<table>
<tr><td colspan="4">Name of Student: Yash Laxman Desai</td></tr>
<tr><td colspan="2">Roll Number: 10</td><td colspan="2">Practical Number: 1</td></tr>
<tr><td colspan="4">Title of Assignment: To develop a multi-client chat server application where multiple clients chat with each other concurrently, where the messages sent by different clients are first communicated to the server and then the server, on behalf of the source client, communicates the messages to the appropriate destination client.</td></tr>
<tr><td colspan="2">DOP: 08/08/24</td><td colspan="2">DOS: 08/08/24</td></tr>
<tr><td>CO Mapped: CO1</td><td>PO Mapped:
PO3,PO5,PO7,
PO12, PSO1, PS02</td><td>Faculty Signature:</td><td>Marks:</td></tr>
</table>

Aim: To develop a multi-client chat server application where multiple clients chat with each other concurrently, where the messages sent by different clients are first communicated to the server and then the server, on behalf of the source client, communicates the messages to the appropriate destination client

1. Remote Procedure Call (RPC)

Definition: Remote Procedure Call (RPC) is a powerful technique used in distributed computing that allows a program to cause a procedure (subroutine) to execute on a different address space (commonly on another physical machine). It abstracts the complexity of the underlying network communication, making remote interactions appear like local function calls to the developer.

How RPC Works:

- **Client Stub:** The client-side representation of the procedure call. It translates the procedure call into a network request.
- **Server Stub:** The server-side counterpart that receives the network request, invokes the actual procedure, and sends back the result.
- **Marshalling and Unmarshalling:** Marshalling is the process of converting the procedure parameters into a format suitable for transmission over the network. Unmarshalling is the reverse process performed on the server side.

Advantages:

- Simplifies the development of distributed applications.
- Allows for more modular and maintainable code by separating concerns between client and server.
- Can be used to call procedures across different programming languages and platforms.

Challenges:

- Handling network failures and latency.
- Ensuring data integrity and security during transmission.
- Overhead due to marshalling and unmarshalling.

Use Cases:

- Distributed systems and microservices architecture.
- Client-server applications that require communication over a network.

2. Socket Class

Definition: The Socket class in Java represents a client-side endpoint for communication between two machines over a network. It is part of the java.net package and provides a rich set of methods to manage network connections.

Key Methods:

- **Constructor:**
  - Socket(String host, int port): Creates a stream socket and connects it to the specified port number on the named host.
- **Communication:**
  - getInputStream(): Returns an input stream for reading bytes from this socket.
  - getOutputStream(): Returns an output stream for writing bytes to this socket.
- **Connection Management:**
  - close(): Closes the socket and releases any associated resources.
  - connect(SocketAddress endpoint): Connects the socket to the specified endpoint address.

Advantages:

- Provides a simple and flexible way to establish a network connection.
- Supports both TCP (stream) and UDP (datagram) communication.

Challenges:

- Managing socket connections efficiently, especially in a multi-threaded environment.
- Handling various I/O exceptions and network failures gracefully.

3. ServerSocket Class

Definition: The ServerSocket class in Java represents a server-side endpoint that listens for incoming client connections. It is also part of the java.net package and is used to create servers that can accept client requests over a network.

Key Methods:

- **Constructor:**
  - ServerSocket(int port): Creates a server socket bound to the specified port.
- **Listening for Connections:**
  - accept(): Listens for and accepts an incoming connection to this socket. This method blocks until a connection is made.
- **Socket Management:**
  - close(): Closes the server socket and releases any associated resources.
  - bind(SocketAddress endpoint): Binds the server socket to a specific address (IP address and port).

Advantages:

- Simplifies the process of establishing a server that can handle multiple client connections.
- Provides a blocking method (accept()) to wait for incoming connections, ensuring efficient use of resources.

Challenges:

- Managing multiple client connections efficiently, often requiring multi-threading or a thread pool.
- Handling network and I/O exceptions to maintain server stability and reliability.

Description: A multi-client chat server application where multiple clients chat with each other concurrently. The messages sent by different clients are first communicated to the server and then the server, on behalf of the source client, communicates the messages to the appropriate destination client.

Code:

ChatServer.java

```
package com.chat.server;
import java.io.*;
import java.net.*;
import java.util.*;
public class ChatServer {
  private static final int PORT = 12345;
  private static Set<ClientHandler> clientHandlers = new HashSet<>();
  public static void main(String[] args) {
    System.out.println("Server is started...");
    try (ServerSocket serverSocket = new ServerSocket(PORT)) {
      while (true) {
        Socket clientSocket = serverSocket.accept();
        ClientHandler clientHandler = new ClientHandler(clientSocket);
        clientHandlers.add(clientHandler);
        clientHandler.start();
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
  private static void broadcast(String message, ClientHandler sourceClient) {
    for (ClientHandler clientHandler : clientHandlers) {
      if (clientHandler != sourceClient) {
        clientHandler.sendMessage(message);
      }
    }
  }
  private static class ClientHandler extends Thread {
    private Socket clientSocket;
    private PrintWriter out;
    private String name;
```

```
    public ClientHandler(Socket socket) {
        this.clientSocket = socket;
    }
    @Override
    public void run() {
        try (
            BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true)
        ) {
            this.out = out;
            // Ask for the client's name
            out.println("Enter your name:");
            this.name = in.readLine();
            System.out.println(name + " has joined chat-room.");
            broadcast(name + " has joined chat-room.", this);
            String message;
            while ((message = in.readLine()) != null) {
                System.out.println(name + ": " + message);
                broadcast(name + ": " + message, this);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                clientHandlers.remove(this);
                clientSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
            System.out.println(name + " has left the chat-room.");
            broadcast(name + " has left the chat-room.", this);
        }
    }
    private void sendMessage(String message) {
        out.println(message);
    }
  }
}
```

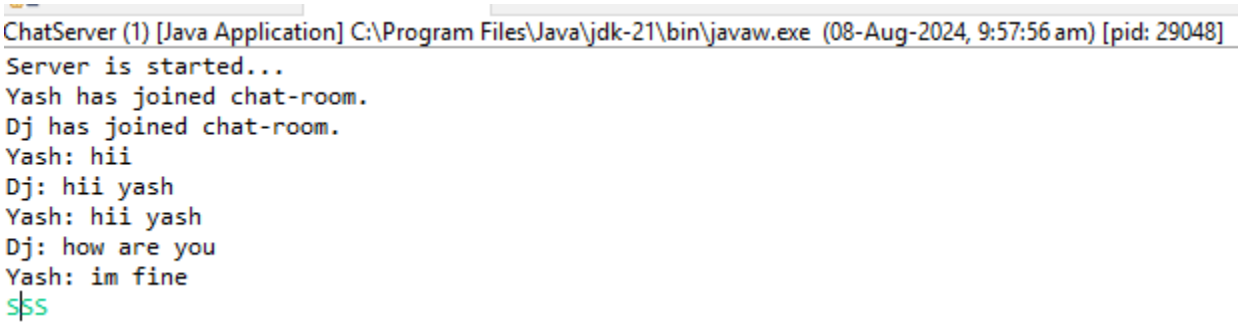
ChatClient.java

```
package com.chat.client;
import java.io.*;
import java.net.*;
import java.util.Scanner;
public class ChatClient {
```

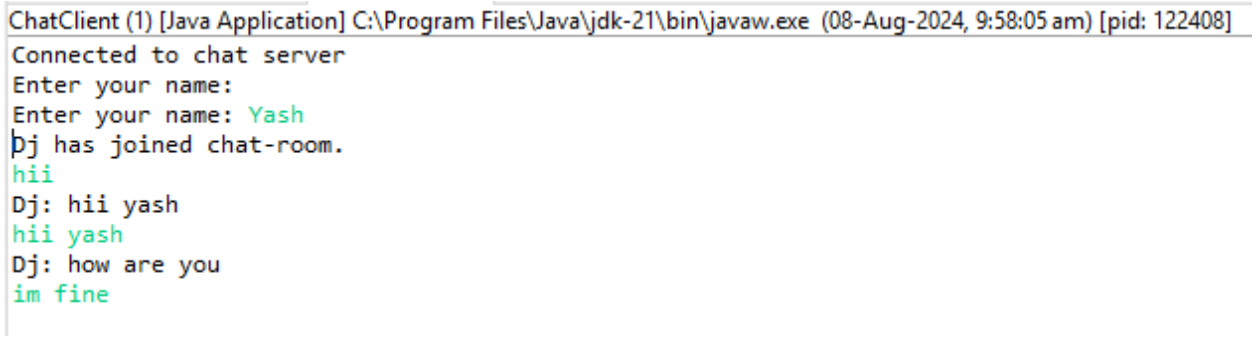
```
    private static final String HOST = "127.0.0.1"; // Server IP address
    private static final int PORT = 12345;         // Server port
    public static void main(String[] args) {
        try (Socket socket = new Socket(HOST, PORT)) {
            System.out.println("Connected to chat server");
            // Start a thread to listen for messages from the server
            new Thread(new ReceivedMessagesHandler(socket)).start();
            // Read messages from the console and send them to the server
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            Scanner scanner = new Scanner(System.in);
            // Enter user name
            System.out.print("Enter your name: ");
            String name = scanner.nextLine();
            out.println(name);
            while (scanner.hasNextLine()) {
                String message = scanner.nextLine();
                out.println(message);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    private static class ReceivedMessagesHandler implements Runnable {
        private Socket socket;
        public ReceivedMessagesHandler(Socket socket) {
            this.socket = socket;
        }
        @Override
        public void run() {
            try (BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream())))
{
                String message;
                while ((message = in.readLine()) != null) {
                    System.out.println(message);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Output:

Server



Client1:



Client2: