**A Simple Multi-threaded Chat Server**

**1. Introduction**

This document details the design and implementation of a basic multi-threaded chat server and its corresponding client. The system allows multiple clients to connect, communicate within chat rooms, send private messages, and transfer files. The server is written in Python and leverages the socket and threading libraries for network communication and concurrency.

**2. System Architecture**

The system follows a client-server architecture.

* **Server:** The central component responsible for managing client connections, chat rooms, message broadcasting, and file transfers.
* **Client:** The application that users interact with to connect to the server, send/receive messages, and transfer files.

**3. Server-Side Implementation (server.py)**

**3.1. Core Functionality**

* **Connection Handling:** The server uses the socket library to listen for incoming client connections. The threading library is used to create a new thread for each client, allowing for concurrent handling of multiple clients.
* **User Management:** Clients are prompted to enter a username upon connection. The server stores the username and the client's current room.
* **Room Management:** The server allows the creation, joining, and leaving of chat rooms. It maintains a list of users in each room.
* **Message Broadcasting:** The server broadcasts messages to all users in the same room.
* **Command Handling:** The server processes commands entered by users (e.g., /help, /join, /create).
* **File Transfer:** The server facilitates file transfers between clients using a chunking mechanism and JSON packets to coordinate the transfer.
* **Private Messaging:** The server supports private messages to specific users.
* **Message History:** The server maintains a history of recent messages in each room, allowing users to recall previous messages.

**3.2. Code Walkthrough**

* **Imports:** The code begins by importing necessary libraries: socket, threading, datetime, collections, and json.
* **Global Variables:** Key data structures like clients, user\_rooms, rooms, msg\_id\_counter, history, and active\_transfers are initialized. A data\_lock is used to ensure thread-safe access to these shared resources.
* timestamp()**:** A utility function to generate timestamps for messages and logs.
* broadcast\_message()**:** Sends a text message to all clients in a room, except for an optional excluded client.
* broadcast\_packet()**:** Sends a JSON packet to a list of recipients.
* handle\_client()**:** This function is executed in a separate thread for each client. It handles:
  + Receiving the username.
  + Receiving and processing messages from the client.
  + Calling handle\_text\_message(), handle\_file\_start(), handle\_file\_chunk(), or handle\_private\_msg() as appropriate.
  + Handling client disconnection.
* handle\_text\_message()**:** Processes regular text messages and calls handle\_command() if the message starts with a /.
* handle\_command()**:** Parses and executes commands entered by the user. This function contains the logic for all the chat commands (e.g., /join, /create, /sendfile).
* handle\_file\_start()**,** handle\_file\_chunk()**,** handle\_private\_msg()**:** Handle the respective packet types.
* disconnect\_client()**:** Cleans up client data when a client disconnects.
* main()**:** The main server loop that creates the socket, binds to a port, listens for connections, and creates a new thread for each incoming connection.

**4. Client-Side Implementation (client.py)**

**4.1. Core Functionality**

* **Connection:** The client connects to the server using the socket library.
* **Sending Messages:** The client sends text messages and commands to the server.
* **Receiving Messages:** The client receives and displays messages from the server.
* **File Sending:** The client allows users to select a file and send it to other users using the /sendfile command. It breaks the file into chunks and sends them to the server.
* **File Receiving:** The client receives file chunks from the server, reconstructs the file, and saves it to the "downloads" directory.
* **Private Messaging:** The client supports sending and receiving private messages.
* **User Highlighting:** The client highlights usernames with different colors to improve readability.

**4.2. Code Walkthrough**

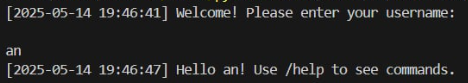
* **Imports:** The client imports socket, threading, datetime, re, sys, filedialog, json, base64, os.
* **Global Variables:** username, room, user\_colors, and file\_receivers are initialized.
* get\_color\_for\_user()**:** Generates an ANSI color code for a given username.
* timestamp()**:** Generates a timestamp.
* print\_message()**:** Prints a message, highlighting tagged users with colors.
* receive\_messages()**:** This function runs in a separate thread and handles receiving data from the server. It parses both text messages and JSON packets, handling file chunks and private messages appropriately.
* send\_messages()**:** This function handles sending data to the server. It reads user input, handles the /sendfile and /msg commands, and sends the data to the server.
* main()**:** The main client function that connects to the server, receives the welcome message, gets the username, starts the receive thread, and calls the send thread.

**5. Client Usage Examples**

The following examples illustrate common interactions with the chat client.

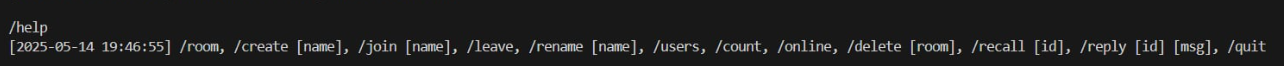
**5.1. Connecting and Setting Username**

Upon starting the client, the server prompts for a username. In this example, the user enters "an". The server acknowledges the connection and displays a welcome message along with the /help command.



**5.2. Viewing Available Commands**

The user enters the /help command to see a list of available commands supported by the server.



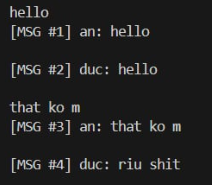
**5.3. Joining a Chat Room**

The user uses the /join command to enter the "room1" chat room.



**5.4. Sending and Receiving Messages**

Users in the same chat room can send and receive messages. Each message is prefixed with a message ID and the sender's username.



**5.5. Renaming, Managing Rooms, and Joining a New Room**

This image demonstrates several actions:

* /rename duc: The user "an" changes their username to "duc".
* /room: The user lists the available chat rooms.
* /create test: The user creates a new chat room named "test".
* /room: The user lists the available chat rooms again, now including "test".
* /join test: The user joins the newly created "test" room.



**5.6. Quitting the Client**

The user enters the /quit command to disconnect from the server. The server acknowledges the lost connection.



**6. Data Structures and Algorithms**

* **Dictionaries:** Dictionaries (clients, user\_rooms, rooms) are used extensively for efficient lookups.
* **Deques:** Deques (history) provide efficient insertion and removal from both ends, making them suitable for managing message history.
* **Threading:** The threading library is crucial for handling multiple clients concurrently, ensuring that the server remains responsive.
* **JSON:** JSON is used for structured data exchange, particularly for file transfer metadata and private messages.
* **Base64:** Base64 encoding is used to transmit binary file data within JSON packets.
* **File Chunking:** Files are broken down into smaller chunks for transmission, improving reliability and allowing for progress tracking.

**7. Potential Improvements**

* **Error Handling:** The code could benefit from more robust error handling (e.g., using try-except blocks more extensively, logging errors).
* **Scalability:** For a large number of users, the server's architecture might need to be optimized (e.g., using asynchronous I/O instead of threading).
* **Security:** The current implementation lacks security features (e.g., encryption, authentication). Adding these would be essential for a production system.
* **Message Persistence:** Currently, message history is stored in memory. Persisting messages to a database would allow for message retrieval after the server restarts.
* **File Storage:** Downloaded files are saved in a local "downloads" directory. A more sophisticated file management system could be implemented.
* **Protocol:** A more robust protocol could be defined for communication between client and server to handle various scenarios and data types more effectively.

**8. Conclusion**

The provided code implements a functional, albeit basic, multi-threaded chat server. It demonstrates core concepts such as socket programming, threading, data management, and client-server interaction. While there are areas for improvement, it serves as a good foundation for building a more advanced and feature-rich chat application. The use of threading enables concurrent client handling, and the use of JSON facilitates structured data exchange. Further development could focus on enhancing scalability, security, and user experience.