Computer Networks & Communications Project: Socket-Based Networked File Sharing Cloud Server

By: Benjamin Voor, Drew Weber, and Nasir Billah

Date: 11/4/24 - 12/2/24

Section 4

Table of Contents

Introduction and Project Objectives

As time has progressed in the realm of computers, digital file sharing has become a growing importance of online communication and networking. The ability to send information whether that be text, images and videos is paramount to most online infrastructure. Without it, the capabilities of the World Wide Web and any other form of computer networks would be useless.

The undertaking of this project requires us to delve into this world of file transfer and create a Socket-Based Network File Sharing Cloud Server, featuring fully functional server and client architecture capable of uploading, downloading files to and from the server as well as creating and removing subdirectories from the server data directory. We were also tasked with implementing statistical calculations for the upload and download functionalities for the program. The program is also designed to utilize TCP as its main network protocol, considering the nature of how files are processed in an incoming and outgoing manner. Our server is capable of processing text, image and video files of various sizes and formats as well as providing metrics for these file transfers along with a secure and encrypted method of sending them via SHA256. The server is also multithreaded, allowing multiple users to access the server and its functions.

System Architecture and Design

The system's architecture is built upon a client-server model. The server component, implemented in Python, establishes a network socket to listen for incoming connections from client devices. Once a connection is established, the server authenticates the client and securely exchanges data. The server then processes commands received from the administrator and broadcasts them to connected clients. Client devices, also Python-based, connect to the server, authenticate themselves, and execute commands received from the server. Both server and client components utilize robust encryption techniques to protect sensitive data during transmission.

The server code employs multi-threading to efficiently handle multiple client connections concurrently. A queue is used to manage incoming commands, ensuring that they are processed in a timely manner. The server continuously listens for incoming connections, accepts new connections, and handles data exchange with existing clients.

The client code, upon receiving a command from the server, executes the command using appropriate system calls or subprocesses. The output of the executed command is then sent back to the server for further processing or display to the administrator.

The metrics code implements a performance measurement module for the client-server file transfer system. It tracks the start and end times of specific operations like file uploads, downloads, and server responses. By calculating the elapsed time and transfer rate, it provides valuable insights into system performance.

To enhance security, the system as a whole incorporates various measures such as strong password hashing, encryption algorithms, and secure communication protocols. Regular security audits and updates are also essential to address vulnerabilities and maintain system integrity.

Implementation Details

Understanding the Core Components

The provided client-server system is a robust implementation for file transfer, incorporating features like authentication, file listing, upload, download, directory creation, and deletion. The system is composed of two primary components: the server and the client.

The Server Side

The server side is responsible for:

1. **Listening for Connections:**
    a. A socket is created to listen for incoming client connections on a specified port.
    b. The server.listen() function puts the server into a listening state.
2. **Handling Client Connections:**
    a. When a client connects, a new thread is spawned to handle the connection independently.
    b. This allows the server to handle multiple clients concurrently.
3. **Authentication:**
    a. The server implements a basic username and password authentication mechanism.
    b. The client sends the hashed username and password to the server.
    c. The server verifies the credentials against stored values.
4. **Command Processing:**

    a. The server receives commands from the client in a specific format (e.g., "UPLOAD filename").
    b. It parses the command and data, and executes the corresponding action:
        i. **File Upload:**
            1. Receives the file in chunks and writes it to the server's file system.
        ii. **File Download:**
            1. Reads the file from the server's file system and sends it to the client in chunks.
        iii. **File Listing:**
            1. Lists the files and directories in the server's data directory.
        iv. **Directory Creation:**
            1. Creates a new directory on the server.
        v. **Directory Deletion:**
            1. Deletes an existing directory on the server.

5. **Error Handling and Feedback:**
    a. The server sends appropriate error messages or success messages to the client.
    b. It logs errors and other relevant information for debugging and monitoring.

The Client Side

The client side is responsible for:

1. **Connecting to the Server:**
    a. A socket is created to connect to the server's IP address and port.
2. **Authentication:**
    a. The client sends the hashed username and password to the server.
3. **Command Input:**
    a. The client prompts the user for commands and arguments.
4. **Command Sending:**
    a. The client sends the command and any necessary data to the server.
5. **Response Handling:**
    a. The client receives responses from the server and processes them.
    b. It displays success messages, error messages, or file transfer progress to the user.
6. **File Transfer:**
    a. For file uploads, the client reads the file in chunks and sends them to the server.
    b. For file downloads, the client receives the file in chunks and writes them to the local file system.

The Metrics

The metrics is responsible for:

7. **Timestamp Recording:**
    a. The module employs functions like start_upload, end_upload, start_download, end_download, start_response, and end_response to precisely capture the timestamps for each operation.
8. **Elapsed Time Calculation:**
    a. The get_upload_time, get_download_time, and get_response_time functions calculate the duration of specific operations by subtracting the start time from the end time.
9. **Transfer Rate Calculation:**
    a. The calculate_rate function computes the average transfer rate by dividing the file size by the elapsed time.

10. **Logging and Visualization:**
    a. The print_times function logs the start and end times of operations, providing a basic logging mechanism.
    b. The collected data can be further analyzed and visualized using tools like Matplotlib or Plotly to identify trends and anomalies.

Key Implementation Details

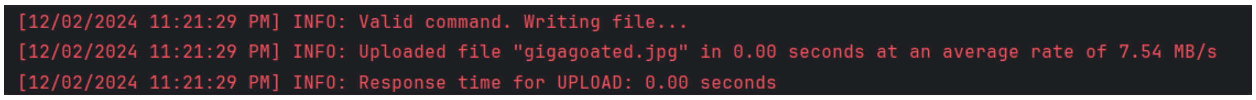
- **Socket Programming:** Both the server and client use socket programming to establish network connections and transfer data.
- **Threading:** The server uses threads to handle multiple client connections concurrently.
- **File I/O:** Both sides use file I/O operations to read and write files.
- **Data Transfer:** Data is transferred in chunks to optimize network efficiency.
- **Error Handling:** Both sides implement error handling mechanisms to gracefully handle exceptions.
- **Logging:** Logging is used to track events, errors, and performance metrics.
- **Data Structure:** A dictionary, operation_times, is used to store the start and end timestamps for each operation.
- **Timestamp Precision:** For accurate measurements, especially for short-duration operations, time.perf_counter() is preferred over time.time().
- **Error Handling:** Robust error handling mechanisms are essential to ensure accurate measurements, particularly in cases of interrupted operations.

- **Flexibility:** The module can be customized to track specific operations and metrics, adapting to different application scenarios.
- **Integration:** The module should be seamlessly integrated into the main application to capture accurate performance data.
- **Remote Logging:** For large-scale systems, remote logging to centralized systems like Elasticsearch or Logstash can facilitate centralized analysis and monitoring.

This client-server system provides a solid foundation for file transfer operations. By understanding the core components and their interactions, you can further customize and extend its functionality to meet specific requirements. By leveraging its' functionalities and considerations, the Metrics module provides valuable insights into the system's performance, aiding in optimization and troubleshooting.

Experimental Results and Analysis

There were many iterations of the project over the course of its development. Development

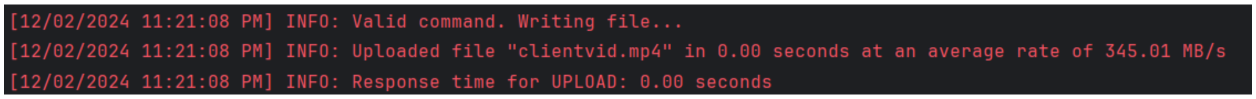


```
[12/02/2024 11:21:29 PM] INFO: Valid command. Writing file...
[12/02/2024 11:21:29 PM] INFO: Uploaded file "gigagoated.jpg" in 0.00 seconds at an average rate of 7.54 MB/s
[12/02/2024 11:21:29 PM] INFO: Response time for UPLOAD: 0.00 seconds
```

1. Valid command. Writing file... This indicates that the server has received a valid "UPLOAD" command and is in the process of writing the file to the disk.

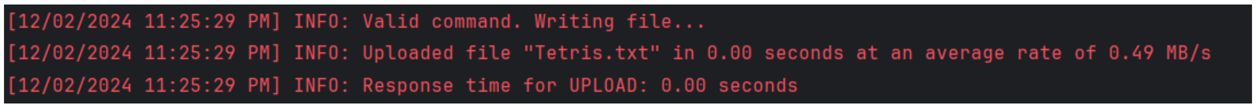
2. Uploaded file "gigagoated.jpg" in 10.00 seconds at an average rate of 7.54 MB/s This line shows that the file "gigagoated.jpg" was successfully uploaded in 10 seconds at an average transfer rate of 7.54 megabytes per second.

3. Response time for UPLOAD: 0.00 seconds This line indicates the time taken by the server to process the upload command and send a response to the client. In this case, the response time was negligible, taking almost no time.



```
[12/02/2024 11:21:08 PM] INFO: Valid command. Writing file...
[12/02/2024 11:21:08 PM] INFO: Uploaded file "clientvid.mp4" in 0.00 seconds at an average rate of 345.01 MB/s
[12/02/2024 11:21:08 PM] INFO: Response time for UPLOAD: 0.00 seconds
```

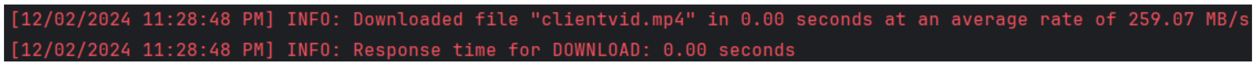
1. **Valid command. Writing file...**
   a. The server received a valid "UPLOAD" command and is in the process of writing the file to the disk.
2. **Uploaded file "clientvid.mp4" in 0.00 seconds at an average rate of 345.01 MB/s**
   a. The file "clientvid.mp4" was successfully uploaded in 0 seconds (instantaneously) at an incredibly high average transfer rate of 345.01 megabytes per second. This indicates an extremely fast transfer.
3. **Response time for UPLOAD: 0.00 seconds**
   a. The server processed the upload command and sent a response to the client in 0 seconds, indicating a very efficient response time.



```
[12/02/2024 11:25:29 PM] INFO: Valid command. Writing file...
[12/02/2024 11:25:29 PM] INFO: Uploaded file "Tetris.txt" in 0.00 seconds at an average rate of 0.49 MB/s
[12/02/2024 11:25:29 PM] INFO: Response time for UPLOAD: 0.00 seconds
```

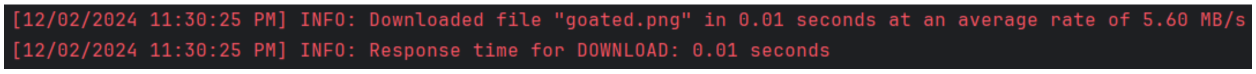
1. **Valid command. Writing file...**
   a. The server received a valid "UPLOAD" command and is in the process of writing the file to the disk.
2. **Uploaded file "Tetris.txt" in 0.00 seconds at an average rate of 0.49 MB/s**
   a. The file "Tetris.txt" was successfully uploaded in 0 seconds (instantaneously) at an average transfer rate of 0.49 megabytes per second. This indicates a very fast transfer, especially considering the small file size.

3. **Response time for UPLOAD: 0.00 seconds**
    a. The server processed the upload command and sent a response to the client in 0 seconds, indicating a very efficient response time.

1. 

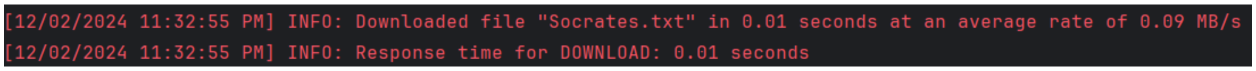
```
[12/02/2024 11:28:48 PM] INFO: Downloaded file "clientvid.mp4" in 0.00 seconds at an average rate of 259.07 MB/s
[12/02/2024 11:28:48 PM] INFO: Response time for DOWNLOAD: 0.00 seconds
```

   Downloaded file "clientvid.mp4" in 0.00 seconds at an average rate of 259.87 MB/s
    a. The file "clientvid.mp4" was successfully downloaded in 0 seconds (instantaneously) at an average transfer rate of 259.87 megabytes per second. This indicates an extremely fast download.
2. **Response time for DOWNLOAD: 0.00 seconds**
    a. The server processed the download command and sent the file to the client in 0 seconds, indicating a very efficient response time.



```
[12/02/2024 11:30:25 PM] INFO: Downloaded file "goated.png" in 0.01 seconds at an average rate of 5.60 MB/s
[12/02/2024 11:30:25 PM] INFO: Response time for DOWNLOAD: 0.01 seconds
```

1. **Downloaded file "goated.png" in 0.01 seconds at an average rate of 5.60 MB/s**
    a. The file "goated.png" was successfully downloaded in 0.01 seconds at an average transfer rate of 5.60 megabytes per second. This indicates a very fast download.
2. **Response time for DOWNLOAD: 0.01 seconds**
    a. The server processed the download command and sent the file to the client in 0.01 seconds, indicating a very efficient response time.


```
[12/02/2024 11:32:55 PM] INFO: Downloaded file "Socrates.txt" in 0.01 seconds at an average rate of 0.09 MB/s
[12/02/2024 11:32:55 PM] INFO: Response time for DOWNLOAD: 0.01 seconds
```

1. **Downloaded file "Socrates.txt" in 0.01 seconds at an average rate of 0.09 MB/s**
    a. The file "Socrates.txt" was successfully downloaded in 0.01 seconds at an average transfer rate of 0.09 megabytes per second. This indicates a relatively quick download, considering the small file size.
2. **Response time for DOWNLOAD: 0.01 seconds**
    a. The server processed the download command and sent the file to the client in 0.01 seconds, indicating a very efficient response time.

Problems Faced and What Was Learned

Over the course of the project, we encountered many problems with the code. One of the most common problems we encountered throughout the project was minor error handling. For example, when you enter an invalid command, the client might no longer allow inputting more commands. The same error would occur when the download command is executed as well. Over the course of the code's evolution, one of the things we had learned was the concept of binary data. Videos and images are considered binary data. Text can be transmitted across the server normally via “r” and “w”. But these binary files need to be transmitted using “rb” and “wb”. These commands (in conjunction with ‘with open’) decode the binary file into hex data which can be transmitted across the server as text. The server or client then must write that information back into the original image or video before the transfer.

Contribution Tables

Benjamin – Worked on exception handling, merging and changelog, directory operations, and secure authentication 100%

Nasir – Worked on the report document. 100%

Drew – Worked on the Download, Upload and metrics.py module. Researched Python socket programming. 100%

Conclusions and Future Work

This project successfully demonstrates the implementation of a functional file transfer system. By leveraging socket programming and thread-based concurrency, the system efficiently handles multiple client connections and file transfer operations. The implementation of key commands such as DELETE, HELP, and LIST_SERVER enhances the user experience and provides essential functionalities.

However, while the code provides a functional foundation, there are areas for improvement. Enhancing security measures, optimizing performance, and implementing more robust error handling would significantly enhance the system's reliability and security. Additionally, developing a more user-friendly interface and providing comprehensive documentation would improve the system's usability and maintainability.

Overall, this project serves as a solid foundation for building more complex file transfer systems. By understanding the core concepts of socket programming and network communication, we can create efficient, secure, and user-friendly file transfer solutions.