Socket-Based Networked File Sharing Cloud Server

**Introduction and project objectives**

The main objective of this project is to design, implement, and evaluate a distributed file-sharing system based on a server-client architecture. The connection should be established securely and allow for efficient file transfers from multiple clients to the centralized server. The project gives a focused experimental examination of the server-client model where multiple clients can receive and send files to one server, all on separate computers. Features such as secure authentication, performance analysis, and various file operations demonstrate a full scope of requests. In sum, the aim of this project is to provide a secure and fully capable method of sharing and managing files across various computers in a network.

The file sharing system is built in Python, using network protocols such as TCP for reliable and multithreaded communication to the server. Functionally, the system is meant to connect to the server, authenticate its users, and then allow the client to upload or download files, list files and directories, and manage subfolders through a user interface. The system has implemented security measures through its user authentication which allows for a username and password input needed to use the application. The interface allows the user to select a command and send requests to the server. Additionally, there are error messages so the application remains comprehensive with every attempted and unsuccessful task. Aside from errors, informative feedback is given to the user to indicate the status of file transfers and operations. This performance data includes the file transfer rate and times to provide insight into the efficacy of the system. Another portion of the system and project is to provide performance metrics by running tests of various loads on the client-side of the program.

Ultimately, this project demonstrates the feasibility of implementing a secure, reliable, and efficient distributed file sharing system in a cloud environment. Experimental data is collected through system analytics and compiled in a report along with steps to designing and implementing the application.

**System architecture and design**

The architecture of the file-sharing system is designed for a server-client model, where the client(s) communicate with a central server for file operations. Basically, a client must initiate a connection with the server, which listens for incoming requests. Once connected, a client is authenticated and then holds permission for file operations. Files stored in the server can be manipulated but there is no interaction with other clients and a client cannot control the status of the server. The system is built in Python for both the client-side and server-side applications to allow for simple implementation and scalability. The network allows multiple clients to be connected to the server at the same time as well as managing files concurrently. This communication is done using the network protocol TCP to promote reliable and efficient data transfer. Data transfer is also secured through user authentication by obfuscating and verifying credentials from a database between the server and client.

On the server-side, the application is built to handle multiple client connections using multithreading. This design is done through a socket server, where the server “listens” on a specified IP address (the IP of the server computer) and port for incoming connections. Upon a request, the server creates a new thread using the ‘threading’ Python module to communicate with the particular client. Files are stored in a specific directory on the server, along with metadata such as file size and upload times. The server authenticates users by verifying the received client credentials which are securely hashed using SHA-256. This ensures that credentials are not transmitted in plain text. The server holds all user credentials and grants appropriate access. After authentication, the server continues to listen for commands from the client such as upload, download, delete, subfolder management, and dir. These commands are then received, processed, and an appropriate response (error message or confirmation) is given. Each time a client connects or closes a connection, a message appears on the server. Each command dictates how the server responds, and differs greatly from the client’s interaction. The ‘Upload’ function, for example, tells the server to receive data and save it to the designated path where its files are stored. The ‘Dir’ and ‘Download’ functions tell the server to send the requested information.

On the client-side, the application presents as a user interface to connect to an input IP and port (socket connection). While the server is listening, the client must initiate communication to use the application. The client sends this request for connection, connects, and is prompted to log in. Once logged in, the client can send requests for each command to interact with files stored on the server. Clients are given responses such as transfer times and messages for user and system errors. Functions on the client side are initiated with a command, followed by a response with the requested data, an error message, or confirmation that data was sent. Finally, the network analysis module is responsible for collecting performance data for each file transfer. This component operates on both the client and server sides and is stored locally for examination.

At the core of this system is a client-server architecture that enables efficient file transfers between multiple clients and a centralized server. The server, designed as a multithreaded application, handles concurrent client requests by spawning new threads for each connection. In sum, by using a multithreaded server-client model and implementing security measures such as hashed authentication, the system ensures that multiple users can interact with the server simultaneously while maintaining data integrity and privacy.

**Implementation details**

The Client.py file provides functionality to authenticate users, manage files, and interact with server directories. The libraries it features to achieve this are socket, os, hashlib, and sys. Socket provides the core functionality for client-server communication. We implemented sha256 for authentication security which is hashing for usernames and passwords for authentication. We also implemented chunk-based file transfers so that the files can be split into manageable 1024-byte chunks for efficient network operations. The program also creates a directory for Client\_docs if it does not exist, reducing errors and keeping the storage organized and local.

The client.py starts by an initialization that creates a socket using *IPv4 (AF\_INET*) and *TCP (SOCK\_STREAM)*. It connects to the server using the provided IP and port via *connect\_to\_server* and ensures a local directory exists for file storage on the client side. Once the program successfully establishes a connection to the server, it will authenticate the user. The authentication method prompts the user for a username and password and handles server responses to determine if authentication succeeds. For file management, the upload function verifies the file’s existence in the *Client\_docs* directory, once verified it proceeds to the next step: sending the UPLOAD command along with the filename to the server. The client waits for a READY acknowledgement from the server to then transfer the file in 1024-byte chunks and appends *<EOF>* to signal completion. When the DOWNLOAD command is sent with the filename to the server, similar to the upload command, it waits for a READY acknowledgment and receives the file in 1024-byte chunks until *<EOF>* is detected. It then saves the file in the *Client\_docs* direcctory. For file deletion the command is sent to the server along with the filename and then it prints the server’s response indicating success or failure.

The directory listing (DIR) sends the command to the server and displays the list of files received or an error message. The subfolder management command sends the SUBFOLDER *<action> <path>* command to create or delete subfolders and prints the server’s response for confirmation or errors. The user command prompts the user to input commands to follow up with the appropriate method and handles the invalid commands.

Both files utilize binary mode ‘rb’ and ‘wb’ to read and write files for efficiency and compatibility. The serve.py has libraries socket, threading, os, hashlib, shutil and sys. Threading is used to enable handling multiple clients concurrently using separate threads. Os manages the *Server\_docs* directory and handles file/directory operations like checking existence and creating/deleting files. Shutil provides utility for file operations (although we used it minimally). Sys library assists in system-level operations like server shutdown. There is custom function outside of the class that handles non-empty subfolder deletion. We also provided more specific feedback for various issues throughout the code.

The server.py creates a socket using IPv4 and TCP just like the client.py, which binds the server to the specified host and port. The initialization also ensures that the *Server\_docs* directory exists for file storage prior to doing any commands. The handling client connections authenticates the client using AUTH *<hashed\_username>: <hashed\_password>* by comparing against pre-defined hashed credentials. This function also processes subsequent commands if authenticated and sends appropriate responses for valid commands or errors for invalid actions. The file upload function sends a READY acknowledgement to the client and receives the file in 1024-byte chunks, until *<EOF>* is detected and then saves the file in the *Server\_docs* directory.

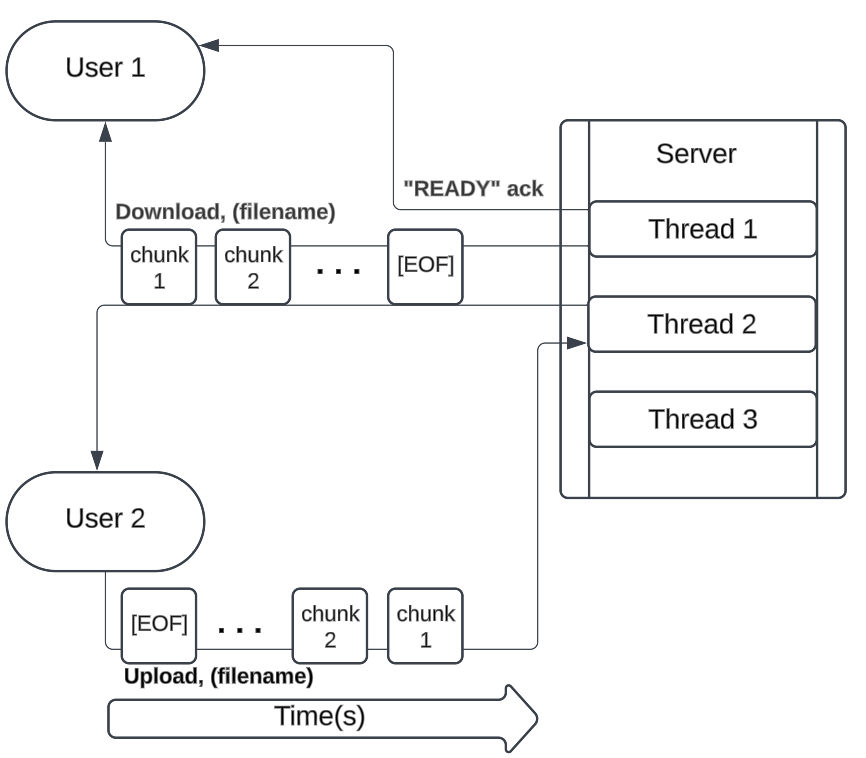
The file download function checks if the requested file exists in *Server\_docs* to send a READY acknowledgement if the file exists or if there is an error otherwise. It then transfers the file in 1024-byte chunks, appending *<EOF>* to signal completion. The file deletion function verifies if the file exists in *Server\_docs* path, deletes the file if it does exist and sends a success response or error if file is not found. The directory listing retrieves the list of files in *Server\_docs* and sends the file list to the client or a message if the list is empty. The subfolder management function handles CREATE command by creating the subfolder if it does not exist and handles the DELETE command by recursively deleting non-empty subfolders using a separate function called *remove\_directory\_tree* which iterates through all contents of a directory and deletes files until the directory is empty and then removes the directory, afterwards a success or error message is displayed when complete. The server start continuously listens for incoming client connections and spawns a new thread for each client, passing the socket to *handle\_client*.

The Analysis.py file provides a source for the server to document the rates and times it interacts with the server. This file is a collection of multiple different methods to allow for the server to easily interact with the log file that will be created. This file is initialized with a dictionary labeled *performance\_data*, where the upload rates, download rates, and transfer times will be stored for each action. When the server starts, it calls the *createLog* function to create the initial log. This function will start by stating that the log file was successfully created and that the server was started along with a timestamp of the current time and date. The *serverConnect* function is then called when the server successfully starts up, this adds a line onto the log file stating the current time and date and that the server was connected. The function *clientConnect* will be called after the server receives an authentication attempt from a client. This function takes a boolean parameter named *authentication* to represent if the client was successful in authentication. This starts by writing a line on the log file stating the current date and time and that the client was connected. The function then checks to see if the client was successful in authentication and adds onto the same line that the client was successfully authenticated. Otherwise, the function will add an error onto the same line stating that the client failed authentication.

The function *timeElapsed* will be called by the server each time the client successfully uploads or downloads a file. This function takes four different parameters, *name* for the file name, *command* to distinguish upload from download, *start* for the start time, and *finish* for the end time. The current date and time will begin the line with the name of the file that the client downloaded or uploaded to the server. The transfer time is then calculated by subtracting the end time from the start time. The upload rate or download rate is then calculated by dividing the file size by the transfer time. This value is then converted to megabytes per second. The function will add that the upload or download was successful and state the seconds it took to download or upload as well as the rate onto the same line. The function error will be called after every error that the server catches. This function takes two different parameters, *commandAttempted* to know which command was attempted and error to state the error that was found. The command is first identified by the function and then added onto the log file with the time and date, stating that the client requested for the command and states the error that was found. The function command is called by the server after the client requests to interact with the subfolders, to list the files, or to delete a file. This function takes one parameter to identify which command was called. This will then add a line into the log file stating date and time and the action that was performed.

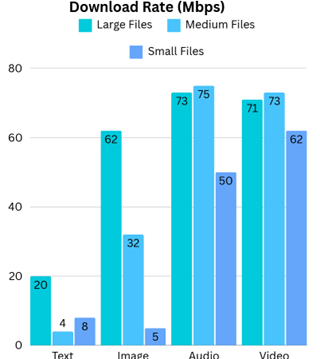
The last function in the Analysis.py file, *clientDisconnected*, is called when the client disconnects from the server. This can be when the client exits the server or is forced out of the server due to an error. This function will add a line into the log file stating the current date and time that the client was disconnected. The total amount of files uploaded and downloaded by the client are then displayed along with the average upload rate and the average download rate. The file is then ended with average transfer time.

**Experimental results and analysis**

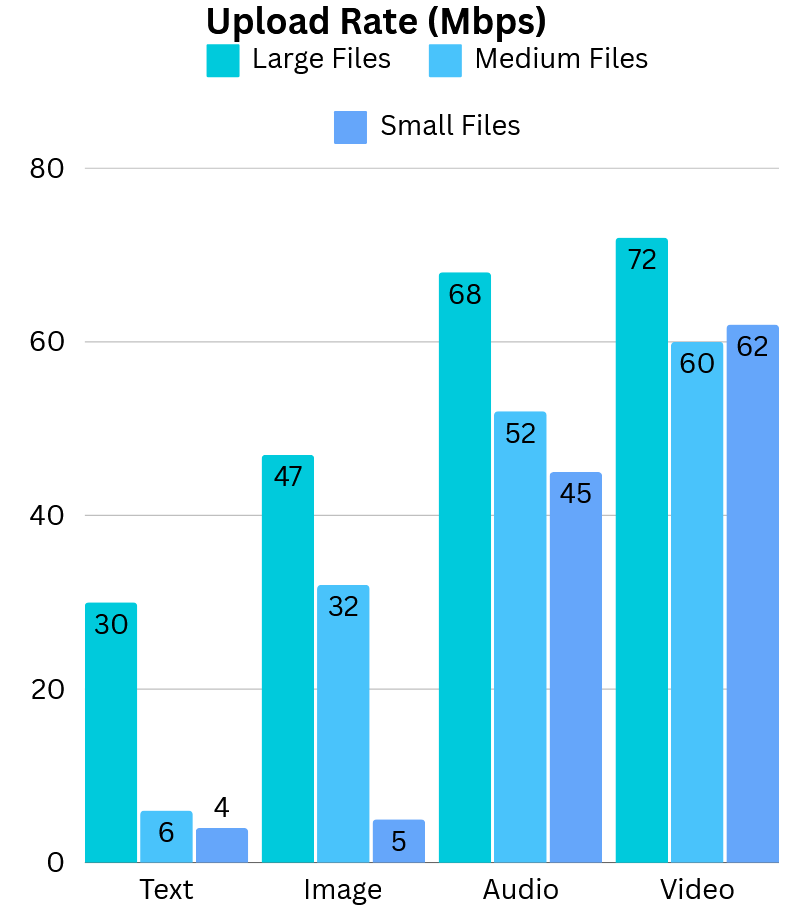


The server-client architecture is demonstrated to affect the functionality of a function. In the case of downloading and uploading this is dependent on the source and destination of data transfer. For the upload command, data is meant to be taken from the client files and sent to the server’s files, while download works the opposite. Before data is sent, an ack message must be received by the client in both cases, to indicate the server is ready to initiate the transfer.

In a TCP connection data can be sent in chunks, which is indicated in the code by the line > client\_socket.send(chunk) to improve efficiency for larger file sizes. The transfer rate is then decided by the maximum rate set by the server and bandwidth. Chunks are sent sequentially until the end of the file is reached, and a confirmation message is sent.



Above is the download rates as tested by 3 different file sizes. Large files were files considered to be 600 Mbps and as seen in the chart, were less predictably affected by slower upload speeds. The analytics on video and audio files illustrates that at larger file sizes the speed is less affected than by other factors such as internet speeds. The difference for downloading smaller files and having a markedly slower transmittance can be noted to be caused by outlying factors, or the lack of necessity to send larger chunks of data. This could also be seen when sending text files, since the largest text file sent for testing was 25Mb. The smallest file sizes simply don’t require the fastest speeds or cannot be calculated any higher because the length of time to upload may be even less than a second (As seen below).



The difference between upload and download speeds is also quite noticeable, as uploads were demonstrated to perform consistently slower. It’s hard to say what exactly causes the disparity, but it may be a result of the bandwidth that is typically allocated for downloading than uploading. However, we also see that at peak performance upload rates can be on par with peak download rates. On the other hand, audio files had slightly slower performance overall. In terms of smallest file sizes, which are usually 25 Mb or under, speeds were reported by the system to be single digits. These outliers behave predictably the same as when they were addressed when examining download speeds. This issue could stem from the method that the computer itself calculates the rate using the time and file size being transferred.

**Problems faced**

A problem we initially faced was ensuring the upload and download functions worked correctly. We noticed that our program would upload only the first file successfully. This was because the data from subsequent uploads was being appended to or overwriting the previously uploaded file due to a lack of proper file separation. This resulted in only one file appearing in the directory, with its content being overwritten. Similarly, during downloads, files were not being correctly distinguished, leading to inconsistent or incorrect file retrieval. We identified that the root cause was the absence of dedicated, structured directories for file storage on both the client and server sides. To resolve this, we introduced two separate directories, Client\_docs and Server\_docs and updated our program to include that directory path.

Another problem we faced was an issue with the subfolder deletion functionality, where subfolders containing files were not being removed because standard directory deletion methods only handle empty directories. We resolved this by creating a custom recursive function in Server.py outside of the class definition. The function iterates through all contents of a directory, deleting files and calling itself recursively to delete any subdirectories. Once all directories have been removed the function then deletes the now-empty directory. We integrated this function into the subfolder management logic, ensuring that subfolders, along with their files and nested subdirectories, are successfully deleted.

**Conclusions and future work**

This project aimed to achieve client-server connectivity and allow the client to interact with the server through different functions. The server will then log the time taken for the client to send and receive different functions from the server, allowing for offline analysis. When reviewing the data provided after testing the code, it becomes clear that the size of the file directly effects the transfer times between the server and client. The highest transfer times for files occurs with sounds and videos, with the smallest transfer times being with text files.

The program created by this project can continue to be improved upon. One of the major improvements that can be seen is adding a graphical user interface to the client’s side of the connection. This will make it easier for the user to understand and see what they’re doing with the server. By adding the graphical user interface, new functions and menus can be added to the program. Some examples of possible new functions are a back button for the client in case they entered the wrong function and the pop-up menu that you can directly select the file you want to upload.

This program can be later optimized for ease of use for the client and faster processing speeds on the server side. An optimization for ease of use that can be implemented in the future is the creation of graphs and plots alongside the log file that was created to further analyze the code. Future optimizations on the code for both the server and client files can improve the processing speed and system response time.

**References/Code Credit**

* Jennings, N. (2023, November 16). *Socket programming in python (guide)*. Real Python. <https://realpython.com/python-sockets/>
* *Time - Time Access and conversions*. Python documentation. (n.d.). <https://docs.python.org/3/library/time.html>
* GeeksforGeeks, & GeeksforGeeks. (2024b, August 13). *File handling in python*. GeeksforGeeks. <https://www.geeksforgeeks.org/file-handling-python/>
* Nik. (2023, September 24). *Python sha256 hashing algorithm: Explained • datagy*. datagy. <https://datagy.io/python-sha256/>
* Kannan, S. (2023, January 18). *How to create a simple python TCP/IP server and client?*. That Girl Coder. <https://thatgirlcoder.com/2021/09/01/how-to-create-a-simple-python-tcp-ip-server-and-client/>
* Tomar, N. (2023, February 7). *Multithreaded TCP file transfer in Python*. Idiot Developer. <https://idiotdeveloper.com/multithreaded-tcp-file-transfer-in-python/>
* GeeksforGeeks. (2024, September 20). *Get current timestamp using Python*. <https://www.geeksforgeeks.org/get-current-timestamp-using-python/>