Lab 9: Client/Server Communication using Sockets and Multithreading

Sockets are endpoints for two-way communication between a client and a server. They enable data transfer over a network, allowing clients and servers to interact in real-time. Multithreading allows multiple threads to run concurrently within a single application. When sockets and multithreading are combined in a client-server (C/S) application, they enable efficient, concurrent communication.

In this tutorial, we will create a simple C/S broadcasting application. Clients connect to the server to start chat, and the server will broadcast the message to all the connected clients.

The Server

The `ChatServer` accepts connections from multiple clients. Each client connection spawns a new `ClientHandler` thread, managing communication with that client. Because of the broadcasting feature, the server should maintain all the connected clients. Since clients come and go in different threads, we need to synchronize access to the set of connected clients. We use a **concurrent HashSet** for this purpose.

```
public class ChatServer {
    public static void main(String[] args) {
        Set<ClientHandler> clients = ConcurrentHashMap.newKeySet();

        try (ServerSocket serverSocket = new ServerSocket(1234)) {
            System.out.println("Server started ......");

            while (true) {
                Socket socket = serverSocket.accept();
                System.out.println("New client connected.");

                ClientHandler clientHandler = new ClientHandler(socket, clients);
                clients.add(clientHandler);
                new Thread(clientHandler).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

`ClientHandler` listens for messages from a single client and broadcasts them to all other connected clients. It also handles client disconnection by closing the socket and removing the client from the set. To distinguish messages from different clients, the client should send his/her name first.

```
class ClientHandler implements Runnable {
    private final Socket socket;
    private final BufferedReader in;
    private final PrintWriter out;
    private final Set<ClientHandler> clients;
    private String clientName;
```

```
    public ClientHandler(Socket socket, Set<ClientHandler> clients) throws
IOException {
        this.socket = socket;
        this.clients = clients;
        this.in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        this.out = new PrintWriter(socket.getOutputStream(), true);
    }

    public void sendMessage(String message) {
        out.println(message);
    }

    private void broadcast(String message, ClientHandler sender) {
        for (ClientHandler client : clients) {
            if (client != sender) {
                client.sendMessage(message);
            }
        }
    }

    @Override
    public void run() {
        try {
            clientName = in.readLine();
            broadcast(clientName + " joined the chat!", this);

            String message;
            while ((message = in.readLine()) != null) {
                broadcast(clientName + ": " + message, this);
            }
        } catch (IOException e) {
            System.out.println(clientName + " disconnected");
        } finally {
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
            clients.remove(this);
            broadcast(clientName + " left the chat!", this);
        }
    }
}
```

The Client

`ChatClient` connects to the server, gets user name, then handles message sending and receiving concurrently

```
public class ChatClient {
    public static void main(String[] args) {
        try {
            Socket socket = new Socket("localhost", 1234);
            System.out.println("Connected to server");

            System.out.print("Enter your name: ");
            Scanner scanner = new Scanner(System.in);
            String name = scanner.nextLine();

            // separate threads for read and write
            // because read doesn't need to wait for write
            new Thread(new MessageSender(socket, name)).start();
            new Thread(new MessageReceiver(socket)).start();

        } catch (IOException e) {
            System.out.println("Disconnected from server");
        }
    }
}
```

We used two separate threads for reading and writing. If we use a single thread for read and write, the client will be blocked when reading input from the server, as there might be no message being received. Yet, in a chat application, users should be able to send messages at any time.

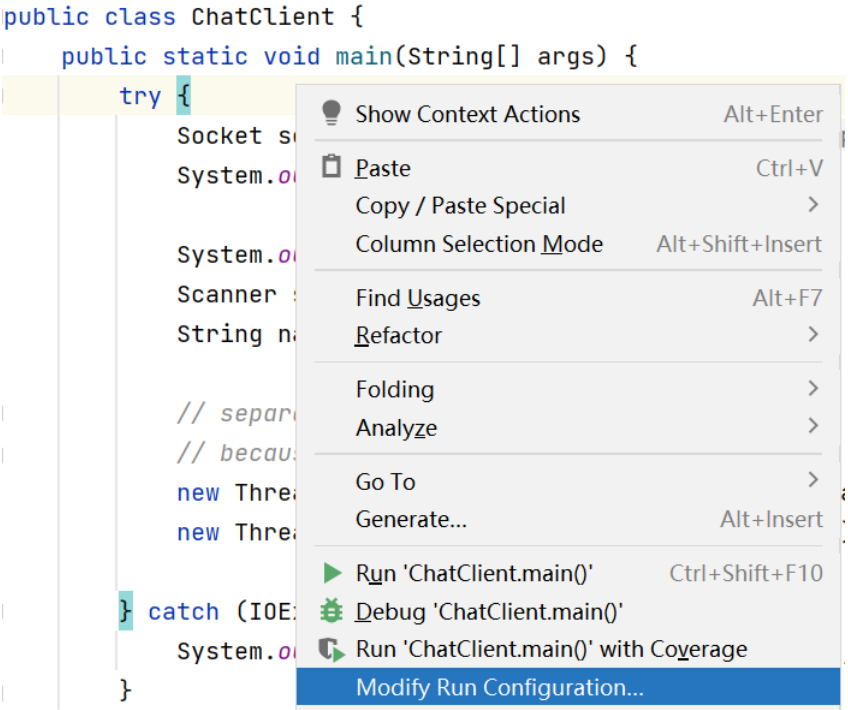
```
class MessageSender implements Runnable {
    private final PrintWriter out;
    private final BufferedReader consoleReader;

    public MessageSender(Socket socket, String name) throws IOException {
        this.out = new PrintWriter(socket.getOutputStream(), true);
        this.consoleReader = new BufferedReader(new InputStreamReader(System.in));
        // Send name immediately
        out.println(name);
    }

    @Override
    public void run() {
        try {
            String message;
            while ((message = consoleReader.readLine()) != null) {
                out.println(message);
            }
        } catch (IOException e) {
            System.out.println("Error sending message");
        }
    }
}
```

```
class MessageReceiver implements Runnable {
    private final BufferedReader in;

    public MessageReceiver(Socket socket) throws IOException {
        this.in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
    }

    @Override
    public void run() {
        try {
            String message;
            while ((message = in.readLine()) != null) {
                System.out.println(message);
            }
        } catch (IOException e) {
            System.out.println("Disconnected from server");
        }
    }
}
```

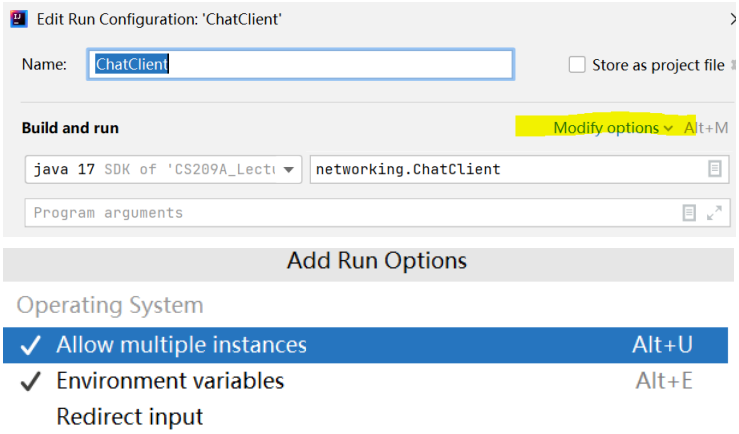
Of course, we could also use the `main` thread for read/write, and create another thread for write/read.

Execution

First run the `ChatServer`, then run multiple `ChatClient` instances. To allow multiple client instances, right-click on the `ChatClient` class and select `Modify Run Configuration`.



Click `Modify options` and select `Allow multiple instances`.



You may also define a specific instance name to distinguish client instances.

Finally, send messages from different clients and observe the results.