Introduction (Updated with helper_node.c)

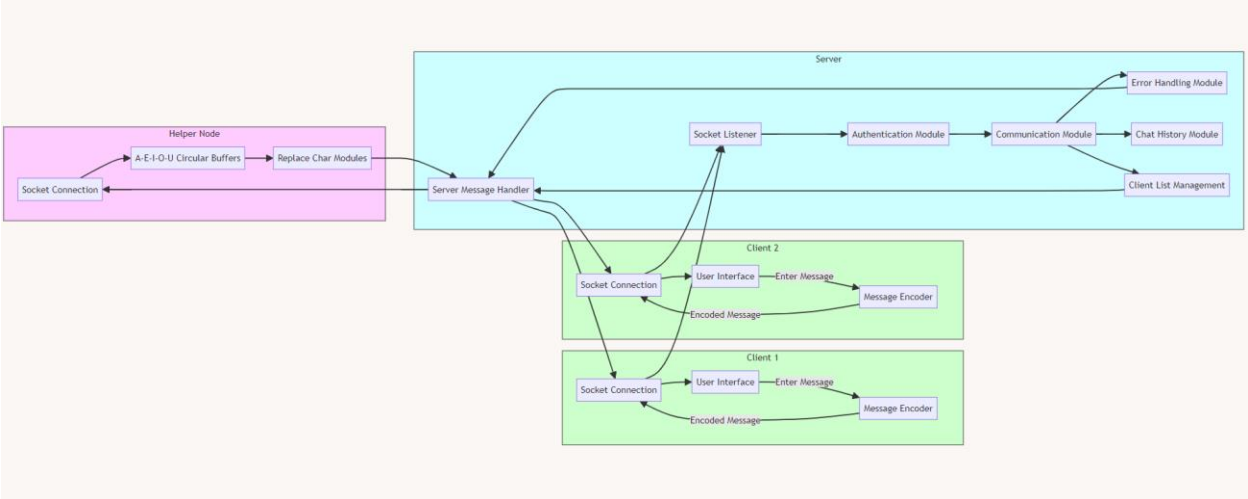
This C program is a chat application that supports communication between multiple clients using Linux TCP sockets. It includes error detection using the CRC algorithm and error detection/correction using Hamming codes. A significant addition to the program is the **helper_node.c** component, which enhances server-side processing by converting lowercase vowels in messages to uppercase using multi-threading. This feature operates alongside the existing error detection and correction mechanisms, providing an additional layer of message processing. The program also allows users to introduce errors for testing purposes, with proper error messages and handling for both CRC and Hamming errors. Users can choose between CRC and Hamming error handling methods, and robust testing has been performed using various input files and scenarios.

Purpose (Updated)

The purpose of this project is to:

1. Gain practical experience in collaborative task execution using socket programming.
2. Implement error detection and correction mechanisms, specifically CRC (Cyclic Redundancy Check) and Hamming codes.
3. Create a client/server application that facilitates one-on-one chat communication between users, with enhanced server-side message processing through **helper_node.c**.
4. Develop robust error-handling capabilities for transmission errors in communication.

DFD:



Explanation:

- **Clients**: Client 1 and Client 2 are each equipped with a User Interface, Message Encoder, and Socket Connection.
- **Server**: Central to the application, handling client connections, authentication, message processing, error handling, chat history, and client management.
- **Helper Node**: Specific to Project 3, it includes a socket connection to the server and modules for processing vowels in messages.
- **Flow of Data**: The diagram shows the flow of messages from clients to the server, through the helper node for vowel processing, and back to the clients.

Key Features and Requirements (Updated)

- **Server Capabilities**: Responds to different client requests, enhanced with **helper_node.c** for vowel conversion in messages.
- **User Authentication**: Unique 8-character usernames adhering to specific format rules.
- **Communication Handling**: Manages communication for up to six users concurrently.
- **Chat History Maintenance**: Keeps a record of chats for reference.
- **Error Detection and Correction**: Implements CRC and Hamming codes for error management.
- **Error Testing Options**: Allows users to introduce errors for robust testing.
- **Comprehensive Error Messaging and Handling**: Ensures proper management of CRC and Hamming errors.
- **Choice of Error Handling Methods**: Users can select between CRC and Hamming methods.
- **Robust Testing**: Conducts extensive testing using various input files and scenarios, inclusive of CRC, Hamming detection, and Hamming correction.
- **Server-Side Encoding/Decoding**: Incorporates **helper_node.c** for converting vowels from lowercase to uppercase, showcasing advanced multi-threading and synchronization in network programming.

List of Routines and Descriptions for helper_node.c

1. init_buffer(CircularBuffer *buf)

- **Purpose**: Initializes a circular buffer.
- **Description**: Sets up the mutex and condition variables for the buffer, and initializes the head and tail pointers.

2. put(CircularBuffer *buf, char *item)

- **Purpose**: Inserts an item into the circular buffer.
- **Description**: Locks the buffer, waits if full, then places the item and updates the tail pointer. Signals that the buffer is not empty before unlocking.

3. get(CircularBuffer *buf)

- **Purpose**: Retrieves an item from the circular buffer.
- **Description**: Locks the buffer, waits if empty, then retrieves the item and updates the head pointer. Signals that the buffer is not full before unlocking.

4. replace_chars(void *args)

- **Purpose**: Thread routine for vowel replacement.
- **Description**: Processes messages from its input buffer, replacing specified characters (vowels) from lowercase to uppercase, then puts the processed message into the next buffer or, for the last buffer, directly into the 'u' buffer.

5. handle_connection(void *arg)

- **Purpose**: Manages individual client connections.
- **Description**: Handles the server-client communication. Receives messages from the client, places them in the first buffer ('a'), retrieves processed messages from the last buffer ('u'), and sends them back to the client.

6. start_helper_node(int port)

- **Purpose**: Starts the helper node server.
- **Description**: Sets up and starts the server socket to listen for incoming connections. For each connection, it creates a new thread to handle the connection using **handle_connection**.

7. main(int argc, char* argv[])

- **Purpose**: Main function to start the helper node.
- **Description**: Initializes buffers and threads for vowel processing. Parses command-line arguments for the port number and starts the helper node server.

Thread Management and Synchronization Routines:

- **pthread_mutex_init, pthread_cond_init**: Initializes mutexes and condition variables for thread synchronization.
- **pthread_create**: Creates threads for handling connections and processing messages.
- **pthread_mutex_lock, pthread_mutex_unlock**: Locks and unlocks mutexes for thread-safe operations on buffers.

- **pthread_cond_wait, pthread_cond_signal**: Manages thread waiting and signaling for buffer operations.

Memory and Network Management Routines:

- **malloc, free**: Allocates and frees dynamic memory.
- **strdup**: Duplicates strings for independent processing in threads.
- **socket, bind, listen, accept**: Network functions for setting up and managing server sockets.

Implementation Details of helper_node.c

Server Implementation:

1. **Threaded Processing**:
    - Each vowel is processed in a separate thread, allowing simultaneous handling of different parts of the message.
    - Threads are launched using **pthread_create** and passed specific arguments through **ReplaceArgs** for targeted vowel processing.
2. **Circular Buffers for Message Handling**:
    - Utilizes a series of circular buffers, each dedicated to a specific vowel ('a', 'e', 'i', 'o', 'u').
    - Manages the flow of messages through these buffers using **put** and **get** functions, ensuring synchronized processing.
3. **Character Replacement Mechanism**:
    - Implements the **replace_chars** function for each vowel thread.
    - Processes messages from its designated buffer, replacing lowercase vowels with uppercase equivalents.
4. **Client Connection Management**:
    - Manages client connections using the **handle_connection** function.
    - Receives messages from clients, pushes them into the first buffer ('a'), and retrieves processed messages from the last buffer ('u') for delivery.
5. **Server Socket Setup**:
    - Establishes server sockets using standard socket programming functions (**socket**, **bind**, **listen**, **accept**).
    - Listens for incoming connections and creates new threads for each connection.

Synchronization and Concurrency:

1. **Mutex Locks and Condition Variables**:
    - Ensures thread-safe operations on buffers with mutex locks (**pthread_mutex_lock**, **pthread_mutex_unlock**).
    - Uses condition variables (**pthread_cond_wait**, **pthread_cond_signal**) to manage buffer full and empty states.
2. **Dynamic Buffer Management**:
    - Handles dynamic message sizes and varying processing times.
    - Ensures that buffer operations do not block or delay the processing of other messages.

Error Handling and Network Communication:

1. **Robust Error Handling**:
    - Incorporates error checking in network operations and thread management.
    - Gracefully handles errors such as failed socket connections or thread creation issues.
2. **TCP Socket Communication**:
    - Maintains consistent and reliable communication over TCP sockets.
    - Ensures that messages are transmitted correctly between the client, server, and helper node.

Resource Management and Scalability:

1. **Memory Management**:
    - Manages memory allocation and deallocation carefully, especially for dynamically created strings and thread arguments.
    - Prevents memory leaks by freeing memory after use.
2. **Scalable Design**:
    - Designed to handle multiple clients and process multiple messages concurrently.
    - Can be scaled to accommodate more vowels or different types of processing by adding more threads and buffers.

Updated Testing Methodology for Project 3

1. Unit Testing:

- **Enhanced Testing**: Includes tests for the **helper_node.c** functions like buffer initialization, put/get operations, and character replacement logic.

- **Component Testing**: Continues with tests on CRC and Hamming encoding/decoding, username validation, and client connection handling, now including the interaction with the helper node.

2. Integration Testing:

- **Helper Node Integration**: Tests the integration of **helper_node.c** with the existing server and client modules, ensuring smooth data flow and processing between these components.
- **Server-Client Communication**: Verifies the server's ability to handle multiple client connections, process and forward messages to **helper_node.c** for processing, and return modified messages to clients.

3. End-to-End Testing:

- **Real-World Scenarios**: Simulates complete application use, including interaction with **helper_node.c** for vowel processing in messages.
- **Functionality Testing**: Covers user authentication, message sending and receiving with vowel conversion, chat history maintenance, and error handling.

4. Error Injection Testing:

- **Message Processing Errors**: Introduces errors not only in message encoding (CRC and Hamming) but also in the processing logic of **helper_node.c**, testing the system's response.
- **Robust Error Handling**: Ensures error detection and correction mechanisms work in tandem with the new vowel processing feature.

5. Multi-Frame Testing:

- **Extended Message Testing**: Tests the application’s handling of multi-frame messages, including those processed by **helper_node.c**, ensuring that longer messages with vowel modifications are transmitted and reconstructed accurately.

Updated Testing Outputs:

Test Case 1: Basic Functionality Test with Vowel Processing

- **Input**: Two clients connect, authenticate, and exchange text messages with vowel processing through **helper_node.c**.
- **Expected Output**: Successful user authentication, accurate vowel conversion in messages, error-free exchange, and proper chat history maintenance.

Test Case 2: Enhanced Error Detection Test

- **Input**: Messages with intentional CRC errors, plus errors in vowel processing.
- **Expected Output**: Accurate detection of CRC and vowel processing errors, with appropriate notifications to clients.

Test Case 3: Comprehensive Error Correction Test

- **Input**: Messages with intentional Hamming errors and processing errors in **helper_node.c**.
- **Expected Output**: Effective detection and correction of single-bit Hamming errors and handling of vowel processing errors.

Test Case 4: Large Message Test with Vowel Processing

- **Input**: Large text messages processed through **helper_node.c**.
- **Expected Output**: Complete transmission and reconstruction of large messages with accurate vowel conversion, without data loss.

Test Case 5: Chat History Test with Processed Messages

- **Input**: Exchange of messages with vowel processing, stored in chat history.
- **Expected Output**: Accurate logging of modified messages in chat history files for each client pair.

Test Case 6: User Authentication Test

- **Input**: Authentication attempts with invalid usernames.
- **Expected Output**: Server rejects invalid usernames, providing appropriate error feedback.

Test Case 7: Error Handling Choice with Vowel Processing

- **Input**: Choice of error handling methods for messages undergoing vowel processing.
- **Expected Output**: Correct application of selected error handling method, with messages processed through **helper_node.c** reflecting these choices.