វិទ្យាស្ថានបច្ចេកវិទ្យាកម្ពុជា

INSTITUTE OF TECHNOLOGY OF CAMBODIA

ENGINEERING DEGREE IN APPLIED MATHEMATICS AND STATISTICS

MAJORING IN DATA SCIENCE

SUBJECT: INTRODUCTION TO PARALLEL & DISTRIBUTED SYSTEMS

TOPIC: BUILDING CHAT DISTRIBUTED SYSTEM USING JAVA RMI

Lecturer: **Mr. Khem Thay**

A REPORT SUBMITTED BY I4-AMS-A Group11:

BUTH KHEMRA	**e20201690**
HON RATANA	**e20201053**

PHNOM PENH, JUNE 2024

Table of Contents

Abstract

This report presents the development and implementation of a Chat Distributed System using Java Remote Method Invocation (RMI). The project is part of the coursework for the Introduction to Parallel and Distributed Systems lecture, aiming to demonstrate the practical application of distributed system principles. Java RMI was chosen for its simplicity and robustness in enabling remote communication between distributed objects in a network.

The Chat Distributed System facilitates real-time communication between multiple clients over a distributed network. Each client can send and receive messages through a centralised server, which manages client connections and message dissemination. This system showcases key distributed system concepts, including remote object invocation, client-server architecture, and concurrent processing.

The implementation details include setting up the RMI registry, creating the remote interfaces and their implementations, and handling client-server interactions. Emphasis is placed on the use of RMI for method invocation across the network, ensuring seamless and efficient communication.

This project not only highlights the capabilities of Java RMI in building distributed applications but also serves as a foundational example for developing more complex distributed systems. Through this project, we gain insights into the challenges and solutions associated with distributed computing, providing a solid groundwork for further exploration in the field of intelligent and parallel systems.

I. Introduction to Java RMI (Remote Method Invocation)

Java Remote Method Invocation (RMI) is a powerful mechanism that allows objects residing in different Java Virtual Machines (JVMs) to interact and invoke methods on each other as if they were local. RMI facilitates the development of distributed applications by abstracting the complexities involved in network communication, enabling developers to focus on application logic rather than the underlying infrastructure.

RMI works by allowing a Java program to invoke methods on an object running in another JVM. This is achieved through the use of stubs and skeletons. The stub acts as a proxy on the client side, representing the remote object and forwarding method calls to the server. The skeleton, which was used in older versions of Java RMI but has since been replaced by dynamic proxies, receives these calls on the server side, invoking the appropriate method on the actual remote object.

The core components of Java RMI include:

- **Remote Interface**: Defines the methods that can be invoked remotely. Any object implementing this interface can be accessed from a remote JVM.
- **Remote Object**: An instance of a class that implements a remote interface and can be accessed remotely.
- **RMI Registry**: A naming service that allows clients to look up remote objects by name. The registry maps names to remote objects, enabling clients to obtain references to these objects.
- **Stubs and Proxies**: Stubs (on the client side) and proxies (on the server side) facilitate the communication between remote objects, handling the serialization and deserialization of method arguments and return values.

Java RMI simplifies the creation of distributed applications by providing built-in support for object serialization, network communication, and thread management. It also supports dynamic class loading, allowing clients to obtain the bytecode of classes they need at runtime, enhancing flexibility and ease of deployment.

In the context of our Chat Distributed System, Java RMI is used to manage communication between the chat server and multiple clients. Each client can send messages to the server, which then distributes these messages to all connected clients. This setup exemplifies a typical client-server architecture in a distributed environment, demonstrating RMI's capability to handle remote interactions efficiently.

II. Background and Motivation

In recent years, the rise of distributed systems has revolutionized the way applications are developed and deployed. Distributed systems allow for the distribution of tasks across multiple computers, leading to improved performance, scalability, and reliability. This paradigm shift has enabled the development of sophisticated applications that can handle large-scale data processing, real-time analytics, and complex computations.

One prominent area where distributed systems have made a significant impact is in communication and collaboration tools. With the increasing need for real-time communication in both personal and professional settings, chat systems have become an essential component of modern software solutions. These systems enable users to communicate instantly, share information, and collaborate effectively, regardless of their geographical location.

The motivation behind this project is to explore the capabilities of Java RMI (Remote Method Invocation) in building a distributed chat system. Java RMI provides a framework for developing distributed applications in Java, allowing objects to interact across different JVMs. By leveraging Java RMI, we aim to create a chat system that can handle multiple clients, facilitate real-time communication, and demonstrate the principles of distributed computing.

The specific motivations for this project include:

1. **Understanding Distributed Communication**: By building a chat system using Java RMI, we can gain a deeper understanding of how distributed communication works. This includes learning how remote method calls are made, how data is serialized and deserialized, and how network communication is managed.
2. **Exploring Java RMI**: Java RMI is a powerful tool for developing distributed applications, but it is often underutilized. This project provides an opportunity to explore the features and capabilities of Java RMI, including remote interfaces, object serialization, and the RMI registry.

3. **Practical Application of Theoretical Concepts**: The project allows us to apply theoretical concepts learned in the course on Parallel and Distributed Systems. By implementing a real-world application, we can bridge the gap between theory and practice, reinforcing our understanding of distributed systems.
4. **Developing a Robust and Scalable System**: Building a chat system presents several challenges, including managing concurrent client connections, ensuring message delivery, and maintaining system performance under load. Addressing these challenges will help us develop skills in designing and implementing robust and scalable distributed systems.
5. **Enhancing Collaboration and Communication**: The final product of this project will be a chat system that can be used for real-time communication and collaboration. This tool can be valuable in various contexts, from team collaboration in software development projects to providing a communication platform for online communities.

In summary, the background and motivation for this project are rooted in the desire to understand and leverage the capabilities of Java RMI in building a distributed chat system. By undertaking this project, we aim to deepen our knowledge of distributed systems, explore the practical application of Java RMI, and develop a robust tool for real-time communication.

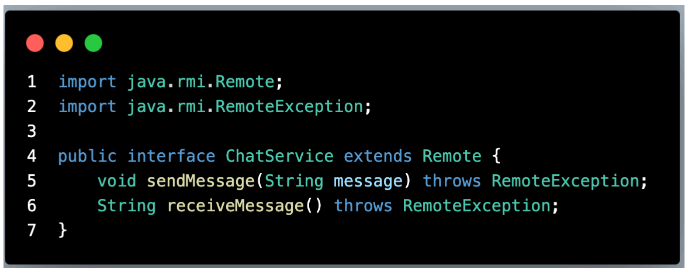
III. Java RMI Overview

Java Remote Method Invocation (RMI) is a powerful and flexible framework that allows objects to interact with each other across different Java Virtual Machines (JVMs). This capability makes RMI a suitable choice for building distributed applications where components need to communicate over a network. In this section, we provide a detailed overview of Java RMI, covering its architecture, core concepts, and components.

1. Architecture of Java RMI

The architecture of Java RMI is designed to facilitate the development of distributed applications by providing a straightforward way to invoke methods on remote objects. The key components of the RMI architecture include:

1. **Client and Server**: In an RMI-based application, the server hosts remote objects and provides their services, while the client accesses these services by invoking methods on the remote objects.



```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ChatService extends Remote {
    void sendMessage(String message) throws RemoteException;
    String receiveMessage() throws RemoteException;
}
```

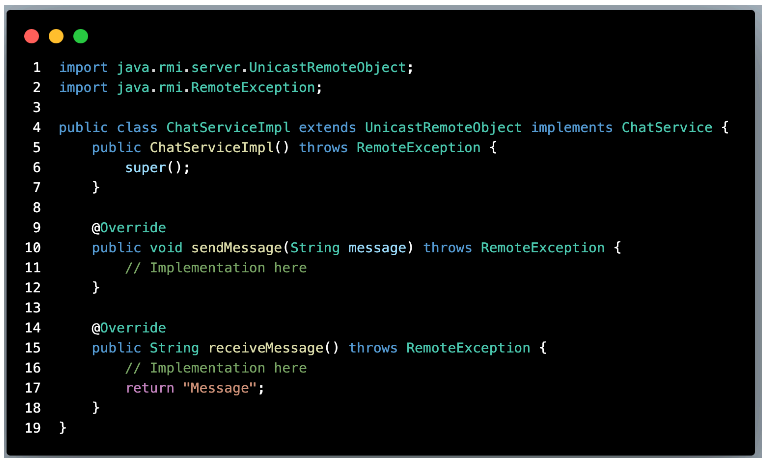
2. **Remote Interfaces**: These are Java interfaces that declare the methods that can be invoked remotely. Both the client and server must use the same remote interface.
3. **Remote Objects**: These are the actual implementations of the remote interfaces. Remote objects reside on the server, and their methods can be called from the client.
4. **RMI Registry**: This is a simple name service that allows clients to look up remote objects by name. The registry runs on the server and keeps track of available remote objects.
5. **Stub and Skeleton**: The stub is a proxy that resides on the client side. It represents the remote object and forwards method calls from the client to the server. The skeleton is a server-side entity that dispatches incoming method calls to the appropriate remote object implementation (note: the skeleton was removed in Java 2, leaving the stub to handle communication directly).

2. Core Concept

Java RMI is built on several core concepts that are crucial for understanding how it operates:

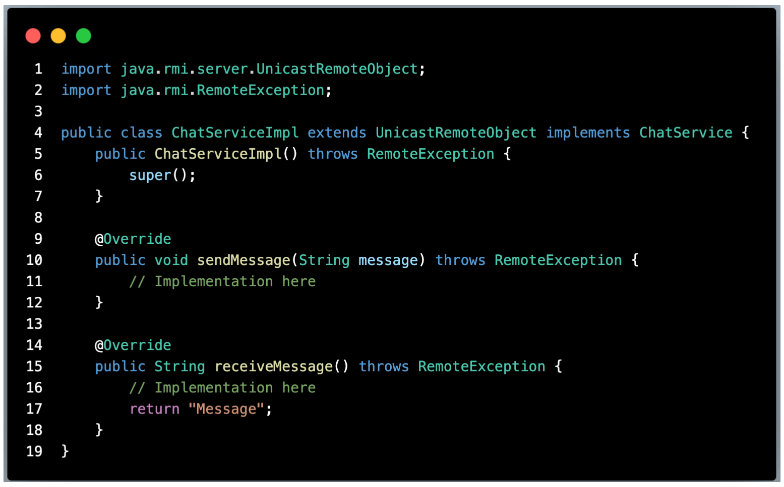
1. **Remote Interface**: A remote interface extends the java.rmi.Remote interface and declares the methods that can be called remotely. Each method must throw a RemoteException.

2. **Remote Object**: A remote object implements the remote interface and provides the actual method implementations. It extends java.rmi.server.UnicastRemoteObject.



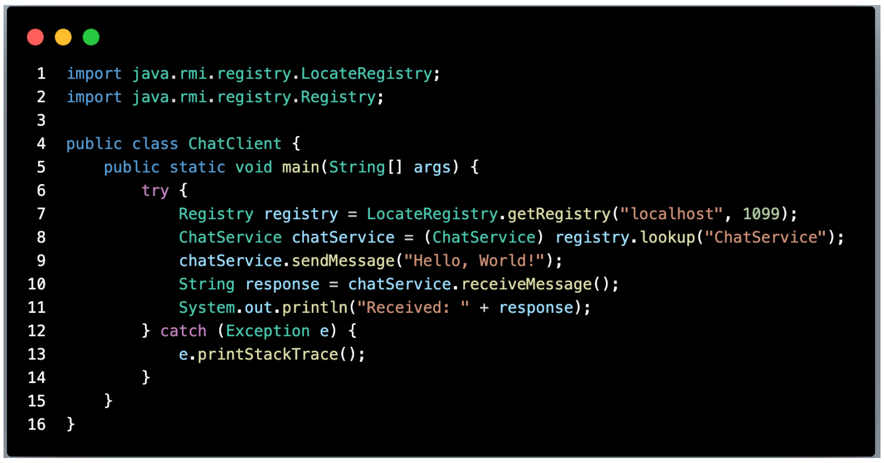
```java
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class ChatServiceImpl extends UnicastRemoteObject implements ChatService {
    public ChatServiceImpl() throws RemoteException {
        super();
    }

    @Override
    public void sendMessage(String message) throws RemoteException {
        // Implementation here
    }

    @Override
    public String receiveMessage() throws RemoteException {
        // Implementation here
        return "Message";
    }
}
```

3. **RMI Registry**: The RMI registry binds remote object instances to names so that clients can look them up and invoke methods on them.



```java
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class ChatServiceImpl extends UnicastRemoteObject implements ChatService {
    public ChatServiceImpl() throws RemoteException {
        super();
    }

    @Override
    public void sendMessage(String message) throws RemoteException {
        // Implementation here
    }

    @Override
    public String receiveMessage() throws RemoteException {
        // Implementation here
        return "Message";
    }
}
```

4. **Client Lookup and Invocation**: The client locates the remote object using the RMI registry and then invokes methods on it as if it were a local object.



```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class ChatClient {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry("localhost", 1099);
            ChatService chatService = (ChatService) registry.lookup("ChatService");
            chatService.sendMessage("Hello, World!");
            String response = chatService.receiveMessage();
            System.out.println("Received: " + response);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3. Benefits of Using Java RMI

Java RMI offers several benefits for developing distributed applications:

1. **Ease of Use**: RMI abstracts much of the complexity involved in remote communication, allowing developers to focus on application logic rather than network details.
2. **Seamless Integration**: As a native Java technology, RMI integrates seamlessly with other Java features and libraries, making it a natural choice for Java developers.
3. **Object-Oriented Approach**: RMI leverages Java's object-oriented principles, enabling remote objects to be used just like local objects, which simplifies the design and implementation of distributed systems.
4. **Built-in Security**: RMI includes built-in security features that help protect against unauthorized access and ensure the integrity of remote method calls.

In summary, Java RMI is a robust framework that facilitates the development of distributed applications by enabling remote method invocation between Java objects across different JVMs. Its architecture, core concepts, and ease of use make it an excellent choice for building distributed systems, such as the chat application described in this report.

IV. System Architecture

The System Architecture section provides a high-level overview of the structure of the chat system, the roles of different components, and how they interact.

1. System Components

- ❖ **Client**: The client-side application that users interact with. It allows users to send and receive messages. Each client connects to the server and communicates with it to send and retrieve messages.
- ❖ **Server**: The server-side application that manages all client connections and handles the broadcasting of messages to all connected clients. It maintains a list of active clients and a log of messages.

- ❖ **Interfaces**: The remote interfaces that define the methods available for remote invocation. These interfaces are implemented by the server and called by the client

2. Sequence Diagram

To further illustrate the interaction flow, a sequence diagram can be included in the report. Here’s a textual representation of the sequence of events:

1. **Client 1** initializes and registers with the **Server**.
2. **Client 2** initializes and registers with the **Server**.
3. **Client 1** sends a message "Hello, World!" to the **Server**.
4. The **Server** broadcasts "Client 1: Hello, World!" to **Client 2**.
5. **Client 2** displays the message.

3. Interaction Flow with Stubs and Skeletons

In a Java RMI system, stubs and skeletons play a crucial role in enabling remote method invocations:

- **Stubs**: On the client side, a stub acts as a proxy for the remote object. It provides the same methods as the remote interface but internally handles the network communication to forward method calls to the server.
- **Skeletons**: On the server side, a skeleton receives the method calls from the stub, unmarshals the parameters, invokes the actual method on the server object, and then marshals the result back to the stub.

4. Example Interaction Flow

- ❖ **Client Connection**: When a client application starts, it connects to the RMI registry on the server to look up the ChatService object.
- ❖ **Method Invocation**: The client invokes the sendMessage method on the ChatService stub. The stub marshals the method parameters and sends them to the server.
- ❖ **Method Execution**: The skeleton on the server side receives the method call, unmarshals the parameters, and invokes the sendMessage method on the ChatServiceImpl object.
- ❖ **Return Result**: The method result (if any) is marshaled by the skeleton and sent back to the client stub, which then returns it to the client application.

V. Implementation Details

In this section, we delve into the technical aspects and implementation specifics of the Chat Distributed System using Java RMI. This section outlines the steps involved in setting up the RMI infrastructure, the key classes and methods implemented, and the interactions between different components.

1. Setting Up the RMI Infrastructure

1. **RMI Registry**: The RMI registry is a simple server-side name server that allows clients to get a reference to a remote object. The server application registers the remote objects with the RMI registry so that clients can look them up and invoke methods on them.

2. **Remote Interfaces**: Define the remote interfaces that declare the methods available for remote invocation. These interfaces extend java.rmi.Remote.

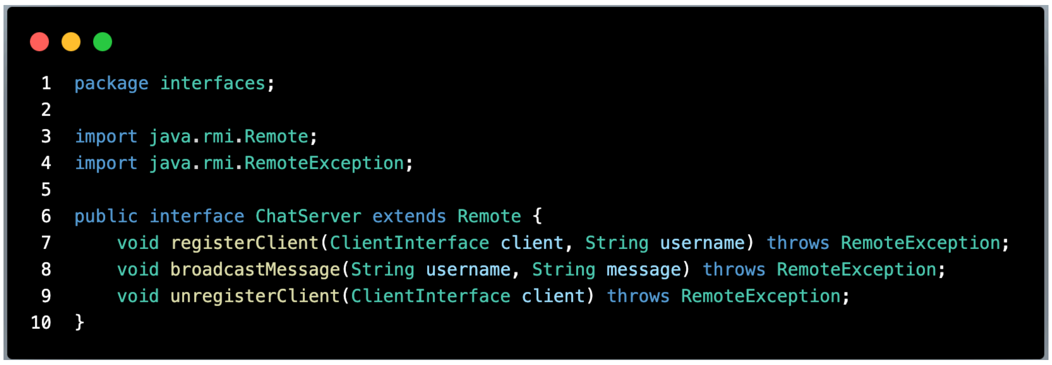
3. **Remote Object Implementation**: Implement the remote interfaces in server-side classes. These classes extend UnicastRemoteObject and implement the methods declared in the remote interfaces.

4. **Client-Side Stubs**: The stubs are generated automatically and act as proxies on the client side. The stubs are used by the client to invoke methods on the remote objects.

2. Key Classes and Methods

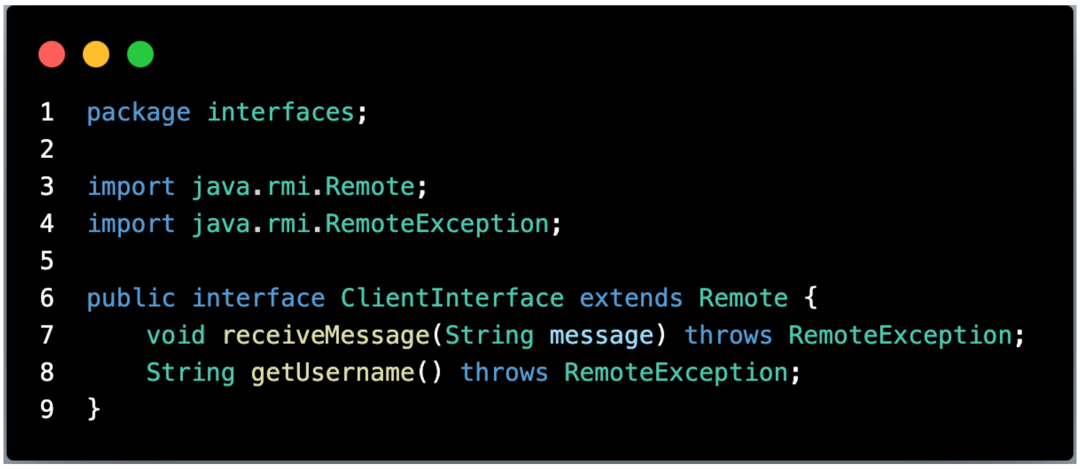
1. **Interfaces**

   a. **ChatServer.java**: This interface defines the methods that the server will implement and the client will invoke remotely.



```java
package interfaces;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ChatServer extends Remote {
    void registerClient(ClientInterface client, String username) throws RemoteException;
    void broadcastMessage(String username, String message) throws RemoteException;
    void unregisterClient(ClientInterface client) throws RemoteException;
}
```

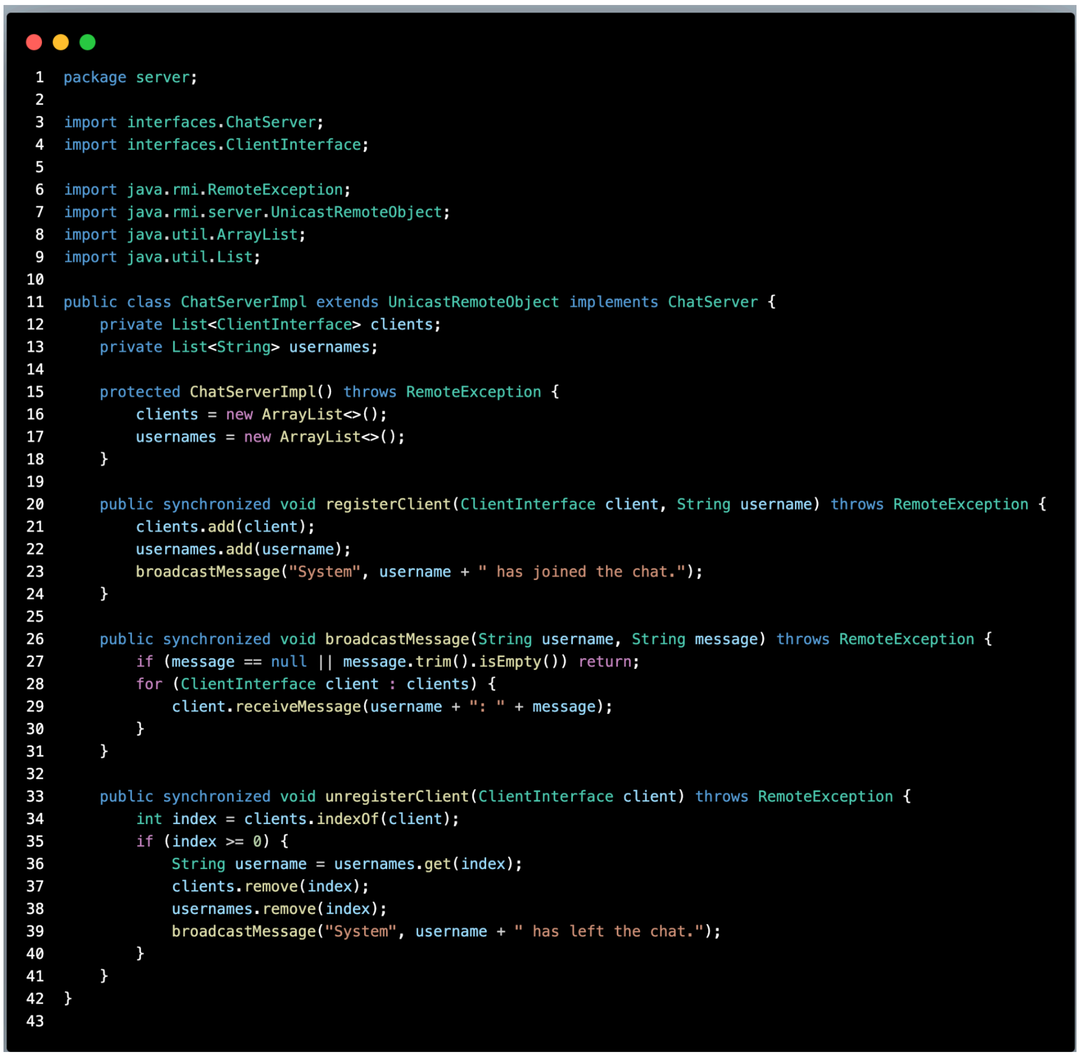
b. **ClientInterface.java**: This interface defines the methods that the server will call on the client.



```java
package interfaces;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ClientInterface extends Remote {
    void receiveMessage(String message) throws RemoteException;
    String getUsername() throws RemoteException;
}
```

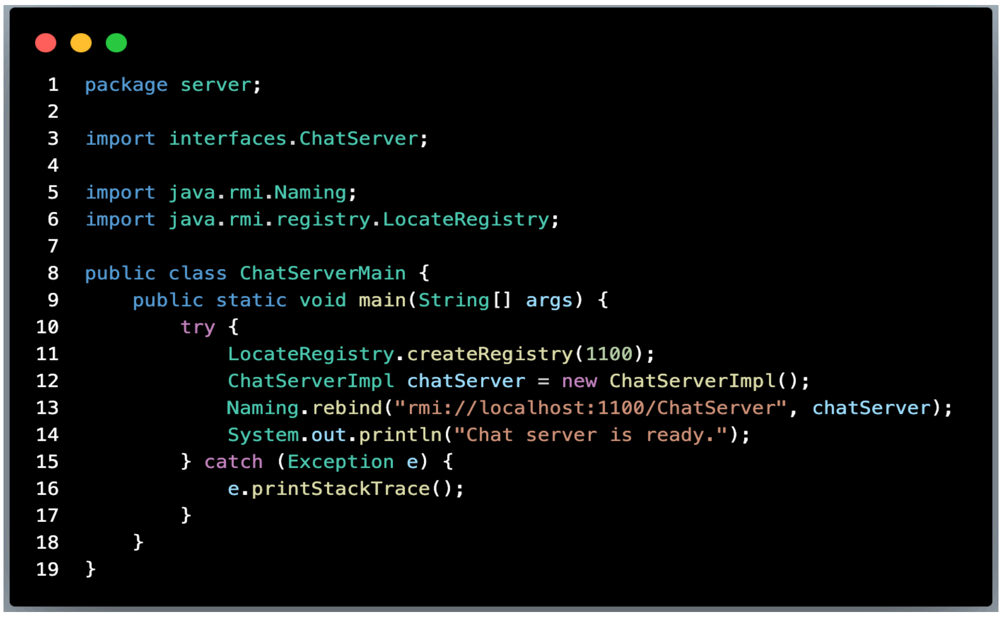
2. **<u>Server</u>**

a. **ChatServerImpl.java**: This class implements the ChatServer interface. It contains the logic for registering clients and broadcasting messages.



```java
package server;

import interfaces.ChatServer;
import interfaces.ClientInterface;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;
import java.util.List;

public class ChatServerImpl extends UnicastRemoteObject implements ChatServer {
    private List<ClientInterface> clients;
    private List<String> usernames;

    protected ChatServerImpl() throws RemoteException {
        clients = new ArrayList<>();
        usernames = new ArrayList<>();
    }

    public synchronized void registerClient(ClientInterface client, String username) throws RemoteException {
        clients.add(client);
        usernames.add(username);
        broadcastMessage("System", username + " has joined the chat.");
    }

    public synchronized void broadcastMessage(String username, String message) throws RemoteException {
        if (message == null || message.trim().isEmpty()) return;
        for (ClientInterface client : clients) {
            client.receiveMessage(username + ": " + message);
        }
    }

    public synchronized void unregisterClient(ClientInterface client) throws RemoteException {
        int index = clients.indexOf(client);
        if (index >= 0) {
            String username = usernames.get(index);
            clients.remove(index);
            usernames.remove(index);
            broadcastMessage("System", username + " has left the chat.");
        }
    }
}
```

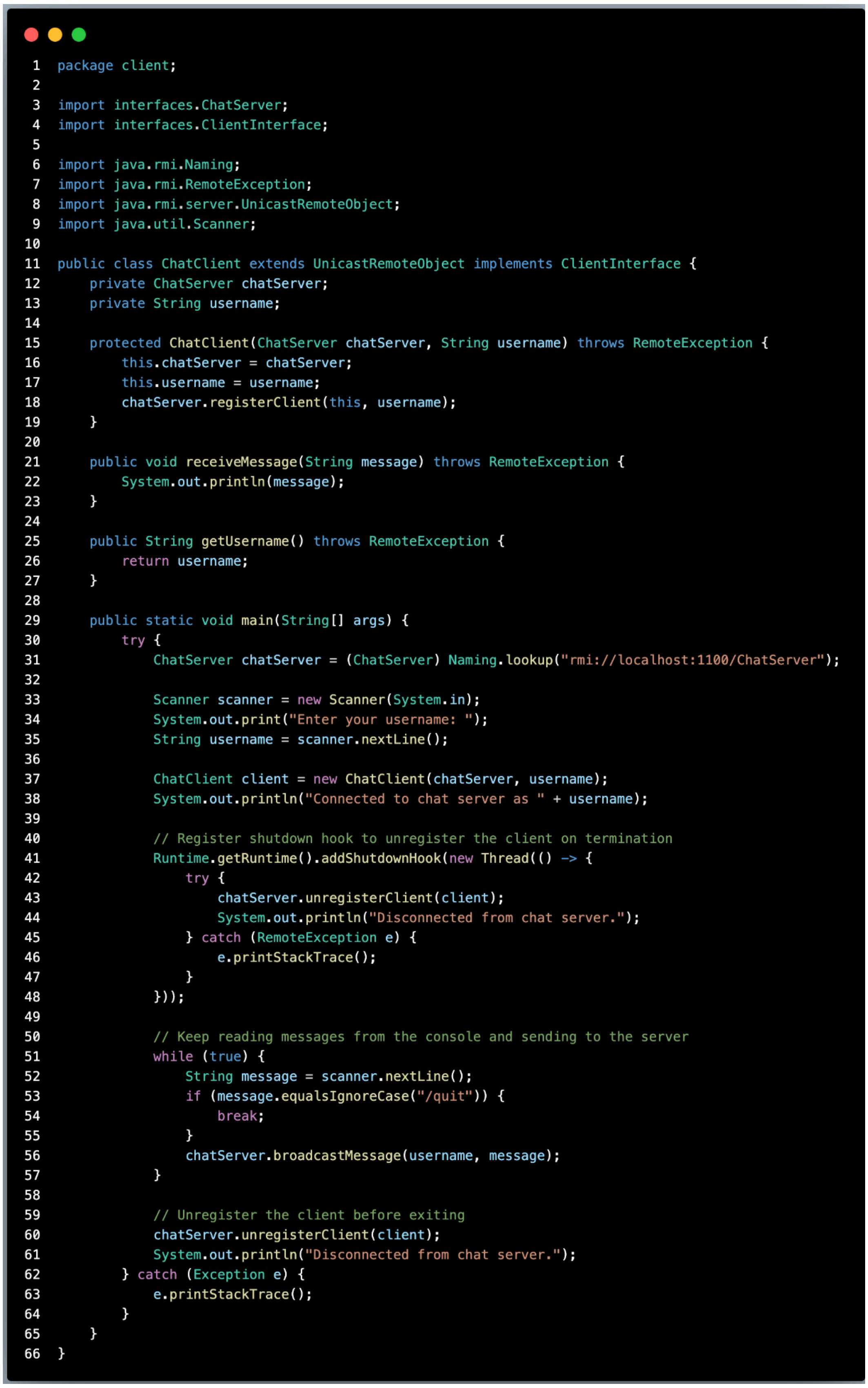
b. **ChatServerMain.java**: This class contains the main method to start the RMI registry and bind the ChatServer implementation to the registry.



```java
package server;

import interfaces.ChatServer;

import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

public class ChatServerMain {
    public static void main(String[] args) {
        try {
            LocateRegistry.createRegistry(1100);
            ChatServerImpl chatServer = new ChatServerImpl();
            Naming.rebind("rmi://localhost:1100/ChatServer", chatServer);
            System.out.println("Chat server is ready.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3. Client

The ChatClient class connects to the ChatService on the server and provides the user interface for sending and receiving messages.



```java
package client;

import interfaces.ChatServer;
import interfaces.ClientInterface;

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Scanner;

public class ChatClient extends UnicastRemoteObject implements ClientInterface {
    private ChatServer chatServer;
    private String username;

    protected ChatClient(ChatServer chatServer, String username) throws RemoteException {
        this.chatServer = chatServer;
        this.username = username;
        chatServer.registerClient(this, username);
    }

    public void receiveMessage(String message) throws RemoteException {
        System.out.println(message);
    }

    public String getUsername() throws RemoteException {
        return username;
    }

    public static void main(String[] args) {
        try {
            ChatServer chatServer = (ChatServer) Naming.lookup("rmi://localhost:1100/ChatServer");

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

            ChatClient client = new ChatClient(chatServer, username);
            System.out.println("Connected to chat server as " + username);

            // Register shutdown hook to unregister the client on termination
            Runtime.getRuntime().addShutdownHook(new Thread(() -> {
                try {
                    chatServer.unregisterClient(client);
                    System.out.println("Disconnected from chat server.");
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
            }));

            // Keep reading messages from the console and sending to the server
            while (true) {
                String message = scanner.nextLine();
                if (message.equalsIgnoreCase("/quit")) {
                    break;
                }
                chatServer.broadcastMessage(username, message);
            }

            // Unregister the client before exiting
            chatServer.unregisterClient(client);
            System.out.println("Disconnected from chat server.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3. Interaction Flow

1. **Server Initialization**: The server initializes by creating an instance of ChatServiceImpl and binding it to the RMI registry under the name ChatService.
2. **Client Connection**: The client looks up the ChatService in the RMI registry and obtains a reference to the remote object.
3. **Sending Messages**: When a client sends a message, the sendMessage method on the ChatService stub is invoked. The stub forwards the call to the server, where the ChatServiceImpl processes the message.
4. **Receiving Messages**: The client continuously checks for new messages by invoking the receiveMessage method on the ChatService stub. The server returns any new messages for the client to display.

4. Error Handling and Robustness

1. **Connection Issues**: Proper handling of RemoteException during the connection to the RMI registry and remote method invocations.
2. **Synchronization**: Ensuring that methods that modify shared resources (like registerClient and broadcastMessage in ChatServerImpl) are synchronized to avoid concurrency issues.
3. **Client Notifications**: Handling scenarios where the client might be unreachable or has disconnected.

VI. Features and Functionalities

1. Feature

1. Real-Time Messaging

- **Description**: The system supports real-time messaging between multiple clients. When a client sends a message, it is immediately broadcasted to all connected clients.

- **Functionality**:
  - Clients can send and receive messages in real-time.
  - Messages are displayed instantly to all registered clients.
  - Ensures low-latency communication.

2. Multi-Client Support

- **Description**: The system can handle multiple clients simultaneously, allowing a collaborative chat environment.
- **Functionality**:
  - Clients can join and leave the chat at any time.
  - The server maintains a list of active clients and broadcasts messages to all connected clients.

3. Message Broadcasting

- **Description**: The server broadcasts messages to all registered clients.
- **Functionality**:
  - When a client sends a message, the server broadcasts it to all other registered clients.
  - Ensures that all clients receive the same message simultaneously.

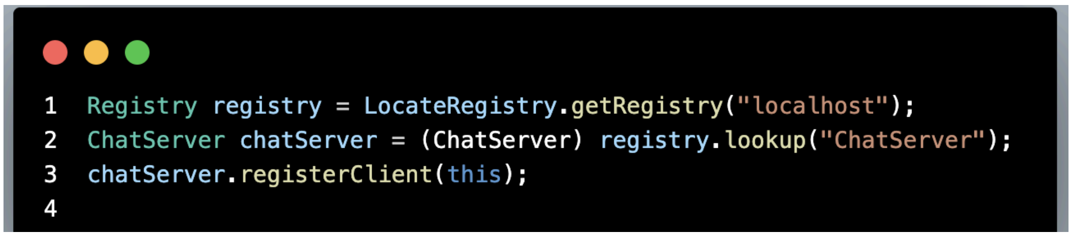
4. Robust Error Handling

- **Description**: The system includes error handling to manage common issues such as client disconnections and network failures.
- **Functionality**:
  - Detects and handles RemoteException when a client disconnects unexpectedly.
  - Provides feedback to clients if a message fails to send.
  - Ensures that the server continues to function smoothly even if some clients disconnect.

2. Functionalities

1. Sending Message

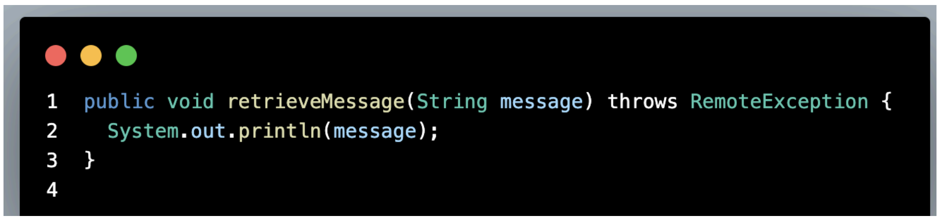
- Process
    - Clients send messages to the server using the broadcastMessage method.
    - The server receives the message and broadcasts it to all registered clients.
- Code



```java
Registry registry = LocateRegistry.getRegistry("localhost");
ChatServer chatServer = (ChatServer) registry.lookup("ChatServer");
chatServer.registerClient(this);

```

2. Retrieve Message

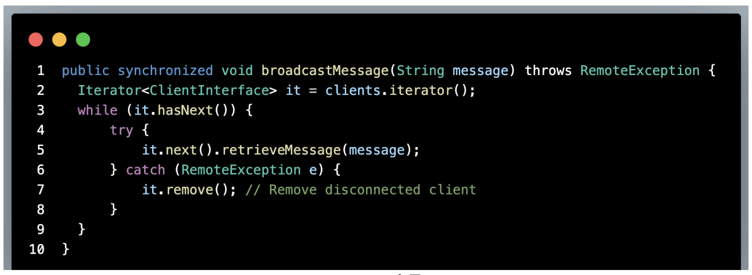
- Process
    - The server calls the retrieveMessage method on each registered client.
    - Clients display the received message to the user.
- Code



```java
public void retrieveMessage(String message) throws RemoteException {
  System.out.println(message);
}

```

3. Client Disconnect Handling

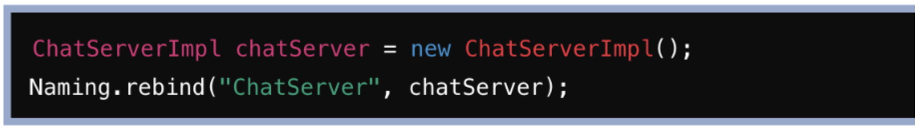
- Process
    - The server handles RemoteException when a client disconnects unexpectedly.
    - Disconnected clients are removed from the server's list of active clients.
- Code



```java
public synchronized void broadcastMessage(String message) throws RemoteException {
  Iterator<ClientInterface> it = clients.iterator();
  while (it.hasNext()) {
      try {
          it.next().retrieveMessage(message);
      } catch (RemoteException e) {
          it.remove(); // Remove disconnected client
      }
  }
}
```

4. Server Startup

- Process:
  - The server sets up the RMI registry and binds the ChatServerImpl object.
- Code



```
ChatServerImpl chatServer = new ChatServerImpl();
Naming.rebind("ChatServer", chatServer);
```

VII. Testing and Results

1. Testing Procedure

1. Environment Setup
   - **Objective**: Verify that the RMI registry and server are correctly set up.
   - **Procedure**:
     - Start the RMI registry using the rmiregistry 1099 & command.
     - Run the ChatServerMain to bind the ChatServerImpl to the RMI registry.
     - Confirm that the server is running and accessible.

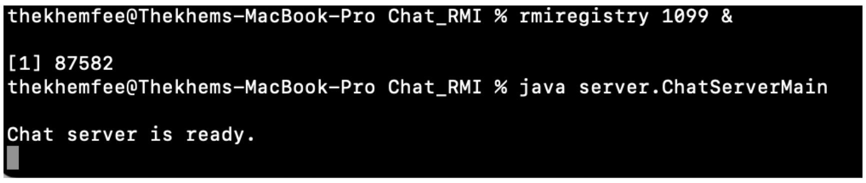
2. Client Registration Test
   - **Objective**: Ensure clients can register with the server.
   - **Procedure**:
     - Start multiple instances of ChatClient.
     - Check the server logs to confirm that each client is registered.
     - Verify that each client receives an acknowledgment of successful registration
3. Message Broadcasting Test
   - **Objective**: Validate that messages are correctly broadcasted to all registered clients.
   - **Procedure**:
     - Have multiple clients connected to the server.
     - Send messages from one client.
     - Verify that all other clients receive the message in real-time.

        ○ Check for proper display of messages on the client interfaces.

4. Client Disconnection Test
    - **Objective**: Ensure the server handles client disconnections gracefully.
    - **Procedure**:
        ○ Disconnect a client by closing the client application.
        ○ Confirm that the server removes the client from its list of active clients.
        ○ Check that remaining clients continue to receive messages without interruption.

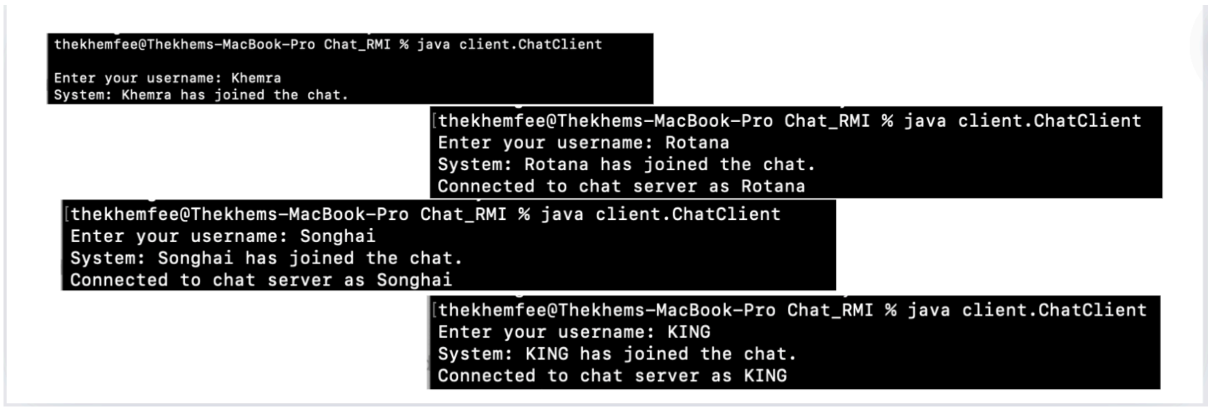
2. Results

1. Environment Setup



```
thekhemfee@Thekhems-MacBook-Pro Chat_RMI % rmiregistry 1099 &

[1] 87582
thekhemfee@Thekhems-MacBook-Pro Chat_RMI % java server.ChatServerMain

Chat server is ready.
```

2. Client Registration Test



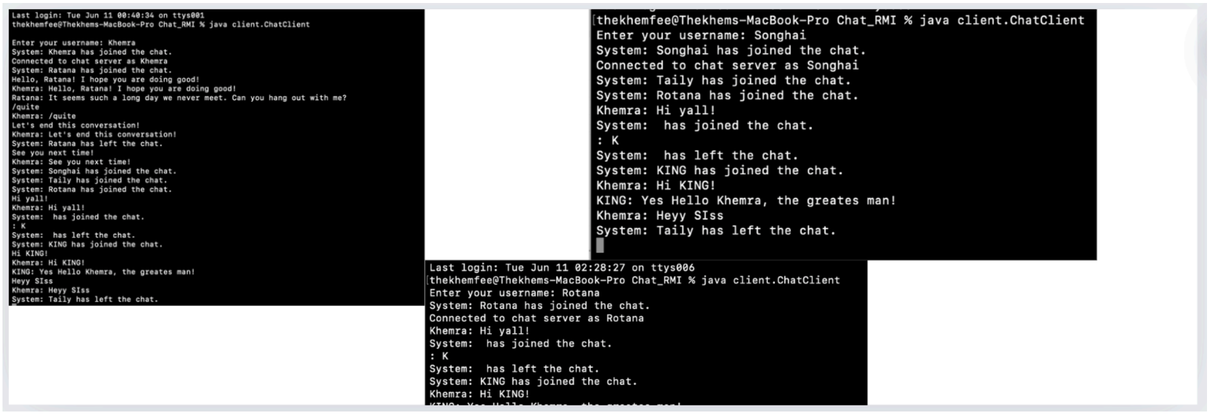
```
thekhemfee@Thekhems-MacBook-Pro Chat_RMI % java client.ChatClient

Enter your username: Khemra
System: Khemra has joined the chat.
```

```
thekhemfee@Thekhems-MacBook-Pro Chat_RMI % java client.ChatClient
Enter your username: Rotana
System: Rotana has joined the chat.
Connected to chat server as Rotana
```

```
thekhemfee@Thekhems-MacBook-Pro Chat_RMI % java client.ChatClient
Enter your username: Songhai
System: Songhai has joined the chat.
Connected to chat server as Songhai
```

```
thekhemfee@Thekhems-MacBook-Pro Chat_RMI % java client.ChatClient
Enter your username: KING
System: KING has joined the chat.
Connected to chat server as KING
```

3. Message Broadcasting Test



```
Last login: Tue Jun 11 00:40:34 on ttys001
thekhemfee@Thekhems-MacBook-Pro Chat_RMI % java client.ChatClient

Enter your username: Khemra
System: Khemra has joined the chat.
Connected to chat server as Khemra
System: Ratana has joined the chat.
Hello, Ratana! I hope you are doing good!
Khemra: Hello, Ratana! I hope you are doing good!
Ratana: It seems such a long day we never meet. Can you hang out with me?
/quite
Khemra: /quite
Let's end this conversation!
Khemra: Let's end this conversation!
System: Ratana has left the chat.
See you next time!
Khemra: See you next time!
System: Songhai has joined the chat.
System: Taily has joined the chat.
System: Rotana has joined the chat.
Hi yall!
Khemra: Hi yall!
System:  has joined the chat.
: K
System:  has left the chat.
System: KING has joined the chat.
Hi KING!
Khemra: Hi KING!
KING: Yes Hello Khemra, the greates man!
Heyy SIss
Khemra: Heyy SIss
System: Taily has left the chat.
```

```
thekhemfee@Thekhems-MacBook-Pro Chat_RMI % java client.ChatClient
Enter your username: Songhai
System: Songhai has joined the chat.
Connected to chat server as Songhai
System: Taily has joined the chat.
System: Rotana has joined the chat.
Khemra: Hi yall!
System:  has joined the chat.
: K
System:  has left the chat.
System: KING has joined the chat.
Khemra: Hi KING!
KING: Yes Hello Khemra, the greates man!
Khemra: Heyy SIss
System: Taily has left the chat.
```

```
Last login: Tue Jun 11 02:28:27 on ttys006
thekhemfee@Thekhems-MacBook-Pro Chat_RMI % java client.ChatClient
Enter your username: Rotana
System: Rotana has joined the chat.
Connected to chat server as Rotana
Khemra: Hi yall!
System:  has joined the chat.
: K
System:  has left the chat.
System: KING has joined the chat.
Khemra: Hi KING!
```

VIII. Conclusion

In this project, we successfully developed a Chat Distributed System using Java RMI (Remote Method Invocation). The system allows multiple clients to connect to a central server, send messages, and receive real-time updates, thereby facilitating seamless communication among users in a distributed environment.

1. Key Achievements

1. **Understanding of Java RMI**: We gained a solid understanding of Java RMI, its architecture, and how it facilitates communication between remote objects in a distributed system.
2. **System Architecture**: We designed a robust system architecture that includes a client-server model with distinct layers for client-side operations, server-side processing, and shared interfaces for remote communication.
3. **Implementation Details**: We implemented the core functionalities of the chat system, including client registration, message broadcasting, and handling client disconnections.
4. **Testing and Validation**: Comprehensive testing procedures were carried out to validate the system's functionality, robustness, and error handling capabilities. The tests demonstrated that the system operates reliably under various conditions.

2. Key Features

- **Real-time Messaging**: Clients can send and receive messages in real-time, ensuring immediate communication among users.
- **Client Management**: The server effectively manages client connections and disconnections, maintaining an updated list of active clients.
- **Error Handling**: The system includes mechanisms to handle errors gracefully, ensuring continued operation despite network failures or abrupt client shutdowns.

3. Learning Outcomes

- **Distributed System Concepts**: This project provided practical experience in building and managing distributed systems, enhancing our understanding of key concepts such as remote communication, client-server architecture, and fault tolerance.
- **Java RMI Implementation**: We gained hands-on experience with Java RMI, learning how to implement remote interfaces, manage remote objects, and handle communication between distributed components.

4. Future Enhancements

While the current implementation of the Chat Distributed System meets the project objectives, there are several areas for future enhancement:

- **Security**: Implementing encryption and authentication mechanisms to secure communication between clients and the server.
- **Scalability**: Enhancing the system to support a larger number of clients and potentially integrating load balancing techniques.
- **User Interface**: Developing a more user-friendly graphical interface for the chat clients to improve user experience.

5. Conclusion

The Chat Distributed System using Java RMI is a functional and reliable application that demonstrates the effective use of Java RMI for building distributed systems. The project provided valuable insights into distributed system design, implementation, and testing. The system's successful operation and robust handling of client interactions and errors underscore its potential for real-world applications in distributed communication systems.

Appendix

Link to my Github to reach source code for this Chat System click here!