Comp9331 Report
By Abhishek Pradeep

In this assignment, I have the opportunity to implement my own version of a messaging and video talk application, based on a client-server model consisting of one server and multiple clients communicating concurrently.

My application uses TCP for text message communication, and UDP for video.

My application supports a range of functions typically found in messaging apps. These include authentication, private chat (message one participant), group chat (build the group chat for part of the participants), and uploading video streams (files in this assignment). I have designed custom application protocols based on TCP and UDP to facilitate these functionalities.

Client-side Code

The **Client class**, which inherits from threading. Thread. It manages socket connections for communication with a server and other clients, and it also implements various functions to handle different types of messages and commands. This class includes a constructor (__init__) that initializes the client socket to connect to the server and sets up a server socket for the client, which listens to incoming connections for possible file transfers.

The **UDP_video_sender** function is designed to send video files using UDP. It opens the file, reads it in chunks, and sends these chunks to a specific address.

The **handle_p2pvideo_command** function orchestrates the peer-to-peer video file transfer by starting two threads, one for sending and the other for receiving the video file. It waits for both threads to complete, which is essential for ensuring that the file transfer process is completed successfully.

The **run** function is the core of my client's interaction with the server. It manages different types of communications by parsing various server responses and executing appropriate actions, like handling login processes, receiving messages, and managing group chats.

My **main execution block** establishes the client application, setting up logging and parsing command-line arguments for server IP and port. It also manages the initial login process and provides a command-line interface for user interaction with the server.

In summary, my application aims to replicate key features of popular messaging and video talk apps, with an emphasis on reliable text communication and fast video file transfers. Balancing these aspects while maintaining a user-friendly interface and ensuring security is the cornerstone of this project.

Server-side Code

Server class manages the server's functionalities. I initialized the server with configurations such as the port, number of login trials, block duration, and timeout. This setup is crucial for handling client-server interactions effectively. Additionally, I have incorporated a mechanism to read user credentials from a file checking some edge cases.

The signal handling function **intHandler** ensures that interrupt signals like Ctrl+C are handled gracefully, allowing for a smooth shutdown of the server.

My implementation of user presence management, through functions like **online** and **who_logged_in**, ensures that the server keeps track of active users and informs others about new logins. This feature is essential for a real-time messaging application.

The **logout** feature, managed by the **offline**_detector method, runs on a separate thread. It periodically checks for user inactivity and logs out users who have been inactive for a specified duration.

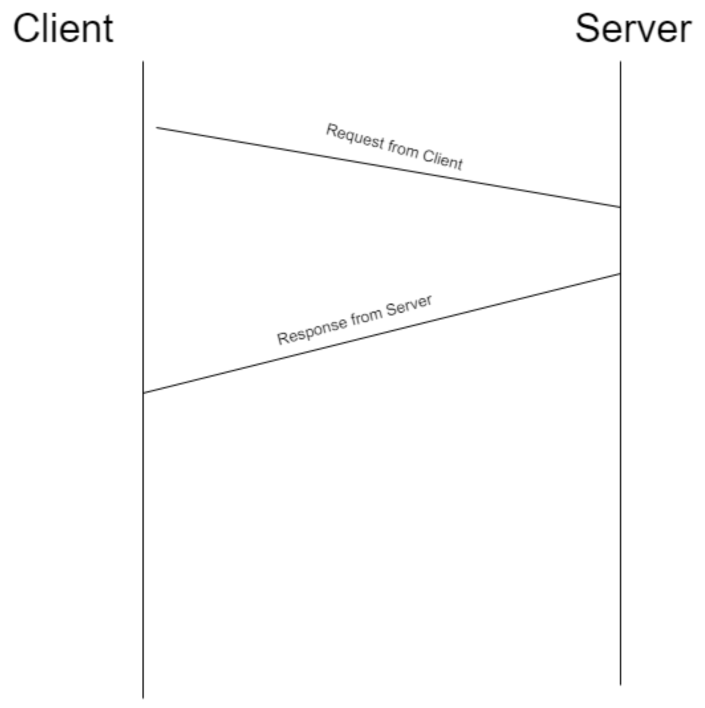
Group chat functionality is another highlight of my server. The **create_group** and **join_group** methods manage the creation and management of group chats. These functions not only handle the logistics of group messaging but also ensure that groups are correctly validated and managed.

The **request processing function** process is the heart of my server's interaction with clients. It handles a variety of client requests, from user logins to messaging and setting up video file transfers.

Lastly, the main execution block of my code, which parses command-line arguments for server configurations and starts the server sets the stage for the server's operation and ensures that the server is configured correctly before it starts accepting client connections.

Diagram of my work:

This is for the basic client server interaction where client sends a request and the server responds to it. The client then processes the server's request.



Final Comments:

While the application successfully implements peer-to-peer (P2P) video file transfer, there are some nuances to its operation that are worth noting. The P2P transfer feature performs reliably when both clients are operating within the same directory. However, challenges arise when clients are in different directories. While the transfer of video files still occurs, this scenario presents some operational complexities. This is an area identified for further refinement to enhance the robustness and flexibility of the P2P transfer functionality across varied operating environments

In terms of user logout functionality, the application effectively manages the disconnection and logout process for active users. However, there is an aspect of log management that could be enhanced. Currently, upon user logout, the system does not remove the user's activity log from the log file as per my design.

I have used UDP based communication for the p2pvideo transfer.
Uses for the features are:
Use of TCP (Transmission Control Protocol) for text messaging to ensure reliable, in-order delivery of messages.
Use of UDP (User Datagram Protocol) for video file transfers to prioritize low latency over reliability.

The application is runnable in a standard Python environment (version 3.11.6 as specified).