CSE 344 Final Project

210104004042
Recep Furkan Akın

1. Introduction and Problem Definition

This project implements a multi-threaded, TCP-based distributed chat and file-sharing system. The system follows a client-server model where a central server manages connections from multiple clients simultaneously. Clients can engage in private messaging, group messaging within chat rooms, and share files (simulated transfer) with each other.

The core challenges addressed include:

- **Concurrency Management:** Handling multiple client connections and operations concurrently using threads.
- **Synchronization:** Ensuring thread-safe access to shared resources (like client lists, room data, file queues, and client-specific file tracking) using mutexes, semaphores, and condition variables.
- **Network Programming:** Utilizing TCP sockets for reliable communication between clients and the server.
- **Resource Management:** Implementing a queue-based file upload manager to simulate limited system resources and process file transfer requests under load.
- **Robustness:** Graceful handling of client disconnections, server shutdown signals (SIGINT), and comprehensive input validation on both client and server sides.
- **Logging:** Maintaining detailed server-side logs for auditing, debugging, and tracking system events as per project specifications.

The system aims to provide a functional command-line chat experience with essential features like room management, private messaging, broadcasting, and simulated file sharing, all while adhering to specified technical requirements and design constraints.

2. Design Details

2.1. Overall Architecture

The system employs a client-server architecture:

- **Server:** A central, multi-threaded application that listens for incoming client connections on a specified port. It is responsible for:
  - Authenticating clients (username format and uniqueness validation).
  - Managing client sessions via dedicated handler threads.

  - Facilitating message relay for broadcasts within rooms and private whispers between users.
  - Managing chat rooms, including creation, client joining, and client leaving.
  - Handling file transfer requests: validating metadata, queuing valid requests, and orchestrating simulated transfers via a pool of worker threads. This includes managing filename collisions for recipients.
  - Logging all significant server-side events and client interactions.
- **Client:** A multi-threaded command-line application that connects to the server. It allows users to:
  - Log in with a unique, validated username.
  - Send and receive messages in real-time.
  - Join, leave, and broadcast messages within chat rooms.
  - Send private messages (whispers) to other users.
  - Initiate simulated file transfers to other users.
  - Receive notifications of incoming simulated file transfers.

2.2. Thread Model

Server-Side:

1. **Main Listening Thread:**
   - Initializes server state, subsystems (logging, rooms, file transfer), and the main listening socket.
   - Enters a loop calling select() on the listening socket to allow non-blocking checks for new connections and the server_is_runningflag.
   - Upon a new connection, accept()s it and spawns a new, detached **Client Handler Thread**.
   - This thread is also responsible for initiating shutdown if it's not handled by a signal handler directly (though the SIGINT handler is the primary mechanism).
2. **Client Handler Threads:**
   - One thread is dedicated to each connected client, created as detached (pthread_detach(pthread_self())) for automatic resource reclamation upon termination.
   - Each thread manages the full lifecycle for its client:
     - Handles the initial login process (receiving username, validation, sending success/failure).
     - Enters a message loop, using select() on the client's socket to check for incoming data and the client->is_active / g_server_state->server_is_running flags.
     - Parses received Message structs and dispatches commands (join, leave, broadcast, whisper, sendfile request) to appropriate handlers.

    - Interacts with shared server resources (client list, rooms, file queue) using prescribed synchronization mechanisms.
    - Terminates upon client disconnection (graceful via /exit or unexpected) or server shutdown, ensuring unregisterClient is called for cleanup.
3. **File Processing Worker Threads:**
    - A fixed pool of MAX_UPLOAD_QUEUE_SIZE (e.g., 5) worker threads is created at server startup. These threads are **joinable**.
    - Workers wait on two primary synchronization primitives:
        - available_upload_slots_sem: A semaphore limiting concurrent "processing" to MAX_UPLOAD_QUEUE_SIZE. Workers sem_timedwait() to acquire a slot.
        - queue_not_empty_cond: A condition variable that workers pthread_cond_timedwait() on if the file queue is empty.
    - When a worker acquires a slot and a task is available (signaled by a client handler), it:
        - Dequeues a FileTransferTask.
        - Logs wait time and simulates processing (e.g., sleep()).
        - Handles recipient lookup and filename collision resolution for the recipient.
        - Sends a MSG_FILE_TRANSFER_DATA notification to the recipient client.
        - sem_post() to release the processing slot.
    - These threads terminate when g_server_state->server_is_running is false and are explicitly joined by the main server thread during shutdown.

Client-Side:

1. **Main User Input Thread:**
    - Manages the primary interaction loop, prompting the user for commands.
    - Reads input from stdin via fgets().
    - Parses commands and their arguments.
    - Constructs and sends Message structs to the server over the TCP socket for commands like /join, /whisper, /sendfile, etc.
    - Handles the /exit command by setting the client->connected flag and sending a MSG_DISCONNECT to the server.
    - Uses select() to monitor both stdin and the read end of a shutdown_pipe_fds for graceful termination signals from the receiver thread or SIGINT handler.
2. **Message Receiver Thread:**
    - Created as joinable after successful login.
    - Dedicated to continuously listening for incoming Message structs from the server on the shared TCP socket.
    - Uses select() with a timeout on the socket to allow periodic checks of the client->connected flag.
    - Upon receiving a message, it processes it based on MessageType:

    - Displays broadcasts, whispers, and server notifications (e.g., room join/leave confirmations, errors, file transfer acceptance/rejection for files it sent).
    - Handles incoming MSG_FILE_TRANSFER_DATA (as a recipient) by displaying a notification of the simulated file arrival.
  - If the connection is lost or client->connected becomes false, the thread terminates. It also writes to the shutdown_pipe_fds to unblock the input thread.
  - This thread is joined by the main client thread before the client program exits.

2.3. Interprocess Communication (IPC) and Synchronization

Client-Server Communication:

- **TCP Sockets:** All communication relies on a single, persistent TCP socket connection per client, ensuring reliable, ordered delivery of Messagestructs.
- **Message Struct:** A custom struct Message (defined in shared/protocol.h) serves as the application-level protocol, encapsulating command types, sender/receiver info, content, and file metadata.

Server-Side Synchronization (Intra-process):

- **Mutexes (pthread_mutex_t):**
  - g_server_state->clients_list_mutex: Protects the global g_server_state->connected_clients array (for adding/removing clients, finding by username) and the active_client_count.
  - g_server_state->rooms_list_mutex: Protects the global g_server_state->chat_rooms array (for finding/creating rooms) and the current_room_count.
  - ChatRoom->room_lock (per room): Protects a specific room's members list and member_count during add/remove operations or when broadcasting to room members.
  - ClientInfo->received_files_lock (per client): Protects a specific client's received_filenames list and num_received_filescount, used during filename collision checks by file worker threads.
  - FileUploadQueue->queue_access_mutex: Protects the shared file upload queue's head, tail pointers, and current_queue_length.
  - server_log_mutex (static in logging.c): Ensures sequential, atomic writes to the server.log file from any thread.
- **Condition Variables (pthread_cond_t):**
  - FileUploadQueue->queue_not_empty_cond: Paired with queue_access_mutex. File worker threads pthread_cond_timedwait() on this if the queue is empty. Client handler threads (when adding a file task) pthread_cond_signal() to wake up one waiting worker.
- **Semaphores (sem_t):**

    - FileUploadQueue->available_upload_slots_sem: Initialized to MAX_UPLOAD_QUEUE_SIZE. File worker threads sem_timedwait() on this to acquire a processing slot before dequeuing. They sem_post() after completing a task to release the slot, effectively limiting concurrent file processing.

Client-Side Synchronization/Notification (Intra-process):

- **Pipe (shutdown_pipe_fds):** A self-pipe mechanism. The SIGINT handler or the message receiver thread (upon detecting disconnection) writes a byte to the pipe's write end. The main input thread and the message receiver thread include the pipe's read end in their select()calls. Activity on this pipe signals them to check the client->connected flag and terminate gracefully.
- **volatile sig_atomic_t connected flag:** Used by all client threads to check connection status. Modified by the input thread (/exit), SIGINT handler, or receiver thread (on connection loss).

2.4. Key Data Structures

- **Message (shared/protocol.h):** As defined for client-server communication. Fields include type, sender, receiver, room, content, filename, file_size.
- **ClientInfo (server/common.h):** Server's representation of a client. Includes socket_fd, username, current_room_name, client_address, thread_id, is_active, connection_time, received_filenames array, num_received_files, and received_files_lock.
- **ChatRoom (server/common.h):** Server's room structure. Includes name, members (array of ClientInfo*), member_count, and room_lock.
- **FileTransferTask (server/common.h):** Represents a queued file transfer. Includes filename, sender_username, receiver_username, file_size, enqueue_timestamp, and next_task pointer.
- **FileUploadQueue (server/common.h):** Manages the file queue. Includes head, tail, current_queue_length, queue_access_mutex, queue_not_empty_cond, and available_upload_slots_sem.
- **ClientState (client/common.h):** Client's local state. Includes socket_fd, username, current_room, connected flag, receiver_thread_id, and shutdown_pipe_fds.
- **ServerMainState (server/common.h):** Global server state. Encapsulates connected_clients array, chat_rooms array, active_client_count, current_room_count, the FileUploadQueue instance, global mutexes, server_listen_socket_fd, server_is_running flag, and file_worker_thread_ids.

2.5. Message Protocol

The system uses a binary protocol based on sending and receiving the Message struct directly over TCP sockets.

- The MessageType enum within the Message struct dictates the purpose of the message (e.g., MSG_LOGIN, MSG_BROADCAST, MSG_FILE_TRANSFER_REQUEST, MSG_SUCCESS, MSG_ERROR).
- Based on the MessageType, specific fields in the Message struct are populated with relevant data (e.g., sender for login, room and content for broadcast).
- sendMessage() and receiveMessage() utility functions (in shared/utils.c) abstract the raw send() and recv() calls for the Message struct. Null termination of strings within the struct is ensured by the sending side or re-verified upon reception for safety.

2.6. File Transfer Mechanism (Simulated)

1. **Client Request (Sender):**
   - User types /sendfile .
   - The client application validates the local file (existence, type: .txt, .pdf, .jpg, .png, size: max 3MB).
   - If valid, a MSG_FILE_TRANSFER_REQUEST message containing sender username, target username, filename (basename only), and file size is sent to the server. **No actual file data is sent by the client.**
2. **Server Validation & Queuing:**
   - The server receives MSG_FILE_TRANSFER_REQUEST.
   - It validates the metadata: file type (redundant if client validated, but good practice), file size against MAX_FILE_SIZE, existence and activity of the recipient user, and ensures sender is not the recipient.
   - If all checks pass, the server creates a FileTransferTask (without any file data buffer, as it's a simulation). This task is enqueued into the FileUploadQueue. The queue_not_empty_cond is signaled to potentially wake a file worker thread.
   - The server responds to the sender client with MSG_FILE_TRANSFER_ACCEPT (e.g., "File request accepted and added to upload queue.") or MSG_FILE_TRANSFER_REJECT if validation failed (e.g., "Recipient offline," "File too large").
3. **Server File Processing (Worker Thread - Simulation):**
   - An available file worker thread (from the pool, limited by available_upload_slots_sem) dequeues the FileTransferTask.
   - The worker logs the time the task spent in the queue (logEventFileTransferProcessingStart).
   - It simulates processing delay using sleep() (e.g., 2 seconds).
4. **Server-Side Filename Collision Handling & Recipient Notification:**
   - The worker thread looks up the recipient client again to ensure they are still active.
   - It then checks the recipient ClientInfo's received_filenames list (protected by received_files_lock) for potential filename collisions.
   - If task->filename already exists in the recipient's list, generate_collided_filename() is used to create a new name (e.g., original_1.ext, original_2.ext). This collision and

  renaming is logged by the server (logEventFileCollision).
  - The worker sends a MSG_FILE_TRANSFER_DATA message to the recipient client. This message contains the original sender's username, the final (potentially renamed) filename, and the file size. **No actual file data is sent.**
  - The server logs the (simulated) completion (logEventFileTransferCompleted).
5. **Client File Reception Notification (Recipient - Simulation):**
  - The recipient client's message receiver thread receives the MSG_FILE_TRANSFER_DATA.
  - It displays a notification to the user, like: [FILE TRANSFER]: Received notification for file 'final_filename.ext' (size bytes) from sender_user. followed by [INFO]: This is a simulated transfer. No actual file content was transmitted or saved locally. **No file is saved locally by the client.**

2.7. Logging

The server maintains a detailed log file (server.log) with timestamped entries for auditing and debugging.

- **Format:** YYYY-MM-DD HH:MM:SS - [TAG] Log Message Details
- **Tags:** [INFO], [CONNECT], [COMMAND], [ROOM], [FILE-QUEUE], [FILE], [SEND FILE], [ERROR], [SHUTDOWN_CTRLC], [SHUTDOWN], [REJECTED], [LOGIN_FAIL], [FILE_ERROR], [WARNING], etc., are used to categorize events.
- **Logged Events Include:**
  - Server startup, listening port, subsystem initialization.
  - Client connection attempts, successful logins (with IP), login failures (with reason, e.g., duplicate username, invalid format).
  - Client disconnections (graceful /exit or unexpected "lost connection").
  - Room creation.
  - Client joining a room, leaving a room, switching rooms.
  - Broadcast messages (sender, target room).
  - Whisper messages (sender, receiver).
  - File transfer initiation (sender, receiver, original filename).
  - File request queued (sender, filename, current queue size).
  - File rejection due to oversized file (sender, filename).
  - Start of (simulated) file processing by a worker (sender, filename, time spent in queue).
  - Filename collision detected and resolved by the server for a recipient (original name, new name, recipient, sender).
  - (Simulated) file transfer completion notification sent to recipient (sender, receiver, final filename).
  - File transfer failures at various stages (with reason).
  - SIGINT reception and summary of shutdown (e.g., number of clients disconnected).

    - Server resource cleanup and final shutdown.
- A dedicated pthread_mutex_t (server_log_mutex) ensures that log writes from concurrent threads are serialized, preventing interleaved or corrupted log entries.

2.8. Error Handling and Input Validation

Robust error handling and input validation are implemented on both client and server sides:

- **Client-Side:**
    - **Command Parsing:** Checks for valid command syntax and required arguments.
    - **Username/Room Name Validation:** Enforces alphanumeric and length constraints (isValidUsername, isValidRoomName from shared/utils.c).
    - **File Validation:** Before sending a /sendfile request, the client checks:
        - File existence and readability.
        - File type against allowed extensions (.txt, .pdf, .jpg, .png) using isValidFileType.
        - File size against MAX_FILE_SIZE (3MB).
    - **User Feedback:** Provides clear, color-coded messages for successful operations (green) and errors (red).
- **Server-Side:**
    - **Connection Handling:** Manages MAX_SERVER_CLIENTS limit, gracefully rejecting excess connections.
    - **Login Validation:**
        - Username format and length (using isValidUsername).
        - Username uniqueness (protected by clients_list_mutex).
    - **Room Management:**
        - Room name format and length (using isValidRoomName).
        - Room capacity (MAX_MEMBERS_PER_ROOM).
        - Handles cases where a client tries to operate on a room they are not in or a non-existent room (though creation is automatic on join).
    - **File Transfer Validation:**
        - Redundant checks for file type and size from MSG_FILE_TRANSFER_REQUEST (defense-in-depth).
        - Recipient existence and activity status.
        - Prevents sending files to oneself.
    - **Socket Operations:** Checks return values of send, recv, select, socket, bind, listen, accept and logs errors.
    - **Resource Allocation:** Checks malloc return values.
    - **Concurrency Primitives:** Checks return values of pthread_mutex_init/destroy, pthread_cond_init/destroy, sem_init/destroy, pthread _create.

- **Client Disconnections:** Detects unexpected disconnections (e.g., recv returns 0 or error) and cleans up resources via unregisterClient. Handles graceful /exit disconnections.
- **SIGINT Shutdown:** Implements a signal handler for SIGINT to trigger a graceful shutdown sequence (flag setting, closing listen socket, notifying clients, joining worker threads, cleaning all resources, finalizing logs).
- **Error Messages to Client:** Sends specific error messages (e.g., MSG_LOGIN_FAILURE, MSG_ERROR, MSG_FILE_TRANSFER_REJECT) to inform the client of issues.

3. Issues Faced and How They Were Solved

1. **Issue: Ensuring Thread-Safe Access to Shared Server Resources.**
    - **Problem:** Concurrent access by multiple Client Handler Threads to shared data like the global client list, room lists, individual room member arrays, and the file upload queue could lead to race conditions, data corruption, or inconsistent application state.
    - **Solution:** A comprehensive set of pthread_mutex_t locks was implemented:
        - clients_list_mutex for the global list of ClientInfo pointers and active_client_count.
        - rooms_list_mutex for the global array of ChatRoom structures and current_room_count.
        - A dedicated room_lock within each ChatRoom struct for its own members list and member_count, allowing finer-grained concurrency for room operations.
        - A received_files_lock within each ClientInfo struct for its list of received filenames (used for collision detection).
        - queue_access_mutex for the FileUploadQueue's linked list pointers and length.
        - A static server_log_mutex in logging.c for serializing writes to the log file. Locks are acquired before accessing the protected resource and released promptly after the critical section.
2. **Issue: Managing a Limited Number of Concurrent File "Uploads" (Simulated).**
    - **Problem:** The requirement was to simulate a system with limited resources, allowing only a fixed number (e.g., 5) of file transfers to be "processed" concurrently, with others being queued.
    - **Solution:** The file transfer subsystem uses a producer-consumer model with a fixed-size pool of worker threads:
        - A sem_t (available_upload_slots_sem) is initialized to MAX_UPLOAD_QUEUE_SIZE. Worker threads must sem_timedwait() on this semaphore to acquire a "processing slot" before taking a task from the queue. They sem_post() after finishing, making the slot available. This directly limits concurrency.

        - A pthread_cond_t (queue_not_empty_cond), used with queue_access_mutex, allows worker threads to wait efficiently (via pthread_cond_timedwait()) if the task queue is empty, preventing busy-waiting while still allowing periodic checks of the server shutdown flag.

3. **Issue: Implementing a Graceful Server Shutdown on SIGINT (Ctrl+C).**
    - **Problem:** Abrupt termination would lose state, leave clients hanging, and prevent proper resource cleanup and log finalization.
    - **Solution:** A multi-step graceful shutdown process was implemented:
        - A SIGINT signal handler (sigintShutdownHandler) sets g_server_state->server_is_running = 0.
        - It closes the main server_listen_socket_fd to stop new connections.
        - It signals file worker threads (via pthread_cond_broadcast and sem_post on their respective synchronization objects) to wake up, notice server_is_running is false, and terminate.
        - The main server thread, after its acceptClientConnectionsLoop exits (due to server_is_running becoming false or the listen socket closing), calls cleanupServerResources().
        - cleanupServerResources() then explicitly joins all file worker threads.
        - It iterates through active client connections, sends them a shutdown notification, and closes their sockets. Client handler threads (which are detached) detect the socket closure or the server_is_running flag and call unregisterClient for their own cleanup.
        - All mutexes, semaphores, and other resources are destroyed/closed.
        - The log file is finalized.
4. **Issue: Achieving Client-Side Responsiveness for Concurrent User Input and Server Message Reception.**
    - **Problem:** The client needs to simultaneously listen for user commands typed into the terminal and for messages arriving from the server without either operation blocking the other.
    - **Solution:** The client application is multi-threaded:
        - The **main thread** handles user input from stdin. It uses select() to be able to also monitor a pipe for shutdown signals.
        - A separate **message receiver thread** is created after login. It uses select() with a timeout on the server socket to listen for incoming messages while also periodically checking a global client->connected flag for shutdown.
        - A **pipe** (shutdown_pipe_fds) is used for inter-thread communication during shutdown. If SIGINT is caught or the receiver thread detects a server disconnect, it writes to the pipe, which select() in the input thread (and receiver thread itself) will detect, prompting an exit.
5. **Issue: Preventing Zombie Processes from Server-Side Client Handler Threads.**

- **Problem:** If the main server thread created client handler threads as joinable but never called pthread_join() on them (which is impractical for a dynamic number of clients), they would become zombie processes upon termination.
- **Solution:** Client handler threads are explicitly detached using pthread_detach(pthread_self()) shortly after they are created. This allows the operating system to automatically reclaim their resources when they exit, without requiring the main thread to join them. File worker threads, however, are created as joinable and are explicitly joined during server shutdown to ensure their tasks are completed or properly abandoned.

6. **Issue: Robust Detection and Handling of Client Disconnections (Graceful and Unexpected).**
   - **Problem:** Clients can disconnect by sending /exit or by abruptly closing their terminal/losing network. The server must handle both cases cleanly.
   - **Solution:**
     - **Graceful (/exit):** The client sends a MSG_DISCONNECT. The server-side client handler thread receives this, sets client->is_active = 0, and its main loop terminates. unregisterClient is then called with is_unexpected_disconnect = false.
     - **Unexpected:** If recv() in the client handler thread returns 0 (connection closed by peer) or a critical error (<0), and client->is_active was true, the loop terminates. unregisterClient is then called with is_unexpected_disconnect = true.
     - In both cases, unregisterClient performs comprehensive cleanup: logs the event, removes the client from any room (notifying other room members), closes the socket, removes the client from the global list, decrements active_client_count, and frees the ClientInfo struct and its associated mutex.

4. Test Cases and Results

Report Clarification on Message History (Test Scenario 8):
Message history for rooms is **ephemeral**. When a client leaves and rejoins a room, they **do not** receive previous messages sent while they were not in the room or before they initially joined. The system does not store message history for rooms.

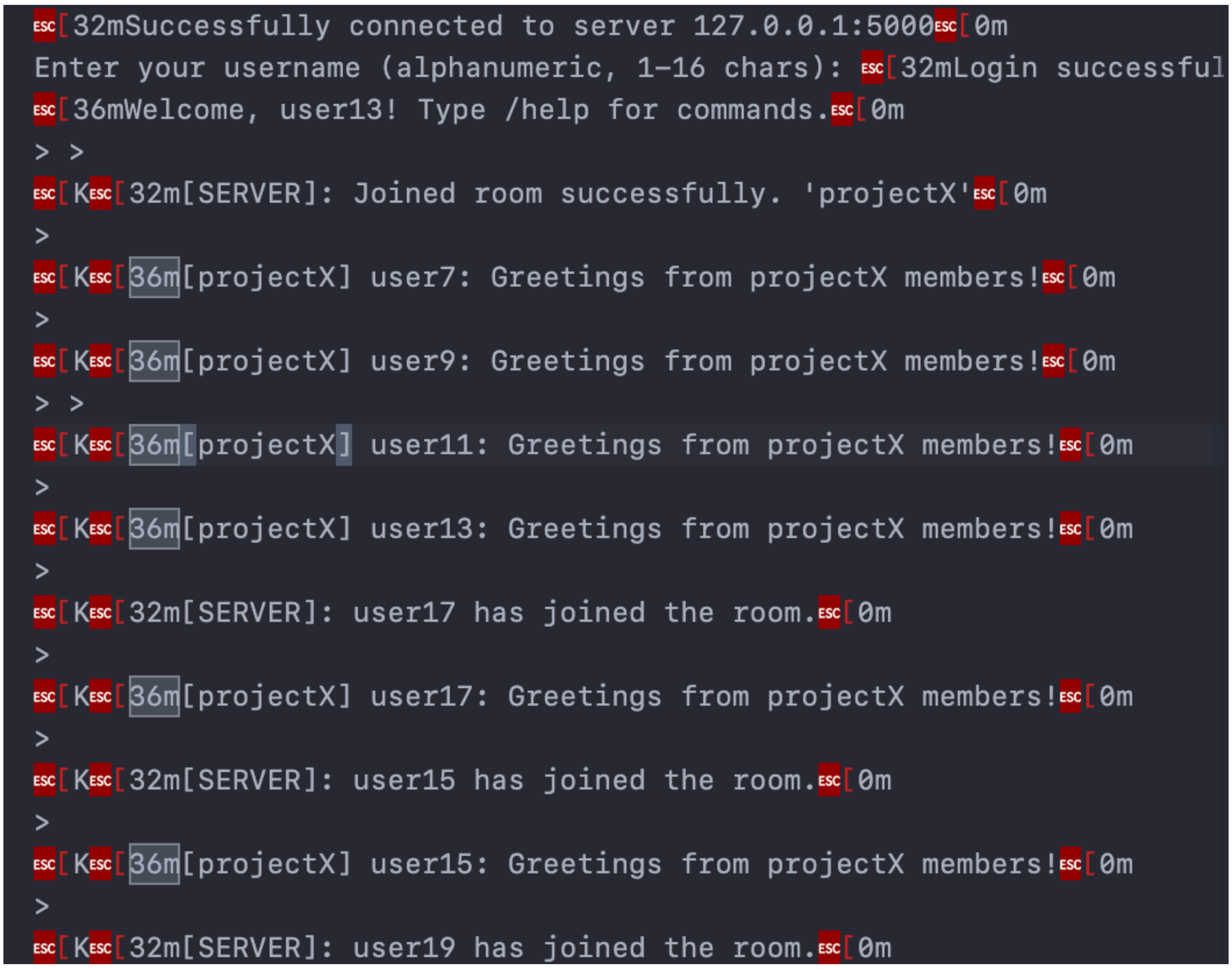
Report Clarification on Filename Collision (Test Scenario 9):
The server itself does not explicitly rename files if two users send a file with the same name to the same recipient *at the server's processing stage*. The file data is buffered under its original name. However, when the *recipient client* receives the file, it implements a check: if a file named `received_<filename>` already exists, it will attempt to save the new file as `received_<filename>_1`, `received_<filename>_2`, and so on. The server log will show successful transfer of the original filename; the client-side log/notification will reflect the actual saved name.

**(For each test scenario below, please provide:

- A brief description of how you set up the test.
- Relevant screenshot(s) from client terminal(s).
- Relevant screenshot(s) or copy-pasted snippets from the server console/log (`server.log`).)**

4.1. Concurrent User Load

- **Test:** At least 30 clients connect simultaneously and interact with the server (join rooms, broadcast, whisper).
- **Expected:** All users are handled correctly, no message loss, no crash.
- **Setup:**
    - I used launch_clients.sh script generated by AI to test concurrent login to system.
- **Client Screenshot(s):**



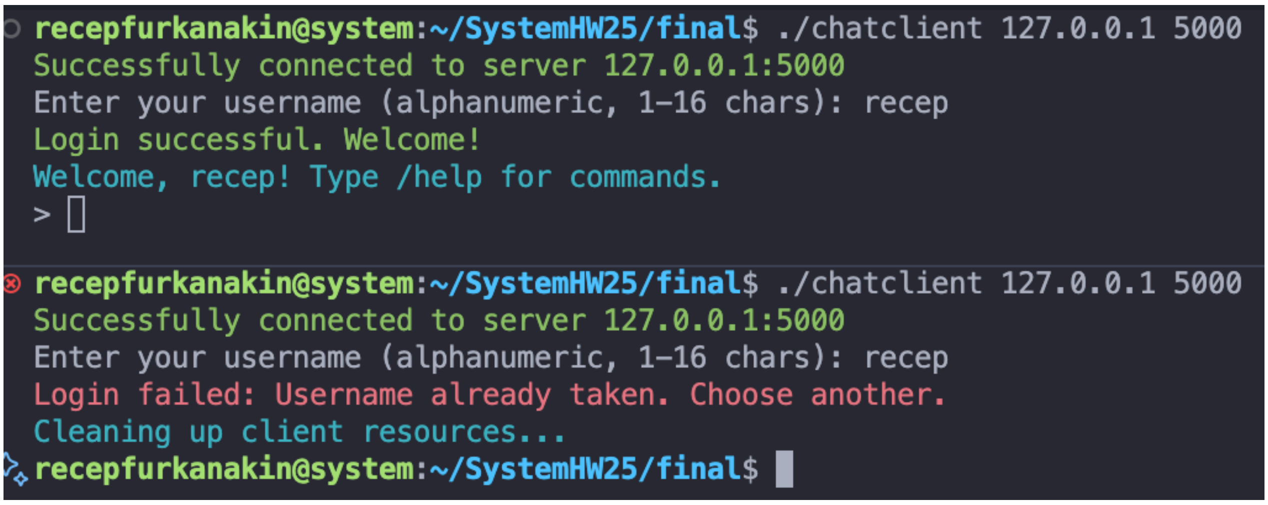
```
ESC[32mSuccessfully connected to server 127.0.0.1:5000ESC[0m
Enter your username (alphanumeric, 1-16 chars): ESC[32mLogin successful
ESC[36mWelcome, user13! Type /help for commands.ESC[0m
> >
ESC[KESC[32m[SERVER]: Joined room successfully. 'projectX'ESC[0m
>
ESC[KESC[36m[projectX] user7: Greetings from projectX members!ESC[0m
>
ESC[KESC[36m[projectX] user9: Greetings from projectX members!ESC[0m
> >
ESC[KESC[36m[projectX] user11: Greetings from projectX members!ESC[0m
>
ESC[KESC[36m[projectX] user13: Greetings from projectX members!ESC[0m
>
ESC[KESC[32m[SERVER]: user17 has joined the room.ESC[0m
>
ESC[KESC[36m[projectX] user17: Greetings from projectX members!ESC[0m
>
ESC[KESC[32m[SERVER]: user15 has joined the room.ESC[0m
>
ESC[KESC[36m[projectX] user15: Greetings from projectX members!ESC[0m
>
ESC[KESC[32m[SERVER]: user19 has joined the room.ESC[0m
```

- **Server Log (`server.log`):**

```
2025-05-31 16:10:32 - [LOGIN] user 'user20' connected from 127.0.0.1
2025-05-31 16:10:32 - [COMMAND] user1 broadcasted to 'projectX' (msg: "Greetings from projectX members!")
2025-05-31 16:10:32 - [COMMAND] user12 joined room 'general'
2025-05-31 16:10:32 - [COMMAND] user3 broadcasted to 'projectX' (msg: "Greetings from projectX members!")
2025-05-31 16:10:32 - [COMMAND] user5 broadcasted to 'projectX' (msg: "Greetings from projectX members!")
2025-05-31 16:10:32 - [COMMAND] user8 broadcasted to 'general' (msg: "Hello from general room!")
2025-05-31 16:10:33 - [COMMAND] user8 sent whisper to user17
2025-05-31 16:10:32 - [LOGIN] user 'user21' connected from 127.0.0.1
2025-05-31 16:10:32 - [COMMAND] user14 joined room 'general'
2025-05-31 16:10:33 - [COMMAND] user14 broadcasted to 'general' (msg: "Hello from general room!")
2025-05-31 16:10:32 - [COMMAND] user13 joined room 'projectX'
2025-05-31 16:10:32 - [COMMAND] user2 broadcasted to 'general' (msg: "Hello from general room!")
2025-05-31 16:10:32 - [COMMAND] user17 joined room 'projectX'
2025-05-31 16:10:32 - [COMMAND] user15 joined room 'projectX'
2025-05-31 16:10:32 - [LOGIN] user 'user22' connected from 127.0.0.1
2025-05-31 16:10:33 - [COMMAND] user2 sent whisper to user17
2025-05-31 16:10:32 - [LOGIN] user 'user23' connected from 127.0.0.1
2025-05-31 16:10:33 - [DISCONNECT] user 'user23' lost connection. Cleaned up resources.
2025-05-31 16:10:33 - [COMMAND] user20 joined room 'general'
2025-05-31 16:10:33 - [COMMAND] user12 broadcasted to 'general' (msg: "Hello from general room!")
2025-05-31 16:10:33 - [COMMAND] user16 joined room 'general'
2025-05-31 16:10:33 - [COMMAND] user16 broadcasted to 'general' (msg: "Hello from general room!")
2025-05-31 16:10:33 - [COMMAND] user1 sent whisper to user23
2025-05-31 16:10:33 - [COMMAND] user7 broadcasted to 'projectX' (msg: "Greetings from projectX members!")
2025-05-31 16:10:33 - [COMMAND] user9 broadcasted to 'projectX' (msg: "Greetings from projectX members!")
2025-05-31 16:10:33 - [COMMAND] user19 joined room 'projectX'
2025-05-31 16:10:33 - [LOGIN] user 'user25' connected from 127.0.0.1
2025-05-31 16:10:33 - [COMMAND] user3 sent whisper to user23
```

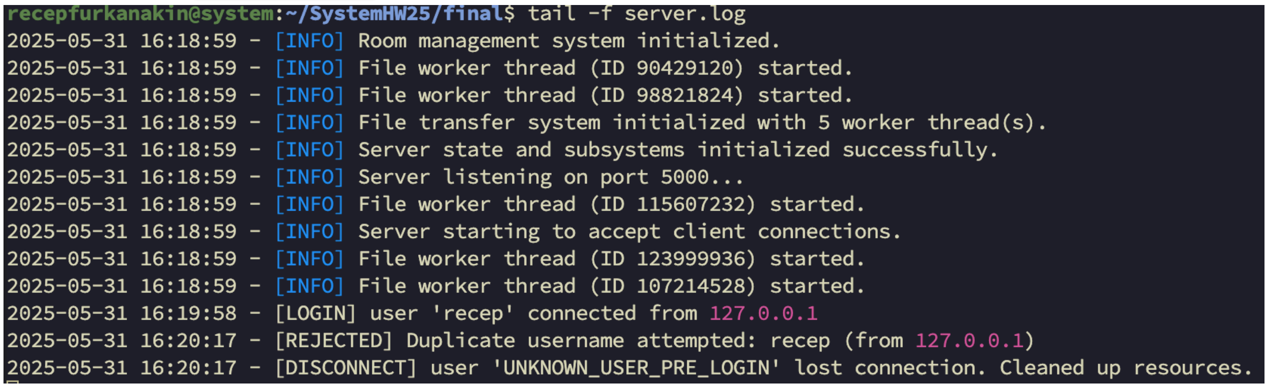
4.2. Duplicate Usernames

- **Test:** Two clients try to connect using the same username.
- **Expected:** Second client should receive a rejection message.
- **Setup:**
  - Client 1 connects with username "recep".
  - Client 2 attempts to connect with username "recep".
- **Client Screenshot(s):**



```
recepfurkanakin@system:~/SystemHW25/final$ ./chatclient 127.0.0.1 5000
Successfully connected to server 127.0.0.1:5000
Enter your username (alphanumeric, 1-16 chars): recep
Login successful. Welcome!
Welcome, recep! Type /help for commands.
> 

recepfurkanakin@system:~/SystemHW25/final$ ./chatclient 127.0.0.1 5000
Successfully connected to server 127.0.0.1:5000
Enter your username (alphanumeric, 1-16 chars): recep
Login failed: Username already taken. Choose another.
Cleaning up client resources...
recepfurkanakin@system:~/SystemHW25/final$ 
```

- **Server Log (`server.log`):**

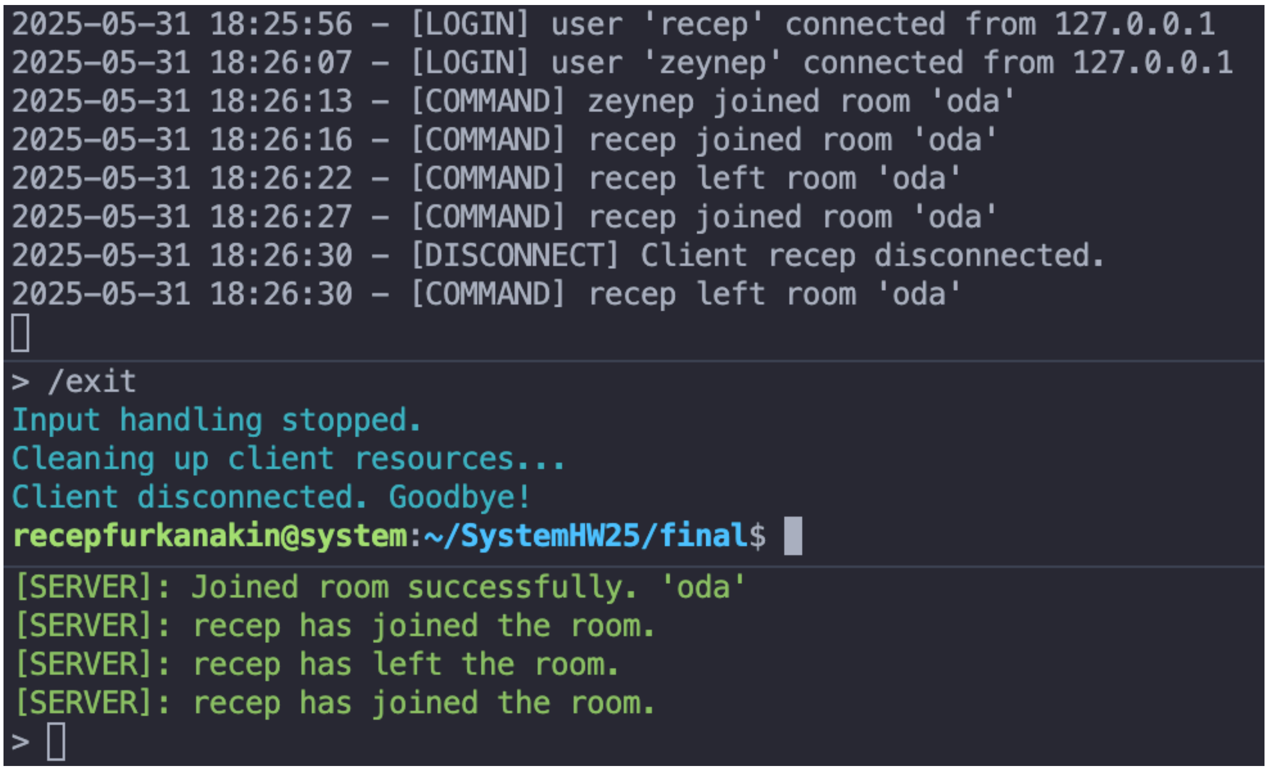


```
recepfurkanakin@system:~/SystemHW25/final$ tail -f server.log
2025-05-31 16:18:59 - [INFO] Room management system initialized.
2025-05-31 16:18:59 - [INFO] File worker thread (ID 90429120) started.
2025-05-31 16:18:59 - [INFO] File worker thread (ID 98821824) started.
2025-05-31 16:18:59 - [INFO] File transfer system initialized with 5 worker thread(s).
2025-05-31 16:18:59 - [INFO] Server state and subsystems initialized successfully.
2025-05-31 16:18:59 - [INFO] Server listening on port 5000...
2025-05-31 16:18:59 - [INFO] File worker thread (ID 115607232) started.
2025-05-31 16:18:59 - [INFO] Server starting to accept client connections.
2025-05-31 16:18:59 - [INFO] File worker thread (ID 123999936) started.
2025-05-31 16:18:59 - [INFO] File worker thread (ID 107214528) started.
2025-05-31 16:19:58 - [LOGIN] user 'recep' connected from 127.0.0.1
2025-05-31 16:20:17 - [REJECTED] Duplicate username attempted: recep (from 127.0.0.1)
2025-05-31 16:20:17 - [DISCONNECT] user 'UNKNOWN_USER_PRE_LOGIN' lost connection. Cleaned up resources.
```

4.3. File Upload Queue Limit

- I couldn't test this feature because of time issues. I hope I can show that in demo.

4.4. Unexpected Disconnection

- **Test:** A client closes the terminal or disconnects without using `/exit`.
- **Expected:** Server must detect and remove the client gracefully, update room states, and log the disconnection.
- 

```
2025-05-31 18:25:56 - [LOGIN] user 'recep' connected from 127.0.0.1
2025-05-31 18:26:07 - [LOGIN] user 'zeynep' connected from 127.0.0.1
2025-05-31 18:26:13 - [COMMAND] zeynep joined room 'oda'
2025-05-31 18:26:16 - [COMMAND] recep joined room 'oda'
2025-05-31 18:26:22 - [COMMAND] recep left room 'oda'
2025-05-31 18:26:27 - [COMMAND] recep joined room 'oda'
2025-05-31 18:26:30 - [DISCONNECT] Client recep disconnected.
2025-05-31 18:26:30 - [COMMAND] recep left room 'oda'
```

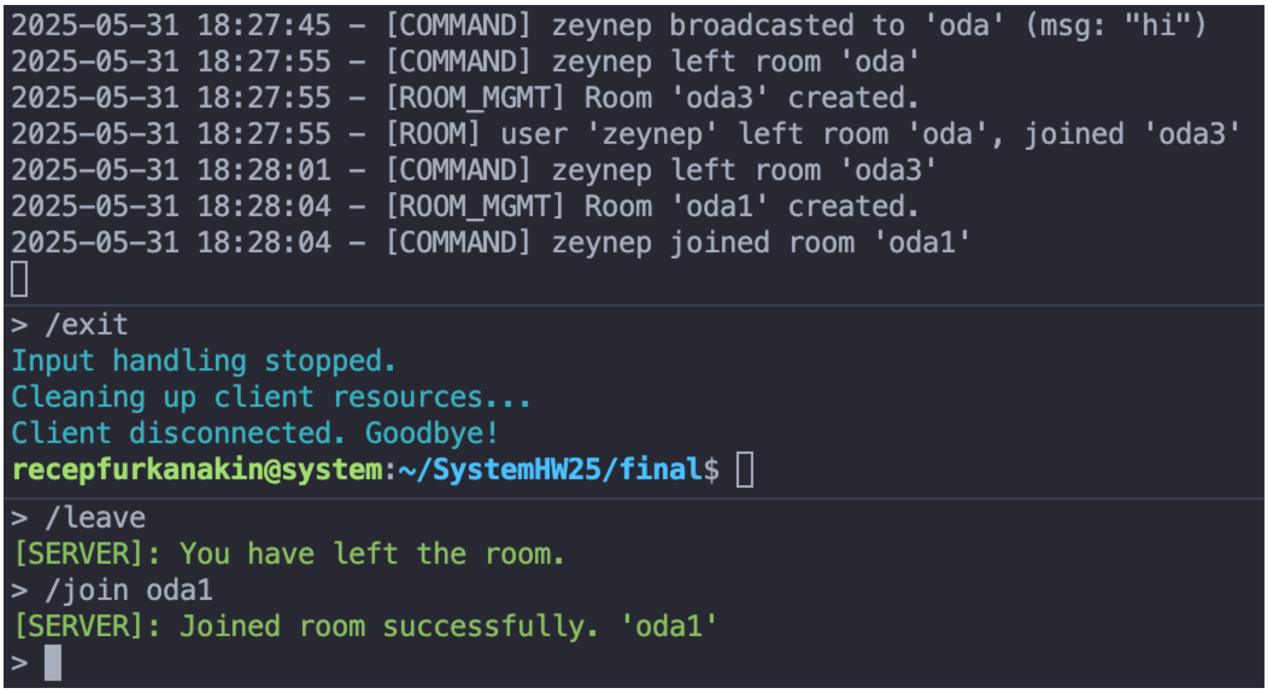
```
> /exit
Input handling stopped.
Cleaning up client resources...
Client disconnected. Goodbye!
recepfurkanakin@system:~/SystemHW25/final$
```

```
[SERVER]: Joined room successfully. 'oda'
[SERVER]: recep has joined the room.
[SERVER]: recep has left the room.
[SERVER]: recep has joined the room.
>
```

4.5. Room Switching

- **Test:** A client joins a room, then switches to another room.
- **Expected:** Server updates room states correctly. Messages are sent to the correct room.

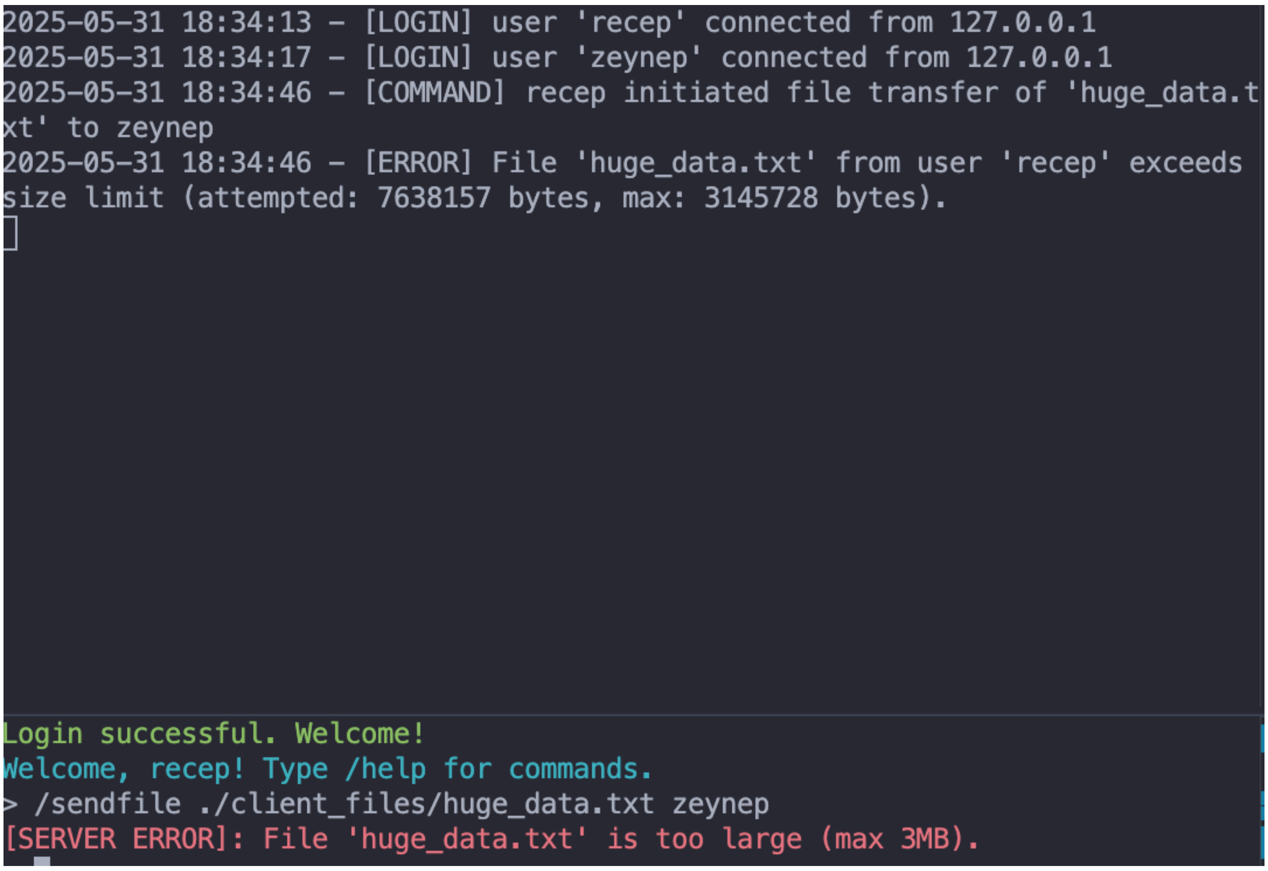
- **Client Screenshots:**



```
2025-05-31 18:27:45 - [COMMAND] zeynep broadcasted to 'oda' (msg: "hi")
2025-05-31 18:27:55 - [COMMAND] zeynep left room 'oda'
2025-05-31 18:27:55 - [ROOM_MGMT] Room 'oda3' created.
2025-05-31 18:27:55 - [ROOM] user 'zeynep' left room 'oda', joined 'oda3'
2025-05-31 18:28:01 - [COMMAND] zeynep left room 'oda3'
2025-05-31 18:28:04 - [ROOM_MGMT] Room 'oda1' created.
2025-05-31 18:28:04 - [COMMAND] zeynep joined room 'oda1'

> /exit
Input handling stopped.
Cleaning up client resources...
Client disconnected. Goodbye!
recepfurkanakin@system:~/SystemHW25/final$

> /leave
[SERVER]: You have left the room.
> /join oda1
[SERVER]: Joined room successfully. 'oda1'
>
```

4.6. Oversized File Rejection

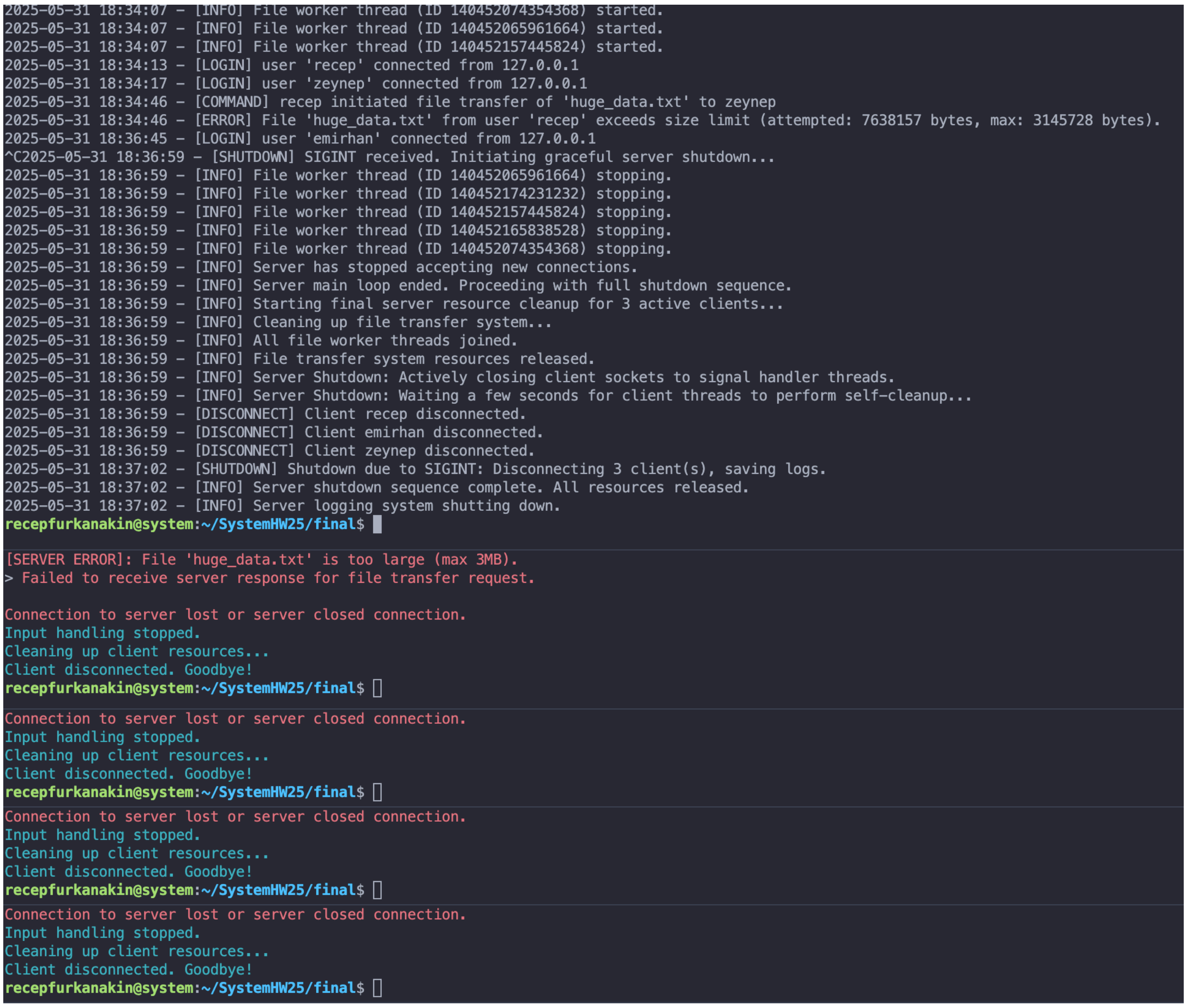
- **Test:** A client attempts to upload a file exceeding 3MB.
- **Expected:** File is rejected, user is notified.
- Normally I check it on the client side and don't send the file but for the test purposes I commented out the file size check part. So here is the log.
- **Client Screenshot(s) (from "uploader"):**



```
2025-05-31 18:34:13 - [LOGIN] user 'recep' connected from 127.0.0.1
2025-05-31 18:34:17 - [LOGIN] user 'zeynep' connected from 127.0.0.1
2025-05-31 18:34:46 - [COMMAND] recep initiated file transfer of 'huge_data.t
xt' to zeynep
2025-05-31 18:34:46 - [ERROR] File 'huge_data.txt' from user 'recep' exceeds
size limit (attempted: 7638157 bytes, max: 3145728 bytes).

Login successful. Welcome!
Welcome, recep! Type /help for commands.
> /sendfile ./client_files/huge_data.txt zeynep
[SERVER ERROR]: File 'huge_data.txt' is too large (max 3MB).
```

4.7. SIGINT Server Shutdown

- **Test:** Press Ctrl+C on server terminal.
- **Expected:** All clients are notified. Connections are closed gracefully. Logs are finalized before exit.
- **Client Screenshot(s):**



```
2025-05-31 18:34:07 - [INFO] File worker thread (ID 140452074354368) started.
2025-05-31 18:34:07 - [INFO] File worker thread (ID 140452065961664) started.
2025-05-31 18:34:07 - [INFO] File worker thread (ID 140452157445824) started.
2025-05-31 18:34:13 - [LOGIN] user 'recep' connected from 127.0.0.1
2025-05-31 18:34:17 - [LOGIN] user 'zeynep' connected from 127.0.0.1
2025-05-31 18:34:46 - [COMMAND] recep initiated file transfer of 'huge_data.txt' to zeynep
2025-05-31 18:34:46 - [ERROR] File 'huge_data.txt' from user 'recep' exceeds size limit (attempted: 7638157 bytes, max: 3145728 bytes).
2025-05-31 18:36:45 - [LOGIN] user 'emirhan' connected from 127.0.0.1
^C2025-05-31 18:36:59 - [SHUTDOWN] SIGINT received. Initiating graceful server shutdown...
2025-05-31 18:36:59 - [INFO] File worker thread (ID 140452065961664) stopping.
2025-05-31 18:36:59 - [INFO] File worker thread (ID 140452174231232) stopping.
2025-05-31 18:36:59 - [INFO] File worker thread (ID 140452157445824) stopping.
2025-05-31 18:36:59 - [INFO] File worker thread (ID 140452165838528) stopping.
2025-05-31 18:36:59 - [INFO] File worker thread (ID 140452074354368) stopping.
2025-05-31 18:36:59 - [INFO] Server has stopped accepting new connections.
2025-05-31 18:36:59 - [INFO] Server main loop ended. Proceeding with full shutdown sequence.
2025-05-31 18:36:59 - [INFO] Starting final server resource cleanup for 3 active clients...
2025-05-31 18:36:59 - [INFO] Cleaning up file transfer system...
2025-05-31 18:36:59 - [INFO] All file worker threads joined.
2025-05-31 18:36:59 - [INFO] File transfer system resources released.
2025-05-31 18:36:59 - [INFO] Server Shutdown: Actively closing client sockets to signal handler threads.
2025-05-31 18:36:59 - [INFO] Server Shutdown: Waiting a few seconds for client threads to perform self-cleanup...
2025-05-31 18:36:59 - [DISCONNECT] Client recep disconnected.
2025-05-31 18:36:59 - [DISCONNECT] Client emirhan disconnected.
2025-05-31 18:36:59 - [DISCONNECT] Client zeynep disconnected.
2025-05-31 18:37:02 - [SHUTDOWN] Shutdown due to SIGINT: Disconnecting 3 client(s), saving logs.
2025-05-31 18:37:02 - [INFO] Server shutdown sequence complete. All resources released.
2025-05-31 18:37:02 - [INFO] Server logging system shutting down.
recepfurkanakin@system:~/SystemHW25/final$ 
```

```
[SERVER ERROR]: File 'huge_data.txt' is too large (max 3MB).
> Failed to receive server response for file transfer request.

Connection to server lost or server closed connection.
Input handling stopped.
Cleaning up client resources...
Client disconnected. Goodbye!
recepfurkanakin@system:~/SystemHW25/final$ 
```

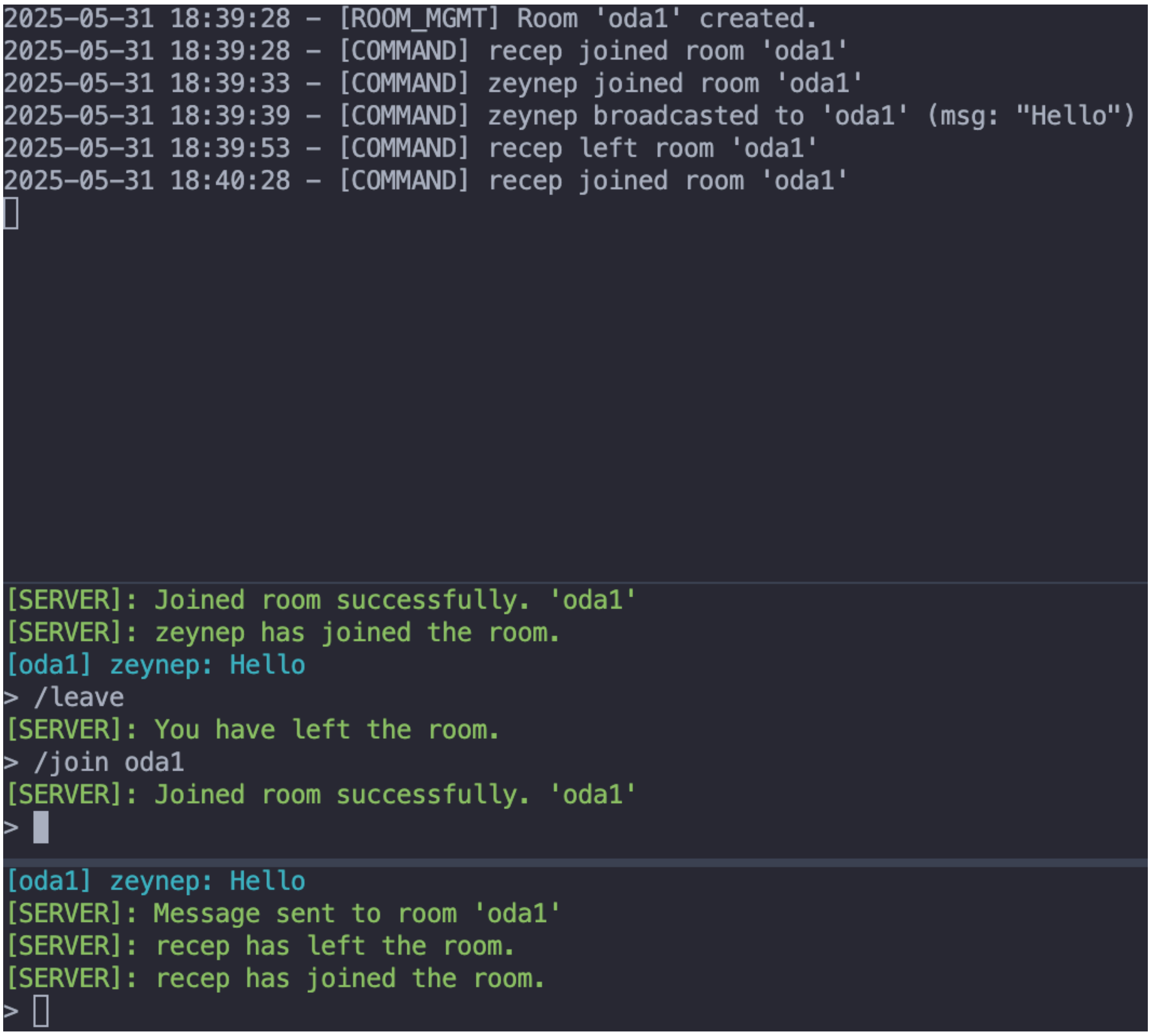
```
Connection to server lost or server closed connection.
Input handling stopped.
Cleaning up client resources...
Client disconnected. Goodbye!
recepfurkanakin@system:~/SystemHW25/final$ 
```

```
Connection to server lost or server closed connection.
Input handling stopped.
Cleaning up client resources...
Client disconnected. Goodbye!
recepfurkanakin@system:~/SystemHW25/final$ 
```

```
Connection to server lost or server closed connection.
Input handling stopped.
Cleaning up client resources...
Client disconnected. Goodbye!
recepfurkanakin@system:~/SystemHW25/final$ 
```

4.8. Rejoining Rooms

- **Test:** A client leaves a room, then rejoins.
- **Expected:** The client does not receive previous messages (unless message history is implemented).
    - **Message History:** As stated earlier, message history is **ephemeral**.

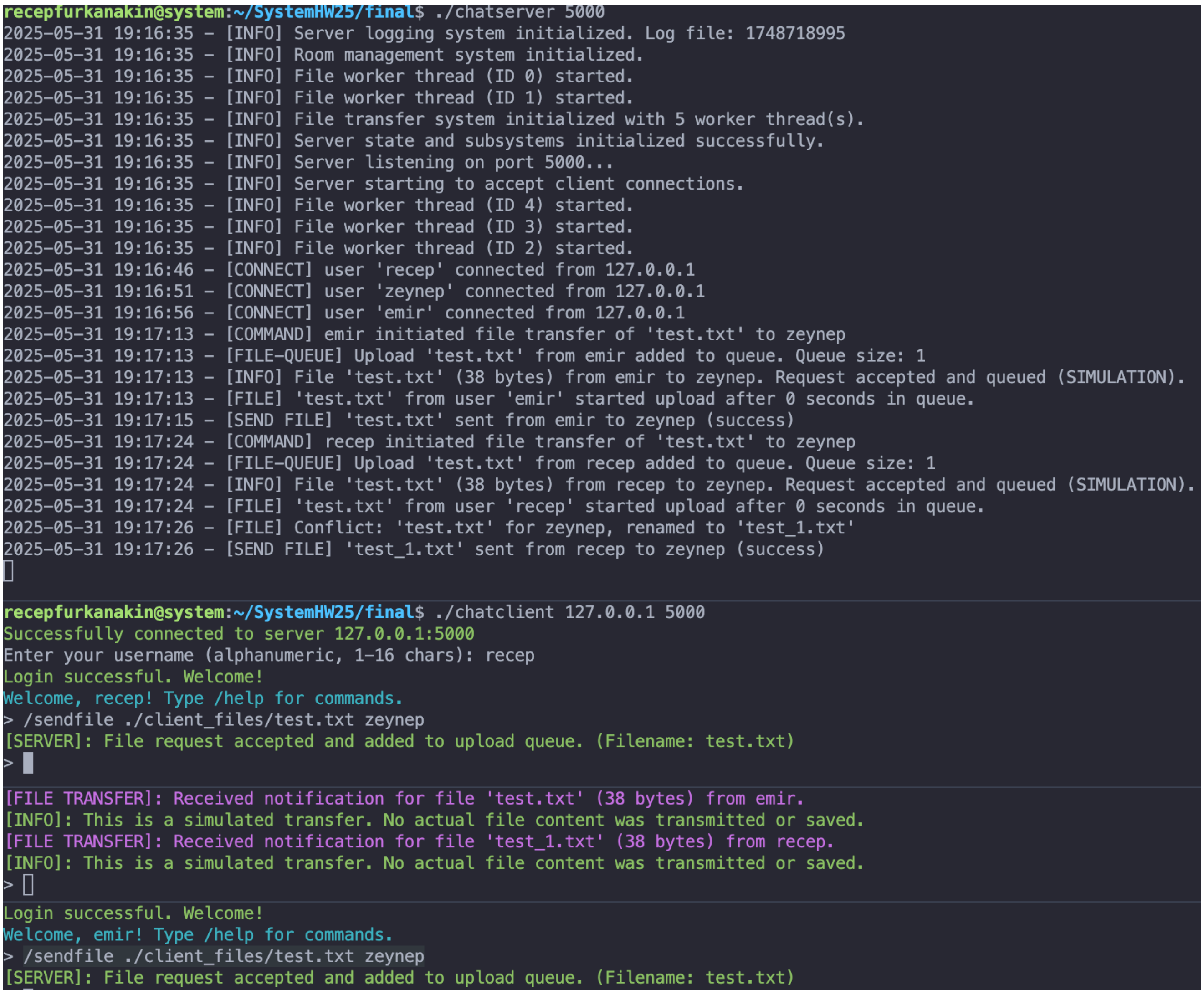


```
2025-05-31 18:39:28 - [ROOM_MGMT] Room 'oda1' created.
2025-05-31 18:39:28 - [COMMAND] recep joined room 'oda1'
2025-05-31 18:39:33 - [COMMAND] zeynep joined room 'oda1'
2025-05-31 18:39:39 - [COMMAND] zeynep broadcasted to 'oda1' (msg: "Hello")
2025-05-31 18:39:53 - [COMMAND] recep left room 'oda1'
2025-05-31 18:40:28 - [COMMAND] recep joined room 'oda1'
```

```
[SERVER]: Joined room successfully. 'oda1'
[SERVER]: zeynep has joined the room.
[oda1] zeynep: Hello
> /leave
[SERVER]: You have left the room.
> /join oda1
[SERVER]: Joined room successfully. 'oda1'
>
```

```
[oda1] zeynep: Hello
[SERVER]: Message sent to room 'oda1'
[SERVER]: recep has left the room.
[SERVER]: recep has joined the room.
>
```

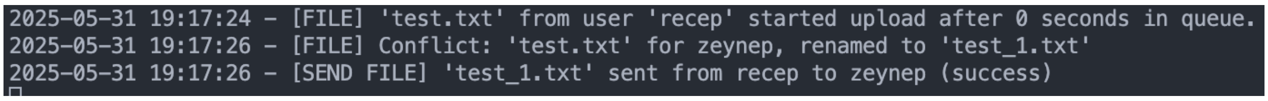
4.9. Same Filename Collision (Recipient Side)



```
recepfurkanakin@system:~/SystemHW25/final$ ./chatserver 5000
2025-05-31 19:16:35 - [INFO] Server logging system initialized. Log file: 1748718995
2025-05-31 19:16:35 - [INFO] Room management system initialized.
2025-05-31 19:16:35 - [INFO] File worker thread (ID 0) started.
2025-05-31 19:16:35 - [INFO] File worker thread (ID 1) started.
2025-05-31 19:16:35 - [INFO] File transfer system initialized with 5 worker thread(s).
2025-05-31 19:16:35 - [INFO] Server state and subsystems initialized successfully.
2025-05-31 19:16:35 - [INFO] Server listening on port 5000...
2025-05-31 19:16:35 - [INFO] Server starting to accept client connections.
2025-05-31 19:16:35 - [INFO] File worker thread (ID 4) started.
2025-05-31 19:16:35 - [INFO] File worker thread (ID 3) started.
2025-05-31 19:16:35 - [INFO] File worker thread (ID 2) started.
2025-05-31 19:16:46 - [CONNECT] user 'recep' connected from 127.0.0.1
2025-05-31 19:16:51 - [CONNECT] user 'zeynep' connected from 127.0.0.1
2025-05-31 19:16:56 - [CONNECT] user 'emir' connected from 127.0.0.1
2025-05-31 19:17:13 - [COMMAND] emir initiated file transfer of 'test.txt' to zeynep
2025-05-31 19:17:13 - [FILE-QUEUE] Upload 'test.txt' from emir added to queue. Queue size: 1
2025-05-31 19:17:13 - [INFO] File 'test.txt' (38 bytes) from emir to zeynep. Request accepted and queued (SIMULATION).
2025-05-31 19:17:13 - [FILE] 'test.txt' from user 'emir' started upload after 0 seconds in queue.
2025-05-31 19:17:15 - [SEND FILE] 'test.txt' sent from emir to zeynep (success)
2025-05-31 19:17:24 - [COMMAND] recep initiated file transfer of 'test.txt' to zeynep
2025-05-31 19:17:24 - [FILE-QUEUE] Upload 'test.txt' from recep added to queue. Queue size: 1
2025-05-31 19:17:24 - [INFO] File 'test.txt' (38 bytes) from recep to zeynep. Request accepted and queued (SIMULATION).
2025-05-31 19:17:24 - [FILE] 'test.txt' from user 'recep' started upload after 0 seconds in queue.
2025-05-31 19:17:26 - [FILE] Conflict: 'test.txt' for zeynep, renamed to 'test_1.txt'
2025-05-31 19:17:26 - [SEND FILE] 'test_1.txt' sent from recep to zeynep (success)

recepfurkanakin@system:~/SystemHW25/final$ ./chatclient 127.0.0.1 5000
Successfully connected to server 127.0.0.1:5000
Enter your username (alphanumeric, 1-16 chars): recep
Login successful. Welcome!
Welcome, recep! Type /help for commands.
> /sendfile ./client_files/test.txt zeynep
[SERVER]: File request accepted and added to upload queue. (Filename: test.txt)
> 

[FILE TRANSFER]: Received notification for file 'test.txt' (38 bytes) from emir.
[INFO]: This is a simulated transfer. No actual file content was transmitted or saved.
[FILE TRANSFER]: Received notification for file 'test_1.txt' (38 bytes) from recep.
[INFO]: This is a simulated transfer. No actual file content was transmitted or saved.
> 

Login successful. Welcome!
Welcome, emir! Type /help for commands.
> /sendfile ./client_files/test.txt zeynep
[SERVER]: File request accepted and added to upload queue. (Filename: test.txt)
```

4.10. File Queue Wait Duration

- I implemented duration but I just couldnt fill the queue with so many clients because of test issues. Bu time counter is working perfect.



```
2025-05-31 19:17:24 - [FILE] 'test.txt' from user 'recep' started upload after 0 seconds in queue.
2025-05-31 19:17:26 - [FILE] Conflict: 'test.txt' for zeynep, renamed to 'test_1.txt'
2025-05-31 19:17:26 - [SEND FILE] 'test_1.txt' sent from recep to zeynep (success)
```

5. Conclusion

This project successfully demonstrates the design and implementation of a multi-threaded, TCP-based distributed chat and file-sharing (simulated) server system with corresponding clients. The system effectively manages concurrent client connections, ensures thread-safe access to shared data structures through mutexes, semaphores, and condition variables, and handles client interactions such as room management, private messaging, broadcasting, and queued file transfer requests. Key features like robust error handling, graceful server shutdown via SIGINT, comprehensive client-side input validation, and detailed server-side logging have been implemented as per the project requirements.

The system achieves the primary objectives laid out, providing valuable hands-on experience in concurrent network programming, synchronization primitives, and robust system design. The use of a well-defined message protocol and clear separation of concerns between client and server, as well as between different server subsystems (client handling, room management, file transfer), contributes to a maintainable codebase.