OPERATING SYSTEMS - Homework 3

Itamar Kolodny 211490362 Tamara Bluzer 315287441

In this homework assignment, we implemented a multi-client chat program using the TCP protocol. The system allows multiple users to connect to a central server to communicate through public broadcasts or private "whisper" messages.

We split the code into three main files to keep the logic organized:

1. hw3server.c: Contains the server-side logic for managing connections and routing messages.
2. hw3client.c: Contains the client-side logic for user input and displaying chat data.
3. chat.h: Defines the shared handshake structure and constants used by both components.

Implementation Details

Initialization and Handshake

The server binds to a specified TCP port and listens for incoming connections.Then, when a client connects, it immediately sends a chat_handshake_t structure containing its name. The server receives this name and prints a connection message including the client's IP address.

Concurrency with Select

Both the server and client use the select() system call to handle multiple tasks in parallel without using multiple threads. The server uses select() to monitor its listening socket for new connections while simultaneously watching existing client sockets for incoming messages, while the client uses select() to monitor both the keyboard and the server socket. This ensures the user can see new messages even while they are typing.

Message Routing

- Normal Messages: Any text entered by a user is sent to the server, which prefixes it with the sender's name and broadcasts it to all connected clients.
- Whisper Messages: If a message starts with @friend, the server parses the destination name and sends the message only to that specific user.
- Exiting: If a user types !exit, the client notifies the server, prints "client exiting," and shuts down.

Synchronization and Logic

- To avoid blocking, we used select() to manage file descriptors efficiently.
- The server maintains an array of client structures to keep track of active sockets and their associated names.
- We implemented a write_all helper function in the client to ensure that large messages are fully transmitted over the socket without data loss.

Challenges

We faced several challenges during this assignment. The most challenging part was ensuring the client could handle user input and server messages simultaneously. Using select() allowed us to avoid the complexity of threading while keeping the keyboard responsive. Another challenge that we faced was properly splitting the "whisper" messages (identifying the @ symbol, the target name, and the actual message) required careful string manipulation in the server to ensure messages were routed correctly. Managing the client list when someone disconnects was another challenge which was a bit tricky. We implemented logic to detect a closed socket, print the disconnection message, and then clean up the client array by shifting elements to fill the gap.

Testing

We tested the program with several scenarios:

1. Connecting multiple clients and sending broadcast messages.
2. Verifying that whisper messages are only seen by the intended recipient.
3. Testing the “!exit” command to ensure the client terminates cleanly and the server detects the departure.
4. Stress testing with long messages to ensure the buffer handling works correctly.
5. connecting 16 clients (the maximum allowed) and verifying that all can send/receive broadcasts and whispers correctly. We also attempted to connect a 17th client and verified the server rejects it gracefully without affecting existing connections.
6. connecting with a very long name and verifying the server stores/prints it safely.
7. sending a message at the maximum allowed length and verifying correct formatting, correct delivery to all clients, and no truncation-related bugs or buffer overflows.
8. sending many messages quickly to confirm the server remains responsive and correctly routes messages without freezing or losing connections.