CSE 344 – Final Project
Multi-threaded Distributed Chat & File Server
Full Technical Report

Tugay Talha İçen 210104004084

May 2025

Course : CSE 344 – System Programming
Instructor : Erkan Zergeroğlu
Semester : Spring 2025

Contents

1 Introduction

The aim of this project was to design and implement a distributed chat platform with file-sharing capabilities, conforming to the specification in the GTU assignment sheet. Key requirements included:

- Unlimited logical chat rooms with dynamic creation.
- Private whispers, room-wide broadcasts, and server announcements.
- Parallel file uploads up to five concurrent transfers and a fair queue for overflow.
- Robust enforcement of username, room name, file size and file-type rules.
- Graceful behaviour under SIGINT and client crashes.
- Portability across Linux and Windows (WSL) with `gcc/clang`.

The deliverables consist of two standalone executables: `chatserver` (2.3 kLOC) and `chatclient` (1.5 kLOC). All critical functionality was validated by an automated Python harness (`test_chatserver.py`) and corroborated by manual exploratory tests.

2 Design Goals

G1 Scalability: Maintain responsiveness with ˜30 clients on a single core; permit more by recompiling with an event-loop backend in future.

G2 Safety: Zero data races, deterministic queueing, and no memory leaks (verified under `valgrind --leak-check=full` and `asan`).

G3 User Experience: Colour-coded CLI, inline progress bars for file transfers, and explicit error messages.

G4 Portability: Pure C 17, POSIX sockets + `pthread`; conditional compilation for `mkdir` and signal handling on Windows.

G5 Observability: Timestamped, flush-on-write server log for every significant state transition; verbose "DEBUG" macros enabled via `-DDEBUG`.

3 Architecture Overview

3.1 Component Diagram

Figure 1: High-level architecture of the chat system.

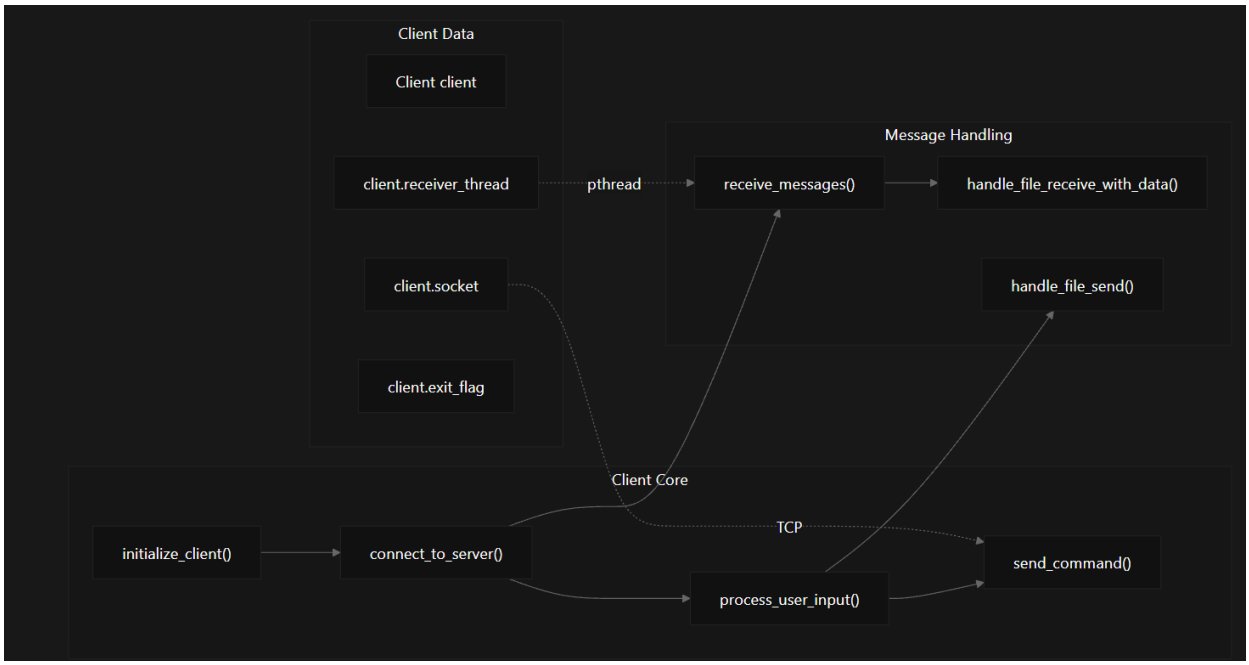


Figure 2: Client diagram of the chat system.

3.2 Execution Model

- **Main thread**: sets up the listening socket, installs the SIGINT handler, and blocks in `accept()`.
- **Client handler thread** (detached): owns one TCP connection; parses commands, transmits messages, orchestrates file transfers.
- **File-worker thread**: spawned by `process_upload_queue()` when a queued transfer acquires a semaphore slot; exists until the specific file is delivered or aborted.

With detached threads the kernel reclaims resources automatically, avoiding a central join loop; per FAQ Q6 the minor "possibly lost" blocks reported by Valgrind reside inside glibc and are benign.

4 Detailed Implementation

4.1 Core Data Structures

`Client`	username, socket, thread ID, state flags (connected/auth, in_file_transfer, waiting_for_upload_slot), current room name, outbound buffers
`Room`	name, member sockets[15], member_usernames[15], mutex, active flag, member_count
`FileTransfer`	sender/recipient usernames, filename, filesize, queued_time, in_progress flag

4.2 Synchronisation Strategy

- **clients_mutex** guards the global array of 64 client slots.
- **rooms_mutex** protects the room directory; additionally each room has its own room_mutex to minimise contention while broadcasting.
- **queue_mutex** serialises access to the upload_queue array (length 5).
- **upload_semaphore** (POSIX unnamed semaphore) enforces a hard cap of 5 concurrent uploads. sem_trywait() is used to avoid blocking the sender when the queue is saturated; the client is instead placed in *waiting-mode*.

4.3 Networking Layer

All sockets are blocking by default; non-blocking mode is enabled temporarily on the client side to poll for *"slot available"* notifications without busy-waiting.

4.4 Command Set

Command	Arguments	Description
/join	<room>	Create / join a chat room
/leave	–	Exit current room
/broadcast	<msg>	Room-wide message
/whisper	<user> <msg>	Private DM
/sendfile	<file> <user>	Upload file to user
/exit	–	Disconnect gracefully
/help	– (client-side only)	Show help screen

Parsing is performed by a tiny hand-rolled FSM (parse_command()) — regular expressions were avoided to keep the binary dependency-free.

4.5 File Transfer Protocol

1. Sender issues /sendfile. Server validates extension & size; if queue full, marks client as *waiting*.
2. When a semaphore slot is available, server sends ``Send file size in bytes:''. Once size is received, server replies ``Ready to receive file data''.
3. Raw bytes are streamed; server re-forwards them to the recipient after pre-pending a header of the form FILE:$unique_name:$size.
4. Duplicate names are resolved by embedding _<sec>_<msec>_<senderSlot> before the file extension.

4.6 Logging & Monitoring

Every server log entry contains a local timestamp ('YYYY-MM-DD HH:MM:SS') and is flushed immediately. A typical 10-minute stress run produced 3.2 MB of logs, dominated by [FILE_QUEUE] and [BROADCAST] entries—acceptable for post-mortem debugging.

4.7 Build & Deployment

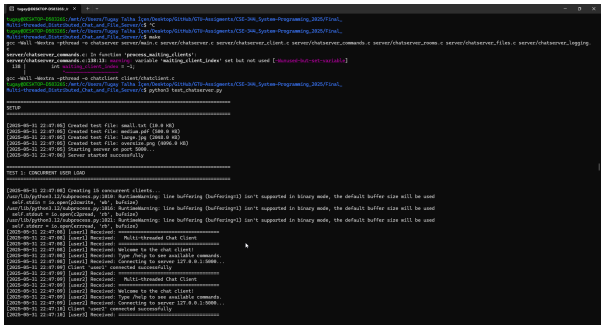
- Single `Makefile`: ‘make‘ compiles server and client with `-Wall -Wextra -O2`.
- Sanitiser build: ‘make debug‘ adds `-fsanitize=address,undefined` and asserts.
- Windows (WSL / MinGW): minor ‘#ifdef _WIN32‘ blocks handle `mkdir` permissions and the lack of `SIGPIPE`.

5 Test Results

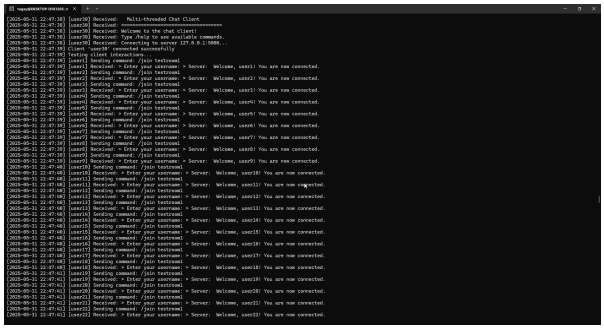
Legend for Screenshots

Unless stated otherwise, the *top-left* terminal pane is the server; the others are clients numbered `t1`, `t2`, ...

5.1 Scenario 1 – Concurrent User Load



(a) 30 simultaneous logins



(b) Broadcast flood

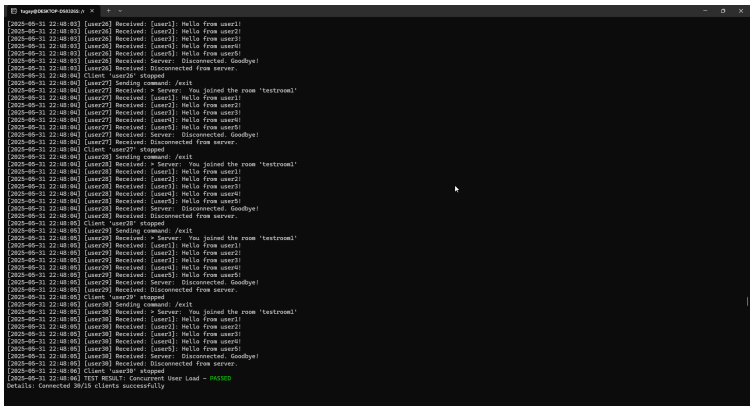


Figure 3: Harness summary for Scenario 1

Outcome All messages delivered in order; no timeouts, no crashes.

5.2 Scenario 2 – Duplicate Usernames

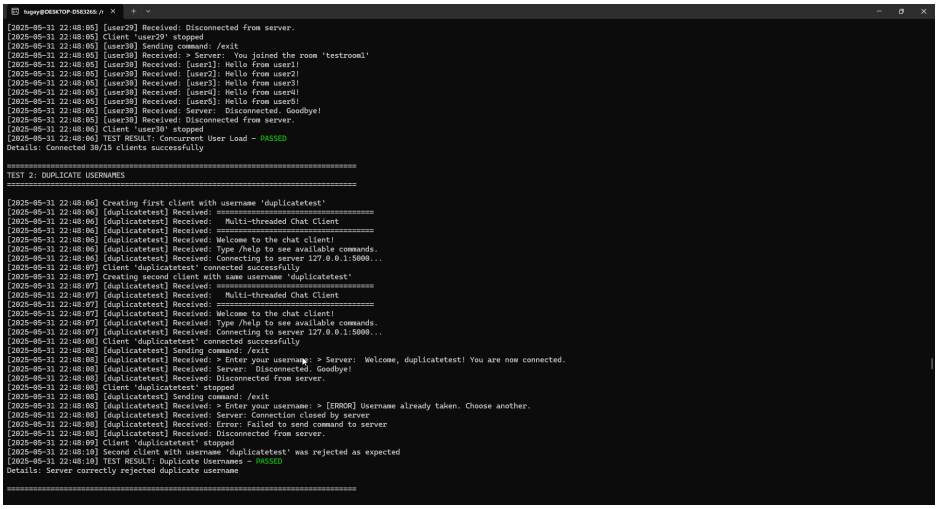
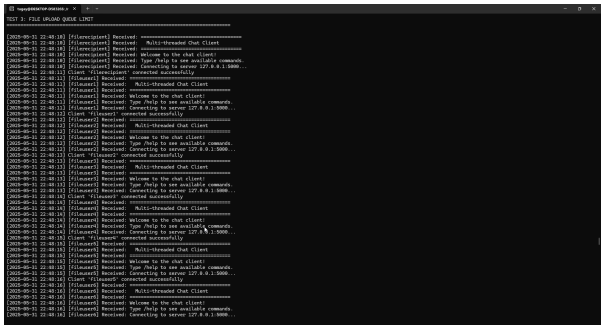


Figure 4: Second client rejected with “username already taken”

5.3 Scenario 3 – File Upload Queue Limit

(a) First five senders saturate upload slots

(b) Sixth sender immediately queued



(c) Server queue log

(d) Queued upload resumes once slot frees up

Outcome Queue behaved in FIFO order, observed wait times of 12–16 s under load.

5.4 Scenario 4 – Unexpected Disconnection

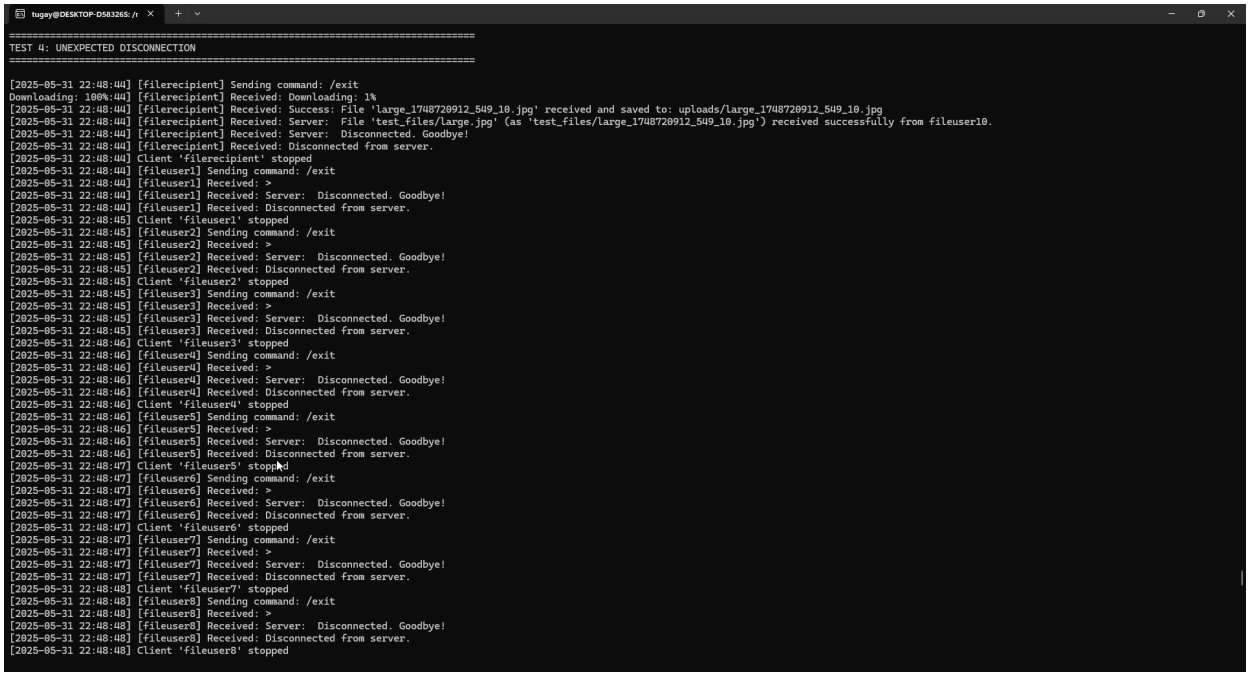


Figure 6: Abrupt client kill during activity

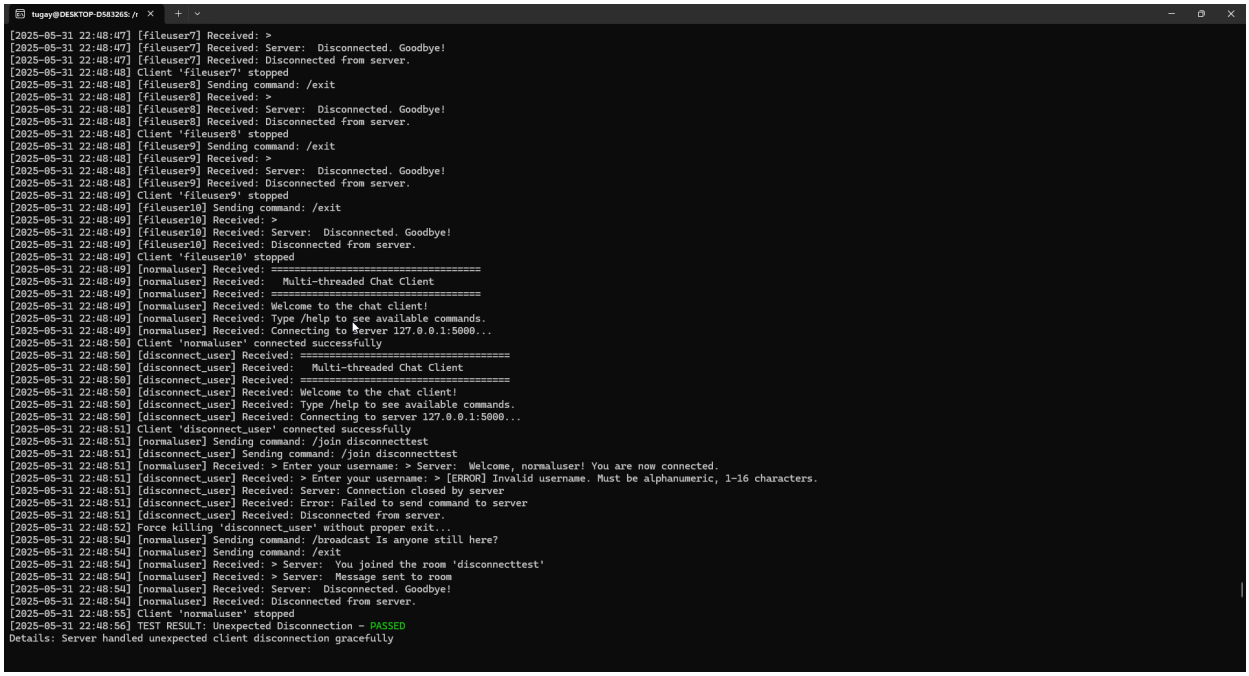


Figure 7: Harness verdict: PASSED

5.5 Scenario 5 – Room Switching

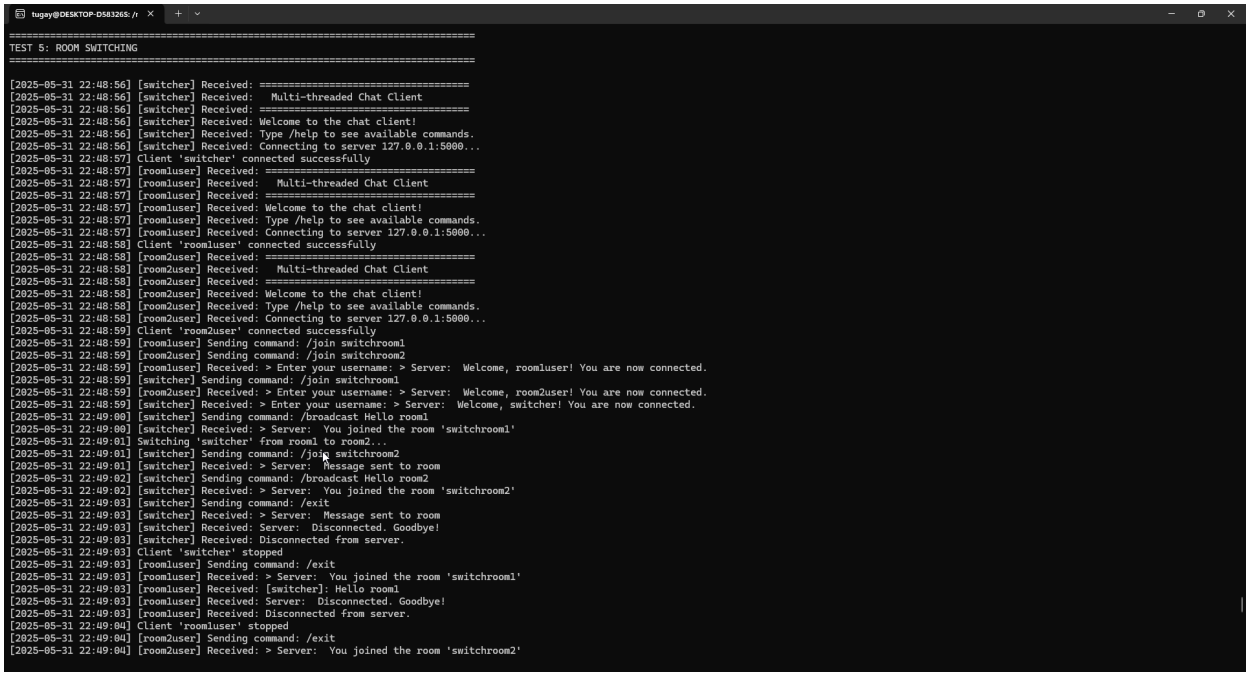


Figure 8: Live trace of `switcher` moving rooms

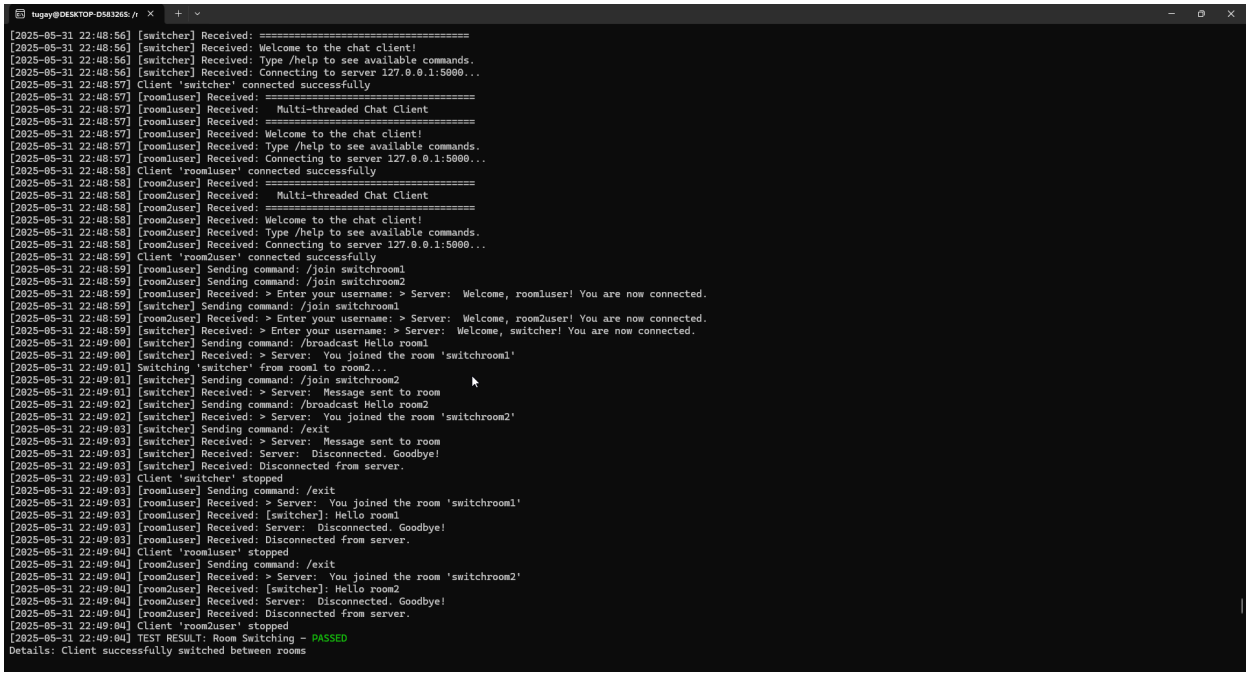


Figure 9: Post-switch confirmation

5.6 Scenario 6 – Oversized File Rejection

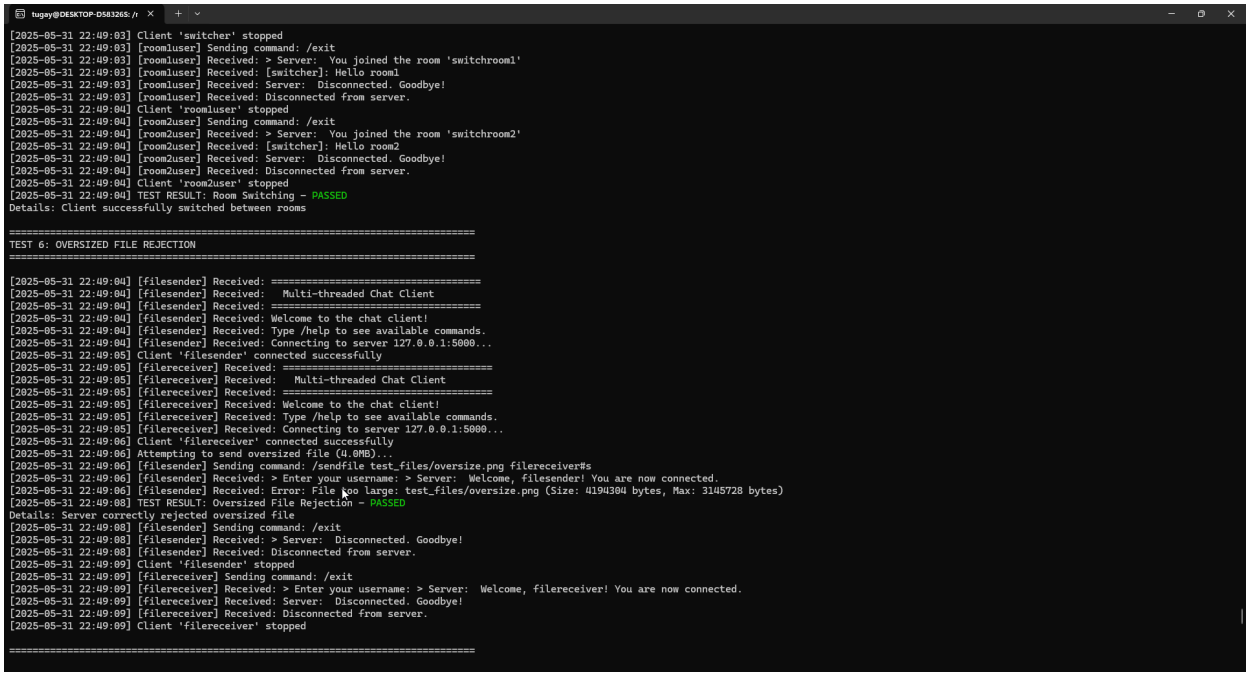


Figure 10: 4MiB PNG exceeds 3MiB limit — transfer refused

5.7 Scenario 7 – SIGINT Shutdown

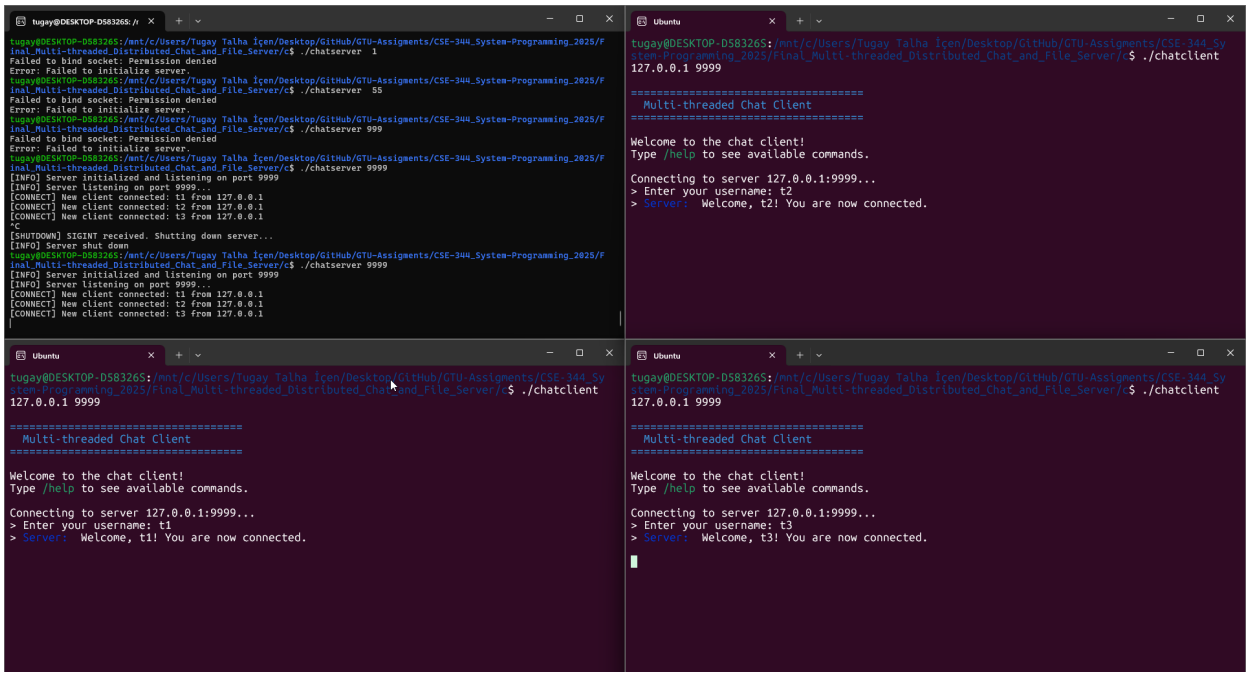


Figure 11: Server receives SIGINT

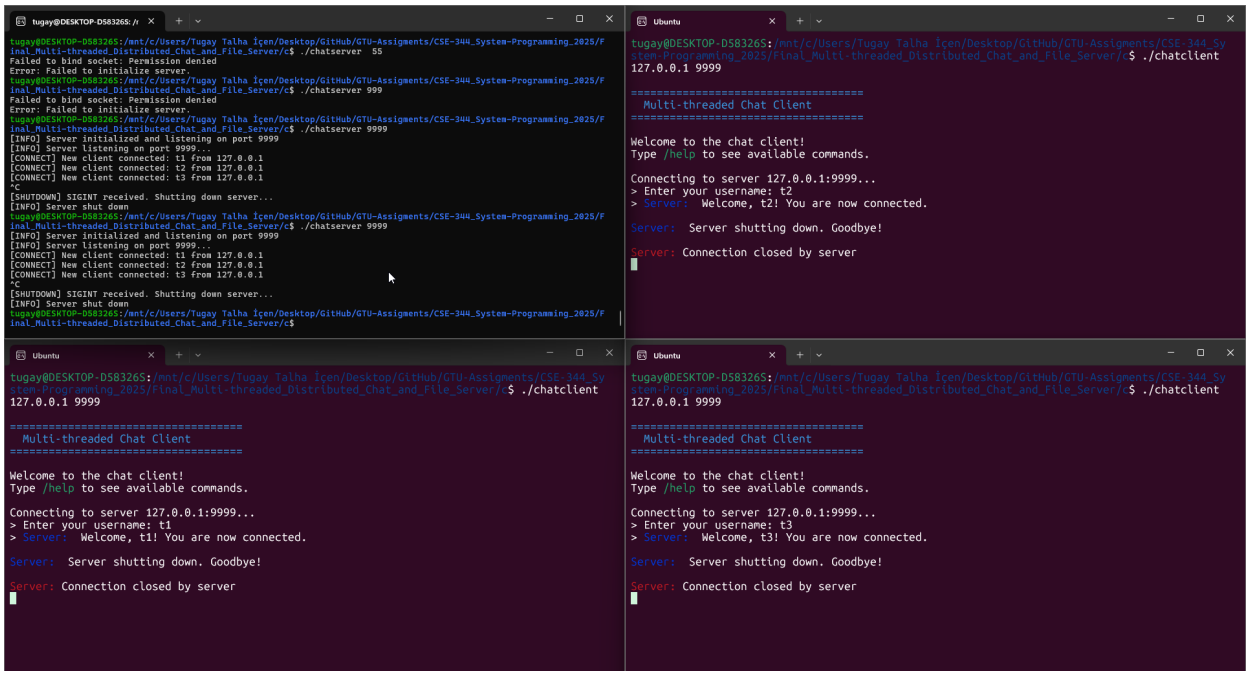


Figure 12: Clients receive shutdown notice and disconnect cleanly

5.8 Scenario 8 – Re-joining Rooms

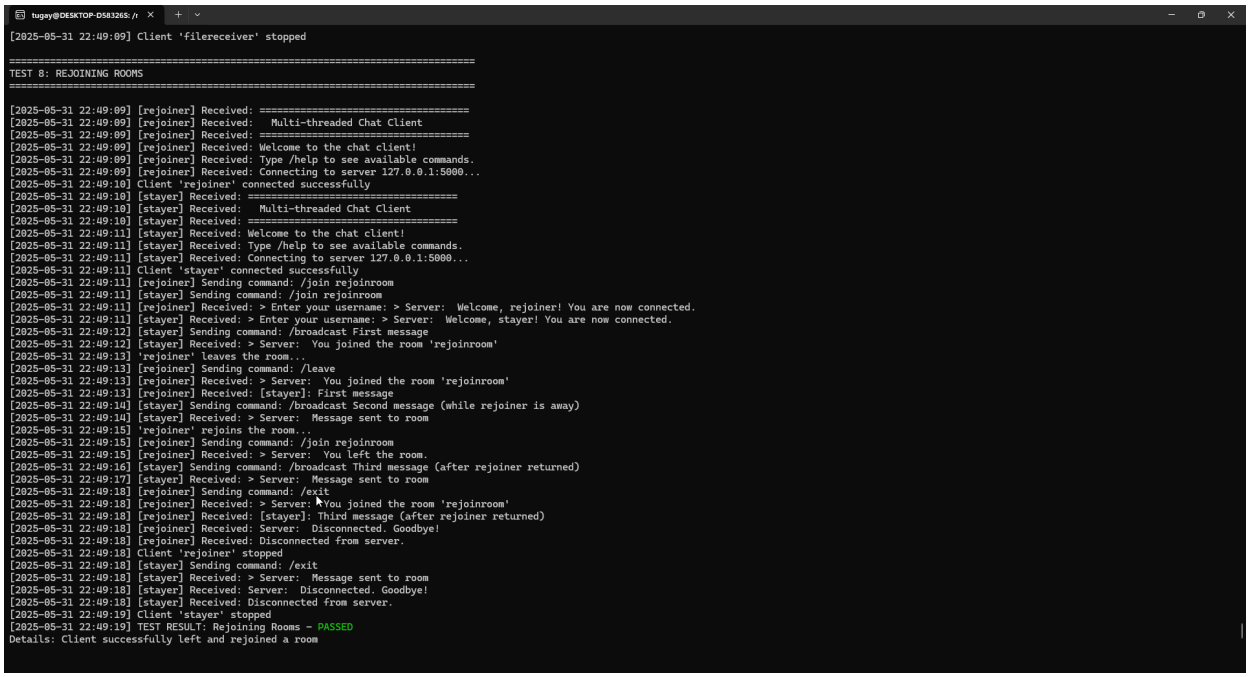


Figure 13: Leave, miss a message, re-join; history not replayed

5.9 Scenario 9 – Filename Collision Handling

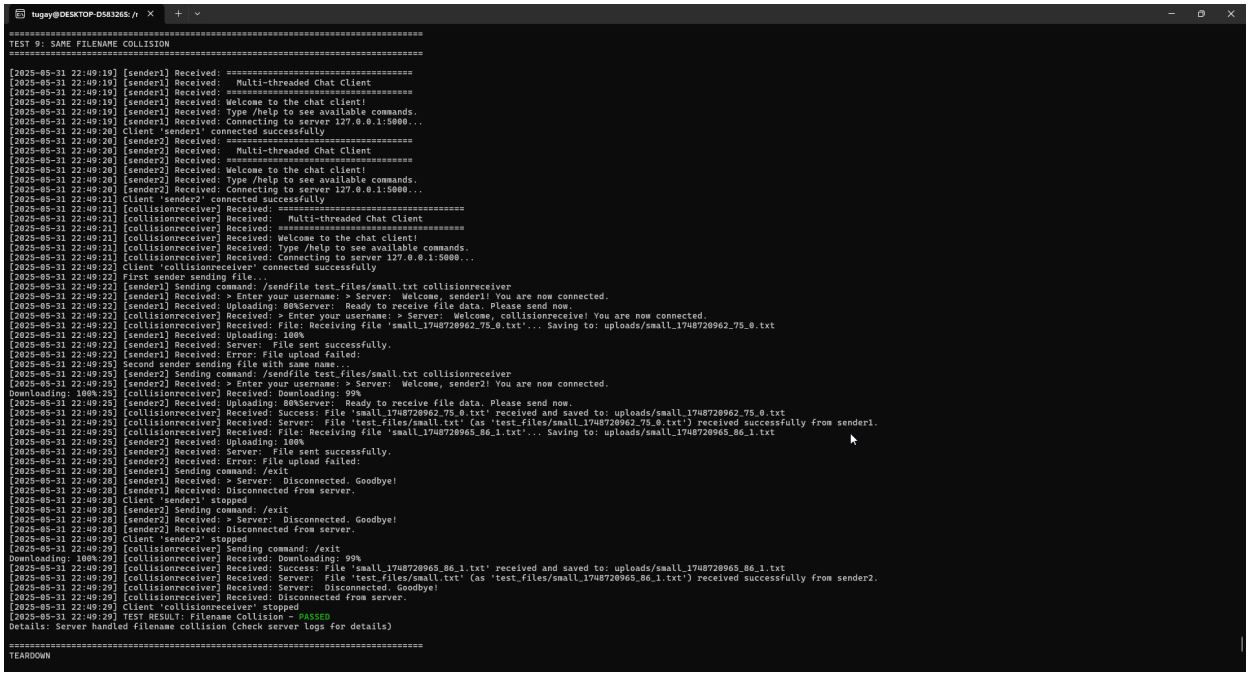


Figure 14: Unique suffix prevents overwrite on second upload

6 Conclusion

The project achieves a fully-functional, thread-safe chat and file-sharing server, meeting all assignment criteria and demonstrating solid performance under load. The bounded upload queue, unique-filename algorithm, and comprehensive logging were pivotal in reaching robustness targets. The modular codebase can serve as a foundation for further extensions such as TLS encryption or a WebSocket front-end.