CSE 344

Final Project

Due Date: 31 May 23:59

This assignment will be tested and run on Debian 12 (64-bit) in VirtualBox.
Identical or highly similar submissions will receive -100 points.
A report with screenshots is mandatory. No report = 0 points.
Late submissions will not be accepted.

Multi-threaded Distributed Chat and File Server

In this project, you will implement a multi-threaded, TCP-based chat and file-sharing system using the client-server model. A central server will accept connections from multiple clients simultaneously. Clients will be able to exchange private and group messages, as well as share files with each other.

The server must handle concurrency correctly using threads and synchronization mechanisms (mutexes, semaphores, queues, etc.). You are also expected to use Interprocess Communication (IPC) for queuing file transfers and implement a robust design capable of managing limited resources under load.

Objectives

By completing this project, students will:

- Gain hands-on experience with multi-threading and synchronization
- Understand and apply network programming with TCP sockets
- Implement and manage IPC mechanisms like queues, semaphores, mutexes
- Handle concurrent connections and ensure thread-safe operations
- Build a command-based communication system with clear message protocols
- Design a queue-based file upload manager to simulate limited system resources
- Maintain detailed server-side logging for auditing and debugging

Server Features

- Listens for incoming TCP connections on a given port
- Creates a dedicated thread for each connected client
- Supports at least 15 concurrent clients

- Validates and stores unique usernames (max 16 characters, alphanumeric only)
- Manages rooms (channels) for group messaging
- Allows private messages, room broadcasts, and file transfers
- Implements an upload queue for file sharing (simulate limited resources)
- Logs every user action with timestamps
- Applies mutexes and semaphores to ensure thread-safe queue access
- Responds gracefully to client disconnections or crashes
- If the server receives a SIGINT (Ctrl+C), it must close all active connections gracefully, notify clients, and write all logs before shutting down.

Client Features

- Command-line interface with a simple menu and live interaction
- Connects to the server via TCP (IP and port as arguments)
- User provides a valid username (max 16 chars, alphanumeric)
- Command support:
    - */join <room_name>* → Join or create a room
    - */leave* → Leave the current room
    - */broadcast <message>* → Send message to everyone in the room
    - */whisper <username> <message>* → Send private message
    - */sendfile <filename> <username>* → Send file to user (within file size limit)
    - */exit* → Disconnect from the server
- Color-coded message responses (e.g., green for success, red for error)

Technical Requirements

1. Multi-threading

- Use threading to handle multiple client sessions
- Shared resources must be protected by locks or semaphores

2. Networking

- Use TCP sockets for all communication
- Server must remain responsive even under high load

3. IPC and File Transfer Queue

- Implement a shared queue (max 5 uploads at a time)
- Incoming file requests are enqueued and processed in order
- Synchronization using mutex, semaphore, or condition variable (depending on language)

4. Logging

- Server logs all activities to a file with timestamped entries
- Example:

```
2025-05-18 14:02:31 - [USER: alice12] - joined room 'project1'

2025-05-18 14:03:01 - [LOGIN] user 'john45' connected from 192.168.1.44

2025-05-18 14:03:05 - [JOIN] user 'john45' joined room 'team1'

2025-05-18 14:03:12 - [BROADCAST] user 'john45': Hello team!

2025-05-18 14:03:22 - [SEND FILE] 'project.pdf' sent from john45 to alice99
(success)
```

5. Input Validation & Error Handling

- Handle incorrect commands and show user-friendly messages
- Prevent duplicate usernames, oversized files, invalid commands

Example Commands & Output

Client Terminal:

```
$ ./chatclient 127.0.0.1 5000
Enter your username: emre2025
> /join teamchat
[Server]: You joined the room 'teamchat'
> /broadcast Hello team!
[Server]: Message sent to room 'teamchat'
> /whisper john42 Can you check this?
[Server]: Whisper sent to john42
> /sendfile homework.pdf john42
[Server]: File added to the upload queue.
> /exit
[Server]: Disconnected. Goodbye!
```

Server Console:

```
[INFO] Server listening on port 5000...
[CONNECT] New client connected: emre2025 from 192.168.1.104
[COMMAND] emre2025 joined room 'teamchat'
[COMMAND] emre2025 broadcasted to 'teamchat'
[COMMAND] emre2025 sent whisper to john42
[COMMAND] emre2025 initiated file transfer to john42
[DISCONNECT] Client emre2025 disconnected.
```

Client Terminal:

```
> /join team1
[Server]: You joined the room 'team1'
> /broadcast Hello
[Server]: Message sent to room
> /sendfile project.pdf john
[Server]: File sent successfully
```

Server Console:

```
[CONNECT] Client connected: user=alice12
[INFO] alice12 joined room 'team1'
[FILE] alice12 sent file 'project.pdf' to john
```

Test Scenarios

You must test your system under the following conditions. Include screenshots and corresponding log entries in your report.

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
```

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
```

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.

Optional: Include estimated wait time or feedback in client.
Log example:

```
[FILE-QUEUE] Upload 'project.pdf' from emre02 added to queue. Queue size: 5
```

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.
```

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'
```

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.
```

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.
```

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.

Log example:

```
[ROOM] user 'ayse99' rejoined 'group2'
```

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'
```

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.
```

Design Constraints & Rules

- Usernames: Max 16 chars, alphanumeric only
- File transfer:
    - Accepted file types: .txt, .pdf, .jpg, .png
    - Max file size: 3MB
- Each room has a max capacity of 15 users
    - Room names must be alphanumeric and up to 32 characters. No spaces or special characters allowed.
- Upload queue capacity must be limited (e.g., MAX_UPLOAD_QUEUE = 5)
- Use of high-level languages is allowed:
    - C/C++: pthread, sockets, semaphore.h
    - Python: threading, socket, queue
    - Java: Thread, Socket, Semaphore, etc.

Submission Format

Submit a compressed `.zip` file with the following structure:

```
studentID>_<NAME_SURNAME>.zip

├── report.pdf          → Design explanation, diagrams, and test results

├── Makefile            → For compilation (if using C/C++)

├── server/            → Server source code

│   └── ...

├── client/            → Client source code

│   └── ...

└── example_log.txt     → Sample log file from a test run
```

To ensure a uniform evaluation process, the names of the executables **must be exactly as specified below**:

- Server executable: **chatserver**
- Client executable: **chatclient**

You must ensure that running the following commands starts your program correctly:

$ ./chatserver <port>

$ ./chatclient <server_ip> <port>

❗ **Any deviation from these executable names will result in a point penalty.**

Do not name your executables differently (e.g., server_v2, main, client_app, etc.).
If your submission includes differently named binaries, it will not be tested.

Report Must Include

- Introduction and problem definition
- Design details (IPC, thread model, architecture diagram)
- Issues faced and how they were solved
- Test cases and results (with screenshots or logs)
- Conclusion and potential improvements

For your questions, please contact: **zbilici@gtu.edu.tr**