LAU
الجامعة اللبنانية الأميركية
Lebanese American University

School of
Arts and Sciences

CSC430 – Computer Networks

Advanced File Sharing System
28/04/2025

Miryam El Helou 202104354
Kamal Dbouk 202203415

I. High Level Approach

This project entails a basic file sharing system implemented using Python socket programming in a client-server architecture. This system allows the user to simply upload files and retrieve them over the server. This communication is handled over TCP. File integrity is also a priority, utilizing the SHA-256 hashing to ensure safe transfer. Comprehensive file logging can be found for both the client and the server as well. The client is provided with a user-friendly interface using Tkinter. The server is multithreaded to handle multiple clients at the same time. We utilized a custom text-based protocol over TCP. The custom texts are "UPLOAD", "DOWNLOAD", "LIST", and "UPLOAD_COMPLETE". It listens to up to 5 clients at a time (Meaning it can assign a thread to 5 at a time – so more than 5 clients can be running at the same time). To start the server, you must run *python server.py* in the terminal. Similarly, run *python client.py* in another terminal to initiate the client.

II. Features

A. Multi-threaded Server:

The server utilizes the threading module to handle multiple clients simultaneously. The main server thread is listening for new connections. Then it assigns one thread per client connection and passes *handle_client*. Thus, each thread is decoding the received command and calling the corresponding functions from the following:

- receive_file(client_socket, filename, filesize, address)
- send_file(client_socket, filename, address)
- send_file(client_socket, filename, address)
- send_file_list(client_socket, address)

```
while True:
    client_socket, address = server.accept()
    client_thread = threading.Thread(target=handle_client, args=(client_socket, address))
    client_thread.start()
```

B. Upload Files:

The client allows users to upload files to the server. This is found under *upload_file* in the client code. The client will send it to the server in chunks of 4096 at a time instead of the full file. This is done for efficiency. These chunks are then used to update the progress bar at the bottom. The server here will receive the file from the client under *receive_file* in the server code. The server will verify if a file with the same name exists and calculates the hash, then sends the file back with the server hash and our protocol message "UPLOAD_COMPLETE". The server has one last job here, to log what they had received. An example of what the server could log is:
2025-04-19 16:31:24,335 - INFO - Received file from ('127.0.0.1', 49533): ID_v2.pdf (518250 bytes).
The client receives this response and compares the local hash with the server hash to preserve integrity. Here, the client returns to the user a message and logs if the file was renamed or not. An example of what the client could log is:
2025-04-17 17:35:22,657 - INFO - File uploaded successfully: 1732982517620_v6.jpeg OR
2025-04-17 17:35:22,657 - INFO - Hash mismatch for uploaded file: 1732982517620_v6.jpeg

C. Download Files:
The client can download files from the server by entering its name in the input box and clicking on download. Here, the client is sending a “DOWNLOAD” protocol command to the server with the filename. The server here will check if the file exists, if it does it will send it by chunks of 4096 as well. The server will log something like the following:
2025-04-19 16:31:30,167 - INFO - Sent file list to ('127.0.0.1', 49533)
The client here will receive the file and download it. Throughout this process, the client will be logging such as the following:
2025-04-19 16:30:29,704 - INFO - Downloading file: ID.pdf (518250 bytes)
2025-04-19 16:30:29,704 - INFO - File downloaded successfully: ID.pdf (518250 bytes).
The results will be reflected to the user by the client and the progress bar will be updated accordingly.

D. List Files:
When the user clicks the “Refresh Files” button, the client sends the server a “LIST” protocol command. The server returns an array of filenames and logs the following:
2025-04-19 16:31:35,427 - INFO - Sent file list to ('127.0.0.1', 49537).
The client utilizes Tkinter to display the list of names. The client logs its actions such as the following:
2025-04-19 16:31:35,428 - INFO - Fetched file list from server.

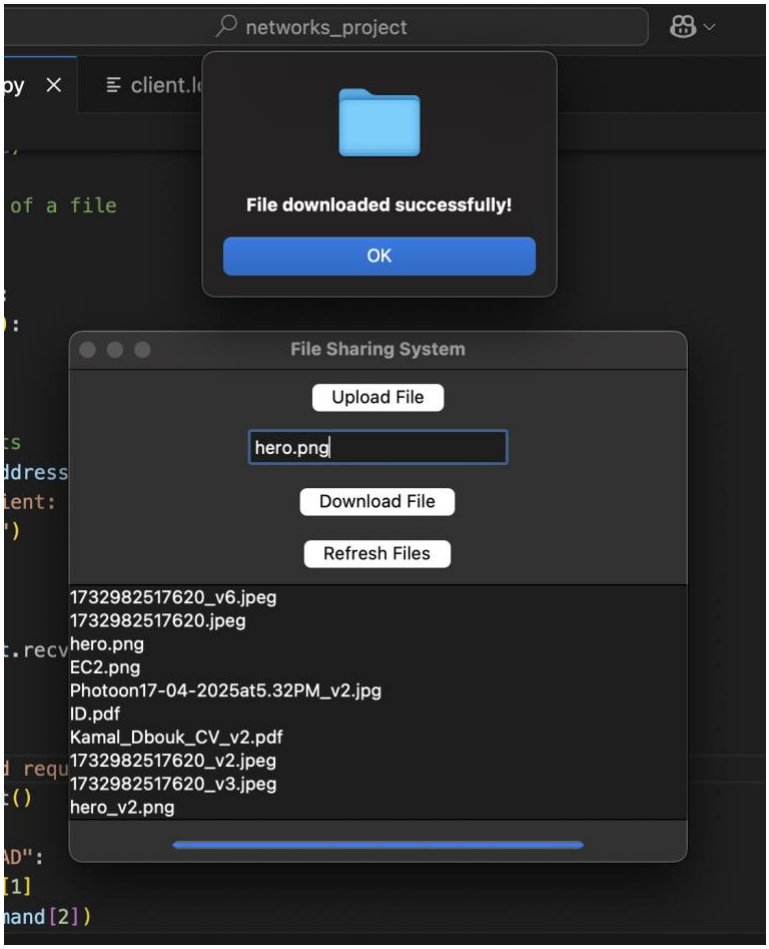
E. Logging:
Logging is implanted separately between the client and server. For each action, the timestamp and context are logged. This aided us in the testing phase. Each possible log is mentioned in the previous functions above.

```
2025-04-19 16:31:03,203 - INFO - Connected to client: ('127.0.0.1', 49533)
2025-04-19 16:31:04,782 - INFO - Received request from ('127.0.0.1', 49533): LIST
2025-04-19 16:31:04,782 - INFO - Sent file list to ('127.0.0.1', 49533)
2025-04-19 16:31:07,261 - INFO - Received request from ('127.0.0.1', 49533): DOWNLOAD d
2025-04-19 16:31:07,261 - WARNING - File not found for ('127.0.0.1', 49533): d
2025-04-19 16:31:11,313 - INFO - Received request from ('127.0.0.1', 49533): LIST
2025-04-19 16:31:11,314 - INFO - Sent file list to ('127.0.0.1', 49533)
2025-04-19 16:31:15,151 - INFO - Received request from ('127.0.0.1', 49533): DOWNLOAD id.pdf
2025-04-19 16:31:15,155 - INFO - Sent file to ('127.0.0.1', 49533): id.pdf (518250 bytes)
2025-04-19 16:31:24,331 - INFO - Received request from ('127.0.0.1', 49533): UPLOAD ID.pdf 518250
2025-04-19 16:31:24,335 - INFO - Received file from ('127.0.0.1', 49533): ID_v2.pdf (518250 bytes)
2025-04-19 16:31:30,165 - INFO - Received request from ('127.0.0.1', 49533): LIST
2025-04-19 16:31:30,167 - INFO - Sent file list to ('127.0.0.1', 49533)
```

F. SHA-25 Hashing:
To prevent data corruption during transfer, we utilized SHA-256 hashing to compare a file before and after upload. The client calculates the hash and the server recalculates as mentioned above.

```python
# Function to compute SHA-256 hash of a file
def compute_hash(filepath):
    hasher = hashlib.sha256()
    with open(filepath, "rb") as f:
        while chunk := f.read(4096):
            hasher.update(chunk)
    return hasher.hexdigest()
```

G. Progress Bar:
The GUI includes a visual progress bar that updates in real time. This feedback allows users to monitor the progress of their file upload. The progress bar number is based on the value of the progress array which is updated with each byte transferred divided by the total file size. Since we upload by chunks of 4096, the progress bar will update by $\frac{4096}{total\ file\ size}$ until the last chunk is less than or equal to 4096.



H. Version Handling:
If a user uploads a file with the same name, the system will rename add "*_v2*" at the end. This prevents overwriting a file and allows version control.

```
while os.path.exists(filepath):
    version += 1
    new_filename = f"{base}_v{version}{ext}"
    filepath = os.path.join(FILE_DIR, new_filename)
```

I. GUI:
The application features a GUI built with Tkinter. It includes buttons, entry fields, listbox and a progress bar.

III. Challenges Faced

A. Logging System:

A challenge we faced was attempting to test the system before implementing the logging system. With the logging system, errors became readable and debugging became easier.

B. Version Handling:

Implementing version handling was relatively challenging. We faced issues with implementing it at the level of duplicate versions past v2. After extensive research, we figured that logically, looping over the files and incrementing the version for every version found worked best.

IV. Testing

To validate the system, a series of manual tests were conducted. We attempted multiple clients running at the same time to ensure the threading was working. We also tested the version control functionality by uploading the same files multiple times, ensuring a unique version is found at the server every time. Logs were also reviewed for each functionality to ensure it was working as intended.

The work was split evenly between Kamal Dbouk and Myriam El-Helou. Kamal Dbouk primarily worked on the client while Myriam El-Helou primarily worked on the server.