths for the files that are being stored so that when the user requests INF the server doesn't need to do anything complicated to return a nice built structure. (user database)

3.7 The max file size that the server can handle will be 100MB although this could go up if I feel the need.

3.8 A backup feature should be available to the server so that inside the files directory there will be a backup directory that of course the client doesn't have access to. But in case anything goes wrong it will be able to load from that.

4.0 User Access Protocol:

4.1 The client will act as a "dumb client" between the server and the user by just passing whatever the server gives it to the user, and whatever the user gives it to the server.

4.2 This client will send a 'hi' message that will be encoded as follows:

HI\r\n\r\n

In the future this message could include more than just HI to provide the server some information. But for now, this will do so I can keep the request response architecture.

Note: If the server is connected to and does not receive just this message it will disconnect immediately.

4.3 The server will send back a message that will inform the client a success happened, and it will provide the info to ask the client to get the username and password from the user. Formatted by the server:

1. Success, Error or Disconnect Character
2. Number of items the client needs to ask the user to fill in
3. Item number 1's text to display to inform the user on how to fill it in correctly
4. Item number 2's text to display to inform the user on how to fill it in correctly
5. For all numbers of course

Example server response message to "HI\r\n\r\n"

S\r\n2\r\nUsername: \r\nPassword: \r\n\r\n

4.4 The client will be built upon this structure so that it will loop for the number of items it needs to get from the user and just display in this case Username: to the user where it will read what the user types and accordingly format a message around what the user types.

Given the user entered their username as Matthew, and their password as yeet123 the message will be formatted as below:

4.5 Client Message format:

Matthew\r\nyeet123\r\n\r\n

As you can see it will just need to send the data the server requested. The server will be able to decode this easily, although if anything

4.6 Once the user successfully logs in then the client will follow the protocol specified in **2.0**

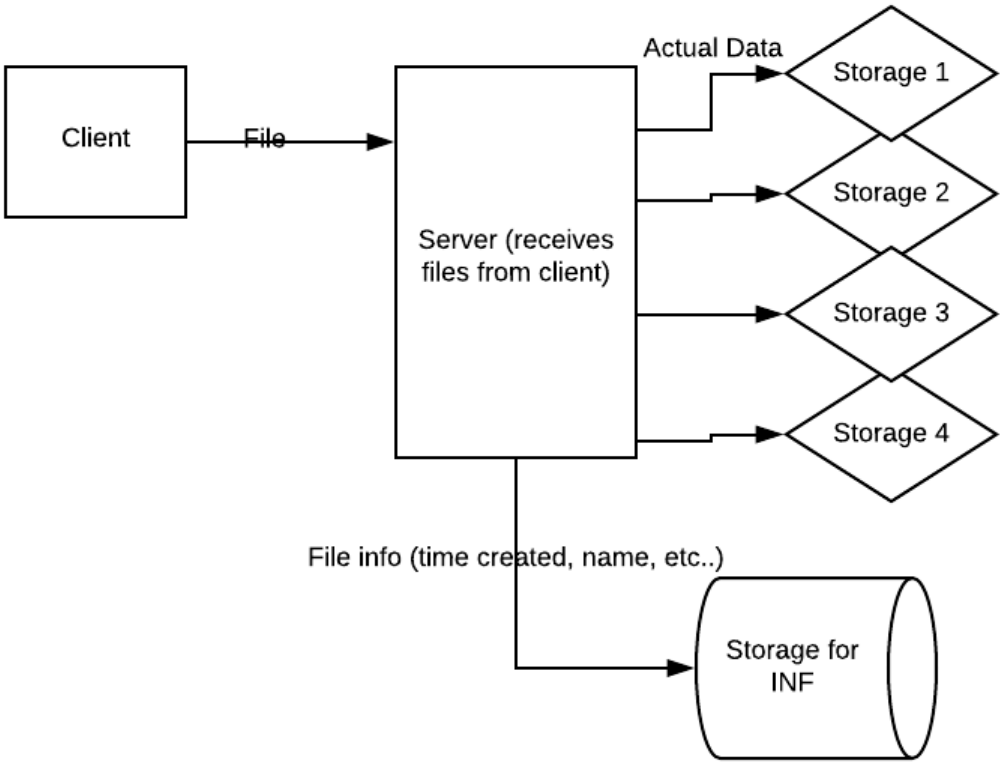
4.7 The server will continue to accept the messages as in protocol **2.0** with the only difference being the responses that it gives will depend upon what the client sends now, meaning the server will follow the same response as in ***4.3***.

5.0 Storage Server Design

(RTV DOES NOT WORK YET FOR THIS, ONLY STR AND DEL)

5.1 To run all the file servers run:

bash fs_start.sh



5.2 The file servers currently are organized with

Storage 1 having A1, A2
Storage 2 having B1, B2
Storage 3 having A1, B2
Storage 4 having A2, B1

Note: This will have to change in the future to account for storage 3 and 4 to do the math that is not implemented yet.