Aim:

To implement a simple client-server chat application in C++ using socket programming, where the server can handle multiple clients simultaneously, and clients can send and receive messages from the server.

Theory:

In computer networking, a client-server architecture is a model in which multiple clients request and receive services from a centralized server. The server provides a resource or service, and the clients initiate communication with the server to use that service.

A chat application is an example of a client-server model where the server listens for incoming connections from clients. The clients connect to the server, and once the connection is established, they can exchange messages. The server typically manages multiple client connections simultaneously, forwarding messages received from one client to others.

This lab involves creating a simple chat application using C++ with sockets. The server is designed to handle multiple clients concurrently using multi-threading, allowing real-time communication between clients.

How the Client-Server Chat Application Works:

1. **Server Setup:**
    - The server creates a socket and binds it to a specific port.
    - It listens for incoming client connections and accepts them.
    - For each client connection, the server spawns a new thread to handle communication, enabling multiple clients to connect simultaneously.
2. **Client Setup:**
    - The client creates a socket and attempts to connect to the server using the server's IP address and port.
    - Upon a successful connection, the client can send and receive messages from the server.
3. **Communication:**
    - Both server and client use sockets to send and receive messages.
    - Messages are transmitted in a continuous loop, allowing real-time chat functionality.
    - The server can broadcast messages to all connected clients, ensuring that every client can see messages from others.

Example Scenario:

1. The server is started and begins listening on a designated port (e.g., 8080).
2. Client A connects to the server and sends a message: "Hello Server!".
3. Client B connects to the server and sends a message: "Hi there!".
4. The server, running in a multi-threaded mode, receives both messages and broadcasts them to all connected clients.
5. Client A sees: "Client B: Hi there!"
6. Client B sees: "Client A: Hello Server!"

This process illustrates how the server maintains communication with multiple clients and distributes messages among them.

Significance:

Implementing a client-server chat application is essential for understanding network communication, socket programming, and concurrency in C++. It provides a foundation for building more complex distributed systems and services, such as online games, collaborative applications, or real-time data exchange systems.

Function and Modules Used:

- **`std::thread`:** Allows the server to handle multiple clients concurrently by spawning new threads for each client connection.
- **`std::mutex`:** Ensures thread-safe access to shared resources, such as the list of connected clients.
- **`std::vector<std::thread>`:** Keeps track of all active threads handling client connections.
- **`std::string` and `char` arrays:** Used for storing and transmitting messages between clients and the server.
- **`socket` API functions (`socket()`, `bind()`, `listen()`, `accept()`, `connect()`, `send()`, `recv()`):** Used to set up, manage, and communicate over network connections.

Output:

When the application is executed:

1. The server prints a message indicating that it is waiting for client connections.
2. Upon a client connection, the server logs the connection details (e.g., client IP and port).
3. As clients send messages, the server broadcasts them to all other connected clients.
4. Clients display the messages received from other clients in the chat.

Analysis:

The implemented client-server chat application effectively demonstrates real-time communication between multiple clients over a network. The server's multi-threaded design allows it to handle multiple clients concurrently, enabling efficient message exchange.

By using sockets for communication and threads for concurrency, the application ensures that each client can send and receive messages independently. The use of mutexes prevents race conditions when accessing shared resources, such as the list of connected clients.

This implementation also highlights the challenges of network programming, such as handling multiple connections, synchronizing data across threads, and managing socket communication. The application serves as a practical example of how to build a simple distributed system with a focus on concurrency and network communication.

Conclusion:

The successful implementation of a client-server chat application in C++ demonstrates the principles of network communication and concurrent programming. The application shows how sockets can be used to establish a network connection between clients and a server and how multi-threading enables handling multiple clients simultaneously.

This project underscores the importance of synchronization mechanisms, such as mutexes, in a multi-threaded environment, ensuring that shared resources are accessed safely. The knowledge gained from this lab can be applied to more complex networked applications, such as multiplayer games, online collaboration tools, or distributed computing platforms.