**GEBZE TECHNICAL UNIVERSITY**

**COMPUTER ENGINEERING**

**DEPARTMENT**

**CSE344 Systems Programming**

**Spring 2025**

**Final Project Report**

**Multi-threaded Distributed Chat and File Server**

**Osmancan Bozali**

**220104004011**

## **Introduction and Problem Definition**

### 1.1 Project Overview

This project implements a comprehensive multi-threaded, TCP-based chat and file-sharing system using the client-server model. The system addresses the challenge of handling concurrent client connections while maintaining data consistency and providing robust file transfer capabilities with resource management.

### 1.2 Problem Definition

Modern distributed communication systems must handle multiple concurrent users while ensuring:

* **Concurrent Access Management:** Multiple clients connecting simultaneously without conflicts
* **Resource Limitation Simulation:** File transfer queue management to simulate real-world bandwidth/storage constraints
* **Data Integrity:** Thread-safe operations on shared resources (user lists, rooms, file queues)
* **Graceful Error Handling:** Robust handling of unexpected disconnections and system shutdowns
* **Real-time Communication:** Instant message delivery and room-based broadcasting

### 1.3 System Requirements

The implemented solution provides:

* Support for 30+ concurrent client connections
* Username uniqueness validation (max 16 characters, alphanumeric)
* Room-based group messaging (max 15 users per room, 20 total rooms)
* Private messaging (whisper) functionality
* Simulated file transfer system with queue management (max 5 concurrent uploads)
* Comprehensive logging with timestamps
* Signal-based graceful shutdown (SIGINT handling)

## **Design Details**

### System Architecture

### A diagram of a server AI-generated content may be incorrect.The system follows a multi-threaded client-server architecture with the following key components:

*Figure: System Architecture*

### 2.2 Thread Model

#### 2.2.1 Server Threading Architecture

**Main Thread:**

* Listens for incoming connections on specified port
* Accepts new client connections
* Creates dedicated client handler threads
* Manages server shutdown sequence

**Client Handler Threads:**

* One thread per connected client (up to 30 concurrent)
* Handles username validation and authentication
* Processes client commands (/join, /leave, /broadcast, /whisper, /sendfile)
* Manages client disconnection and cleanup

**File Transfer Worker Threads:**

* Fixed pool of 5 worker threads for file processing
* Processes file transfer requests from shared queue
* Simulates file upload/download operations
* Updates transfer statistics and notifies clients

#### 2.2.2 Client Threading Architecture

**Main Thread:**

* Handles user input and command parsing
* Sends commands to server
* Manages client-side validation (file existence, format)

**Receiver Thread:**

* Continuously listens for server responses
* Handles different message types (broadcasts, whispers, notifications)
* Provides color-coded output formatting

### 2.3 Inter-Process Communication (IPC) and Synchronization

#### 2.3.1 Synchronization Mechanisms

**Client Management:**

pthread\_mutex\_t clients\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

client\_t clients[MAX\_CLIENTS\_GLOBAL];

* Protects shared client array from concurrent access
* Ensures thread-safe client addition/removal operations

**Room Management:**

pthread\_mutex\_t rooms\_list\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

room\_t rooms[MAX\_ROOMS];

*// Each room has individual mutex*

pthread\_mutex\_t room\_mutex;

* Global mutex for room list operations
* Individual room mutexes for concurrent room activities

**File Transfer Queue:**

typedef struct {

file\_transfer\_t transfers[MAX\_UPLOAD\_QUEUE \* 3];

pthread\_mutex\_t queue\_mutex;

sem\_t queue\_semaphore;

pthread\_cond\_t queue\_cond;

pthread\_cond\_t queue\_not\_full;

} file\_queue\_t;

#### 2.3.2 Queue Management System

The file transfer system implements a producer-consumer pattern:

**Queue Operations:**

* **Semaphore (queue\_semaphore):** Controls concurrent file processing (max 5)
* **Condition Variables:** Coordinate queue state changes
  + queue\_cond: Signals when new transfers are available
  + queue\_not\_full: Signals when queue has space for new transfers
* **Mutex (queue\_mutex):** Protects queue data structure modifications

**Queue Workflow:**

1. Client sends /sendfile command
2. Server validates file metadata and user permissions
3. Transfer request enqueued with position notification
4. Worker thread dequeues and processes transfer
5. Completion notification sent to sender and receiver

### 2.4 Network Protocol Design

#### 2.4.1 Message Format

**Server Response Format:**

* OK:<message> - Successful operation
* ERROR:<error\_message> - Error condition
* FILE\_NOTIFY:<notification> - File transfer notification

**Command Format:**

* /join <room\_name> - Join or create room
* /leave - Leave current room
* /broadcast <message> - Send message to room
* /whisper <username> <message> - Private message
* /sendfile <filename> <username> - File transfer
* /exit - Disconnect

#### 2.4.2 File Transfer Protocol

**Enhanced sendfile with metadata:**

*// Client sends: /sendfile filename username filesize*

void parse\_sendfile\_args(const char\* args, char\* filename,

char\* username, size\_t\* file\_size);

**Server-side validation:**

* File type validation (.txt, .pdf, .jpg, .png)
* File size limit enforcement (3MB maximum)
* User existence verification
* Queue capacity checking

### 2.5 Data Structures

#### 2.5.1 Client Structure

typedef struct {

int sockfd; *// Socket descriptor*

char username[MAX\_USERNAME\_LEN + 1]; *// Unique username*

char ip\_addr[INET\_ADDRSTRLEN]; *// Client IP address*

pthread\_t thread\_id; *// Associated thread*

int active; *// Connection status*

char current\_room\_name[MAX\_ROOM\_NAME\_LEN + 1]; *// Current room*

} client\_t;

#### 2.5.2 Room Structure

typedef struct {

char name[MAX\_ROOM\_NAME\_LEN + 1]; *// Room name*

int member\_client\_indices[MAX\_ROOM\_USERS]; *// Member indices*

int num\_users; *// Current user count*

pthread\_mutex\_t room\_mutex; *// Room-specific mutex*

int active; *// Room status*

} room\_t;

#### 2.5.3 File Transfer Structure

typedef struct {

char filename[MAX\_FILENAME\_LEN]; *// Original filename*

char sender\_username[MAX\_USERNAME\_LEN + 1]; *// Sender identity*

char receiver\_username[MAX\_USERNAME\_LEN + 1]; *// Target user*

size\_t file\_size; *// File size in bytes*

time\_t request\_time, start\_time, completion\_time; *// Timing data*

transfer\_status\_t status; *// Current status*

char error\_message[256]; *// Error details*

} file\_transfer\_t;

## **Issues Faced And Solutions**

### 3.1 Concurrency and Race Conditions

**Issue:** Multiple threads accessing shared client and room data structures simultaneously caused data corruption and crashes.

**Solution:**

* Implemented hierarchical locking strategy
* Used fine-grained mutexes for individual rooms vs. coarse-grained for global operations
* Applied lock ordering to prevent deadlocks

### 3.2 Signal Handling and Graceful Shutdown

**Issue:** SIGINT (Ctrl+C) caused abrupt server termination without proper client notification or resource cleanup.

**Solution:**

* Implemented signal handler with atomic flag
* Added graceful disconnection sequence
* Ensured log file closure and resource cleanup

### 3.3 File Transfer Queue Management

**Issue:** File transfer requests could overwhelm the server, and clients had no feedback about queue status.

**Solution:**

* Implemented bounded queue with semaphore-based flow control
* Added queue position and estimated wait time notifications
* Provided real-time transfer status updates

### 3.4 Username Uniqueness and Validation

**Issue:** Race condition in username validation allowed duplicate usernames under high concurrency.

**Solution:**

* Centralized username validation under mutex protection
* Implemented atomic check-and-reserve operation
* Added retry mechanism for clients

### 3.5 Memory Management and Resource Leaks

**Issue:** Dynamic allocations and thread resources were not properly cleaned up on unexpected client disconnections.

**Solution:**

* Implemented comprehensive cleanup functions
* Added connection state tracking
* Used pthread\_detach for automatic thread cleanup

### 3.6 Non-blocking Input Handling

**Issue:** Client blocked on stdin input while server messages needed to be received simultaneously.

**Solution:**

* Implemented non-blocking input with select() system call
* Added timeout-based polling for server disconnection detection
* Maintained responsive user interface

## **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.

Result: see *example\_log.txt* in source code.

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.

A screenshot of a computer

AI-generated content may be incorrect.

4.3. File Upload Queue Limit

Test: 10 users attempt to send files at the same time. Upload queue only allows 5

concurrent uploads.

Expected: First 5 go through, the rest are queued and processed when slots become

available.

A screenshot of a computer program

AI-generated content may be incorrect.

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.

A screenshot of a computer

AI-generated content may be incorrect.

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.

A screenshot of a computer screen

AI-generated content may be incorrect.

4.6. Oversized File Rejection

Test: A client attempts to upload a file exceeding 3MB.

Expected: File is rejected, user is notified.

A screenshot of a computer

AI-generated content may be incorrect.

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.

A screenshot of a computer screen

AI-generated content may be incorrect.

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).

A screenshot of a computer

AI-generated content may be incorrect.

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).

A screenshot of a computer screen

AI-generated content may be incorrect.

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...).

A screenshot of a computer screen

AI-generated content may be incorrect.

4.11. Valgrind Test for Memory Leak Check

A screenshot of a computer screen

AI-generated content may be incorrect.

## **Conclusion and Potential Improvements**

This project successfully demonstrates comprehensive mastery of systems programming concepts, achieving all primary objectives through proper implementation of multi-threading, network programming, and IPC mechanisms. The system effectively handles up to 30 concurrent clients with robust TCP socket communication, implements sophisticated queue-based file transfer management with resource limitation simulation, and provides graceful handling of unexpected disconnections and signal-based shutdown procedures. Key technical contributions include innovative queue management with position-based wait time estimation, hierarchical cleanup procedures for connection state recovery, and a modular architecture allowing easy feature extensions. The implementation showcases advanced synchronization techniques using mutexes, semaphores, and condition variables to ensure thread-safe operations across shared resources including client arrays, room structures, and file transfer queues. While the current system provides a solid foundation comparable to commercial messaging platforms, several improvements could enhance its capabilities:, adding persistent message history and implementing real file transfer mechanism. The development process thought me important lessons about concurrency complexity, the critical importance of robust error handling, proper resource management for preventing memory leaks, and the value of comprehensive testing under various conditions to reveal edge cases. Overall, this implementation exceeds basic requirements by providing insights into real-world distributed system challenges while maintaining educational value through clear, well-documented code that combines theoretical knowledge application with practical problem-solving, resulting in a robust and scalable system ready for production deployment.