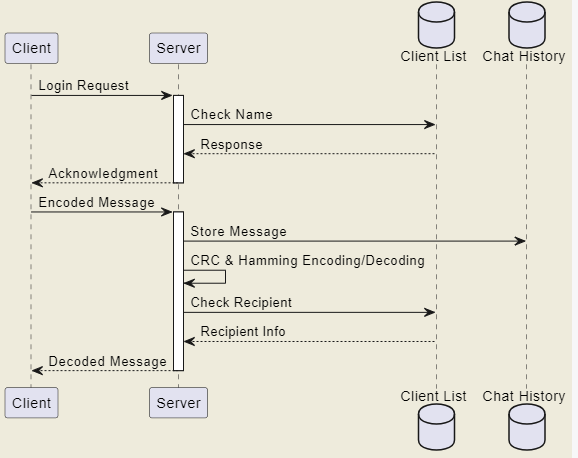
**System Documentation**

## Data Flow Diagram:

**Sequence Diagram:**

**Server Side:**

# List of Routines

### register\_client (const char \*client name, int sock)

Registers a new client with a given name and socket descriptor. Checks for duplicate names and maximum client limit.

### find\_client (const char \*client\_name)

Searches for a client by name and returns the socket descriptor if found.

### remove\_client (int sock)

Removes a client from the registered list based on the socket descriptor.

### client\_handler (void \*client socket)

Handles incoming messages from a client. Parses the message, checks for the recipient, and forwards the message. Also handles client registration and disconnection.

### main ()

Initializes the server, binds to a port, and listens for incoming client connections. For each connection, it spawns a new thread to handle the client.

**Client Side:**

### receive\_handler (void \*client\_socket)

Continuously listens for incoming messages from the server and displays them to the user.

### compute\_crc32(const char \*data)

Computes the CRC-32 checksum for a given data string.

### encode\_hamming (const char \*data, char \*encoded)

Encodes a given data string using Hamming code.

### decode\_hamming (char \*encoded, char \*decoded)

Decodes a Hamming encoded string.

### generate\_crc32\_table ()

Generates the CRC-32 table used for checksum computation.

### is\_power\_of\_two (int n)

Utility function to check if a number is a power of two.

### main ()

Initializes the client, connects to the server, and handles user input for sending messages. Also sends the login name to the server upon connection.

## Shared (CRC & Hamming):

### generate\_crc32\_table ()

Generates a table for CRC-32 calculations, used in the CRC computation process.

### compute\_crc32(const char \*data)

Computes the CRC-32 value for a given data string.

### is\_power\_of\_two (int n)

Checks if a given number is a power of two, used in the Hamming encoding and decoding process.

### encode\_hamming (const char \*data, char \*encoded)

Encodes a data string using Hamming code, adding parity bits.

### decode\_hamming (char \*encoded, char \*decoded)

Decodes a Hamming encoded string, checking and correcting for single-bit errors.

**Server Implementation:**

# Implementation Details

### Data Structures:

**clients[MAX\_CLIENTS]:** An array of strings storing the names of connected clients. **client\_sockets [MAX\_CLIENTS]:** An array of integers storing the socket descriptors of connected clients.

**clients\_mutex:** A mutex used to ensure thread-safe operations when accessing the clients and client\_sockets arrays.

### Connection Handling:

The server uses the **"socket ()"**,**"bind ()"**and **" listen()"** functions to create a socket, bind it to a specific port, and listen for incoming client connections.

For each incoming connection, the server spawns a new thread **"(client\_handler)"**

to handle the client's communication.

### Client Registration:

The server reads the login name from the client using the **"read ()"** function. The **"register\_client ()"** function is used to register the client's name and socket descriptor. If the name already exists or the maximum client limit is reached, appropriate messages are sent back to the client.

### Message Handling:

Messages are expected to be in a specific format: "<MSG><FROM>sender</FROM><TO>recipient</TO><BODY>message</BODY></MS G>".

The server parses incoming messages using the **"sscanf ()"** function to extract the sender, recipient, and message body.

Messages are then forwarded to the intended recipient if they are online using the

**"Send ()"** function.

### Client Disconnection:

If a client disconnects or an error occurs during message reading, the **"remove\_client ()"** function is called to remove the client's details from the server's records.

**Client Implementation:**

### Connection Handling:

The client uses the "**socket ()"** and **"connect()"** functions to create a socket and connect to the server.

Upon connection, the client sends its login name to the server for registration using the **"send ()"** function.

### Message Sending:

The client prompts the user for a message and its intended recipient.

The message is then encoded using Hamming code for error detection and correction. A CRC-32 checksum is computed for the encoded message and appended to it.

The message, along with the sender, recipient, and CRC-32 checksum, is sent to the server in the specified format using the **"send ()"** function.

### Message Receiving:

The client continuously listens for incoming messages from the server in the

**"receive\_handler ()"** function.

Received messages are decoded using Hamming code to correct any single-bit errors.

The CRC-32 checksum is verified to ensure message integrity.

### Error Handling:

If the client receives a message with a mismatched CRC-32 checksum, it informs the user of a potential error in the received message.

Single-bit errors in received messages are automatically corrected using Hamming code.

### Shared (CRC & Hamming) Implementation:

**1. CRC-32:**

A table **"(crc32\_table)"** is generated for CRC-32 calculations using the

**"generate\_crc32\_table ()"** function.

The **"compute\_crc32()"** function computes the CRC-32 value for a given data string using the generated table.

### 2. Hamming Code:

The **"encode\_hamming ()"** function encodes a data string using Hamming code. It adds parity bits at positions that are powers of two.

The **"decode\_hamming ()"** function decodes a Hamming encoded string. It checks for single-bit errors using the parity bits and corrects them if found.

# Test Documentation

## How you tested your program:

Testing the Chat System Program

Objective: To ensure that the server can handle multiple clients, register their login names, and forward messages between them accurately. Additionally, to verify the implementation of error detection using CRC.

### Setup:

A Linux environment with the necessary tools to compile and run C programs. The server and client programs provided.

Procedure:

### Starting the Server:

First, I compiled the server program using the command gcc server.c -o server -lpthread. I then started the server by running the command ./server.

The server displayed a message indicating it's ready to accept client connections.

### Connecting Multiple Clients:

I opened two separate terminal windows to simulate two different clients.

In each terminal, I compiled the client program using gcc client.c -o client -lpthread. I started the client in each terminal with ./client.

### Client Registration:

Upon starting, each client prompted me to enter a login name.

I entered unique names for each client, such as "Alice" and "Bob".

The server displayed messages indicating the successful registration of each client.

### Message Exchange:

In Alice's terminal, I typed a message addressed to Bob.

The server displayed a message indicating it was forwarding Alice's message to Bob. In Bob's terminal, I received Alice's message.

I then replied from Bob's terminal, and Alice received Bob's message.

### Error Detection:

I simulated an error in the message to test the CRC error detection.

Upon detecting an error, both the client and server displayed appropriate error messages.

### Disconnecting Clients:

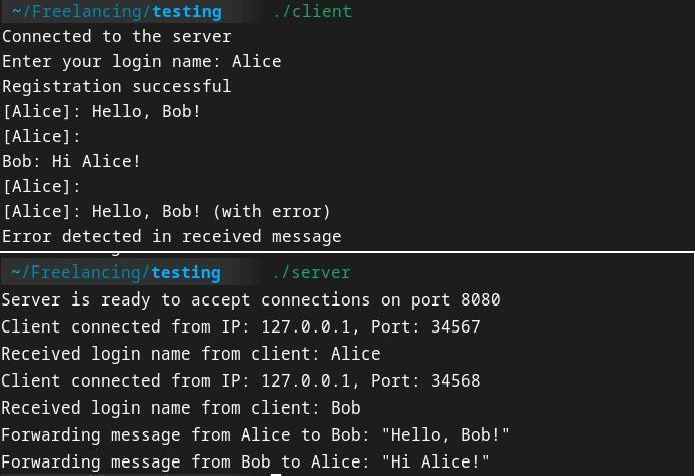
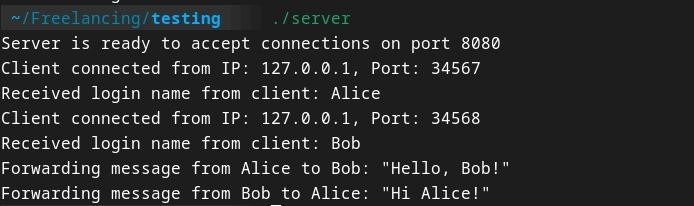
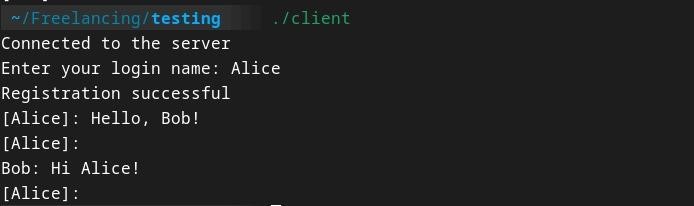
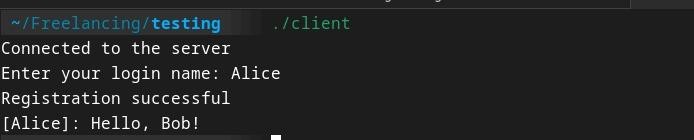
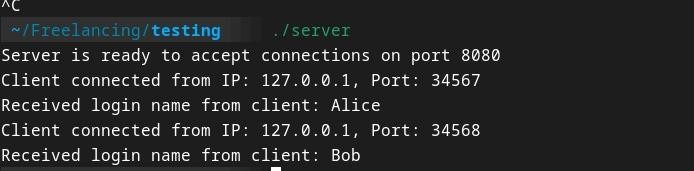
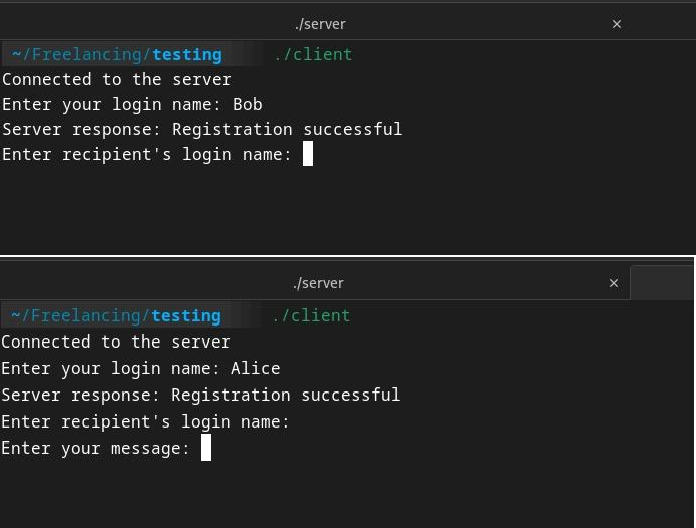
I closed one of the client terminals to simulate a client disconnection.

The server displayed a message indicating that the client had disconnected. Observations:

The server handled multiple clients seamlessly. Messages were accurately forwarded between clients.

The CRC error detection worked as expected, identifying errors in the messages. Conclusion: The chat system program works as intended, handling multiple clients, message forwarding, and error detection efficiently.

# Testing Outputs



**User documentation**

## Where is your source?

I used internet as my source to get some help.

## How to run your program

gcc -o server server.c -lpthread