



**AMERICAN  
UNIVERSITY<sup>OF</sup> BEIRUT**

**MAROUN SEMAAN FACULTY OF  
ENGINEERING & ARCHITECTURE**

**American University of Beirut**

**School of Engineering and Architecture**

**Department of Electrical and Computer Engineering**

**Project Title**

Design and Implementation of a Multi-threaded Chat Room with Socket-based IPC

By

**Hussam Al Basha**

(hoa12@mail.aub.edu)

**Lewaa Thebian**

(Int01@mail.aub.edu)

**A REPORT**

submitted to Dr. Khalil Hariss in fulfillment of the requirements of the project for  
the course EECE432 – Operating Systems

December 2023

# Table of Contents

<b>Introduction.....</b>	<b>3</b>
<b>Background/Theoretical Framework.....</b>	<b>4</b>
<b>Project Design and Development.....</b>	<b>7</b>
<b>Primary Implementation (C Language): .....</b>	<b>7</b>
<b>Secondary Implementation (Python Language): .....</b>	<b>10</b>
<b>Challenges and Solutions.....</b>	<b>14</b>
<b>Demonstration and Testing .....</b>	<b>15</b>
<b>Primary Implementation (C Language): .....</b>	<b>15</b>
<b>Secondary Implementation (Python Language): .....</b>	<b>16</b>
<b>Conclusion .....</b>	<b>17</b>
<b>References.....</b>	<b>18</b>
<b>Appendices.....</b>	<b>19</b>

## Introduction

In the fast-evolving landscape of modern computing, the client-server model stands as a fundamental architecture, powering a myriad of applications and services that shape our digital experiences. Our project delves into this essential architecture through the design and implementation of a chat room, encapsulating a client-server scenario. More than a mere demonstration of technical proficiency, this chat room serves as a practical application of intricate networking and programming concepts.

The primary goal of our project was to build a functional chat room within a client-server framework. Central to achieving this was the establishment of a robust communication protocol between the clients and the server, leveraging the power of socket programming as the principal Inter-Process Communication (IPC) tool. Our exploration into socket programming was crucial, providing deep insights into the intricacies of network communication and underlining the effectiveness of this method in real-world applications.

A pivotal aspect of the project was the exploration of multi-threading, particularly its implementation in a client-server context. We aimed to demonstrate how multi-threading can augment the responsiveness and efficiency of networked applications. To ensure the seamless operation and reliability of the chat room, especially under varying loads, we employed synchronization techniques to manage concurrent operations effectively.

Integrating various programming and operating system concepts was key to constructing an efficient chat room. This integration encompassed network communication, process management, and concurrent programming, each playing a vital role in shaping the project's architecture. Unlike typical chat room implementations that might focus on graphical user interfaces (GUIs), our project, written in C, concentrated on backend server-client architecture. We chose C for its direct application of POSIX libraries and functions covered in our class, capitalizing on its strengths in handling sockets, threading, and other system-level operations as part of our educational curriculum.

Python, often selected for its versatility in network and threading operations and its support for GUI development with libraries like tkinter, was not our choice. Instead, we focused on applying the material learned in class through C's capabilities, especially in systems-level programming, which provided a more hands-on experience with low-level network communication and thread management.

## **Background/Theoretical Framework**

This section provides a theoretical foundation for the chat room application developed as part of this project. The application is grounded in several key concepts of computer networking, operating systems, and programming. Understanding these concepts is crucial for comprehending the design and implementation of the project.

- Basic Networking Concepts

### **TCP/IP Model:**

The TCP/IP model is the backbone of internet communications. It is a hierarchical set of protocols that govern the exchange of data across networks. This project primarily leverages the Transport Layer (TCP) and Internet Layer (IP) for establishing connections and data transmission.

### **IP Addresses and Port Numbers:**

An IP address is a unique identifier assigned to each device on a network, facilitating its location and communication. Port numbers, on the other hand, help in identifying specific applications or processes running on a device. This project utilizes specific IP addresses and port numbers to establish socket connections between the chat server and clients.

### **TCP vs. UDP:**

TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are core protocols for data transmission. TCP, known for its reliability and ordered data transmission, was chosen for this project to ensure message integrity and proper sequencing in the chat application. UDP, while faster, lacks these reliability features and is more suited to applications where speed is prioritized over accuracy.

- Process and Inter-Process Communication (IPC)

**Definition of a Process:**

In computing, a process is an instance of a computer program that is being executed. It contains the program code and its current activity. Each process in an operating system has a unique process identifier (PID).

**Importance of IPC:**

IPC (Inter-Process Communication) allows different processes to communicate and synchronize their actions while executing concurrently. This project leverages IPC mechanisms, specifically sockets, to enable communication between the server and multiple clients.

**Methods of IPC:**

Various methods of IPC exist, such as pipes, message queues, shared memory, and sockets. Sockets, used in this project, are endpoints for sending and receiving data between processes over a network.

- Socket Programming

**Socket Definition:**

In the context of networking, a socket is an endpoint for sending or receiving data across a computer network. In this project, sockets are crucial for establishing and managing the TCP connections between the chat server and its clients.

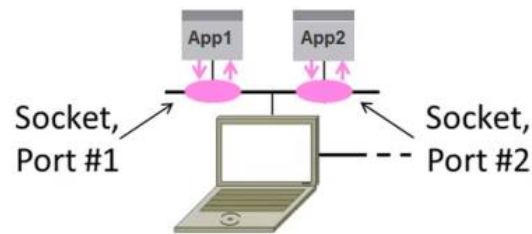
**Socket Types:**

The project uses stream sockets provided by the TCP protocol. These sockets offer a reliable, connection-oriented, and byte-stream service.

**Socket Operations:**

Key operations in socket programming include creating a socket, binding it to a specific address and port, listening for incoming connections, accepting connections, and sending/receiving data. These operations are integral to the server-client communication in the chat application.

**Sockets** let apps attach to the local network at different **ports**



- Multi-threading & Synchronization

### **Multi-threading in Applications:**

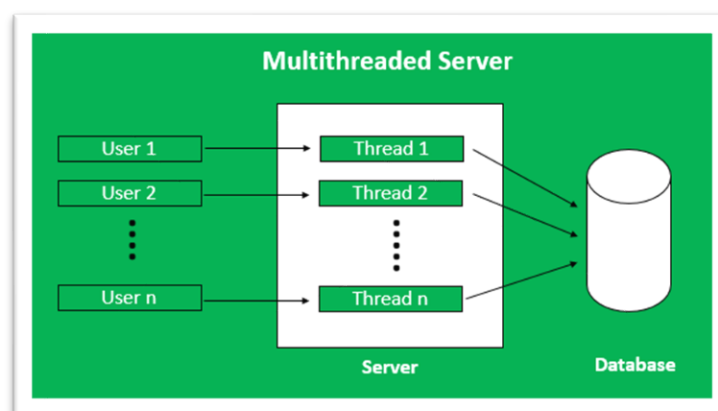
Multi-threading allows concurrent execution of multiple parts of a program to maximize resource utilization and improve application performance. In the chat room application, multi-threading enables the server to handle multiple client connections simultaneously.

### **Thread Synchronization:**

Synchronization is vital in a multi-threaded application to prevent concurrent threads from interfering with each other. This project employs thread synchronization to manage shared resources, such as the list of connected clients, ensuring consistent and error-free operations.

### **Application in Client-Server Model:**

The use of multi-threading in the server component of the chat application is a practical example of applying these concepts. It demonstrates how multiple clients can interact with the server concurrently without data corruption or loss.



# Project Design and Development

## Primary Implementation (C Language):

This section outlines the key components of the chat room project, which demonstrates a client-server architecture using C. The project is divided into two main parts: **server.c** and **client.c**, which play critical roles in the application's functionality.

## Server Architecture

The server, implemented in C (**server.c**), utilizes socket programming for network communication and threading for handling multiple client connections concurrently.

### 1. Initialization and Setup:

- The server initializes a TCP/IP socket, binding it to the specified host and port using the **socket()** and **bind()** functions. It listens, using **listen()** function, for incoming connections and maintains an array of connected client sockets.
- A pthread mutex (**clients\_mutex**) is used to manage concurrent access to this client sockets array, ensuring thread safety.

### 2. Handling Client Connections:

- The server accepts incoming connections in a loop using the **accept()** function. For each new connection, a thread is spawned using **pthread\_create()** to handle client-specific communication.
- Each thread runs the **client\_handler** function, which continuously listens for messages from its assigned client. Received messages are broadcast to all other clients using the **broadcast\_message** function.

### 3. Broadcasting Messages:

- The **broadcast\_message** function sends messages to all clients except the sender, enabling real-time communication among clients.

### 4. Client Disconnection:

- The server handles client disconnections. When a client disconnects, the corresponding socket is removed from the client sockets array and closed.

## Client Functionality

The client script (**client.c**) enables user interaction with the server, handling message sending and receiving.

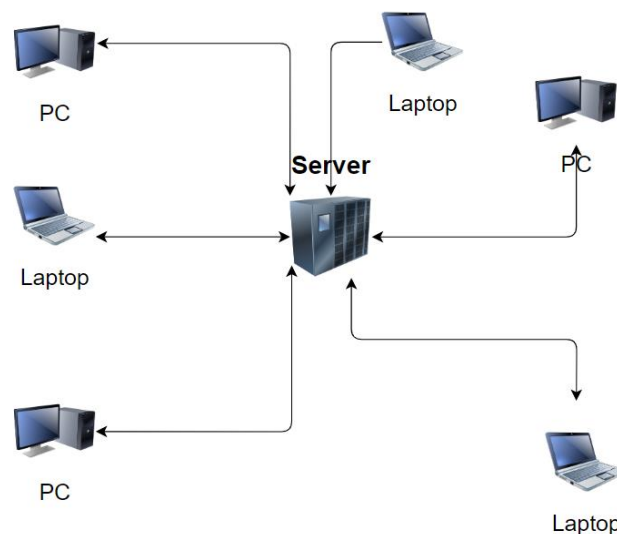
### 1. Connection Setup:

- Upon launching, the client establishes a connection with the server using the **socket()** and **connect()** functions. The client sends a nickname to the server for identification.

### 2. Receiving and Sending Messages:

- The client uses two main functions: **receive\_messages** and the main loop for sending messages.
- **receive\_messages**, running in a **separate thread**, continuously listens for incoming messages from the server and displays them.
- The main loop (**main thread**) allows the user to input messages and send them to the server using the **send()** function.

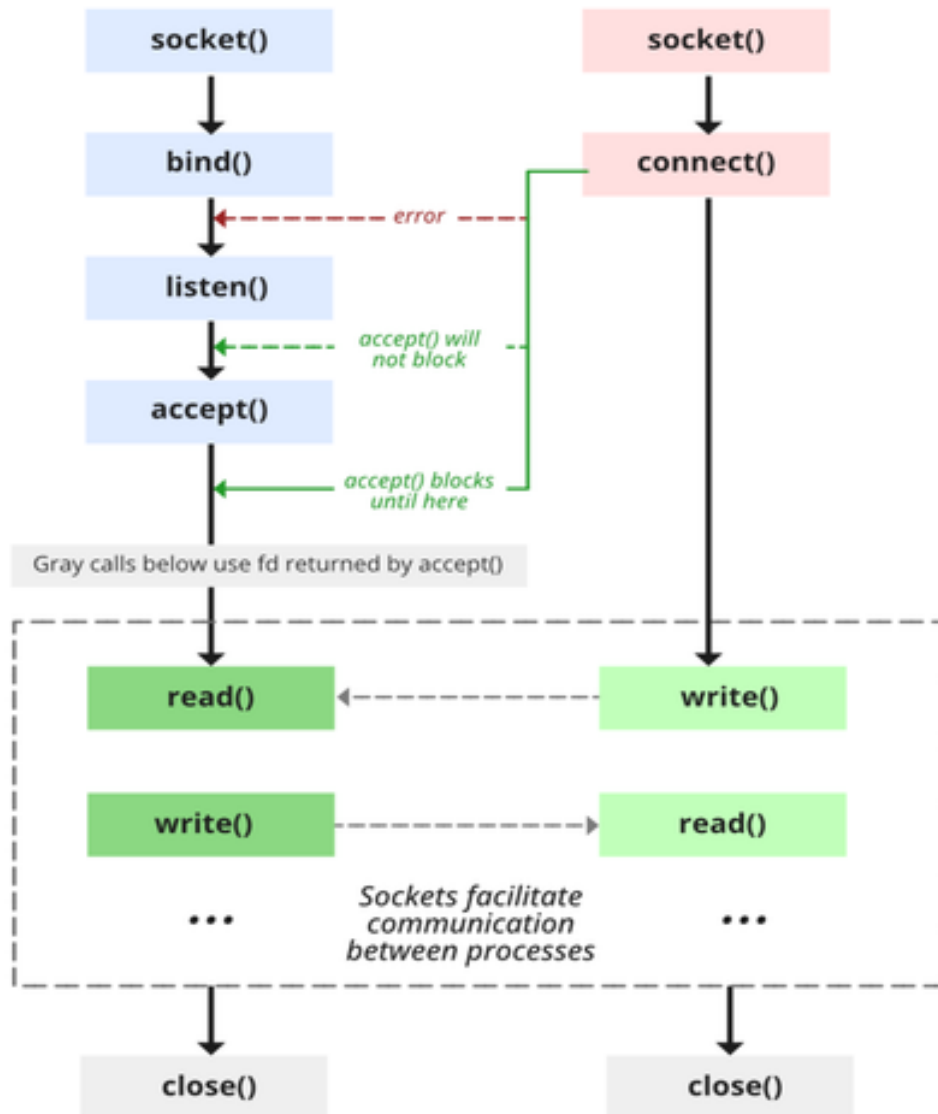
This chat room application demonstrates the use of sockets and threads in C for creating a multi-client chat server. The server can handle multiple client connections simultaneously, while the client program allows users to send and receive messages in real time.





## Server Process

## Client Process



State diagram for server and client model of Socket

## Secondary Implementation (Python Language):

This section outlines the key components of the chat room project, which demonstrates a client-server architecture using Python. The project is divided into several main scripts:

**server.py**, **client.py**, **chat\_gui.py**, & **main.py**, each play a critical role in the application's functionality.

## Server Architecture

The server acts as the central node in our chat room application. Written in Python (**server.py**), it utilizes socket programming for network communication and threading for handling multiple client connections concurrently.

### 1. Initialization and Setup:

- The **ChatServer** class initializes a TCP/IP socket, binding it to the specified host and port. It listens for incoming connections and maintains a list of connected clients and their nicknames.
- A threading lock (**self.lock**) is used to manage concurrent access to this clients list, ensuring thread safety.

### 2. Handling Client Connections:

- The server accepts incoming connections in a loop. For each new connection, a thread is spawned to handle client-specific communication (**handle\_client** method). This design allows the server to manage multiple clients simultaneously.
- The **handle\_client** method continuously listens for messages from its assigned client. Received messages are broadcast to all other clients.

### 3. Broadcasting Messages:

- The **broadcast** method is responsible for sending messages to all clients except the sender. This method is critical for chat functionality, enabling real-time communication among clients.

### 4. Client Disconnection:

- The server gracefully handles client disconnections. The **remove\_client** method removes the client from the active list and closes the socket connection.

## Client Functionality

The client script (client.py) facilitates user interaction with the server, handling message sending and receiving.

### 1. Connection Setup:

- Upon launching, the client establishes a connection with the server using sockets. It sends a nickname to the server, which is used to identify messages from this client.

### 2. Receiving and Sending Messages:

- Two main functions handle communication: **receive\_messages** and **write\_messages**.
- **receive\_messages** runs in a continuous loop, listening for incoming messages from the server and displaying them to the user.
- **write\_messages** allows the user to input messages and send them to the server.

## Graphical User Interface

The chat application features a graphical user interface (GUI) developed using Python's tkinter library.

### 1. GUI Components:

- The GUI includes a text area for displaying chat messages, an input field for typing messages, and send/exit buttons.
- User interactions, like clicking the send button or pressing the enter key, trigger events to send messages through the socket.

### 2. Integration with Backend:

- The GUI seamlessly integrates with the client's backend. It updates the chat display in real-time as messages are received and sends user-inputted messages to the server.

## Main Application Logic

**main.py** serves as the entry point for running the chat room application.

### 1. Starting the Server and Client:

- This script initiates the server in a separate thread, allowing it to run concurrently with the client.
- It then launches the client GUI, enabling user interaction with the chat room.

## Code Documentation:

Server.py: Managing Chat Room Connections and Messages

### 1. Class Initialization (`__init__` Method)

- Describe the instantiation of the server socket: **`self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`**.
- Explain setting socket options for reusability: **`self.server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)`**.
- Discuss the binding process and the significance of listening for incoming connections.

### 2. Broadcast Method (`broadcast`):

- Detail the iteration over the **`self.clients`** list within a lock context to ensure thread safety.
- For each client, except the sender, the message is sent using **`client_socket.send(message)`**.
- Include error handling for broken connections, using **`except ConnectionResetError`**.

### 3. Client Handling Method (`handle_client`):

- Explain the while loop for continuously receiving messages from a client: **`message = client_socket.recv(1024)`**.
- Discuss the use of **`self.broadcast(message, client_socket)`** to relay messages.
- Detail the handling of client disconnection and errors, and subsequent removal of the client from the server's list.

### 4. Client Removal Method (`remove_client`):

- Describe the removal process: locating the client in **self.clients**, removing it, and closing the socket.
- Emphasize the broadcast of a departure message to inform other clients.

Client.py: Facilitating User Interaction

### 1. **Receiving Messages (receive\_messages Function)**

- Describe the loop for receiving messages: **while True: message = sock.recv(1024).**
- Include how special cases like receiving 'NICK' are handled.
- Discuss error handling and socket closure on exception.

### 2. **Writing Messages (write\_messages Function)**

- Explain how messages are composed with the user's nickname and sent to the server.
- Detail the continuous loop for user input and message sending.

### 3. **Connection and Thread Initialization**

- Outline how the client socket is created and connected to the server.
- Describe the creation and starting of threads for message receiving and sending.

Chat\_gui.py: User Interface and Interaction

### 1. **GUI Setup (\_\_init\_\_ Method)**

- Discuss the initialization of the tkinter window and layout.
- Detail the creation of GUI elements like the chat area, message field, and buttons.
- Explain how the user's nickname is requested and used.

### 2. **Message Handling Methods (send\_message, receive\_messages)**

- Describe how **send\_message** retrieves text from the input field and sends it to the server.
- Discuss the **receive\_messages** method for continuously updating the GUI with incoming messages.

### 3. **Server Connection and Event Handling**

- Explain the method for connecting to the server and handling server responses.
- Detail the binding of GUI events (e.g., button clicks) to corresponding functions.

Main.py: Orchestrating the Application

### 1. **Server Thread Creation**

- Describe how **main.py** starts the server in a separate thread to allow simultaneous operation with the client GUI.
- Explain the use of **threading.Thread(target=start\_server)** and **server\_thread.start()**.

## 2. Client GUI Launch

- Discuss the instantiation of the **ChatClient** class from **chat\_gui.py** and starting the GUI loop.

## Challenges and Solutions

### Challenge 1: Managing Concurrent Client Connections

- **Solution Implemented:**

The C implementation involves creating a new thread for each client connection using **pthread\_create**. This approach allows the server to handle multiple clients concurrently. The server listens for incoming connections in a loop, and for each new connection, it spawns a new thread.

### Challenge 2: Synchronization of Shared Resources

- **Solution Implemented:**

In the C code, a mutex (**pthread\_mutex\_t clients\_mutex**) is used to synchronize access to the array **client\_sockets**, which tracks active client connections. This prevents data inconsistencies when multiple threads attempt to read from or write to the client sockets array.

### Challenge 3: Network Latency and Message Ordering

- **Solution Implemented:**

The use of TCP (implied by **SOCK\_STREAM** in the **socket()** call) ensures reliable and ordered delivery of messages. The TCP protocol handles the challenges of network latency and message reordering, ensuring that messages are delivered in the order they were sent.

### Challenge 4: Handling Client Disconnections and Server Shutdown

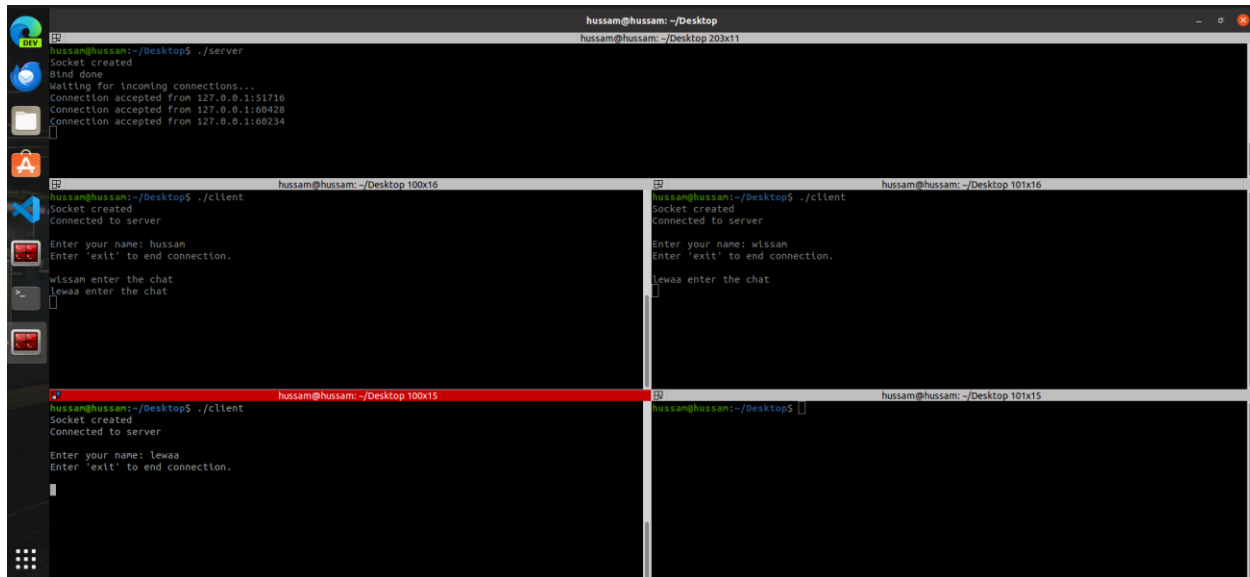
- **Solution Implemented:**

The server code handles client disconnections by checking the return value of **recv()**. If a client disconnects, the server removes the client's socket from the **client\_sockets** array and closes the socket. For server shutdowns, incorporating a controlled shutdown process where the server closes all client sockets and releases resources would be ideal, though this isn't explicitly covered in the provided code.

# Demonstration and Testing

## Primary Implementation (C Language):

Server is running and accepting clients.



The screenshot shows a Linux desktop environment with a sidebar on the left containing icons for IDEs, a terminal, and other applications. The main area displays four terminal windows. The top-left window is the server, titled 'hussam@hussam: ~/Desktop', showing the execution of './server'. It outputs 'Socket created', 'Bind done', 'Waiting for incoming connections...', and then three 'Connection accepted' messages from IP addresses 127.0.0.1:51716, 127.0.0.1:60428, and 127.0.0.1:60234. The other three windows are client terminals, each titled 'hussam@hussam: ~/Desktop' with a specific width (100x16, 101x16, and 101x15). Each client window shows the execution of './client', 'Socket created', 'Connected to server', and a prompt 'Enter your name:'. The first client (100x16) has 'hussam' entered. The second client (101x16) has 'wissam' entered. The third client (101x15) has 'lewaa' entered. All clients also show the prompt 'Enter \'exit\' to end connection.'.

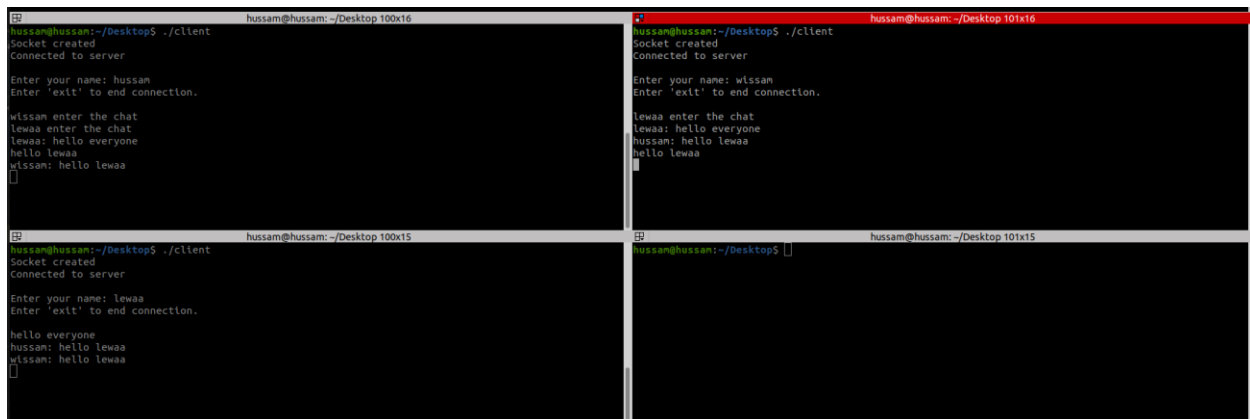
```
hussam@hussam: ~/Desktop
hussam@hussam: ~/Desktop 203x11
hussam@hussam:~/Desktop$ ./server
Socket created
Bind done
Waiting for incoming connections...
Connection accepted from 127.0.0.1:51716
Connection accepted from 127.0.0.1:60428
Connection accepted from 127.0.0.1:60234

hussam@hussam:~/Desktop 100x16
hussam@hussam:~/Desktop$ ./client
Socket created
Connected to server
Enter your name: hussam
Enter 'exit' to end connection.
wissam enter the chat
lewaa enter the chat

hussam@hussam:~/Desktop 101x16
hussam@hussam:~/Desktop$ ./client
Socket created
Connected to server
Enter your name: wissam
Enter 'exit' to end connection.
lewaa enter the chat

hussam@hussam:~/Desktop 101x15
hussam@hussam:~/Desktop$ ./client
Socket created
Connected to server
Enter your name: lewaa
Enter 'exit' to end connection.
```

Clients can chat with each other through the server, which receives messages and then broadcasts them to all connected clients, excluding the sender of each message.



This screenshot shows the same four terminal windows as the previous one, but now displaying chat history. The server window is not visible in this view. The three client windows (100x16, 101x16, and 101x15) show the chat messages received from the server. The 100x16 client shows 'wissam enter the chat', 'lewaa enter the chat', 'lewaa: hello everyone', 'hello lewaa', and 'wissam: hello lewaa'. The 101x16 client shows 'lewaa enter the chat', 'lewaa: hello everyone', 'hussam: hello lewaa', and 'hello lewaa'. The 101x15 client shows 'hello everyone', 'hussam: hello lewaa', and 'wissam: hello lewaa'.

```
hussam@hussam:~/Desktop 100x16
hussam@hussam:~/Desktop$ ./client
Socket created
Connected to server
Enter your name: hussam
Enter 'exit' to end connection.
wissam enter the chat
lewaa enter the chat
lewaa: hello everyone
hello lewaa
wissam: hello lewaa

hussam@hussam:~/Desktop 101x16
hussam@hussam:~/Desktop$ ./client
Socket created
Connected to server
Enter your name: wissam
Enter 'exit' to end connection.
lewaa enter the chat
lewaa: hello everyone
hussam: hello lewaa
hello lewaa

hussam@hussam:~/Desktop 101x15
hussam@hussam:~/Desktop$ ./client
Socket created
Connected to server
Enter your name: lewaa
Enter 'exit' to end connection.
hello everyone
hussam: hello lewaa
wissam: hello lewaa
```

When a client, such as 'Lewaa', leaves the chat room, all remaining clients are notified about the departure, and the application continues to function normally, accommodating this change. Furthermore, when a new client, for example, 'Ahmad', joins the chat room, all existing clients are promptly notified about the new participant's arrival.

```
hussam@hussam: ~/Desktop$ ./client
Socket created
Connected to server

Enter your name: hussam
Enter 'exit' to end connection.

wissam enter the chat
lewaa enter the chat
lewaa: hello everyone
hello lewaa
wissam: hello lewaa
I am going to leave
bye
exit
hussam@hussam: ~/Desktop$

hussam@hussam: ~/Desktop$ ./client
Socket created
Connected to server

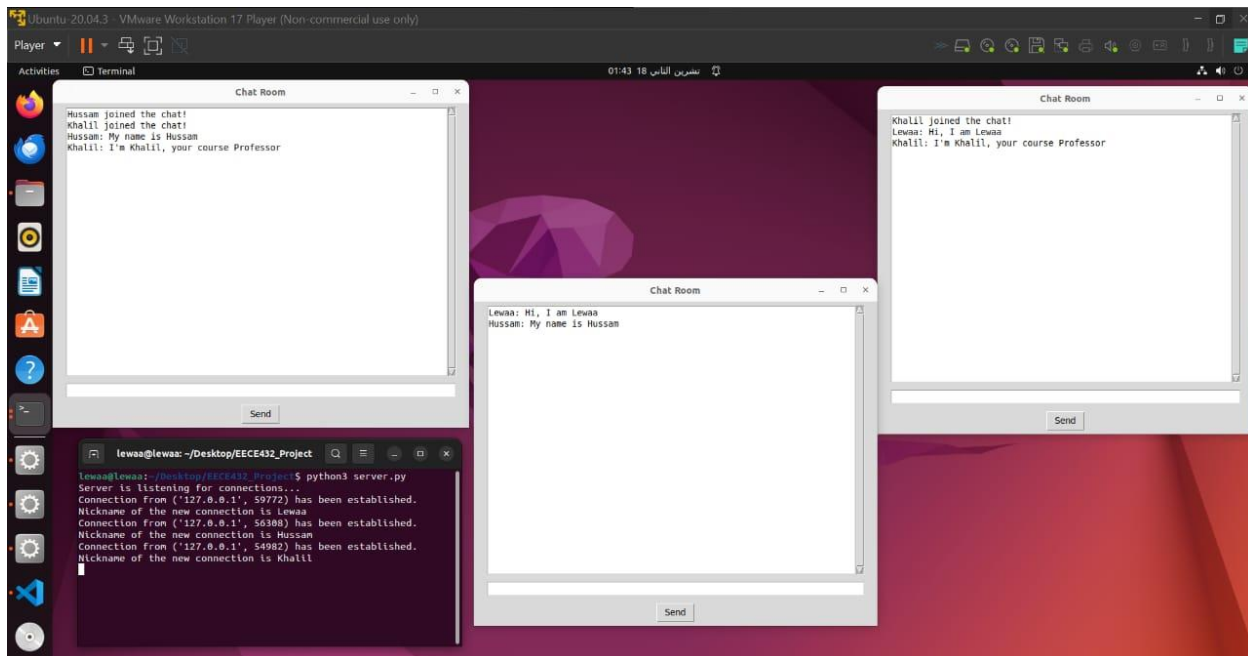
Enter your name: lewaa
Enter 'exit' to end connection.

hello everyone
hussam: hello lewaa
wissam: hello lewaa
hussam: I am going to leave
hussam: bye
hussam left the chat
Ahmad enter the chat
hussam@hussam: ~/Desktop$

hussam@hussam: ~/Desktop$ ./client
Socket created
Connected to server

Enter your name: Ahmad
Enter 'exit' to end connection.
```

## Secondary Implementation (Python Language):





## Conclusion

Reflecting on the accomplishments of this project, which entailed designing and implementing a chat room application using a client-server model, it's evident that the primary objectives were successfully met. This project served as an immersive exploration into key computer science concepts, particularly in the realms of networking and multi-threading, and it provided a practical application of the material covered in our coursework.

### Key Achievements:

#### 1. Effective Server-Client Communication:

- The fundamental triumph of this project was the creation of a robust chat room application. It adeptly manages multiple client connections via a central server and facilitates real-time message exchange. This was achieved through the adept use of socket programming in C, demonstrating a solid grasp of network communication principles.

#### 2. Technical Mastery Demonstrated:

- The project showcased the efficient implementation of a multi-threaded server in C, handling concurrent client connections. This not only emphasized the power of threading in C but also highlighted the effectiveness of synchronization techniques, such as mutexes, to ensure data integrity and thread safety.

### Learning and Development:

#### 1. Practical Application of Theoretical Knowledge:

- This venture provided a valuable opportunity to apply theoretical concepts, like TCP/IP networking and IPC mechanisms, particularly sockets, in a tangible setting. It also offered insights into the intricacies of multi-threading and its challenges in real-world applications.

#### 2. Problem-Solving and Technical Skill Enhancement:

- Addressing various technical challenges, especially in managing concurrent operations in a networked environment, honed problem-solving skills. The project significantly bolstered programming abilities, particularly in C, enhancing understanding and proficiency in network programming and system-level design.

### Future Directions and Improvements:

#### 1. Expanding Features and Scalability:

- Future enhancements could include implementing additional functionalities such as file transfers or direct messaging. Improving the server's ability to handle a larger number of simultaneous connections could also be a valuable direction.

#### 2. Security and Robustness:

- Incorporating advanced security features, perhaps through secure socket layers (SSL), would further strengthen the application. Enhancements in error handling and robustness, especially in network communication, would also be beneficial.

## References

- <https://www.geeksforgeeks.org/socket-programming-cc/>
- [https://imgs.search.brave.com/ccJHoumyBEIHpWOX88sgKTXdxS9y\\_MlogpLYaMxjqI/rs:fit:860:0:0/g:ce/aHR0cHM6Ly93d3cu/Ym9nb3RvYm9nby5j/b20vY3BsdXNwbHVz/L2ltYWdlcy9zb2Nr/ZXQvc29ja2V0X3Bv/cnQucG5n](https://imgs.search.brave.com/ccJHoumyBEIHpWOX88sgKTXdxS9y_MlogpLYaMxjqI/rs:fit:860:0:0/g:ce/aHR0cHM6Ly93d3cu/Ym9nb3RvYm9nby5j/b20vY3BsdXNwbHVz/L2ltYWdlcy9zb2Nr/ZXQvc29ja2V0X3Bv/cnQucG5n)
- <https://imgs.search.brave.com/psX4J4VfzAk8Pqvqhs6z8PC2ysyXavGsLNymWOO753w/rs:fit:860:0:0/g:ce/aHR0cHM6Ly9tZWRp/YS5nZWVrc2Zvcmdl/ZWtzLm9yZy93cC1j/b250ZW50L3VwbG9h/ZHMvMjAyMDEwMzEx/MTAzMTEvTXVsdGlz/ZXJ2ZXIucG5n>
- <https://app.diagrams.net/>
- [Slides of the course](#)

# Appendices

## Server.c Code

```
//This header file is part of the C Standard Library. It provides functionalities for input and output operations.
//ex: printf()
#include <stdio.h>

//This header file includes functions involving memory allocation, process control, conversions, and others.
//ex: exit()
#include <stdlib.h>

//This header file provides access to the POSIX operating system API
#include <unistd.h>

//This header file is part of the C Standard Library and provides functions for dealing with strings and arrays
of characters.
// ex: strcpy, strcat, strlen, strcmp, strchr
#include <string.h>

//This header file includes definitions of structures needed for sockets and functions for socket API,
//which is used for network communication.
#include <sys/socket.h>

//This header file contains constants and structures needed for internet domain addresses.
// ex: struct sockaddr_in
#include <netinet/in.h>

//This header file contains functions for internet operations.
#include <arpa/inet.h>

//This header file is used for working with POSIX threads
#include <pthread.h>

//#include <sys/socket.h> include:
// socket(): For creating a new socket.
// bind(): For binding a socket to an address.
// listen(): For listening for connections on a socket.
// accept(): For accepting a connection on a socket.
// connect(): For initiating a connection on a socket.
// send(), recv(): For sending and receiving data on a socket.
// setsockopt(), getsockopt(): For setting and getting options on sockets.
// Structures like sockaddr, sockaddr_in for handling socket addresses.

#define PORT 8080
```

```
#define MAX_CLIENTS 100
#define BUFFER_SIZE 1024

int client_sockets[MAX_CLIENTS];
pthread_mutex_t clients_mutex = PTHREAD_MUTEX_INITIALIZER;

void broadcast_message(int sender, const char *message) {
    pthread_mutex_lock(&clients_mutex);
    for (int i = 0; i < MAX_CLIENTS; i++) {
        if (client_sockets[i] != 0 && client_sockets[i] != sender) {
            send(client_sockets[i], message, strlen(message), 0);
        }
    }
    pthread_mutex_unlock(&clients_mutex);
}

void *client_handler(void *socket_desc) {
    int sock = *(int*)socket_desc;
    char buffer[BUFFER_SIZE];
    ssize_t read_size;

    while ((read_size = recv(sock, buffer, BUFFER_SIZE - 1, 0)) > 0) {
        buffer[read_size] = '\0';
        broadcast_message(sock, buffer);
    }

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

    pthread_mutex_lock(&clients_mutex);
    for (int i = 0; i < MAX_CLIENTS; i++) {
        if (client_sockets[i] == sock) {
            client_sockets[i] = 0;
            break;
        }
    }
    pthread_mutex_unlock(&clients_mutex);

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

```

}

int main() {
    int server_fd, new_socket, *new_sock;
    struct sockaddr_in server, client;
    socklen_t client_size = sizeof(struct sockaddr_in);

    // Create Server socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == -1) {
        perror("Could not create socket");
        return 1;
    }
    puts("Socket created");

    // Set up server address
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_family = AF_INET;
    server.sin_port = htons(PORT);

    // Bind the socket with IP
    if (bind(server_fd, (struct sockaddr *)&server, sizeof(server)) < 0) {
        perror("bind failed");
        return 1;
    }
    puts("Bind done");

    // Start Listening
    // 3 is the maximum length of the queue for pending connections
    listen(server_fd, 3);
    puts("Waiting for incoming connections...");

    // The loop continues accepting new connections as long as accept
    // successfully returns a new socket descriptor for each incoming connection.
    while ((new_socket = accept(server_fd, (struct sockaddr *)&client, &client_size))) {
        printf("Connection accepted from %s:%d\n", inet_ntoa(client.sin_addr), ntohs(client.sin_port));

        pthread_t thread_id;
        new_sock = malloc(sizeof(int));
        *new_sock = new_socket;

        // Updating the Client Sockets Array

```

```

// The mutex clients_mutex is used to lock the critical section where the client_sockets array is updated.
// This prevents race conditions when multiple threads try to update the array simultaneously.
pthread_mutex_lock(&clients_mutex);
for (int i = 0; i < MAX_CLIENTS; i++) {

    //The new socket is added to the first empty slot in the client_sockets array
    if (client_sockets[i] == 0) {
        client_sockets[i] = new_socket;
        break;
    }
}
pthread_mutex_unlock(&clients_mutex);

// For each new connection, a new thread is created
if (pthread_create(&thread_id, NULL, client_handler, (void*) new_sock) < 0) {
    perror("could not create thread");
    return 1;
}
}

if (new_socket < 0) {
    perror("accept failed");
    return 1;
}

return 0;
}

```

## Client.c Code

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>

#define PORT 8080
#define BUFFER_SIZE 1024

void *receive_messages(void *socket_desc) {
    int sock = *(int*)socket_desc;

```

```

char buffer[BUFFER_SIZE];
ssize_t read_size;

while ((read_size = recv(sock, buffer, BUFFER_SIZE - 1, 0)) > 0) {
    buffer[read_size] = '\0';
    printf("%s", buffer);
}

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

return 0;
}

int main() {
    int sock;
    struct sockaddr_in server;
    char message[BUFFER_SIZE], name[30], formatted_message[BUFFER_SIZE+31];
    pthread_t receive_thread;

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

    // Set up server address
    server.sin_addr.s_addr = inet_addr("127.0.0.1"); // Server IP address
    server.sin_family = AF_INET;
    server.sin_port = htons(PORT);

    // Connect to server
    if (connect(sock, (struct sockaddr *)&server, sizeof(server)) < 0) {
        perror("Connect failed. Error");
        return 1;
    }
    puts("Connected to server\n");

    // Prompt for user name

```

```

printf("Enter your name: ");
fgets(name, 30, stdin);
name[strcspn(name, "\n")] = 0; // Remove newline character

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

// Send message to server so the server tell everyone that the client joined the chat
printf("Enter 'exit' to end connection.\n\n");
char m1[100];
strcpy(m1, name);
strcat(m1, " enter the chat\n");
if (send(sock, m1, strlen(m1), 0) < 0) {
    puts("Send failed");
    return 1;
}

// Loop: Send messages to server
while (1) {
    //printf("Enter message: ");
    fgets(message, BUFFER_SIZE, stdin);

    if (strncmp(message, "exit", 4) == 0) {
        char m[100];
        strcpy(m, name);
        strcat(m, " left the chat\n");
        if (send(sock, m, strlen(m), 0) < 0) {
            puts("Send failed");
            return 1;
        }
        break;
    }

    // Format message to include name
    snprintf(formatted_message, BUFFER_SIZE+31, "%s: %s", name, message);

    // Send formatted message to server
    if (send(sock, formatted_message, strlen(formatted_message), 0) < 0) {
        puts("Send failed");
        return 1;
    }
}

```



```
// Clean up  
close(sock);  
return 0;  
}
```