VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

COMPUTER NETWORK (CO3093)

Develop a Network Application
File-sharing Application

Advisor: Mr. Nguyễn Mạnh Thìn
Students: Nguyễn Thành Đạt - 2152506
Nguyễn Minh Khuê - 2153488
Bành Vĩ Khang - 2153423

HO CHI MINH CITY, DECEMBER 2023

Contents

Member list & Workload

No.	Fullname	Student ID	Tasks	Percentage of work
1	Nguyễn Minh Khuê	2153488	- Application design - Application development - Extensional developement	100%
2	Bành Vĩ Khang	2153423	- Application design - Application development - Debug and testing	100%
3	Nguyễn Thành Đạt	2152506	- Report writing - Validation testing - Performance testing	100%

1 Overview of the problem

1.1 System overview

Build a simple file-sharing application with application protocols defined by the group, using the TCP/IP protocol stack. The system will consist of a centralize server, a cluster of storage servers called "Peers" and the end users. The centralized server manages the peers and at the same time listen and serve the end users. Peers or storage servers are going to be the main factor to store clients data remotely and can distribute files to certain people who own it. All of the uploading and transmitting will be done at the clients and the peers side to avoid overhead to the centralize server - which operate mainly on administrating purposes.

1.2 Desired outcomes

Upon the completion of the File-Sharing Application, the following outcomes are essential to be met:

- Efficient File Management: Users should experience an efficient and organized approach to file management through the application. The centralized server must accurately track connected clients and the files they possess in their local repositories.
- Seamless File Retrieval:When a client requires a file not present in its local repository, the server should efficiently identify suitable sources and enable direct file fetching between clients and peers, minimizing server intervention.
- Scalable Multithreading: The peers code must demonstrate scalability through multithreading. Multiple clients should be able to download different files from a target peer simultaneously, ensuring smooth and concurrent data transfer.
- User-Friendly Command Interface: Creating an User-Interface for clients to be be user-friendly. Clients should be able to publish local files to the server, fetch files from cluster of peers.

2 Description of functions

2.1 Client Class

1. `__init__(self, host, port)`
   - **Purpose:** Initializes a `Client` object with the specified host and port.
   - **Functionality:**
     - Sets the `host` and `port` attributes based on the provided arguments.
     - Generates a random `upload_port_num` in the range of 65001 to 65500.
2. `connect_to_server(self, username, password)`
   - **Purpose:** Establishes a connection to the server and authenticates the client with a username and password.
   - **Functionality:**
     - Creates a socket and attempts to connect to the specified host and port.
     - Sends the username and password to the server for authentication.
     - If authentication is successful, informs the server of the client's username and upload port.
     - Returns `True` if the connection and authentication are successful; otherwise, returns `False`.
3. `EXIT(self)`
   - **Purpose:** Sends a termination signal to the server and closes the client's socket.
   - **Functionality:**
     - Sends the "EXIT" signal to the server using pickle serialization.
     - Closes the client's socket.
4. `ADD(self, user_input_file_title)`
   - **Purpose:** Informs the server about the upload of a local file to the client's repository on drive storage.
   - **Functionality:**
     - Sends a pickled message to the server containing the ADD request and file title.
     - Receives and prints the server's response.
5. `GET(self, user_input_file_title)`
   - **Purpose:** Initiates a file retrieval process by requesting the server to find a suitable peer for the specified file.
   - **Functionality:**
     - Sends a pickled message to the server containing the GET request and file title.
     - Receives the server's response, which includes information about potential file sources.
     - If a source is found, initiates a connection to the peer containing desired file.
6. `transfer_file(self, conn, title)`

- **Purpose:** Transfers a file to a remote server through a provided connection.
- **Functionality:**
  - Opens the specified file for reading in binary mode.
  - Reads the file in chunks and sends the data over the provided connection.
  - Uses the "<|:::|>" delimiter to indicate the end of the file.
  - Closes the file and the connection after the transfer.

7. `p2p_listen_thread(self)`
   - **Purpose:** Listens for incoming connections from remote servers to receive files for uploading.
   - **Functionality:**
     - Binds and listens on a socket with a dynamically assigned upload port.
     - Accepts incoming connections and processes file transfer requests from remote peers.
     - Prints information about the received file.
8. `p2s_add_message(self, title)`
   - **Purpose:** Generates a message to inform the server about adding a file to the client's repository.
   - **Functionality:**
     - Constructs a message with ADD request details, including the file title, host, and upload port.
     - Returns the message as a list, this will later be transmitted to the server.
9. `p2s_lookup_message(self, title, lookupCode)`
   - **Purpose:** Generates a message to request file information from the server.
   - **Functionality:**
     - Constructs a message with LOOKUP request details, including the file title, host, and port.
     - Returns the message as a list, this will later be passed to the central server.
10. `p2p_get_request(self, title, peer_host, peer_upload_port)`
    - **Purpose:** Initiates a file transfer request to a remote peer server.
    - **Functionality:**
      - Connects to the specified peer and sends a pickled GET request message.
      - Receives the peer's response, including file information.
      - If the response indicates success (status code 200), initiates file transfer to local machine.
11. `p2p_request_message(self, title)`
    - **Purpose:** Generates a message to request a file from a remote peer in a client-to-peer context.
    - **Functionality:**

- Constructs a message with GET request details, including the file title and client information.
- Returns the message as a list.

12. `p2p_response_message(self, title)`
    - **Purpose:** Generates a response message to handle a peer file request for uploading.
    - **Functionality:**
        - Constructs a message with response details, including the file status and information.
        - Returns the message as a list.

2.2 PeerWorker Class

1. `__init__(self, host, port)`
    - **Purpose:** Initializes a `PeerWorker` object with the specified host and port.
    - **Functionality:**
        - Creates a `Lock` object (`dictLock`) for thread safety when modifying list of files.
        - Initializes instance variables, including `host`, `port`, `upload_port_num` (a random value in the range 65001 to 65500), `dict_list_of_rfcs` (a list to store information about files), `s` (a socket object), and `load` (indicating workload - initially set to 0).
        - Connects to the server using the `connect_to_server` method.
2. `connect_to_server(self)`
    - **Purpose:** Connects to the server and sends peer information.
    - **Functionality:**
        - Connects to the server using the `socket` (`s`).
        - Sends peer information (including `upload_port_num` and `dict_list_of_rfcs`) to the server as a pickled data.
        - Prints server data received as a response.
        - If `dict_list_of_rfcs` is not empty, assigns locks to each file entry for synchronizing when accessing the same file.
3. `listenServer(self)`
    - **Purpose:** Listens for server messages and handles fetch and upload requests.
    - **Functionality:**
        - Runs in a loop to continuously listen for incoming messages from the server.
        - Handles "ping" messages by responding with a "pong" message along the current workload of the peer.
        - Handles "fetch" messages to check if a file is available and responds with appropriate status codes.
        - Handles "upload" messages by initiating a peer-to-client file transfer using `p2p_get_request` method.
        - Closes the socket when the loop exits.

4. `peer_information(self)`
    - **Purpose:** Loads peer information from a pickle file in order to remember stored information if shut down.
    - **Functionality:**
        - Attempts to load peer information from the 'peerIF.pickle' file.
        - Returns the loaded information or default values (None, `upload_port_num`) if the file doesn't exist or an error occurs.
5. `transfer_file(self, conn, title, username)`
    - **Purpose:** Transfers a file to a client through a provided connection by the server.
    - **Functionality:**
        - Acquires the file lock if available.
        - Reads and sends the file in chunks over the provided connection.
        - Uses the "<|:::|>" delimiter to indicate the end of the file.
        - Releases the file lock after the transfer.
6. `p2p_listen_thread(self)`
    - **Purpose:** Listens for incoming client-to-peer connections to receive files.
    - **Functionality:**
        - Binds and listens on a socket with a dynamically assigned upload port.
        - Accepts incoming connections and starts a thread to handle each connection.
7. `handleClients(self, c, addr)`
    - **Purpose:** Handles incoming client-to-peer connections.
    - **Functionality:**
        - Receives peer information and file request from the connecting peer.
        - Sends a response message and initiates a file transfer using the `transfer_file` method.
8. `p2p_get_request(self, title, peer_host, peer_upload_port, owner)`
    - **Purpose:** Initiates a peer-to-client file transfer request to another peer.
    - **Functionality:**
        - Connects to the specified client and sends a request message using the `p2p_request_message` method.
        - Receives the client's response, including file information.
        - If the response indicates success (status code 200), initiates the file transfer.
9. `p2p_request_message(self, title)`
    - **Purpose:** Generates a message to request a file from another peer in a peer-to-peer context.
    - **Functionality:**
        - Constructs a message with GET request details, including the file title and peer information.

- Returns the message as a string.

10. `p2p_response_message(self, title, username)`
    - **Purpose:** Generates a response message to handle a client-to-peer file request.
    - **Functionality:**
      - Constructs a message with response details, including the file status and information.
      - Returns the message as a list.
11. `backup(self)`
    - **Purpose:** Backs up the peer's information to a pickle file to remember states.
    - **Functionality:**
      - Creates a list of dictionaries (`listOfFiles`) containing file information.
      - Serializes the list along with `upload_port_num` to the 'peerIF.pickle' file using pickle.
12. `scheduledBackup(self)`
    - **Purpose:** Runs a scheduled backup of the peer's information every 2 minutes.
    - **Functionality:**
      - Uses the `schedule` library to scan and run the `backup` method every 2 minutes while sleeping for 10 seconds between each scan.

2.3 Server Class

1. `__init__(self, host, port, peer_port)`
    - **Purpose:** Initializes a `P2PServer` object with the specified host, port, and peer port.
    - **Functionality:**
      - Initializes locks for peer and file lists.
      - Sets up socket objects, host, and port.
      - Initializes user information and peer-related lists.
      - Prints a message indicating server initiation.
2. `start_Client(self)`
    - **Purpose:** Starts the server to accept connections from clients.
    - **Functionality:**
      - Binds the server socket to the specified host and port.
      - Listens for incoming connections in a loop.
      - For each connection, starts a new thread (`client_thread`) to handle the client.
3. `authenticate(self, name, pw)`
    - **Purpose:** Authenticates a user based on the provided username and password.
    - **Functionality:**

- Checks if the provided credentials match any user in the `self.users` list.
- Returns `True` if authentication is successful, otherwise `False`.

4. `start_Cluster(self)`
   - **Purpose:** Starts the server to accept connections from peer servers.
   - **Functionality:**
     - Binds the server socket to the specified host and peer port.
     - Listens for incoming connections in a loop.
     - For each connection, starts a new thread (`peer_thread`) to handle the peer.
5. `peer_thread(self, conn, addr)`
   - **Purpose:** Handles communication with a peer server.
   - **Functionality:**
     - Sends a connection confirmation message.
     - Receives data, including information about the peer.
     - Updates the peer and combined lists accordingly.
6. `response_message(self, status)`
   - **Purpose:** Generates a response message based on the provided HTTP-like status code.
   - **Functionality:**
     - Returns a formatted message string with the specified status code.
7. `client_thread(self, conn, addr)`
   - **Purpose:** Handles communication with a client.
   - **Functionality:**
     - Performs authentication.
     - Sends a connection confirmation message.
     - Enters a loop to process client requests, such as listing files or uploading.
8. `ping(self, conn, lock)`
   - **Purpose:** Sends a "ping" message to a peer server to check its status.
   - **Functionality:**
     - Acquires the specified lock before sending the ping and releases it afterward.
     - Returns a response indicating success or failure.
9. `handleADD(self, title, host, up_port, owner)`
   - **Purpose:** Handles the process of adding a file to the system.
   - **Functionality:**
     - Selects an available peer with the lowest load.
     - Sends the file information to that peer for storage.
10. `handleFetch(self, title, host, username)`

- **Purpose:** Handles the process of fetching a file.
- **Functionality:**
  - Searches the combined list for the specified file title.
  - Contacts the respective peer and retrieves the file information if available.

11. `p2s_lookup_response(self, title)`
    - **Purpose:** Generates a response message for file lookup.
    - **Functionality:**
      - Includes the status code, current time, and information about the file if found.
12. `p2s_add_response(self, conn, title, hostname, port)`
    - **Purpose:** Sends a response message to the client after a successful file upload.
    - **Functionality:**
      - Includes the status code and information about the uploaded file.
13. `p2s_list_response(self, conn)`
    - **Purpose:** Sends a response message to the client with a status code indicating success and information about the available files.
    - **Functionality:**
      - Uses the `response_message` function to create and send the response.
14. `send_file(self, filename)`
    - **Purpose:** Sends the contents of a file over the server socket.
    - **Functionality:**
      - Reads the contents of the specified file and sends it over the server socket.
15. `create_combined_list(self, dictionary_list, dict_list_of_files, hostname, port)`
    - **Purpose:** Combines information about files from different peers into a single list (`combined_list`).
    - **Functionality:**
      - Takes a list of file dictionaries from different peers, extracts relevant information, and combines them into a single list.
16. `append_peer_list(self, dictionary_list, hostname, port, conn, up)`
    - **Purpose:** Adds information about a new peer to the peer list.
    - **Functionality:**
      - Adds a new peer's information, including hostname, port, connection object, upload port, and lock, to the peer list.
17. `append_to_combined_list(self, dictionary_list, title, hostname, port)`
    - **Purpose:** Adds information about a file to the combined list.
    - **Functionality:**

  - Adds a new file's information, including title, hostname, and port, to the combined list.

18. `print_dictionary(self, dictionary_list, keys)`
    - **Purpose:** Prints the contents of a dictionary list using the specified keys.
    - **Functionality:**
      - Iterates through the dictionary list and prints the values corresponding to the specified keys.
19. `delete_peers_dictionary(self, dict_list_of_peers, hostname)`
    - **Purpose:** Deletes information about a peer from the peer list.
    - **Functionality:**
      - Uses list comprehension to remove dictionaries with a matching hostname from the peer list.
20. `delete_files_dictionary(self, dict_list_of_files, hostname)`
    - **Purpose:** Deletes information about files stored by a peer from the file list.
    - **Functionality:**
      - Uses list comprehension to remove dictionaries with a matching hostname from the file list.
21. `delete_combined_dictionary(self, combined_dict, hostname)`
    - **Purpose:** Deletes information about files from the combined list.
    - **Functionality:**
      - Uses list comprehension to remove dictionaries with a matching hostname from the combined list.
22. `search_combined_dict(self, title)`
    - **Purpose:** Searches the combined list for information about a specific file and returns the result.
    - **Functionality:**
      - Iterates through the combined list and returns the dictionary with a matching file title, or `False` if not found.
23. `return_dict(self)`
    - **Purpose:** Returns the combined list along with keys.
    - **Functionality:**
      - Returns the combined list and a list of keys used in the dictionaries within the combined list.

3 Detailed application design

3.1 Application Architecture

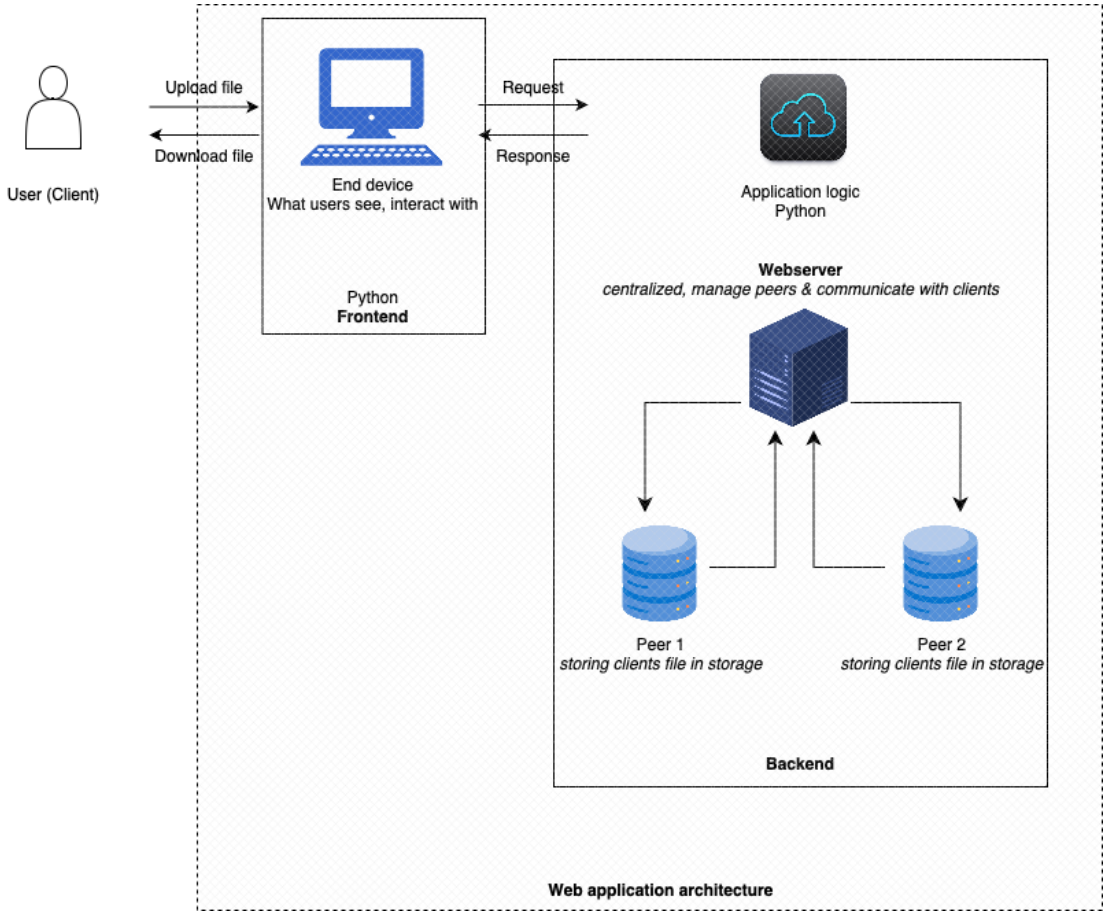

Figure 1. Application Architecture

1. Client-Side:
   - User Interface (UI): The client includes a user interface that allows users to interact with the web application, upload files, and manage their stored data.
   - Communication Module: Handles communication with the centralized server to upload and retrieve files.
2. Web Server:
   - Listens for incoming requests from clients and server peers. Handles communication.
   - Application Logic: Manages file upload and retrieval requests, communicates with server peers.
   - Authentication and Authorization: Ensures that clients have the necessary permissions to upload or retrieve files.

3. Server peers:

   - Server Peers: File Storage: Each server peer has its own file storage where uploaded files are stored. This could be a distributed file system or a simple file storage mechanism. Communication Module: Enables communication with the centralized server and other server peers. Handles file transfer requests.
   - Stores metadata about files, such as file names, owners, and storage locations on server peers.

3.2 Class Diagram

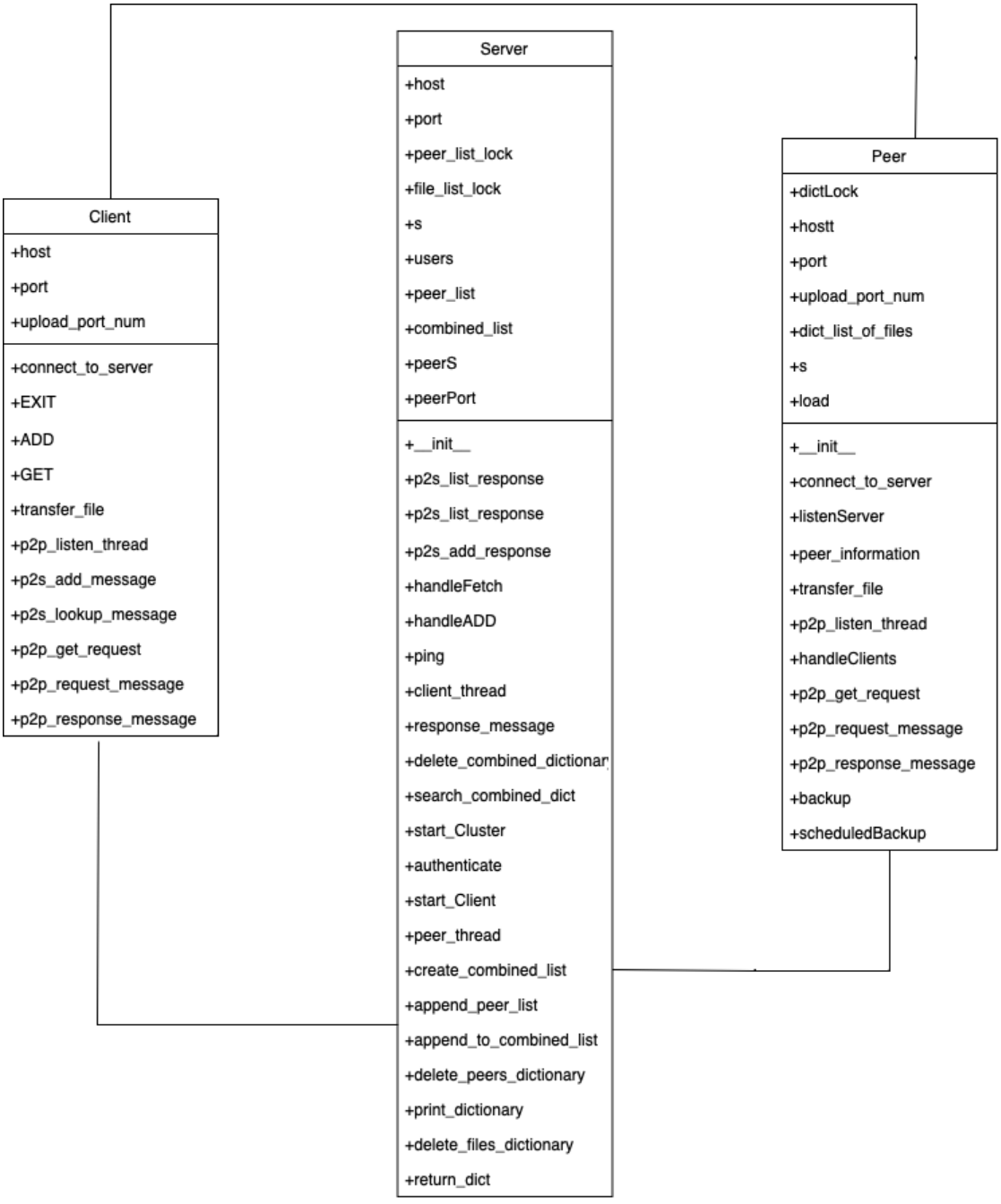

Figure 2. Class Diagram

3.3 Activity Diagrams

- **Client and Server activities:**

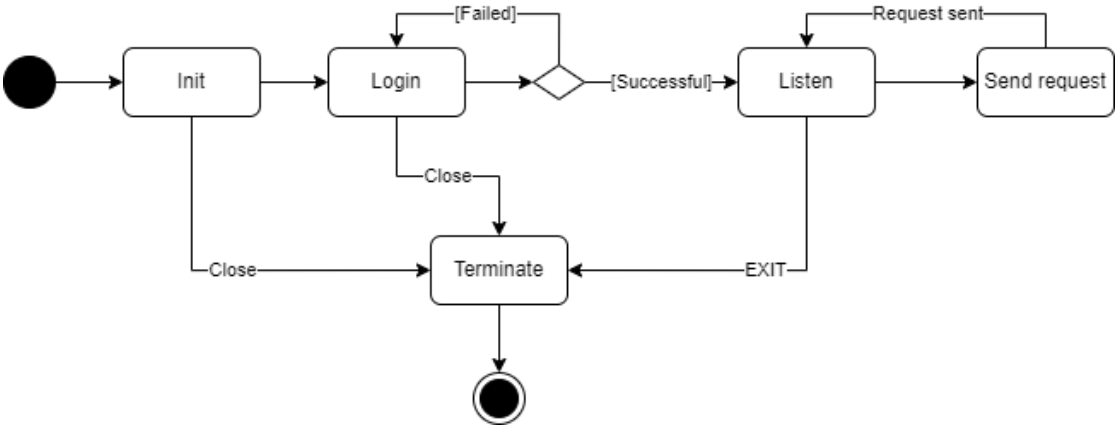

Figure 3. Client process

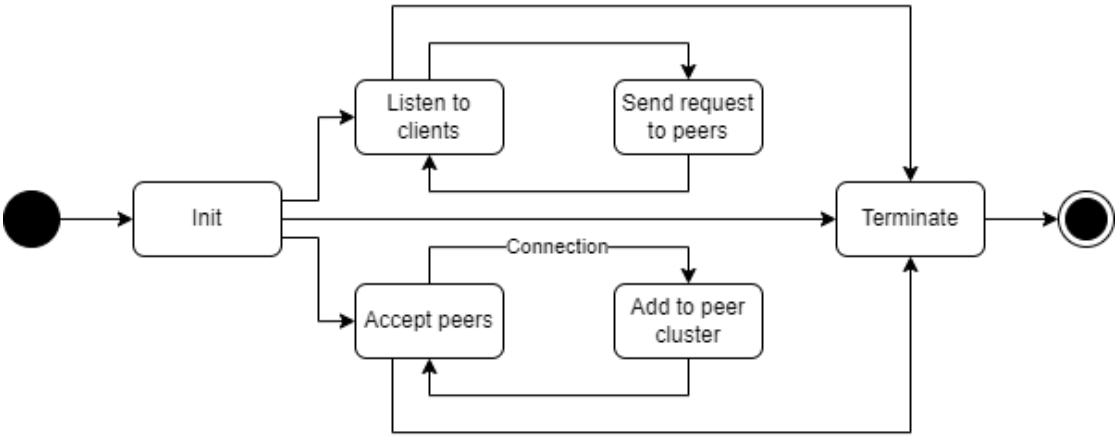

Figure 4. Server process

- **Peer activities:**

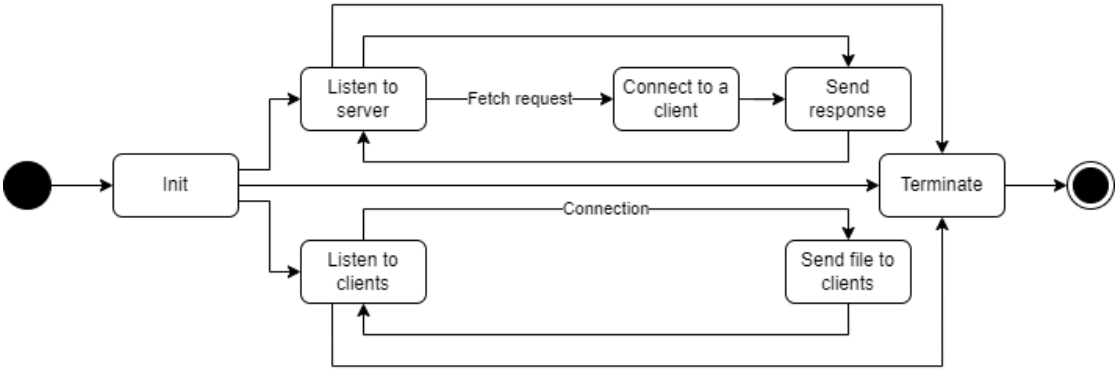

Figure 5. Peer process

4 User Manual

The system requires to have a Central Server, ideally always online and has a static IP address. In order to allow remote access anywhere on the internet, we will have to configure port mapping on the current router to forward incoming connections to your laptop and set up a static IP. In our case, we can only simulate that server on a laptop device, therefore the system can only connect to devices that is in the same LAN.

4.1 Server Administrator

For admins running centralized server, we will provide a simple CLI loggings for them to keep track of the current activities in the system without requiring any involvement.
To run the server, simple open a terminal and type in python server.py. The system will automatically find the host IP address and bind a socket with that address. The ports used for listening to clients and peers are going to be 7737 and 7736 respectively.

Figure 6. Server CLI

If a peer connects to the cluster storage, the Server will print out the information of that peer:

Figure 7. Accepting a peer

If a client connects to the server, it will print out that client's information, this assumes that the client successfully authenticate into the system:

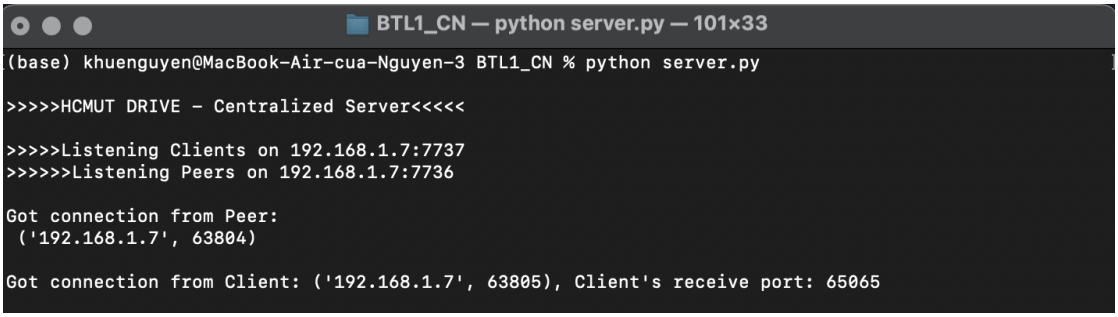

Figure 8. Accepting a client

4.2 Peers Administrator

The drive system will not work without the participation of file storage peers, peers application will run a simple CLI just like server for simplicity loggin report because the system is fully automated and does not require intervention. Simply type in python peer.py and the peer app will run, connecting to the server address at port 7736 as mentioned before

Figure 9. Server CLI

After about 2 minutes, the peers will run a scheduled back up to store the file information in case of sudden power off:

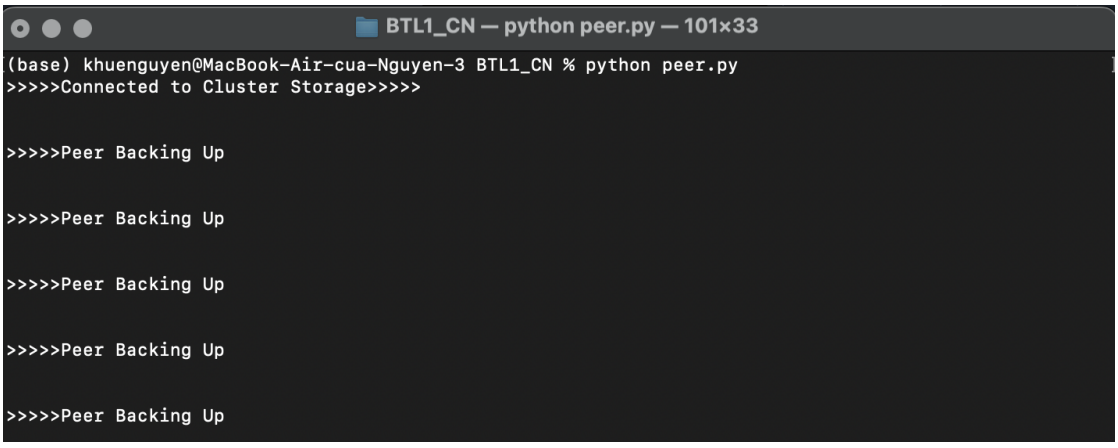

Figure 10. Peer backing up

4.3 Client

4.3.1 Login

To run the application, type in the terminal python client.py, an interface will show up requiring you to login. You will not be able to access the service without successfully logging in

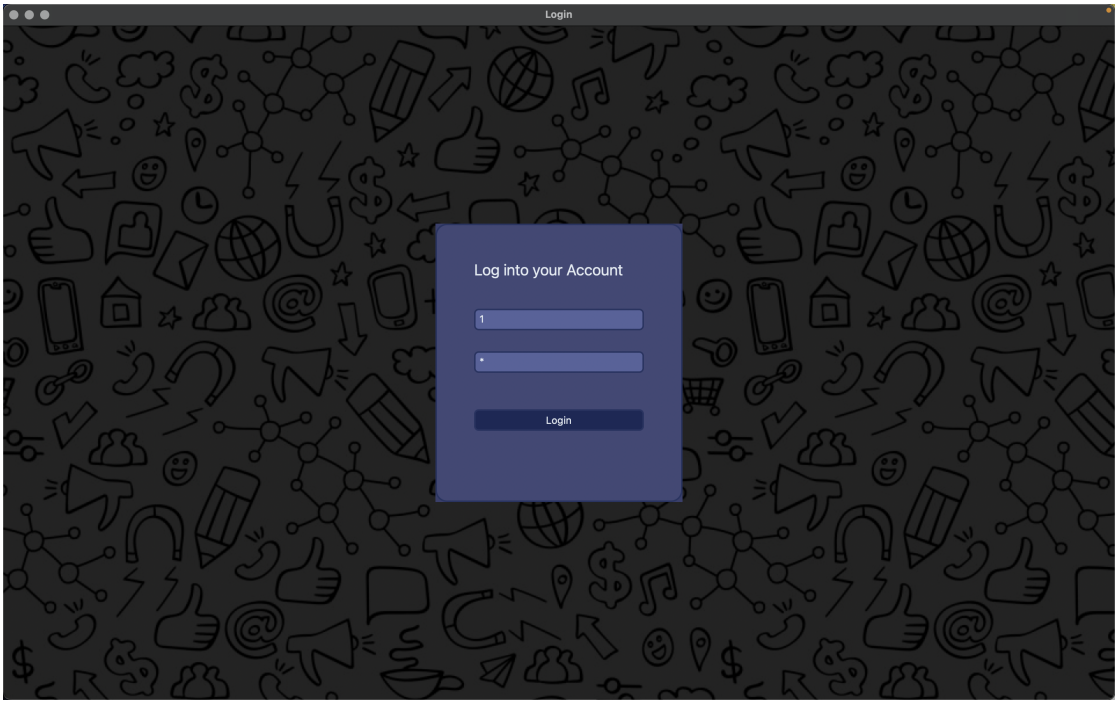

Figure 11. Login interface

4.3.2 Upload

After logging in successfully with the username and password registered to the server, you will now access the services provided by HCMUT DRIVE, a new UI will be shown at service:

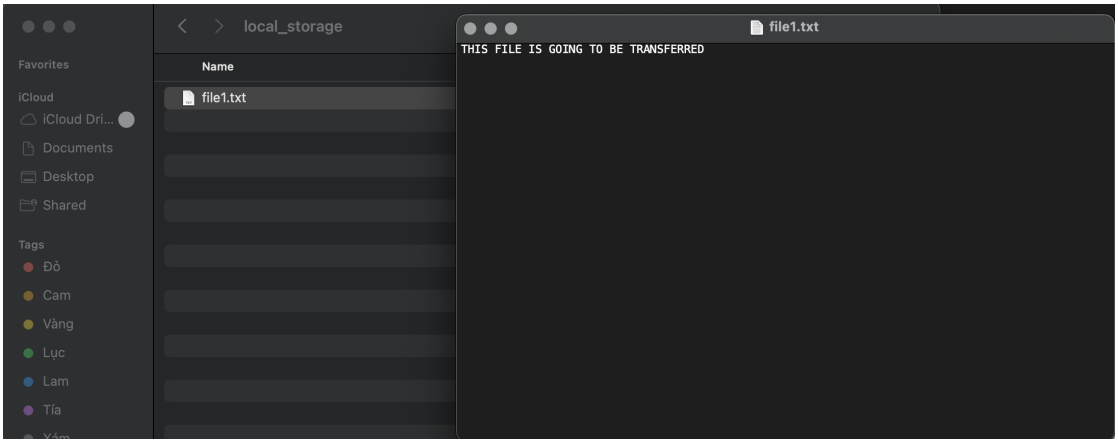

Figure 13. File to be uploaded

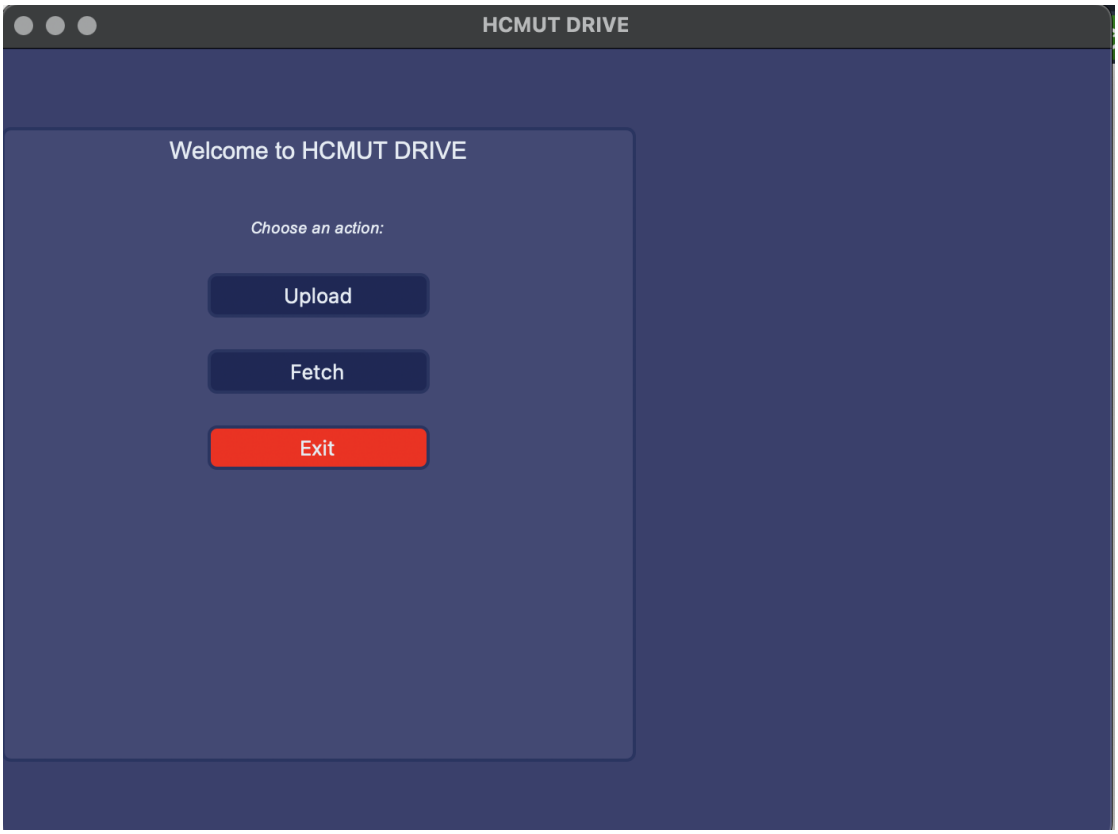

Figure 12. User-interface

Let's try and select upload to upload a file in our local repository to the system's drive. An input will show up to inform you to fill in the file's name and remember to store it in the "local_storage" folder before uploading otherwise it will not be able to locate the file: We will

start by transferring this simple text file, but any type of file is transferable on the system

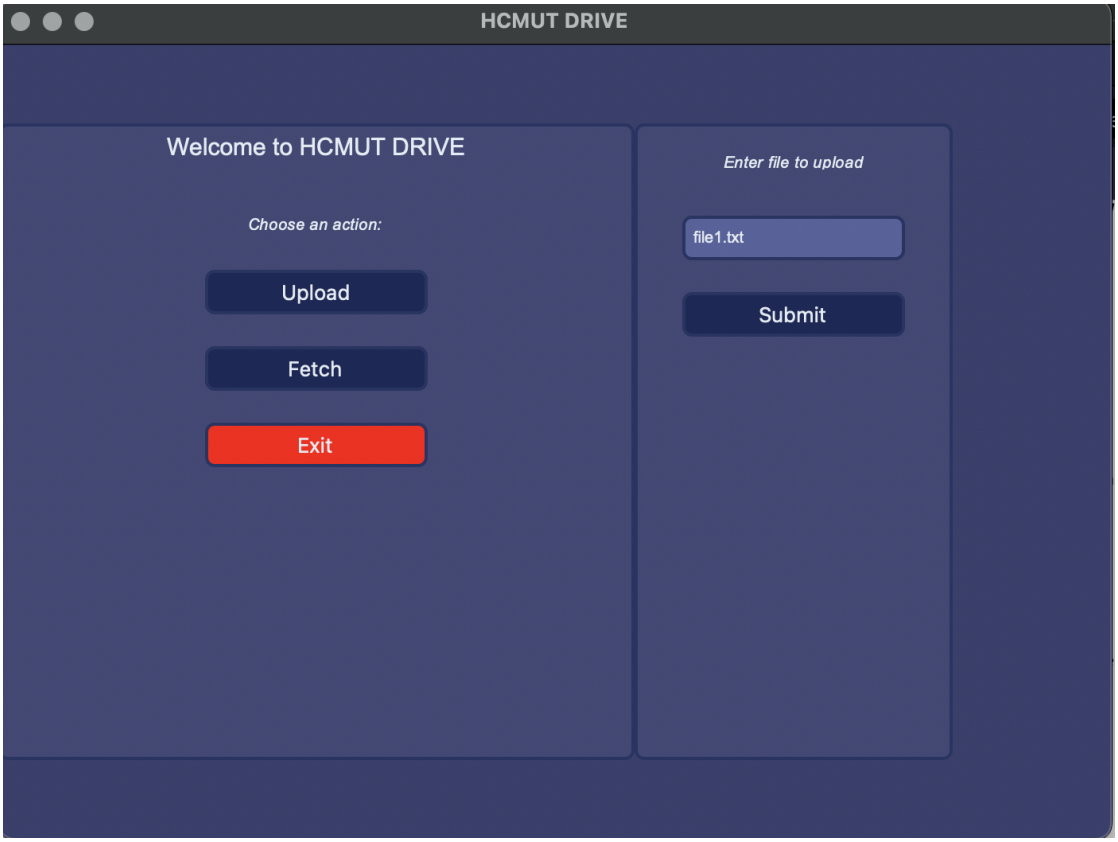

Figure 14. Fill in the file name to upload

After you have submitted the file, access the peer storage folder on peer device to see the changes:

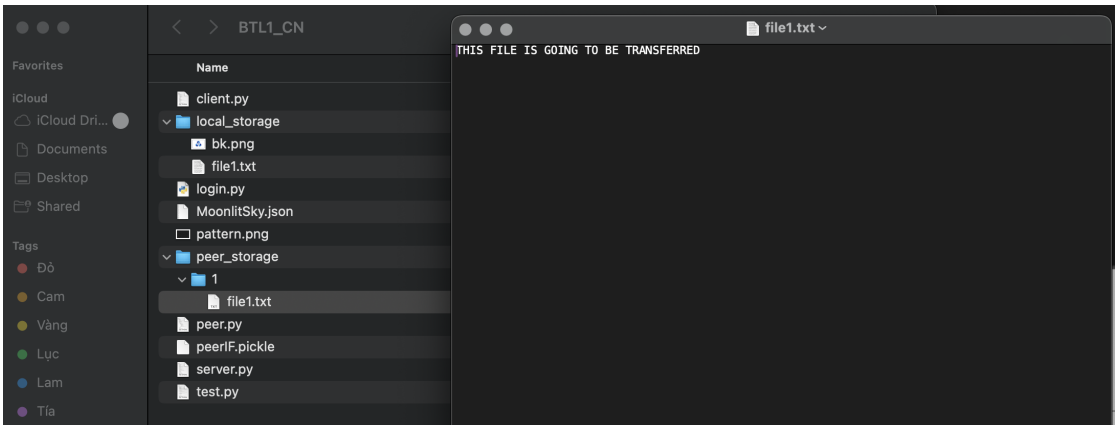

Figure 15. Updated Peer Storage

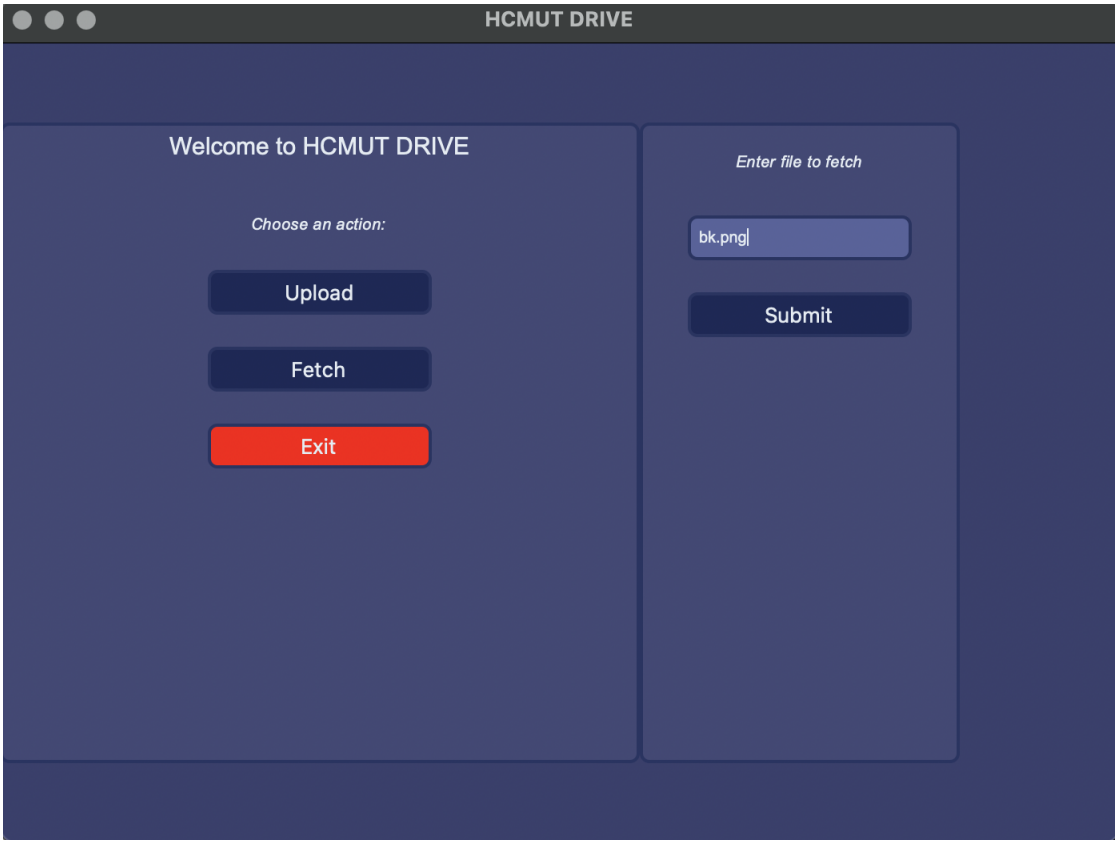

Figure 16. Fetch image

As you can see, the peer created a folder with the username in its storage folder to store all the files for that user, and the file previously was transferred successfully.

4.3.3 Fetch

In order to fetch a file from the system, you perform the simple task similar to uploading, just fill in the file name and press submit to get it. Let's try and fetch a HCMUT logo that was already uploaded by the client: Now let's see the local folder to find the image:

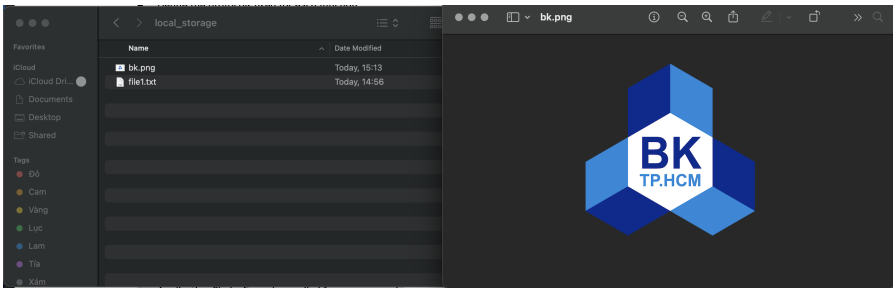

Figure 17. Image in local folder

5 Validation (sanity test) and evaluation of actual result (performance)

The system can handle a number of exception include:

- Server crash: client and peers automatically disconnect, server information about the clients will be backed up before it shutdown.
- Peers crash: clients will notice when peers crash fetching or uploading and will inform the user of the fault. Servers will also remove the peers from the peer list if ping failed
- Peers have back-up mechanism to back up it's information about the clients files in case of sudden shutdown

Overall, the system seems to operate well, although the user interface is lagging sometimes, as far as functionality considered, it can provide the users with a adequate number of functionalities for a drive file storage system including: uploading and fetching files, multi-threading between clients for smoother user experience, handle cases of exception and cross platform application (Windows and Linux OSes)

6 Extension functions of the system

6.1 File sharing

The users can choose to share a file from their drive repository to another user. The file will not be transferred to the shared user directly, but he/she will be able to fetch that file name even though it is not currently in their drive. A file can be shared to nobody or to any number of users as you may want.
The operation to share the file is fairly simple: enter the file name and the user you want to share it to:

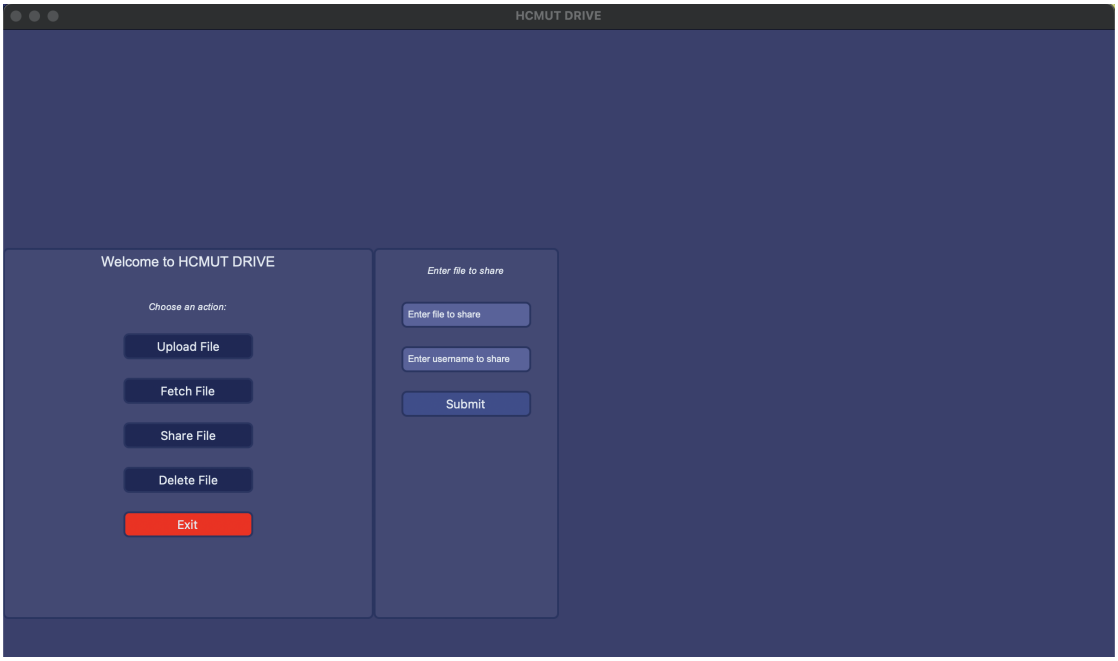

Figure 18. Sharing Interface

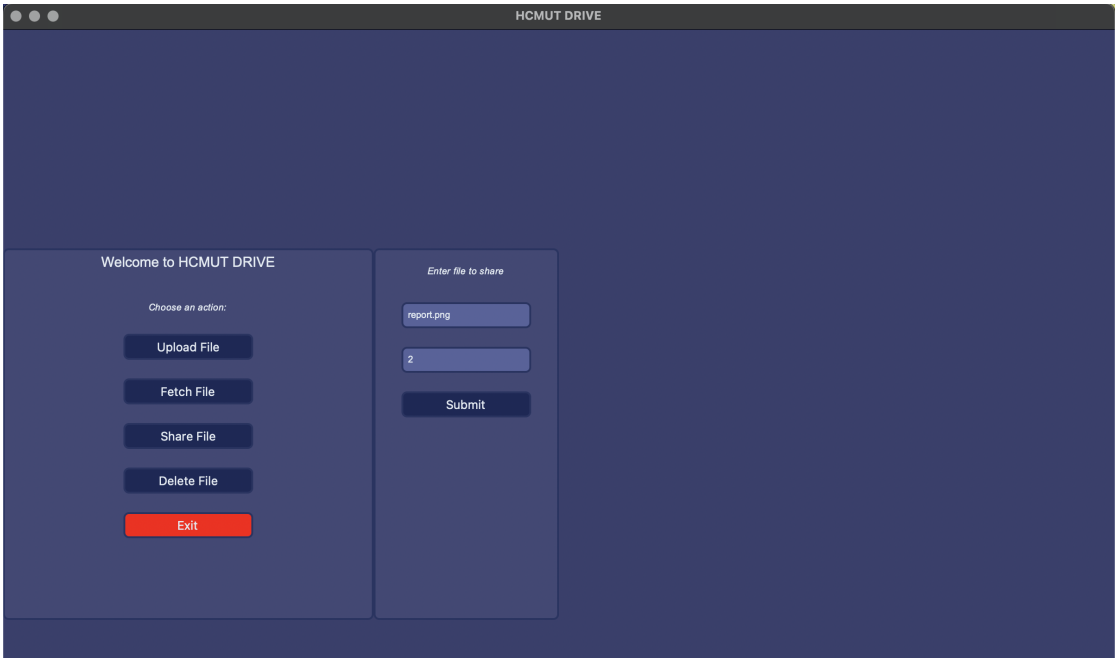

Figure 19. Input

Now user named "2" access the file named report.png anytime they want without having to store it on their drive repository.

6.2 Delete

Adding in deleting feature to provide users with a more flexible set of operations that they can perform. Users can now delete a specified file on their drive (if exist) and that file will no longer be stored on that storage peer
To delete a file on drive, simply enter the file name:

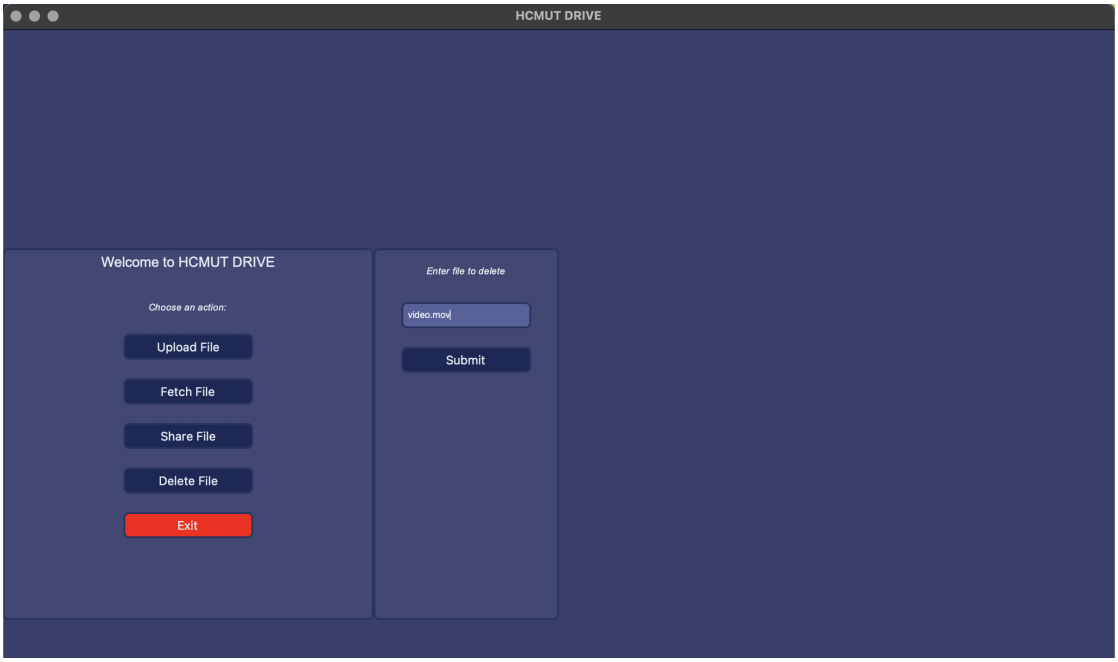

Figure 20. Delete a file

Now the user no longer has the file on drive and will not be able to fetch it again.

6.3 List all files

We noticed that in our initial development of the project, a file listing function was not included. It helps the user know what files are currently in their drive repository so that they can fetch or remove it.

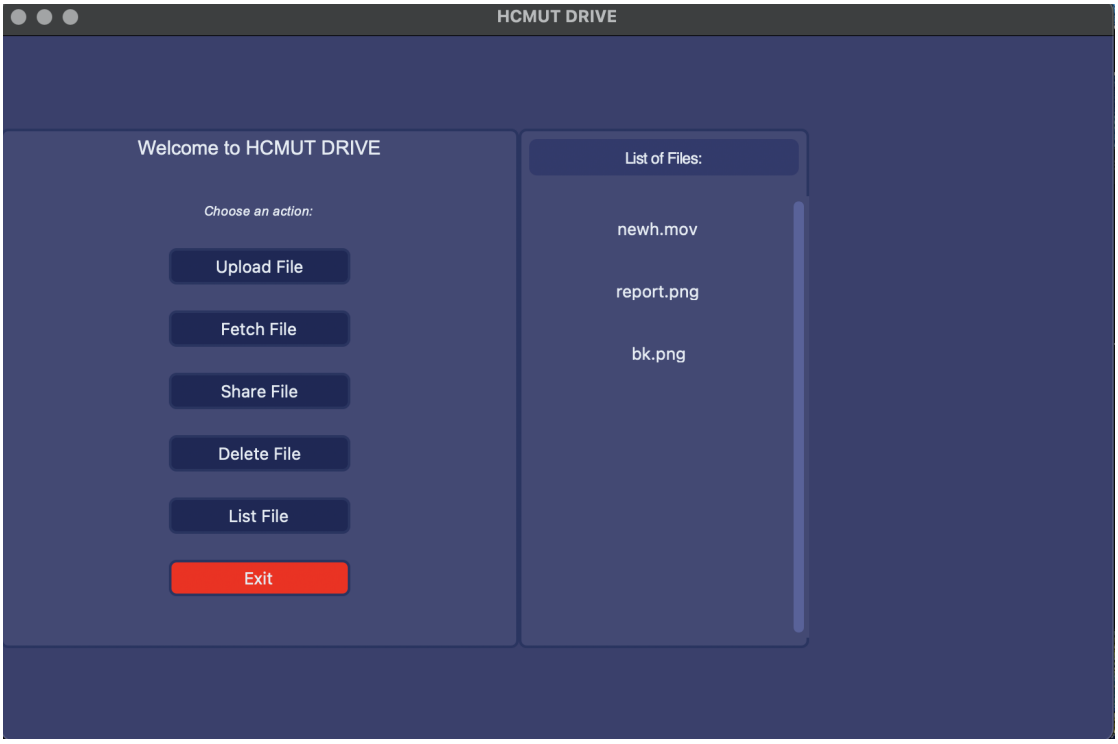

Figure 21. Show Files

You only need to press the button and all the files on your drive will be shown.