Detailed Chat Server Application Project Report

1. Introduction

This project implements a robust, multi-client chat server application using the C programming language. The system facilitates real-time communication between multiple clients through a central server, leveraging socket programming and multi-threading techniques. This report provides an in-depth analysis of the system architecture, component functionalities, and implementation details.

2. System Architecture

The application follows a client-server architecture, designed for scalability and real-time performance.

2.1 Client-Side Architecture

The client program is structured to handle bidirectional communication with the server:

- **Connection Management**: Establishes and maintains a TCP connection with the server.
- **Message Transmission**: Sends user input to the server.
- **Message Reception**: Asynchronously receives and displays messages from the server.
- **User Interface**: Provides a simple command-line interface for user interaction.

2.2 Server-Side Architecture

The server program is designed to handle multiple client connections concurrently:

- **Connection Listening**: Continuously listens for new client connections.
- **Client Management**: Maintains a list of connected clients and their associated information.
- **Message Routing**: Receives messages from clients and broadcasts them to all other connected clients.
- **Concurrency**: Utilizes multi-threading to handle multiple clients simultaneously.

3. Detailed Component Analysis

3.1 Client Program

3.1.1 Main Function

The `main` function serves as the entry point for the client application. Here's a detailed breakdown of its functionality:

```c
int main(int argc, char *argv[]) {
    // Command-line argument validation
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <IP> <port>\n", argv[0]);
        exit(1);
    }
```

```
    int sock;
    struct sockaddr_in server;
    char message[BUFFER_SIZE];
    pthread_t receive_thread;

    // Socket creation
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("Could not create socket");
        return 1;
    }

    // Server address configuration
    server.sin_addr.s_addr = inet_addr(argv[1]);
    server.sin_family = AF_INET;
    server.sin_port = htons(atoi(argv[2]));

    // Establish connection to server
    if (connect(sock, (struct sockaddr *)&server, sizeof(server)) < 0) {
        perror("Connect failed");
        return 1;
    }

    printf("Connected to server\n");

    // Create thread for receiving messages
    if (pthread_create(&receive_thread, NULL, receive_messages,
(void*)&sock) < 0) {
        perror("Could not create receive thread");
        return 1;
    }

    // Main loop for sending messages
    while (1) {
        fgets(message, BUFFER_SIZE, stdin);
        if (send(sock, message, strlen(message), 0) < 0) {
            perror("Send failed");
            return 1;
        }
    }

    close(sock);
    return 0;
}
```

Key aspects:

- **Argument Parsing**: Validates and extracts server IP and port from command-line arguments.
- **Socket Initialization**: Creates a TCP socket using `socket()`.
- **Server Connection**: Configures server address and establishes connection using `connect()`.

- **Thread Creation**: Spawns a separate thread for message reception using `pthread_create()`.
- **Message Sending Loop**: Continuously reads user input and sends it to the server.

3.1.2 receive_messages Function

This function runs in a separate thread to handle incoming messages:

```
void *receive_messages(void *socket_desc) {
    int sock = *(int*)socket_desc;
    char message[BUFFER_SIZE];
    int read_size;

    while ((read_size = recv(sock, message, BUFFER_SIZE, 0)) > 0) {
        message[read_size] = '\0';
        printf("%s", message);
        fflush(stdout);
    }

    if (read_size == 0) {
        puts("Server disconnected");
    } else if (read_size == -1) {
        perror("recv failed");
    }

    return NULL;
}
```

Key features:

- **Continuous Reception**: Uses a while loop with `recv()` to continuously check for incoming messages.
- **Message Display**: Immediately prints received messages to the console.
- **Error Handling**: Detects and reports server disconnection or reception errors.

3.2 Server Program

3.2.1 Main Function

The server's `main` function initializes the server and manages client connections:

```
int main(int argc, char *argv[]) {
    // Command-line argument validation
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <port>\n", argv[0]);
        exit(1);
    }

    int server_socket, client_socket;
```

```
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_len = sizeof(client_addr);
    pthread_t tid;

    // Create server socket
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1) {
        perror("socket");
        exit(1);
    }

    // Configure server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(atoi(argv[1]));

    // Bind socket to address
    if (bind(server_socket, (struct sockaddr *)&server_addr,
sizeof(server_addr)) == -1) {
        perror("bind");
        exit(1);
    }

    // Listen for connections
    if (listen(server_socket, 5) == -1) {
        perror("listen");
        exit(1);
    }

    printf("Server is listening on port %s\n", argv[1]);

    // Main loop to accept client connections
    while (1) {
        client_socket = accept(server_socket, (struct sockaddr
*)&client_addr, &client_len);
        if (client_socket == -1) {
            perror("accept");
            continue;
        }

        pthread_mutex_lock(&clients_mutex);
        if (client_count >= MAX_CLIENTS) {
            pthread_mutex_unlock(&clients_mutex);
            close(client_socket);
            continue;
        }

        // Add new client to the clients array
        for (int i = 0; i < MAX_CLIENTS; i++) {
            if (clients[i].socket == 0) {
                clients[i].socket = client_socket;
                pthread_create(&tid, NULL, handle_client, &clients[i]);
                client_count++;
```

```
                break;
            }
        }
        pthread_mutex_unlock(&clients_mutex);
    }

    close(server_socket);
    return 0;
}
```

Key components:

- **Socket Initialization**: Creates a TCP socket and binds it to the specified port.
- **Connection Listening**: Puts the socket in listening mode to accept incoming connections.
- **Client Acceptance Loop**: Continuously accepts new client connections.
- **Client Management**: Adds new clients to the `clients` array and creates a new thread for each.
- **Concurrency Control**: Uses mutex locks to ensure thread-safe access to shared resources.

3.2.2 handle_client Function

This function manages individual client connections:

```
void *handle_client(void *arg) {
    client_t *client = (client_t *)arg;
    char buffer[BUFFER_SIZE];
    char message[BUFFER_SIZE + USERNAME_SIZE + 4];

    // Ask for username
    send(client->socket, "Enter your username: ", 21, 0);
    int bytes_received = recv(client->socket, client->username,
USERNAME_SIZE - 1, 0);
    if (bytes_received <= 0) {
        close(client->socket);
        return NULL;
    }
    client->username[bytes_received - 1] = '\0'; // Remove newline

    // Announce new user
    sprintf(message, "%s has joined the chat.\n", client->username);
    broadcast_message(message, client->socket);

    // Main message handling loop
    while (1) {
        bytes_received = recv(client->socket, buffer, BUFFER_SIZE, 0);
        if (bytes_received <= 0) {
            break;
        }
        buffer[bytes_received] = '\0';
        sprintf(message, "%s: %s", client->username, buffer);
        broadcast_message(message, client->socket);
```

```
    }

    // Handle client disconnection
    sprintf(message, "%s has left the chat.\n", client->username);
    broadcast_message(message, client->socket);

    pthread_mutex_lock(&clients_mutex);
    for (int i = 0; i < MAX_CLIENTS; i++) {
        if (clients[i].socket == client->socket) {
            clients[i].socket = 0;
            break;
        }
    }
    client_count--;
    pthread_mutex_unlock(&clients_mutex);

    close(client->socket);
    return NULL;
}
```

Key features:

- **Username Management**: Prompts for and stores the client's username.
- **Message Reception**: Continuously receives messages from the client.
- **Message Broadcasting**: Forwards received messages to all other clients.
- **Disconnection Handling**: Manages client disconnection and updates the clients array.

3.2.3 broadcast_message Function

This function sends a message to all connected clients except the sender:

```
void broadcast_message(char *message, int sender_socket) {
    pthread_mutex_lock(&clients_mutex);
    for (int i = 0; i < MAX_CLIENTS; i++) {
        if (clients[i].socket != 0 && clients[i].socket !=
sender_socket) {
            send(clients[i].socket, message, strlen(message), 0);
        }
    }
    pthread_mutex_unlock(&clients_mutex);
}
```

Key aspects:

- **Thread Safety**: Uses mutex locks to ensure safe access to the shared clients array.
- **Selective Sending**: Sends the message to all clients except the original sender.

4. Key Data Structures

4.1 Client Structure

```
typedef struct {
    int socket;
    char username[USERNAME_SIZE];
} client_t;
```

This structure encapsulates client information:

- `socket`: File descriptor for the client's socket.
- `username`: String to store the client's chosen username.

4.2 Clients Array

```
client_t clients[MAX_CLIENTS];
int client_count = 0;
pthread_mutex_t clients_mutex = PTHREAD_MUTEX_INITIALIZER;
```

- `clients`: Array to store information about connected clients.
- `client_count`: Tracks the number of currently connected clients.
- `clients_mutex`: Mutex for thread-safe access to the clients array.

5. Networking and Concurrency

5.1 Socket Programming

The application extensively uses the POSIX socket API:

- `socket()`: Creates an endpoint for communication.
- `bind()`: Assigns a local protocol address to a socket.
- `listen()`: Marks a socket as passive, ready to accept connections.
- `accept()`: Accepts a new connection on a socket.
- `connect()`: Initiates a connection on a socket.
- `send()` and `recv()`: Transmit and receive messages on a socket.

5.2 Multi-threading

POSIX threads (pthreads) are used for concurrent client handling:

- `pthread_create()`: Creates a new thread.
- `pthread_mutex_lock()` and `pthread_mutex_unlock()`: Provide mutual exclusion for shared resources.

6. Error Handling and Robustness

The application implements comprehensive error handling:

- Socket operations are checked for failures, with appropriate error messages using `perror()`.
- Thread creation errors are handled, preventing resource leaks.
- Client disconnections are detected and managed gracefully.

7. Limitations and Potential Improvements

1. **Scalability**: The fixed `MAX_CLIENTS` limit could be replaced with a dynamic allocation system.
2. **Security**: Implement user authentication and encryption for secure communication.
3. **Features**: Add support for private messaging, multiple chat rooms, and message history.
4. **Robustness**: Implement heartbeat mechanisms to detect and handle silent disconnections.
5. **User Experience**: Develop a graphical user interface for easier interaction.
6. **Performance**: Implement non-blocking I/O or event-driven programming for improved scalability.

8. Conclusion

This chat server application demonstrates advanced usage of socket programming and multi-threading in C. It provides a solid foundation for real-time communication systems, showcasing key concepts in network programming, concurrency, and system design. The modular structure and clear separation of client and server components allow for easy extension and improvement, making it an excellent starting point for more complex chat applications.