Aim:

To implement a multi-client chat application in C++ using socket programming, which allows clients to communicate with each other and share files via a centralized server.

Theory:

A multi-client chat application allows multiple clients to connect to a server and exchange messages in real-time. This lab extends the traditional client-server chat model by introducing file-sharing capabilities, enabling clients to transfer files to the server, which can then be accessed or downloaded by other clients.

File sharing over a network involves transmitting file data from one client to another through a server. The server acts as an intermediary, receiving the file from a sender and storing it temporarily before allowing other clients to access or download it. This functionality is achieved using socket programming in C++, where the server manages multiple client connections and handles both text and file data.

How the Multi-Client Chat Application with File Sharing Works:

1. **Server Setup:**
    - The server creates a socket, binds it to a specific port (e.g., port 5000), and listens for incoming connections from clients.
    - For every new client connection, the server spawns a new thread to handle communication with that client, allowing multiple clients to connect concurrently.
    - The server is responsible for broadcasting messages to all connected clients and handling file transfer requests.
2. **Client Setup:**
    - Each client creates a socket and connects to the server using its IP address and port.
    - Clients can send messages to the server, which are then broadcast to all other connected clients.
    - Clients can also send files to the server, which are stored and can be shared with other clients.
3. **File Sharing Mechanism:**
    - When a client wants to share a file, it sends a special command (`/file <filename>`) to the server.
    - The server receives the command and prepares to receive the file data from the client.
    - The client reads the file in binary mode and sends its contents to the server.
    - The server receives the file data, writes it to a designated directory (e.g., `uploads/`), and logs the event.
    - Other clients can access or download the file from the server upon request.

Highlighted Feature: File Sharing

- **File Transfer Initiation:**
  - A client initiates a file transfer by entering the command `/file <filename>`.
  - The server recognizes the command and prepares to receive the file data.
- **Receiving and Storing Files:**
  - The server stores the file in a predefined directory (`uploads/`) and ensures the file is written in binary mode to preserve its content.
- **File Logging:**
  - The server logs all file transfer events, including the file name and status, to a log file (`chat_log.txt`).

Example Scenario:

1. **Client A** connects to the server and sends a message: "Hello everyone!".
2. **Client B** connects and requests to send a file by entering the command `/file example.txt`.
3. **Client B** uploads `example.txt` to the server.
4. The server stores the file in the `uploads/` directory and logs the event.
5. **Client C** connects and can download `example.txt` from the server.
6. All clients receive a broadcast message notifying them about the new file available on the server.

This scenario demonstrates the multi-client chat application with real-time messaging and file-sharing functionality.

Significance:

Implementing a multi-client chat application with file sharing in C++ illustrates important concepts in network programming, such as:

- Socket communication for message exchange.
- Multi-threading to handle multiple clients concurrently.
- File handling and data transfer over a network.
- Synchronization techniques to ensure thread-safe operations.

The application provides a robust framework for building real-time communication platforms where users can chat and share files efficiently.

Functions and Modules Used:

- **`std::thread`:** Handles multiple client connections by creating a new thread for each client.

- **`std::mutex`:** Synchronizes access to shared resources, such as the client list and log file.
- **`std::unordered_map`:** Stores client information, including socket descriptors.
- **`socket` API functions (`socket()`, `bind()`, `listen()`, `accept()`, `connect()`, `send()`, `recv()`):** Used for network communication between clients and the server.
- **`std::ifstream` and `std::ofstream`:** Handles file reading and writing operations for file transfer.
- **`FILE` pointers (`fopen()`, `fwrite()`, `fclose()`):** Used for handling binary file transfer to ensure data integrity.

Output:

When the application is executed:

1. The server logs that it is listening for connections on port 5000.
2. Clients connect and can send messages, which are broadcast to all other clients.
3. A client sends a file, and the server logs the file transfer event.
4. Other clients are notified of the new file, and they can request to download it.

Analysis:

The multi-client chat application with file sharing provides a practical demonstration of network communication and concurrent programming in C++. It effectively shows how to handle multiple clients simultaneously while also enabling file transfer functionality.

The implementation highlights several critical challenges, such as ensuring data integrity during file transfer, managing concurrent access to shared resources, and securely storing files on the server. By integrating file sharing into a chat application, this lab also emphasizes the importance of efficient data handling in distributed systems.

Conclusion:

The implementation of a multi-client chat application with file sharing in C++ successfully demonstrates the principles of network communication, concurrency, and file handling. The application allows clients to communicate in real-time and share files efficiently through a centralized server.

This project underscores the importance of synchronization mechanisms (such as mutexes) in multi-threaded environments and shows how to handle data transmission over a network. The skills learned from this lab can be applied to more complex applications, such as collaborative platforms, cloud storage solutions, or real-time communication systems.