**Title**

**Real-time Chat Application Using Java Sockets and Multithreading**

## Problem Statement

In modern digital communication, real-time chat applications are essential for seamless user interaction. The challenge is to create a simple, efficient chat system that can handle multiple users simultaneously while ensuring low latency and real-time message exchange.

This case study aims to design and implement a **text-based chat application** where:

* Multiple clients can **send and receive messages** concurrently.
* The communication is **real-time and bi-directional**.
* The system utilizes **Sockets** for networking and **Multithreading** for concurrent communication.

## Concepts and Technologies Used

| **Concept/Technology** | **Description** |
| --- | --- |
| **Socket Programming** | Used for establishing connection between server and clients |
| **ServerSocket** | Listens for incoming client connections |
| **Multithreading (Thread/Runnable)** | Allows multiple clients to communicate simultaneously |
| **Java I/O Streams** | Used for message input/output between clients and server |
| **Synchronized Collections** | Ensures thread-safe operations on shared data (client list) |

## System Design and Architecture

### Server

* Listens for client connections on a specific port.
* For each connected client, starts a new **thread** (ClientHandler) to handle communication.
* Maintains a **list of all connected clients** and broadcasts received messages to them.

### Client

* Connects to the server using the IP address and port.
* Has two separate threads:
  + One for sending messages to the server.
  + One for receiving messages from the server.

## Implementation

### Server Code (ChatServer.java)

import java.io.\*;

import java.net.\*;

import java.util.\*;

public class ChatServer {

private static final int PORT = 1234;

private static Set<Socket> clientSockets = Collections.synchronizedSet(new HashSet<>());

public static void main(String[] args) {

try (ServerSocket serverSocket = new ServerSocket(PORT)) {

System.out.println("Server started. Waiting for clients...");

while (true) {

Socket clientSocket = serverSocket.accept();

clientSockets.add(clientSocket);

System.out.println("Client connected: " + clientSocket);

new ClientHandler(clientSocket).start();

}

} catch (IOException e) {

e.printStackTrace();

}

}

static class ClientHandler extends Thread {

private Socket socket;

private BufferedReader input;

public ClientHandler(Socket socket) {

this.socket = socket;

try {

input = new BufferedReader(new InputStreamReader(socket.getInputStream()));

} catch (IOException e) {

e.printStackTrace();

}

}

public void run() {

String msg;

try {

while ((msg = input.readLine()) != null) {

broadcastMessage(msg, socket);

}

} catch (IOException e) {

System.out.println("Client disconnected: " + socket);

} finally {

try {

clientSockets.remove(socket);

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

private void broadcastMessage(String msg, Socket senderSocket) {

synchronized (clientSockets) {

for (Socket s : clientSockets) {

if (s != senderSocket) {

try {

PrintWriter out = new PrintWriter(s.getOutputStream(), true);

out.println(msg);

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

}

}

}

### Client Code (ChatClient.java)

import java.io.\*;

import java.net.\*;

public class ChatClient {

private static final String SERVER\_IP = "localhost";

private static final int SERVER\_PORT = 1234;

public static void main(String[] args) {

try (Socket socket = new Socket(SERVER\_IP, SERVER\_PORT);

BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

BufferedReader serverIn = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {

System.out.println("Connected to chat server.");

// Sending thread

Thread sendThread = new Thread(() -> {

try {

String msg;

while ((msg = input.readLine()) != null) {

out.println(msg);

}

} catch (IOException e) {

e.printStackTrace();

}

});

// Receiving thread

Thread receiveThread = new Thread(() -> {

try {

String serverMsg;

while ((serverMsg = serverIn.readLine()) != null) {

System.out.println("Message: " + serverMsg);

}

} catch (IOException e) {

e.printStackTrace();

}

});

sendThread.start();

receiveThread.start();

sendThread.join();

receiveThread.join();

} catch (Exception e) {

e.printStackTrace();

}

}

}

Output

### Server Console

Server started. Waiting for clients...

Client connected: Socket[addr=/127.0.0.1,port=50222,localport=1234]

Client connected: Socket[addr=/127.0.0.1,port=50224,localport=1234]

### Client 1

Connected to chat server.

Hello from Client 1!

Message: Hi Client 1, this is Client 2!

### Client 2

Connected to chat server.

Message: Hello from Client 1!

Hi Client 1, this is Client 2!

## Conclusion

This case study demonstrates how a simple yet effective real-time chat application can be built using **Java Socket programming and Multithreading**. By using a dedicated thread for each client, the server ensures concurrent communication. This approach is scalable and forms the foundation of many real-world messaging systems.