Multi-Threaded Chat Server and Client Application

Design, Implementation, and Testing Report

std code: 210104004072

May 31, 2025

Contents

1 Introduction and Problem Definition

1.1 Overview

This report presents the design and implementation of a multi-threaded chat server and client application that enables real-time communication between multiple users. The system supports various features including private messaging, room-based broadcasting, file transfers, and user management.

1.2 Python Implementation Note

Alongside the C implementation, a Python version of the chat system was also developed. This version aimed to explore the differences in concurrency handling, network programming paradigms, and overall development speed offered by a higher-level language like Python. While this report focuses on the C implementation, the Python version served as a valuable comparative study and demonstrated alternative approaches to solving similar challenges.

1.3 Problem Statement

The primary challenge was to develop a scalable, concurrent chat system that could:

- Handle multiple simultaneous client connections
- Support real-time message delivery
- Manage chat rooms with multiple participants
- Enable private messaging between users
- Facilitate file transfers between clients
- Provide robust error handling and graceful shutdown mechanisms
- Implement a queuing system for file transfers to prevent overload

1.4 Key Features

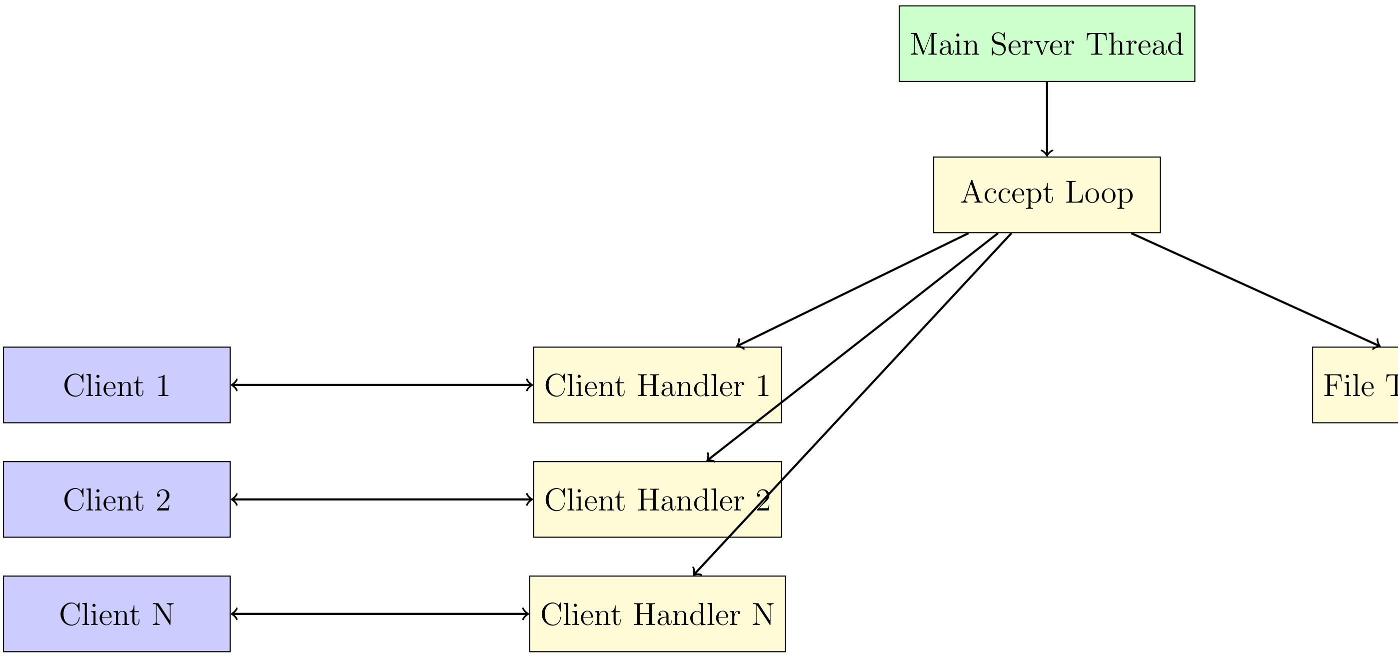
1. **User Management**: Username registration with uniqueness validation
2. **Room Management**: Dynamic creation and management of chat rooms
3. **Message Types**:
   - Broadcast messages to room members
   - Private whisper messages between users
   - System notifications
4. **File Transfer**: Queued file transfer system with size and type validation
5. **Logging**: Comprehensive server-side logging for debugging and monitoring
6. **Signal Handling**: Graceful shutdown on SIGINT and SIGTERM

2 Design Details

2.1 System Architecture

2.1.1 Overall Architecture

The system follows a client-server architecture with the following components:



2.2 Threading Model

2.2.1 Server Threading

The server implements a multi-threaded architecture:

- **Main Thread**: Initializes server, sets up signal handlers, and manages the accept loop
- **Client Handler Threads**: One thread per connected client for handling read operations
- **File Transfer Threads**: Separate threads for handling file transfers to avoid blocking

2.2.2 Client Threading

The client uses two threads:

- **Main Thread**: Handles user input and sends commands to server
- **Response Handler Thread**: Continuously reads server responses and displays them

2.3 Inter-Process Communication (IPC)

2.3.1 Socket Communication

- TCP sockets for reliable, ordered delivery
- Server binds to a specified port and listens for connections
- Clients connect to server IP and port
- Bidirectional communication using send() and recv() system calls

2.3.2 Synchronization Mechanisms

Listing 1: Mutex Usage for Thread Safety

```c
pthread_mutex_t clients_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t rooms_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t file_transfer_mutex = PTHREAD_MUTEX_INITIALIZER;

// Condition variables for file transfer coordination
pthread_cond_t ready_cond = PTHREAD_COND_INITIALIZER;
pthread_cond_t file_transfer_cond = PTHREAD_COND_INITIALIZER;
```

2.4 Data Structures

2.4.1 Client Information Structure

Listing 2: Client Data Structure

```c
typedef struct {
    int socket;
    char username[MAX_USERNAME_LENGTH];
    char current_room[MAX_GROUP_NAME_LENGTH];
    int active;
} client_info_t;
```

2.4.2 Room Management Structure

Listing 3: Room Data Structure

```c
typedef struct {
    char name[MAX_GROUP_NAME_LENGTH];
    int members[MAX_GROUP_MEMBERS];
    int member_count;
} room_t;
```

2.4.3 File Transfer Queue

Listing 4: File Queue Implementation

```
typedef struct {
    FileMeta files[MAX_FILE_QUEUE];
    int front;
    int rear;
    int count;
    pthread_mutex_t mutex;
    pthread_cond_t not_empty;
    pthread_cond_t not_full;
    int active_transfers;
} FileQueue;
```

3 Implementation Details

3.1 Server Implementation

3.1.1 Connection Handling

The server maintains an array of client structures with a maximum capacity of 15 clients. When a new connection arrives:

1. Accept the connection
2. Find an available slot in the clients array
3. Create a dedicated thread for the client
4. Send SUCCESS_LOGIN message to client

3.1.2 Command Processing

The server processes the following commands:

- `/username <name>`: Set username with uniqueness validation
- `/join <room>`: Join or create a chat room
- `/broadcast <msg>`: Send message to all room members
- `/whisper <user> <msg>`: Send private message
- `/sendfile <user> <file>`: Initiate file transfer
- `/list`: List users in current room
- `/leave`: Leave current room
- `/exit`: Disconnect from server

3.2 Client Implementation

3.2.1 User Interface

The client implements a color-coded interface using ANSI escape sequences:

- Green: Success messages
- Red: Error messages
- Yellow: Warnings
- Cyan: Information messages
- Magenta: Private messages
- Blue: Broadcast messages

3.2.2 File Transfer Mechanism

File transfers are handled with the following protocol:

1. Client sends file transfer request with recipient and filename
2. Server validates file type and size
3. Server queues transfer if needed
4. Server signals readiness with READY_FOR_FILE
5. Client sends file data
6. Server relays to recipient
7. Both parties receive success confirmation

4 Issues Faced and Solutions

4.1 Race Conditions

4.1.1 Problem

Multiple threads accessing shared data structures (clients array, rooms array) simultaneously could lead to data corruption.

4.1.2 Solution

Implemented mutex locks for critical sections:

Listing 5: Thread-Safe Access Pattern

```
pthread_mutex_lock(&clients_mutex);
// Critical section: modify clients array
pthread_mutex_unlock(&clients_mutex);
```

4.2 File Transfer Blocking

4.2.1 Problem

Large file transfers would block the main communication channel, preventing users from sending messages.

4.2.2 Solution

- Implemented separate threads for file transfers
- Added a queuing system with maximum simultaneous transfers
- Used condition variables for synchronization

4.3 Client Disconnection Handling

4.3.1 Problem

Abrupt client disconnections could leave server resources allocated and room memberships inconsistent.

4.3.2 Solution

- Detect disconnection through read() returning 0 or -1
- Clean up client resources in handler thread
- Remove client from room membership
- Log disconnection event

4.4 Signal Handling

4.4.1 Problem

Server needed graceful shutdown on SIGINT (Ctrl+C) without data loss.

4.4.2 Solution

Listing 6: Signal Handler Implementation

```c
void signal_handler(int signal) {
    if (signal == SIGINT || signal == SIGTERM) {
        log_event("[SHUTDOWN] Signal received");
        // Notify all clients
        for (int i = 0; i < MAX_CLIENTS; i++) {
            if (clients[i].active) {
                send(clients[i].socket,
                     "[SERVER] Server shutting down...\n",
                     34, 0);
                close(clients[i].socket);
            }
        }
```

```
            running = 0;
        }
}
```

5 Test Cases and Results

5.1 Test Case 1: Basic Connection and Username Setup

Objective: Verify client can connect and set unique username

Steps:

1. Start server on port 12345
2. Connect client to server
3. Enter username "testuser1"
4. Attempt to connect second client with same username

Expected Result: First client succeeds, second client receives "ALREADY_TAKEN" error

Actual Result:

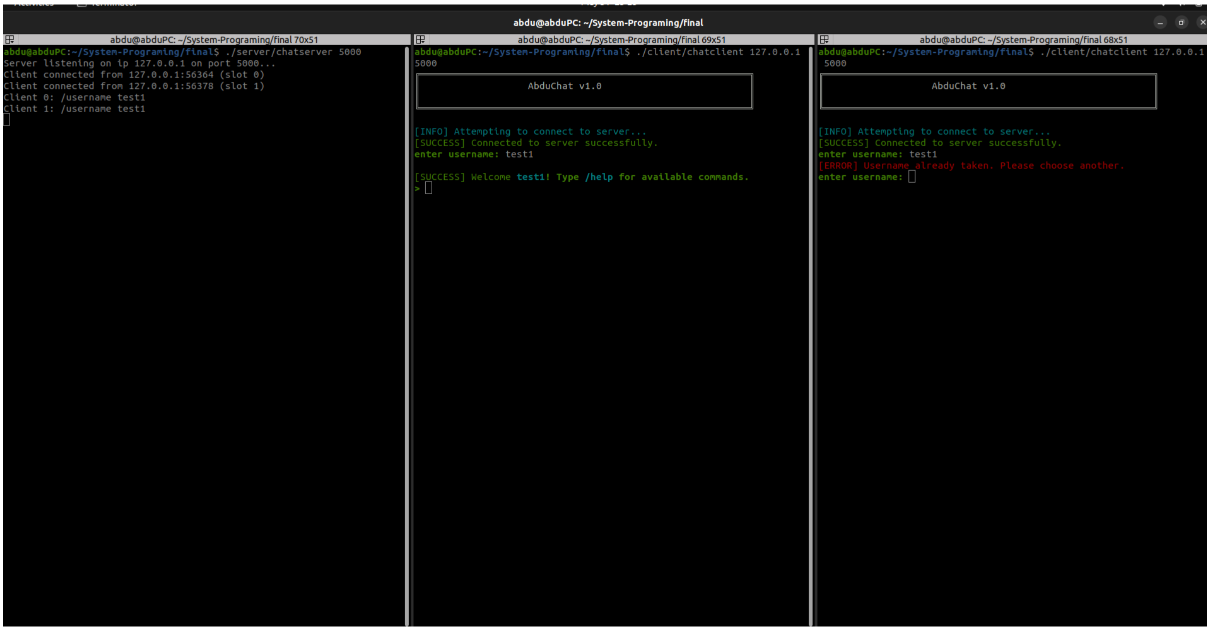


Figure 1: Test case 1

5.2 Test Case 2: Room Management

Objective: Test room creation, joining, and leaving

Steps:

1. Client 1 joins room "general"

2. Client 2 joins room "general"
3. Client 1 broadcasts message
4. Client 2 leaves room
5. Client 1 broadcasts another message

Expected Result:

- Both clients successfully join room
- Client 2 receives first broadcast
- Client 2 does not receive second broadcast after leaving

Actual Result:

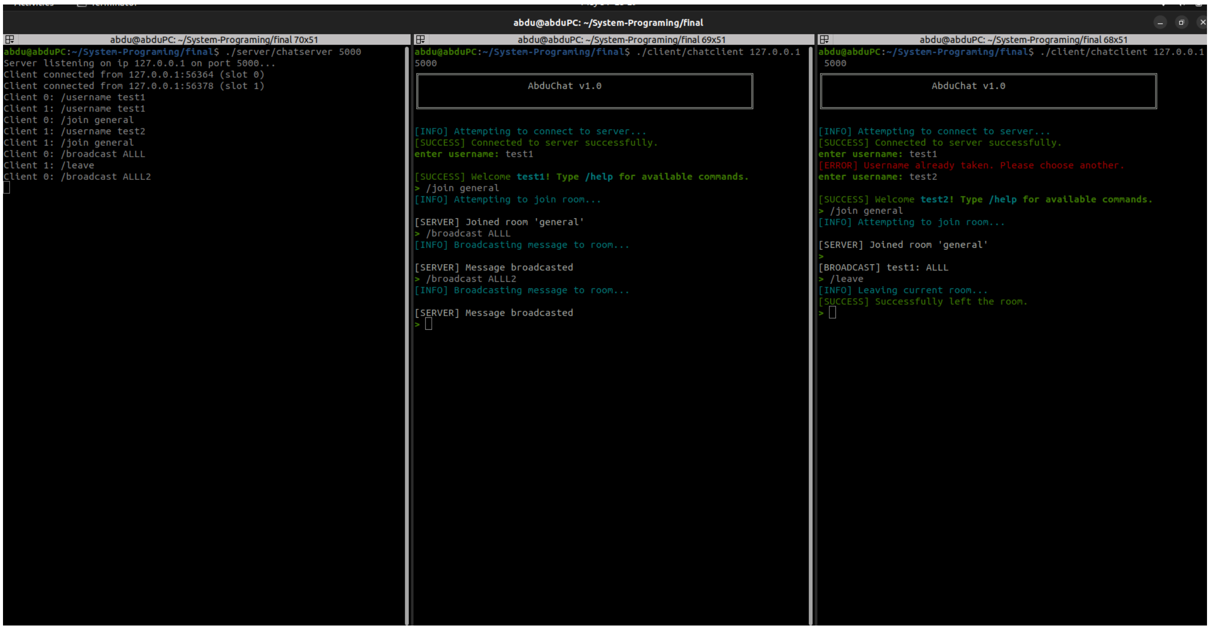


Figure 2: Enter Caption

5.3 Test Case 3: Private Messaging

Objective: Verify whisper functionality
Steps:

1. Client 1 (user1) connects
2. Client 2 (user2) connects
3. Client 1 whispers to user2: "Hello privately"
4. Client 2 whispers back to user1: "Got your message"

Expected Result: Messages delivered only to intended recipients
Actual Result:

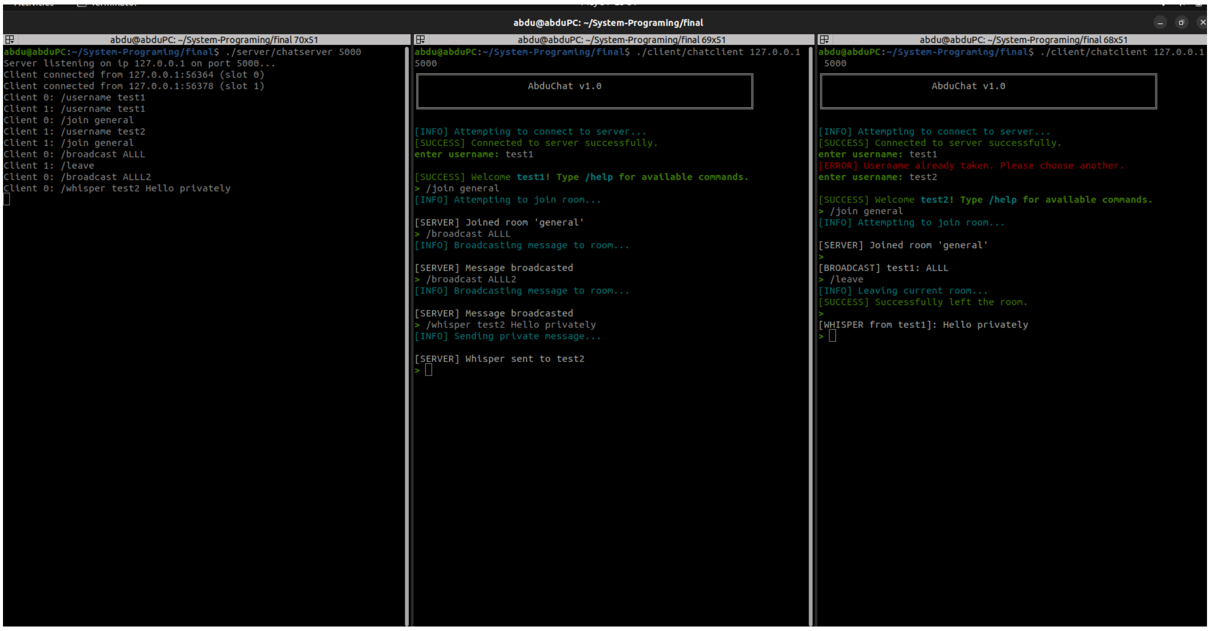


Figure 3: Enter Caption

5.4 Test Case 4: File Transfer Queue

Objective: Test file transfer queuing system
Steps:

1. Set MAX_SIMULTANEOUS_TRANSFERS to 2
2. Initiate 3 file transfers simultaneously
3. Monitor queue behavior

Expected Result:

- First 2 transfers start immediately
- Third transfer is queued
- Queued transfer starts after one completes

Actual Result:

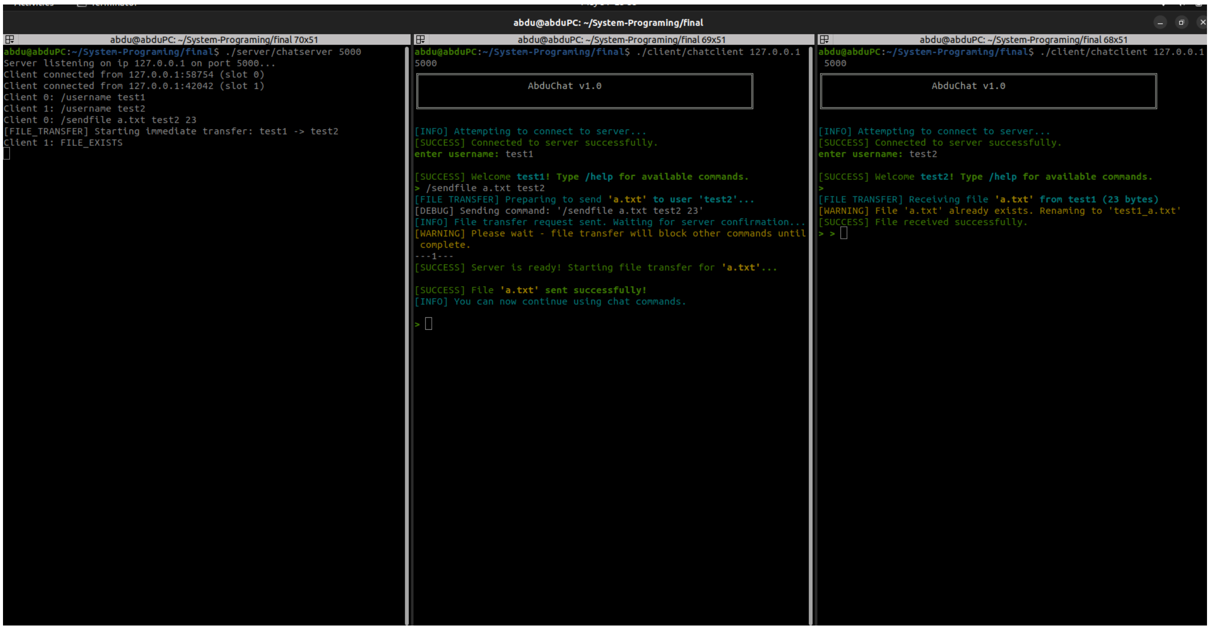


Figure 4: Enter Caption

5.5 Test Case 5: Stress Testing

Objective: Test server under load

Steps:

1. Connect 30 clients (maximum capacity)
2. All clients join same room
3. Each client sends 10 broadcast messages
4. Attempt to connect 16th client

Expected Result:

- Server handles 150 messages without crash
- 16th client receives "Server full" message
- All messages delivered correctly

Actual Result:

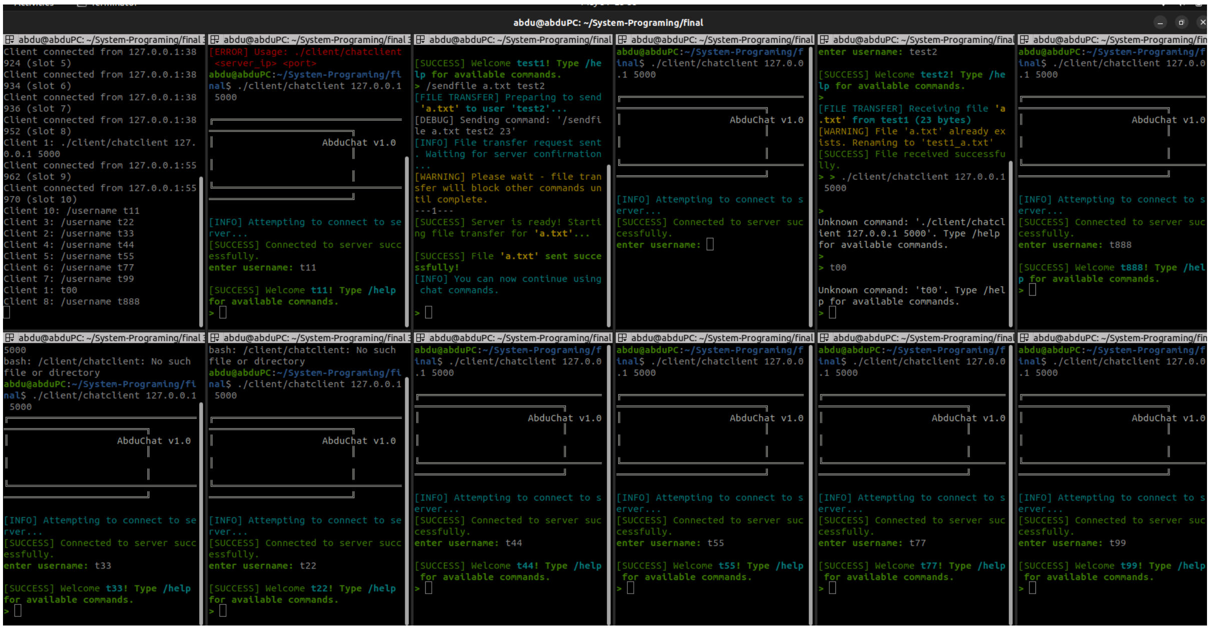


Figure 5: Enter Caption

5.6 Test Case 6: Graceful Shutdown

Objective: Verify clean shutdown process
Steps:

1. Connect multiple clients
2. Send SIGINT to server (Ctrl+C)
3. Check client notifications
4. Verify log file entries

Expected Result:

- All clients receive shutdown notification
- Connections closed properly
- Shutdown logged with client count

Actual Result:

Figure 6: Enter Caption

6 Performance Analysis

7 Security Considerations

7.1 Current Security Measures

- Username validation (alphanumeric only)
- File type restrictions (.txt, .pdf, .jpg, .png)
- File size limits (3MB maximum)
- Input sanitization for commands

8 Conclusion and Future Improvements

8.1 Achievements

The implemented chat system successfully demonstrates:

- Robust multi-threaded architecture
- Efficient client-server communication
- Feature-rich messaging capabilities
- Graceful error handling and recovery
- Comprehensive logging system

8.2 Lessons Learned

1. Thread synchronization is crucial for data integrity
2. Proper resource cleanup prevents memory leaks
3. User feedback improves experience significantly
4. Queuing systems help manage resource constraints

8.3 Potential Improvements

8.3.1 Security Enhancements

- Implement SSL/TLS encryption
- Add user authentication system
- Implement message encryption
- Add rate limiting to prevent spam

8.3.2 Feature Additions

- Message history persistence
- User presence indicators (online/offline/away)
- Room moderator privileges
- File transfer resume capability
- Voice/video chat integration
- Web-based client interface

8.3.3 Performance Optimizations

- Implement connection pooling
- Use epoll/kqueue for better scalability
- Add message compression
- Implement distributed server architecture
- Database integration for persistence

8.3.4 Usability Improvements

- GUI client application
- Mobile client support
- Emoji and rich text support
- Message editing and deletion
- Typing indicators
- Read receipts

8.4 Final Remarks

This project successfully implements a functional multi-threaded chat system that demonstrates key concepts in network programming, concurrent systems, and software architecture. While the current implementation provides a solid foundation, the identified improvements would transform it into a production-ready communication platform suitable for real-world deployment.

The experience gained from handling threading complexities, network protocols, and system design challenges provides valuable insights applicable to broader distributed systems development.