**System Architecture:**

The system we propose to achieve a networked distributed system for a group-based client-server communication is based on Two main components: **Threads & Sockets.**

Clients connect to the server, which coordinates communication among them. The server maintains a list of active clients and coordinates tasks such as message broadcasting, private messaging, and coordinator management.

**Threads :**

Each connected client is managed by a separate thread, enabling concurrent communication and message handling. This prevents the server from blocking while waiting for a single client's response.

Concurrency is essential for handling multiple client connections simultaneously. Threads are employed to manage concurrent communication between the server and clients. Each client connection is handled by a separate ClientThread, allowing for parallel message processing and uninterrupted communication. A ServerListener thread is also utilized to continuously listen for incoming messages from the server in the client application.

**Sockets:**

Sockets provide a communication channel between clients and the server. The server uses a ServerSocket to accept connections, and each ClientThread uses a Socket to communicate with its assigned client.

**System Overview:**

The system comprises three main components:

**Clients:** Represent individual members in the group.

**Server:** Acts as a central hub for message routing and member management.

**Coordinator:** Manages member list updates and responds to member queries.

**Design Patterns:**

**We Implemented many Design Patterns to achieve our system among which:**

**Singleton Pattern:**

The Server class employs the Singleton pattern to ensure that only one instance of the server is created throughout the application's lifecycle. This is achieved by having a private constructor and providing a static method (StartServer()) to instantiate the server if it does not already exist.

**Observer Pattern:**

Each client thread (ClientThread) acts as an observer, continuously listening for messages from the server. The server notifies observers (clients) about events or messages by broadcasting them to all connected clients.

**Thread Pool Pattern:**

The ExecutorService interface in Java represents a thread pool, which is a form of the Thread Pool pattern. We use it to manage a pool of threads for handling client connections and server listening. This improves performance by reusing threads instead of creating new ones for each task.

**3. System Functionalities:**

**1. Functionality Overview**

**Client-Server Communication:** Clients connect to a server, establish usernames, and exchange messages.

**Public and Private Messaging:** Clients can send messages to all participants (public) or specific users (private).

Active User Management: The server tracks connected users and provides a way to list them.

**Coordinator Role:** The first connected user becomes the coordinator, responsible for specific tasks.

**Inactivity Detection:** Both clients and the coordinator are subject to inactivity checks to prevent idle connections.

**2. Client-Side Functionality (client.java):**

**Connection and Login:**

* Connects to the server (specified IP and port or default).
* Prompts the user for a unique username and sends it for validation.
* Receives confirmation from the server to proceed.

**Message Sending and Receiving:**

* Employs a PrintWriter to send messages to the server.
* Uses a BufferedReader to read messages received from the server.

**Message Processing:**

* Identifies private messages with "@" prefix, extracts recipient name and message content.

**Supports commands:**

* /active: checks for active users on the server.
* /quit: Disconnects from the server and terminates the client.
* /yes: Acknowledges activity to the server as a response to /active.
* Relays other messages to the server as public messages (broadcast).

**3. Server-Side Functionality (server.java):**

**Server Setup:**

* Listens for client connections on a specified port (default 8080).
* Creates a ServerSocket for accepting connections.

**Client Management:**

**For each new connection:**

* Creates a separate ClientThread to handle client communication.
* Adds the thread to a list of connected clients (CLIENT\_THREADS).
* Starts the ClientThread to handle message exchange and commands.

**User Management:**

* Maintains a list of connected user names (USER\_NAMES).
* Handles username uniqueness during client registration.

**Communication Handling:**

* Broadcasts messages (except from the sender) to all connected clients.
* Facilitates private messaging by finding the recipient and sending the message.

**Coordinator Role:**

* Assigns the coordinator role to the first connected client (coordinator).
* Uses a scheduled executor (COORDINATOR\_TIMER) to check for coordinator activity (60s) and disconnect them if they are inactive.
* Broadcasts a message and disconnects inactive coordinators.
* Automatically assigns the coordinator role to another active user if needed Without Interrupting the communication of other users.

**Inactivity Detection:**

* Uses a scheduled executor (disconnectTimer) in each ClientThread to monitor inactivity when the coordinator asks for it (/active).
* Monitor Coordinator Activity constantly (60s).
* Broadcasts a disconnection message and removes inactive clients.