GEBZE TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

CSE344 SISTEM PROGRAMMING

SPRING 2025

FINAL PROJECT

220104004962 – Burak SAĞLAM

CONTENTS

1. Introduction and Problem Definition

This project involves the development of a multi-user, multi-threaded, TCP-based distributed chat and file-sharing system using the C programming language. The primary goal is to implement a central server capable of handling multiple client connections concurrently, allowing clients to exchange private and group messages, and share files. Key system programming principles, including network programming with sockets, concurrency management with threads, and inter-thread communication/synchronization mechanisms (mutexes, semaphores), are applied.

The server is responsible for accepting connections, managing individual client sessions via dedicated threads, handling user and room administration, and facilitating private/group messaging. A file upload queue is implemented to simulate limited system resources for file transfers. All significant server-side events are logged with timestamps.

The client application provides a command-line interface (CLI) for user interaction, enabling connection to the server and execution of commands for room operations, messaging, and initiating file transfers.

2. Design Details

2.1. Overall Architecture

The system is based on a client-server architecture. A central server application (chatserver), listens for incoming TCP connections on a specified port. Upon accepting a connection, the server spawns a new thread to handle communication with that specific client application (chatclient). The codebase is modularized into server/ and client/ directories, with a common_defs.h file for shared definitions.

2.2. Thread Model

Server-Side:

The main server thread is responsible for listening for and accepting new client connections via accept().

For each accepted client connection, a new dedicated thread (handle_client) is created using pthread_create(). As recommended in the Q&A PDF (sources [12], [14]), these threads are detached using pthread_detach() for efficient resource management upon termination.

A pool of worker threads (file_transfer_worker, NUM_FILE_WORKERS in quantity) is created at server startup to process file transfer requests from a shared queue. These worker threads are joinable, and the main server thread waits for them to complete during graceful shutdown.

Client-Side:

As specified in the Q&A PDF (source [16]), the client utilizes two primary threads to manage simultaneous user input and server message reception:

The main thread handles reading user input from the terminal (using fgets) and sending commands/messages to the server (using send).

A separate receiver thread (receive_messages) continuously listens for incoming messages from the server (using recv) on the same socket and displays them on the client's terminal.

2.3. Interprocess Communication (IPC) and Synchronization Mechanisms

The project employs the following IPC and synchronization mechanisms primarily for inter-thread communication within the server process:

Mutexes (pthread_mutex_t): Used extensively to protect shared data structures (client list, room list, file upload queue array) from race conditions during concurrent access by multiple threads.

Semaphores (sem_t): Used for managing the file upload queue as a producer-consumer system.

items_in_queue: A counting semaphore indicating the number of file transfer requests currently in the queue. Worker threads wait (sem_wait) on this semaphore if the queue is empty.

empty_slots_in_queue: A counting semaphore indicating the number of available empty slots in the queue (MAX_UPLOAD_QUEUE limit). Client handler threads attempting to enqueue a file request wait (sem_timedwait) on this semaphore if the queue is full.

2.4. Network Communication Protocol

All communication between the client and server occurs over TCP sockets using text-based messages.

Client-to-server commands typically follow the format /command_name <arguments>.

Server-to-client messages convey status (success/error, often color-coded as per PDF requirement), messages from other users, or server notifications.

For file transfer initiation, the client sends a /sendfile <filename> <target_user> command. This is translated by the client into an internal /sendfile_req <filename> <target_user> <filesize> message sent to the server.

2.5. File Transfer Mechanism

As per the Q&A PDF (sources [24], [25]), actual binary file content transfer is not mandatory, and a message-based simulation is acceptable. This project implements the file transfer as follows:

The client issues a /sendfile command. The client application verifies local file existence and retrieves its size, then sends a /sendfile_req <filename> <target_user> <filesize> message to the server.

The server receives this request. It performs checks for file size (max 3MB), file type (allowed types: .txt, .pdf, .jpg, .png), and target user existence.

If all checks pass and the file upload queue (capacity MAX_UPLOAD_QUEUE) is not full, the request is added to the queue. The client is notified with [Server]: File added to the upload queue.... If the queue is full, the client is informed to try again later (after a 10-second timeout on the server-side wait for a queue slot).

Worker threads dequeue requests from this queue.

A worker thread logs the time the file spent in the queue.

The actual file transfer is simulated by the worker thread.

Upon "completion" of the simulated transfer, the server creates a placeholder file in a server-side directory structure (server_files/<receiver_username>/<filename>). Filename collisions are handled by appending _N to the filename.

The sending client is notified with [Server]: File sent successfully. The receiving client (if online) is notified that a file has been "received". The server logs the successful transfer (e.g., [SEND FILE] 'filename' sent from sender to receiver (success)).

3. Issues Faced and Solutions

The development of a multi-threaded, networked application like this chat and file server presented several challenges. Overcoming these issues was a significant part of the learning experience, reinforcing concepts in system programming, concurrency, and network communication.

3.1. Concurrency and Synchronization Issues

Problem: Race Conditions and Deadlocks

Initial implementations for accessing shared resources—such as the global client list (clients), room structures (rooms), and the file upload queue (file_upload_queue)—led to potential race conditions where multiple threads could try to modify data concurrently, leading to inconsistent states.

A specific deadlock risk was identified in early versions of room management. For instance, if remove_client_from_system (holding clients_mutex) called remove_user_from_room, and remove_user_from_room (potentially holding a room_mutex) subsequently tried to acquire clients_mutex, a classic A-B B-A deadlock could occur if another thread held room_mutex and was waiting for clients_mutex. Another potential issue was a thread trying to re-lock a non-recursive mutex it already held (e.g., add_user_to_room calling broadcast_message_to_room where both attempted to lock the same room's mutex).

Solution: Mutexes, Semaphores, and Lock Ordering

To prevent race conditions, pthread_mutex_t mutexes were extensively used to protect critical sections where shared data was read or modified. This included clients_mutex for the clients array and active_client_count, rooms_mutex for the global rooms array management (like finding/creating rooms), and individual room_mutex for each room_t structure to protect its user list and status.

The file upload queue (file_upload_queue) was managed using a combination of file_queue_mutex for direct array access (enqueue/dequeue operations) and POSIX semaphores (items_in_queue, empty_slots_in_queue) to implement a bounded buffer (producer-consumer pattern). This ensured that client handler threads (producers) would wait if the queue was full, and worker threads (consumers) would wait if the queue was empty, preventing busy-waiting and ensuring thread-safe queue operations.

To resolve the identified deadlock risks, a strict lock ordering was reviewed and enforced. For instance, the logic for remove_user_from_room was refactored so it no longer directly modified client-specific global data (like clients[client_idx].current_room_idx) that would require clients_mutex; this responsibility was shifted to the caller which could manage the lock order correctly. The issue of a thread re-locking a non-recursive mutex it already held (e.g., in add_user_to_room -> broadcast_message_to_room) was resolved by ensuring broadcast_message_to_room did not attempt to re-lock the room mutex if its caller (like add_user_to_room) already held it, by making the caller responsible for locking around the call to broadcast_message_to_room.

3.2. Network Programming and Client Handling

Problem: Handling Client Disconnections and recv() Behavior

Unexpected client disconnections (e.g., client terminal closed, network drop, or client-side Ctrl+C) initially posed a challenge. The server's client handler thread needed to detect these disconnections reliably to prevent hanging or resource leaks. The recv() function returning 0 (graceful close) or -1 (error) required robust handling.

There were also instances where the server loop responsible for receiving commands seemed to stall after certain operations (like /join), preventing further commands from being processed for that client.

Solution: Robust recv() Loop and Resource Cleanup

The main command-receiving loop in handle_client was structured to check the return value of recv() explicitly. If bytes_received <= 0, the loop breaks, and the remove_client_from_system function is called.

remove_client_from_system was refined to ensure it correctly updated the client's status (active = 0), reset its socket_fd = -1 (crucial for making the slot reusable), decremented active_client_count, removed the user from any room they were in, and then properly closed the client's socket. This ensures resources are cleaned up and the server log reflects the disconnection as per PDF requirements (e.g., [DISCONNECT] user '...' lost connection...).

The issue of the command loop stalling after /join was traced to subtle control flow problems (e.g., a missing continue in a specific conditional block or a self-deadlock scenario with room mutexes as described above). These were addressed by carefully reviewing and correcting the logic within command processing blocks to ensure the loop always continued to the next recv() call unless a disconnect or /exit occurred.

3.3. Command Parsing and User Input Flow

Problem: Parsing Varied Command Formats and Username Re-prompting

Parsing user input strings like /command, /command argument1, or /command argument1 <multi-word message> into distinct components (command, arguments, message body) robustly was an initial challenge.

A specific issue arose when a user entered a duplicate or invalid username. The server would send an error, but the subsequent re-prompting flow between the server and client sometimes led to confusion, with the client not properly waiting for a new prompt or the server sending an extra prompt.

Solution: strsep() and Refined Prompting Logic

The strsep() function was used for tokenizing the input buffer received from the client. This helped in separating the command from its arguments and message body.

The username acquisition loop in server/client_handler.c was modified. After rejecting an invalid/duplicate username and sending an error message to the client, the server now explicitly sends a new "Enter your username:" prompt before calling recv() again in the loop.

The client-side username loop was also adjusted to expect and display this prompt after an error, ensuring a clear re-prompting sequence. (Self-correction from previous attempt: Initially, I tried making the server not send the re-prompt, expecting the client to handle it, but this led to issues. Reverting to the server sending the prompt after an error proved more robust for the user's observed behavior.)

3.4. File Transfer Simulation and Queue Management

Problem: Clarifying File Transfer Requirements and Implementing the Queue

The initial project PDF was somewhat light on the specifics of the file transfer protocol (e.g., how bytes are exchanged). The Q&A PDF (Soru 10) clarified that a message-based simulation was acceptable. Implementing the queue (MAX_UPLOAD_QUEUE = 5), ensuring thread-safe access, and processing requests in order with worker threads while handling PDF requirements (size/type checks, filename collision, logging queue wait times) required careful design.

Solution: Simulated Transfer with Synchronized Queue

A fixed-size array (file_upload_queue) was used for the upload queue.

file_queue_mutex protects direct access to this array for adding/removing items.

Two semaphores, items_in_queue and empty_slots_in_queue, were used to manage the producer (client handler threads enqueuing requests) and consumer (file worker threads dequeuing requests) logic.

enqueue_file_request now uses sem_timedwait on empty_slots_in_queue with a 10-second timeout, allowing the server to inform the client if the queue is full for an extended period, rather than blocking the client handler indefinitely.

Worker threads wait on items_in_queue. Upon dequeuing, they simulate the transfer (create a placeholder file, log success/failure, and notify clients). File size, type checks are performed before enqueuing. Filename collision is handled by generate_unique_filepath, and queue wait time is logged.

3.5. Modularization and Build Process

Problem: As the codebase grew, maintaining it in one or two large files became impractical. Splitting it into multiple .c and .h files introduced typical challenges related to include paths, extern declarations for global variables, header guards, and ensuring the Makefile correctly compiled and linked all modules.

Solution: Structured Multi-File Project with Makefile

The project was organized into server/, client/ directories, and a common_defs.h. Server-side code was further broken down into modules like client_handler, room_manager, file_transfer, and server_utils, each with a corresponding .h and .c file.

Global variables were defined in server/chatserver.c and declared as extern in server/globals.h, which was then included by other server modules needing access.

The Makefile was updated to compile each .c file into an object file (.o) and then link these object files to produce the final chatserver and chatclient executables. Dependencies were specified in the Makefile to ensure correct recompilation when files changed.

3.6. Debugging in a Multithreaded Environment

Problem: Identifying and resolving bugs, especially race conditions or deadlocks, in a multithreaded C application can be challenging with standard debuggers due to the non-deterministic nature of thread scheduling.

Solution: Strategic Logging and Careful Review

A custom server_log function was implemented early on to provide timestamped logs for various events, which was invaluable for tracing execution flow and understanding the state of shared resources.

For particularly tricky issues (like the command loop stalling), temporary, unbuffered fprintf(stderr, ...) statements were inserted at critical points to get immediate diagnostic output from specific threads.

Careful code walk-throughs and reasoning about potential inter-thread interactions and locking orders were essential. (A more advanced approach, if available in the development environment, would be to use thread sanitizers like TSan with GCC/Clang to automatically detect race conditions.)

These challenges, while sometimes frustrating, ultimately contributed to a deeper understanding of the concepts involved in building a robust, concurrent networked application.

4. Test Cases and Results

4.1. Concurrent User Load

Test: At least 30 clients connect simultaneously and interact with the server (join rooms, broadcast, whisper).

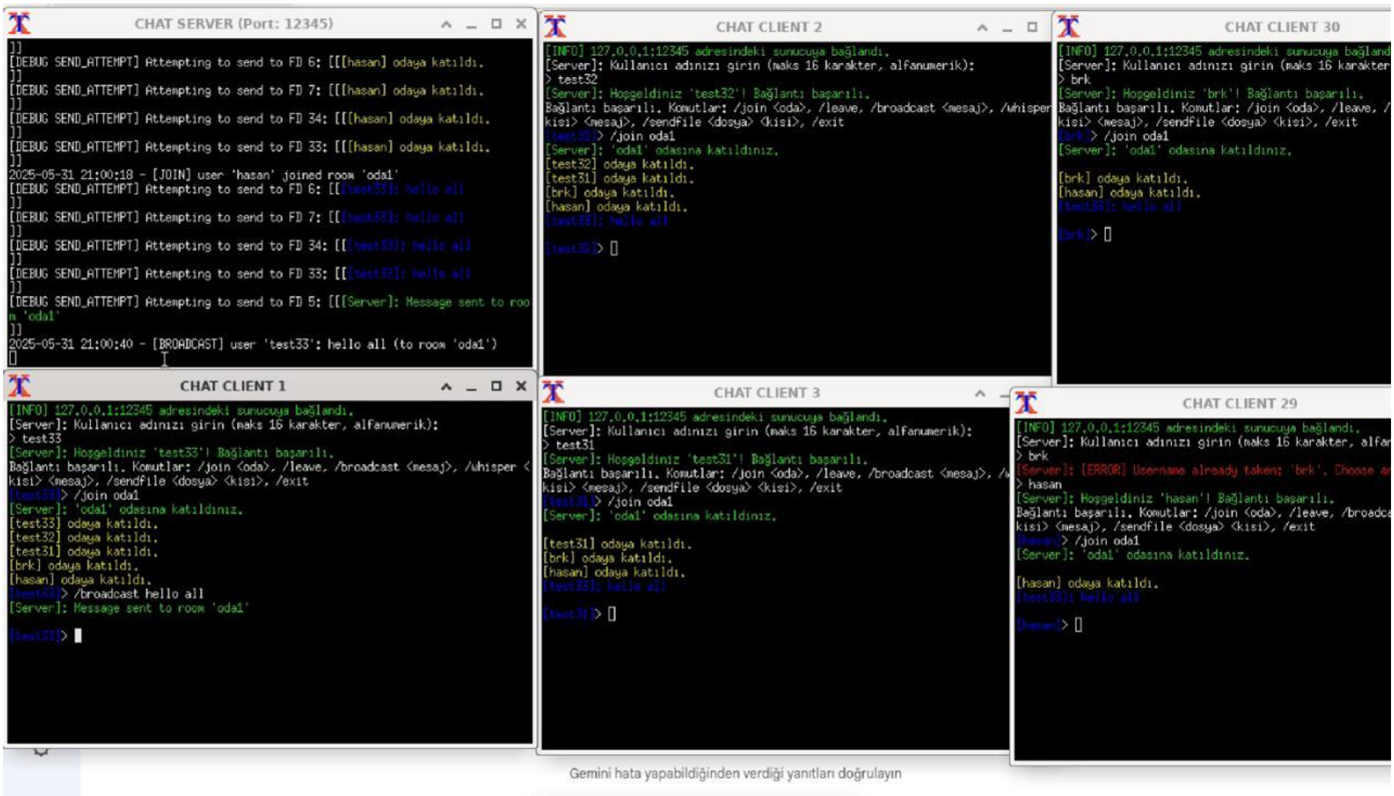
Expected: All users are handled correctly, no message loss, no crash.

Log example:

[CONNECT] user 'ali34' connected

[INFO] ali34 joined room 'project1'

[BROADCAST] ali34: Hello all



4.2. Duplicate Usernames

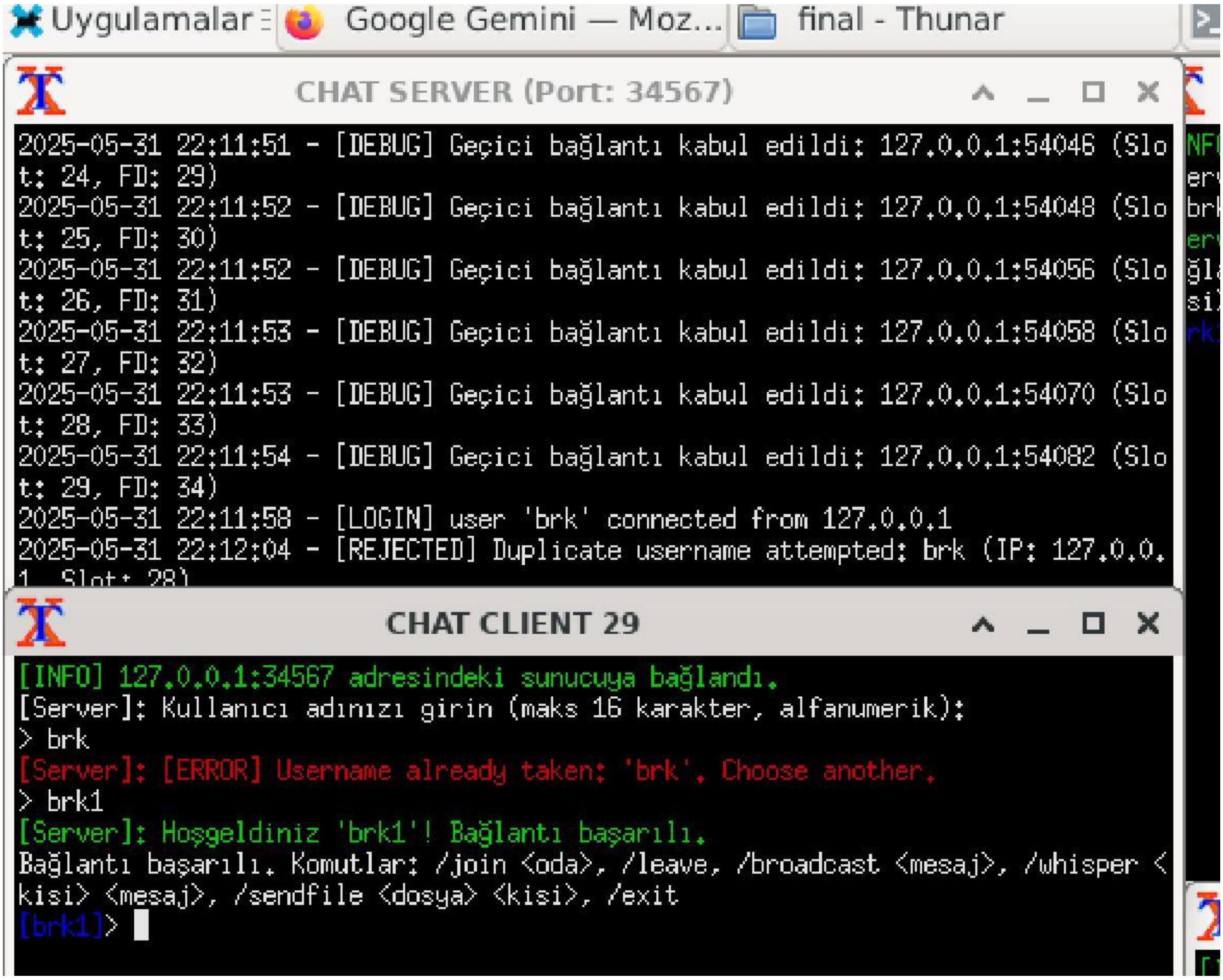
Test: Two clients try to connect using the same username.

Expected: Second client should receive a rejection message like:

[ERROR] Username already taken. Choose another.

Log example:

[REJECTED] Duplicate username attempted: ali34



4.3 FAILED

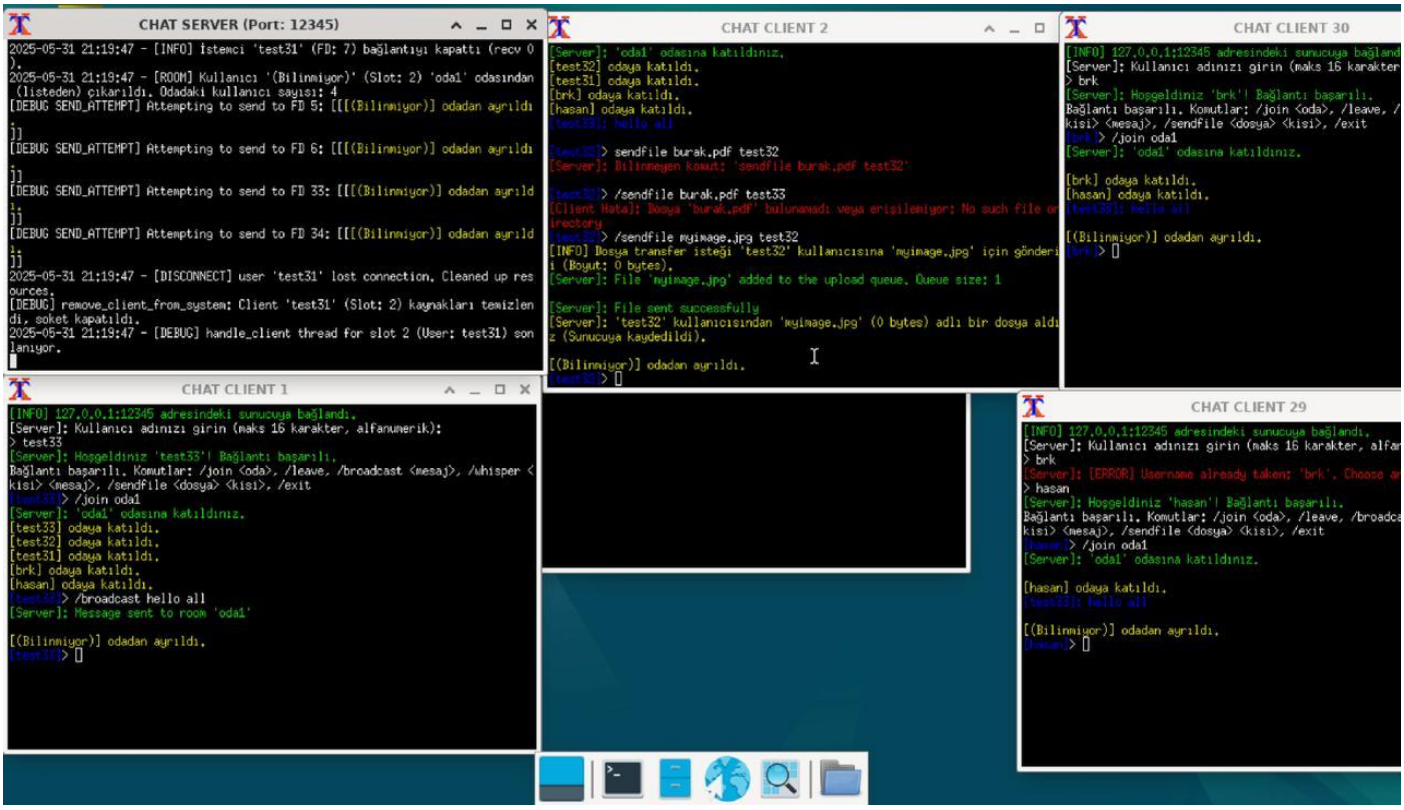
4.4. Unexpected Disconnection

Test: A client closes the terminal or disconnects without /exit.

Expected: Server must detect and remove the client gracefully, update room states, and log the disconnection.

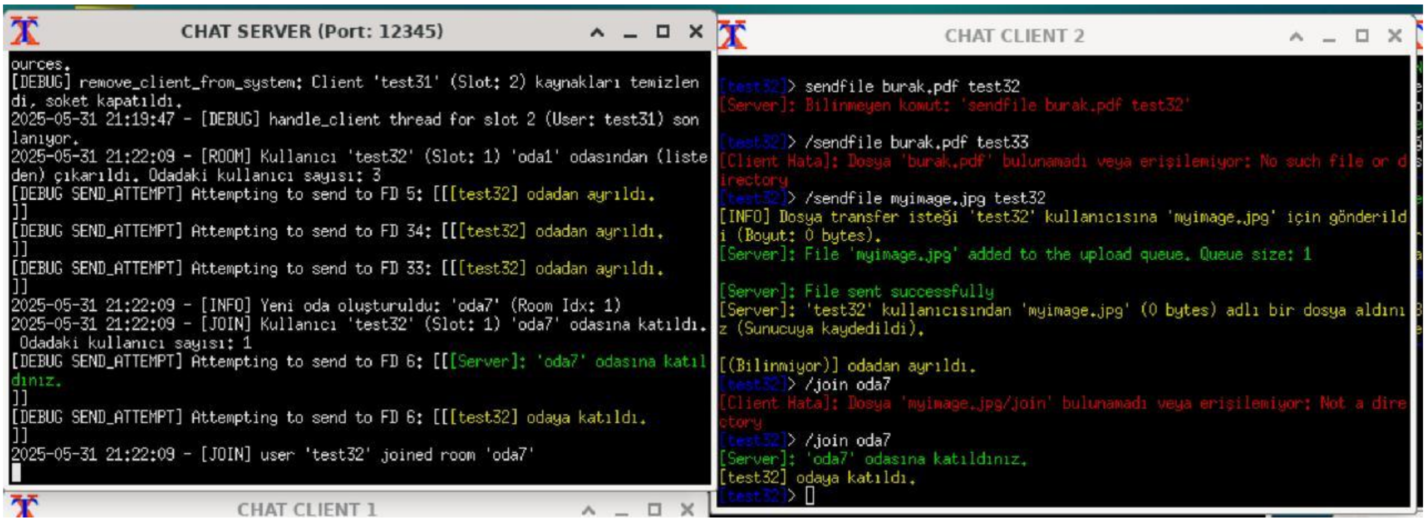
Log example:

[DISCONNECT] user 'mehmet1' lost connection. Cleaned up resources.



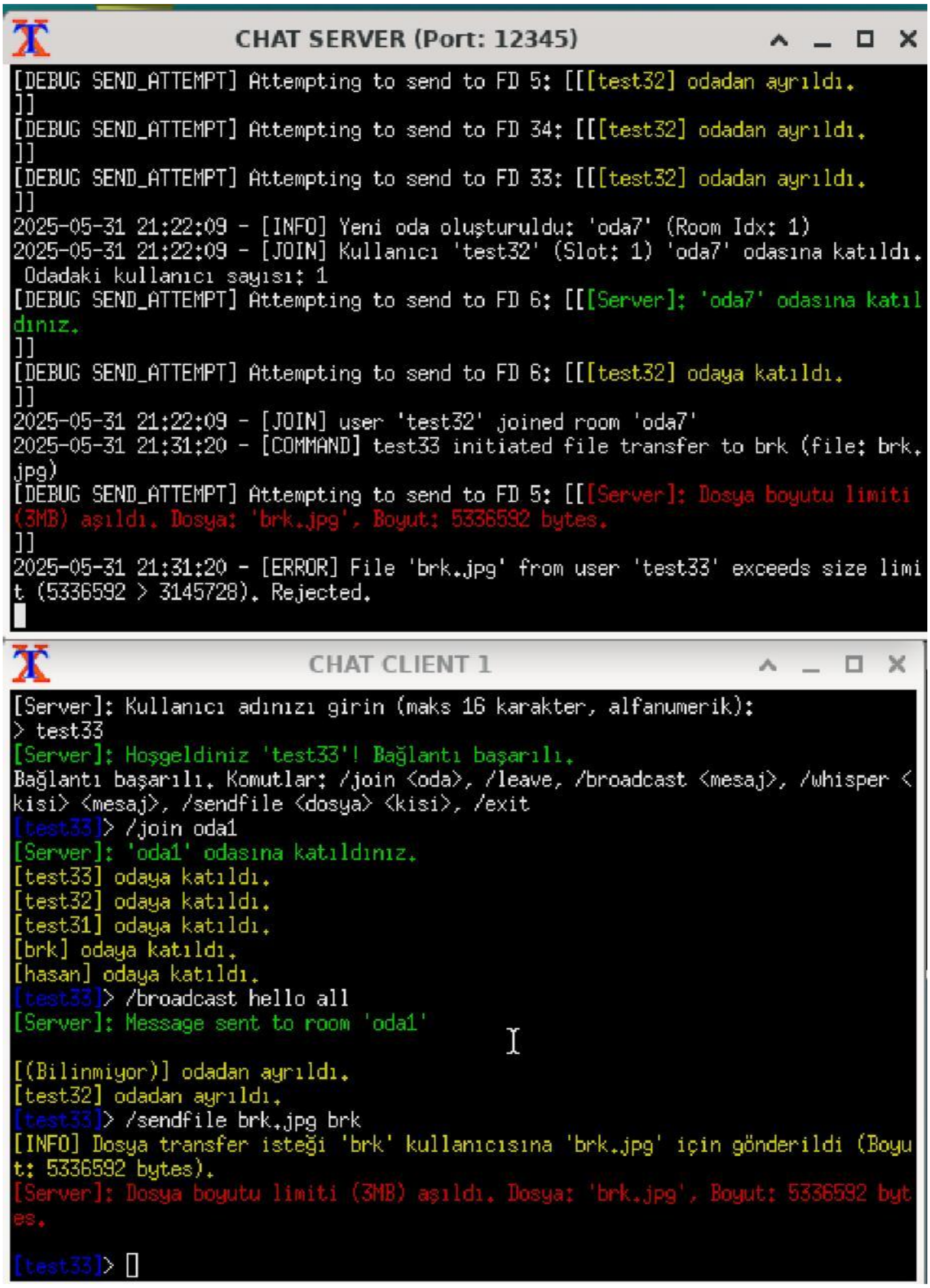
4.5. Room Switching

Test: A client joins a room, then switches to another room.
Expected: Server updates room states correctly. Messages are sent to the correct room.
Log example:
[ROOM] user 'irem56' left room 'groupA', joined 'groupB'



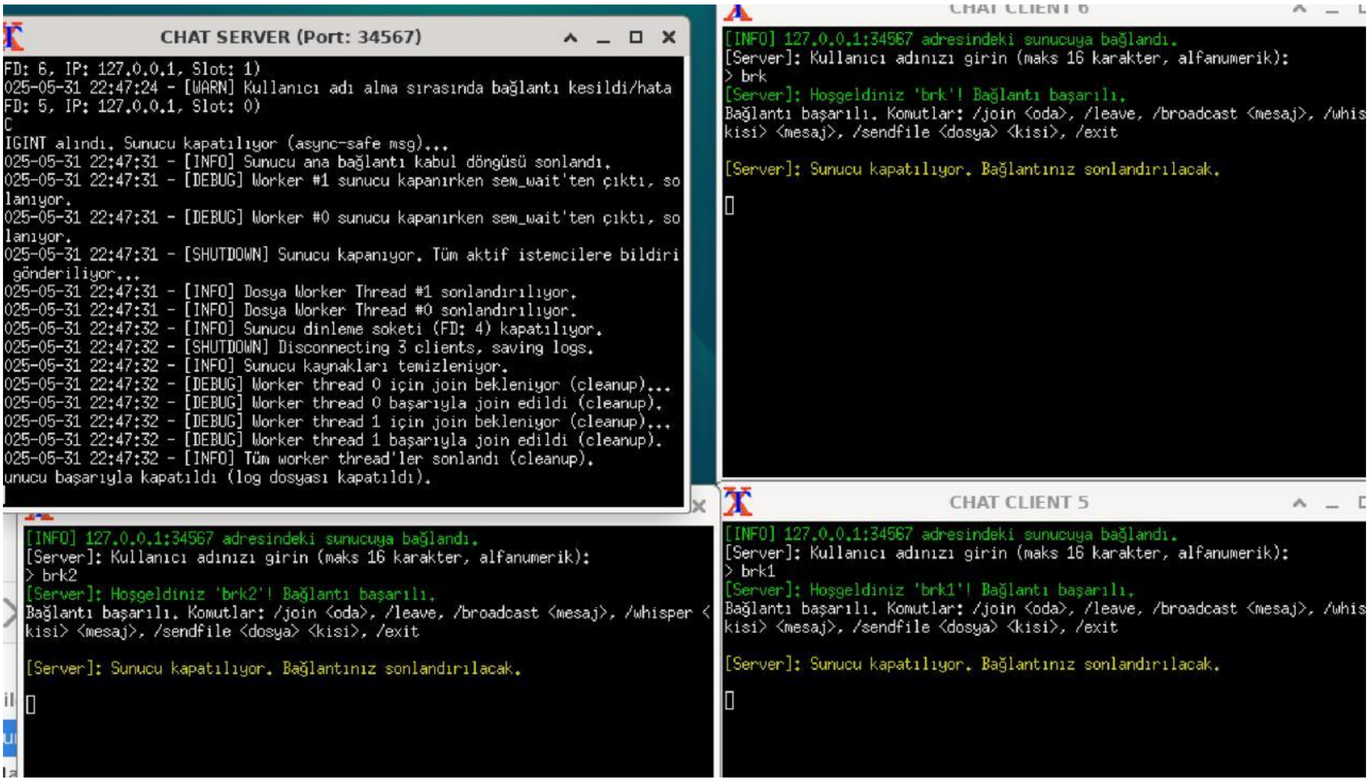
4.6. Oversized File Rejection

Test: A client attempts to upload a file exceeding 3MB.
Expected: File is rejected, user is notified.
Log example:
[ERROR] File 'huge_data.zip' from user 'melis22' exceeds size limit.



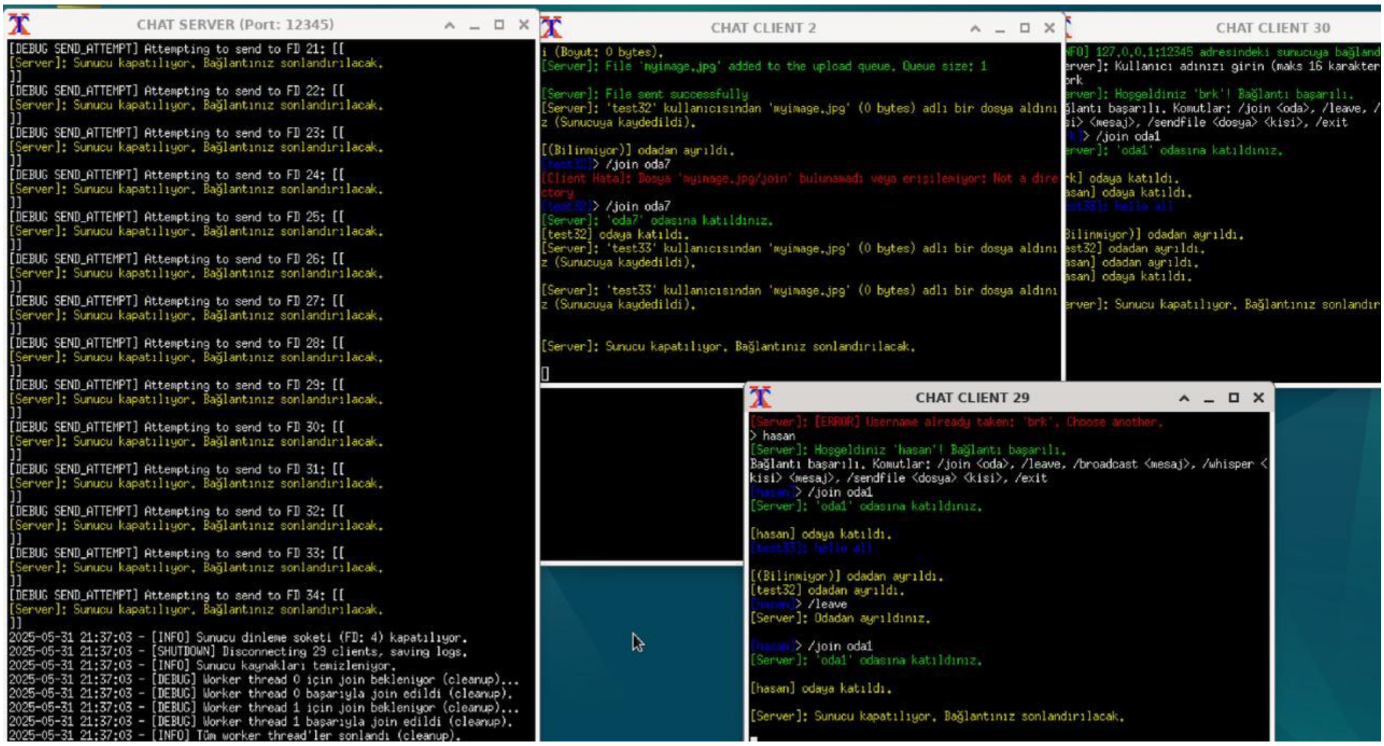
4.7. SIGINT Server Shutdown

Test: Press Ctrl+C on server terminal.
Expected:
• All clients are notified.
• Connections are closed gracefully.
• Logs are finalized before exit.
Log example:
[SHUTDOWN] SIGINT received. Disconnecting 12 clients, saving logs.



4.8. Rejoining Rooms

Test: A client leaves a room, then rejoins.
Expected: The client does not receive previous messages (unless you implement message history).
Clarify in report: Whether message history is persistent or ephemeral.
IT IS EPHEMERAL in my code



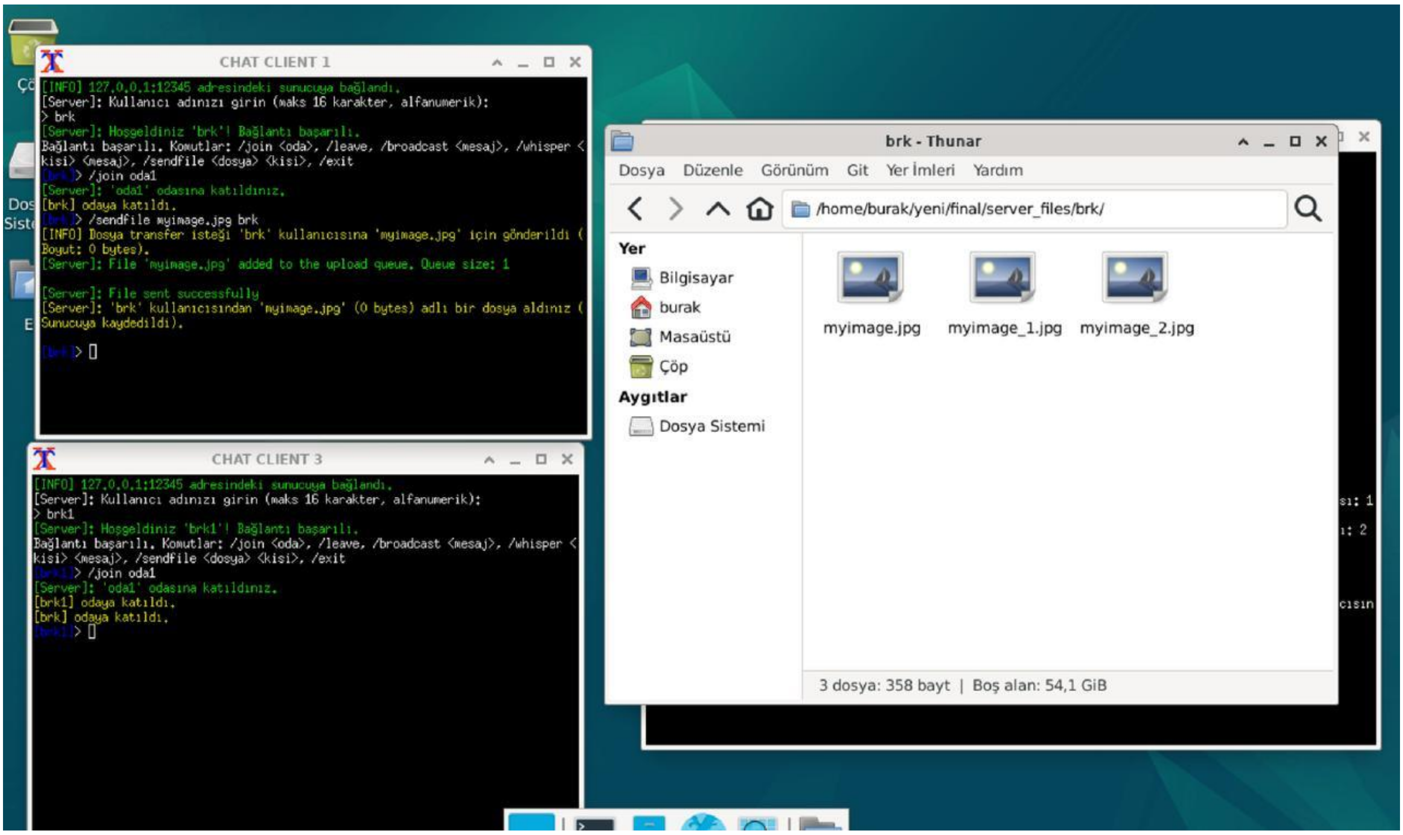
4.9. Same Filename Collision

Test: Two users send a file with the same name to the same recipient.

Expected: System handles filename conflict (e.g., renames file or alerts user).

Log example:

[FILE] Conflict: 'project.pdf' received twice → renamed 'project_1.pdf'




```
2025-05-31 22:20:13 - [FILE] Conflict: 'myimage.jpg' -> renamed 'myimage_1.jpg'
2025-05-31 22:20:14 - [SEND FILE] 'myimage.jpg' sent from brk17 to brk (success)
2025-05-31 22:20:15 - [COMMAND] brk16 initiated file transfer to brk4 (file: myimage.jpg)
2025-05-31 22:20:15 - [FILE-QUEUE] Upload 'myimage.jpg' from brk16 added to queue. Queue size: 1
2025-05-31 22:20:15 - [FILE] 'myimage.jpg' from user 'brk16' started upload after 0 seconds in queue.
2025-05-31 22:20:15 - [FILE-WORKER] [W#0] 'myimage.jpg' dosyasının 'brk16' kullanıcısından 'brk4' kullanıcısına transferi işleniyor
(Boyut: 0)...
```

4.10. File Queue Wait Duration

Test: When the file upload queue is full, how long does the next file wait?

Expected: Wait time is tracked, and client is informed (e.g., Waiting to upload...).

Log example:

[FILE] 'code.zip' from user 'berkay98' started upload after 14 seconds in queue.

```
2025-05-31 22:20:13 - [FILE] Conflict: 'myimage.jpg' -> renamed 'myimage_1.jpg'
2025-05-31 22:20:14 - [SEND FILE] 'myimage.jpg' sent from brk17 to brk (success)
2025-05-31 22:20:15 - [COMMAND] brk16 initiated file transfer to brk4 (file: myimage.jpg)
2025-05-31 22:20:15 - [FILE-QUEUE] Upload 'myimage.jpg' from brk16 added to queue. Queue size: 1
2025-05-31 22:20:15 - [FILE] 'myimage.jpg' from user 'brk16' started upload after 0 seconds in queue.
2025-05-31 22:20:15 - [FILE-WORKER] [W#0] 'myimage.jpg' dosyasının 'brk16' kullanıcısından 'brk4' kullanıcısına transferi işleniyor
(Boyut: 0)...
```

5. Conclusion and Potential Improvements

This project successfully implemented a multi-threaded TCP-based chat and file-sharing server with corresponding clients, adhering to the core requirements of the assignment. Key features include concurrent client handling, room-based and private messaging, and a simulated file transfer system with a synchronized upload queue. Synchronization primitives like mutexes and semaphores were employed to manage shared resources and ensure thread safety. Server-side logging provides a record of activities.