

## Department of Computer Engineering

Academic Term : January - May 2024

Class: B.E. (Computer) Semester VIII

Division: B

Subject Name: Distributed Systems CSC 801

<b>Name</b>	<b>Riya Rajesh Patil</b>
<b>Roll No.</b>	<b>9282</b>
<b>Experiment No.</b>	<b>1</b>
<b>Experiment Title</b>	<b>To implement multithreaded Server and clients</b>

**Aim** - To implement multithreaded server and client

### **Client.java**

```
import java.io.*;
import java.net.*;
import java.util.*;

// Client class
class Client {

// driver code

public static void main(String[] args)
{

// establish a connection by providing host and port
// number
try (Socket socket = new Socket("localhost", 1234)) {

// writing to server
PrintWriter out = new PrintWriter(
```

```

socket.getOutputStream(), true);
// reading from server
BufferedReader in
= new BufferedReader(new InputStreamReader(
socket.getInputStream()));
// object of scanner class
Scanner sc = new Scanner(System.in);
String line = null;
while (!"exit".equalsIgnoreCase(line)) {
// reading from user
line = sc.nextLine();
// sending the user input to server
out.println(line);
out.flush();
// displaying server reply
System.out.println("Server replied "
+ in.readLine());
}
// closing the scanner object
sc.close();
}
catch (IOException e) {
e.printStackTrace();
}
}
}

```

### **Server.java**

```

import java.io.*;

```

```
import java.net.*;

// Server class

class Server {

public static void main(String[] args)

{

ServerSocket server = null;

try {

// server is listening on port 1234

server = new ServerSocket(1234);

server.setReuseAddress(true);

// running infinite loop for getting

// client request

while (true) {

// socket object to receive incoming client // requests

Socket client = server.accept();

// Displaying that new client is connected // to server

System.out.println("New client connected"

+ client.getInetAddress()

.getHostAddress());

// create a new thread object

ClientHandler clientSock

= new ClientHandler(client);

// This thread will handle the client

// separately

new Thread(clientSock).start();

}

}

catch (IOException e) {

e.printStackTrace();

}
```

```

}
finally {
    if (server != null) {
        try {
            server.close();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}

// ClientHandler class

private static class ClientHandler implements Runnable { private final Socket
clientSocket;

// Constructor

public ClientHandler(Socket socket)
{
    this.clientSocket = socket;
}

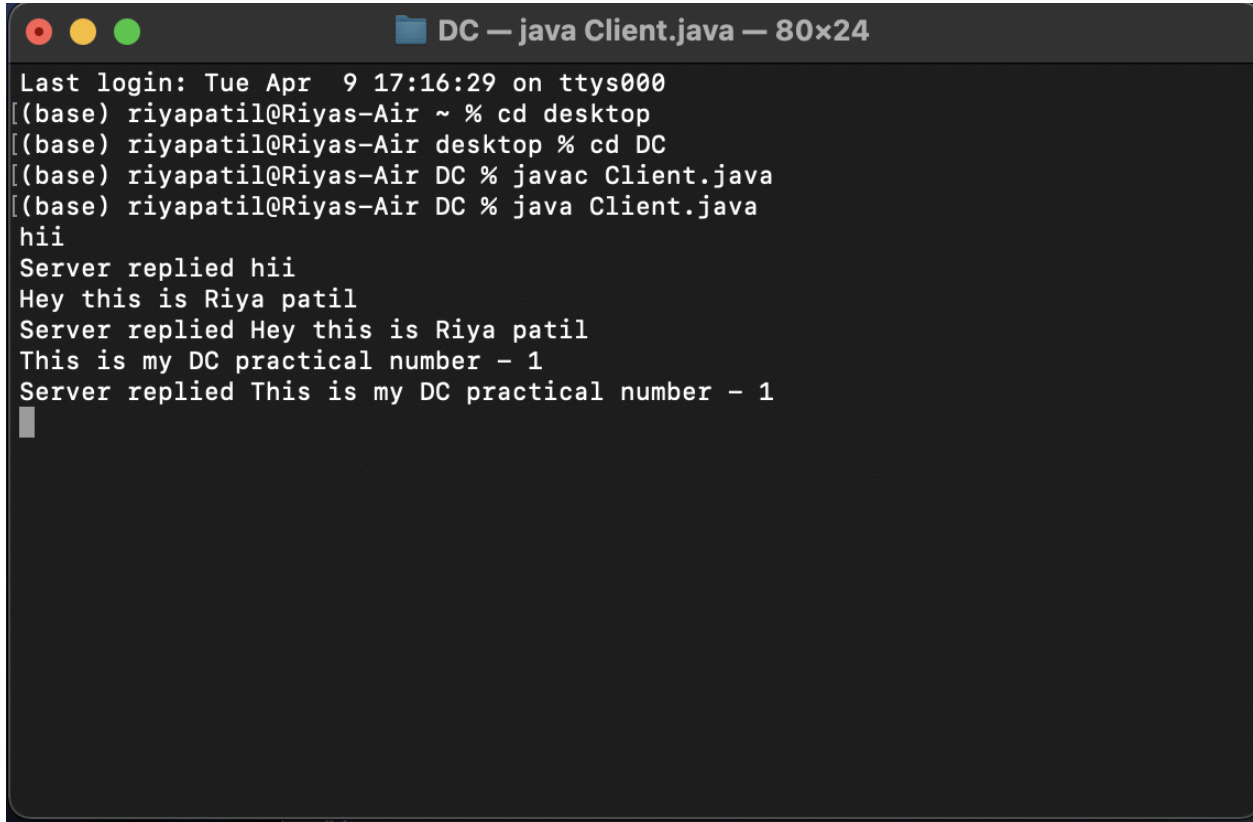
public void run()
{
    PrintWriter out = null;
    BufferedReader in = null;
    try {
        // get the outputstream of client
        out = new PrintWriter(
            clientSocket.getOutputStream(), true);
        // get the inputstream of client

```

```
in = new BufferedReader(
    new InputStreamReader(
        clientSocket.getInputStream()));
String line;
while ((line = in.readLine()) != null) {
    // writing the received message from
    // client
    System.out.printf(
        " Sent from the client: %s\n",
        line);
    out.println(line);
}
}
catch (IOException e) {
    e.printStackTrace();
}
finally {
    try {
        if (out != null) {
            out.close();
        }
        if (in != null) {
            in.close();
        }
        clientSocket.close();
    }
}
catch (IOException e) {
    e.printStackTrace();
}
```

```
}  
}  
}  
}
```

### Output -



```
DC — java Client.java — 80x24  
Last login: Tue Apr  9 17:16:29 on ttys000  
[(base) riyapatil@Riyas-Air ~ % cd desktop  
[(base) riyapatil@Riyas-Air desktop % cd DC  
[(base) riyapatil@Riyas-Air DC % javac Client.java  
[(base) riyapatil@Riyas-Air DC % java Client.java  
hii  
Server replied hii  
Hey this is Riya patil  
Server replied Hey this is Riya patil  
This is my DC practical number - 1  
Server replied This is my DC practical number - 1  
█
```

```
DC — java server.java — 80x24
Last login: Tue Apr  9 17:12:45 on ttys000
(base) riyapatil@Riyas-Air ~ % cd desktop
(base) riyapatil@Riyas-Air desktop % cd DC
(base) riyapatil@Riyas-Air DC % javac server.java
(base) riyapatil@Riyas-Air DC % java server.java
hello
New client connected127.0.0.1
hii
Sent from the client: hii
Sent from the client: Hey this is Riya patil
Sent from the client: This is my DC practical number - 1
```

## Conclusion -

Overall, this code demonstrates the basic functionality of a multithreaded client-server communication system in Java, allowing multiple clients to interact with a single server concurrently. The client class establishes a connection to the server, allowing users to send messages to the server and receive responses. It continuously prompts the user for input until the user types "exit". The server class accepts client connections and creates a new thread to handle each client separately. It echoes back the received messages to the clients. In my output the server prints the message indicating that the new client has been connected. The client sends the hii message to the server and then the server replies with the hii message.

## Postlab -

### 1. What are the advantages of a Multithreaded Server?

1. Concurrency: Multithreaded servers can handle multiple client connections concurrently. Each client connection is managed by a separate thread, allowing the server to serve multiple clients simultaneously. This concurrency improves the server's

responsiveness and scalability, especially in environments with a high volume of client requests.

2. Improved Performance: By utilizing multiple threads, a multithreaded server can leverage the available CPU cores more effectively. This results in better utilization of system resources and improved overall performance, as multiple tasks can be executed in parallel.

3. Responsiveness: Multithreaded servers can respond to client requests more quickly, as each client connection is handled independently in its own thread. This reduces the risk of blocking or delaying other clients' requests, leading to a more responsive and efficient system.

4. Resource Efficiency: Multithreaded servers allow efficient utilization of system resources by multiplexing client connections across multiple threads. This means that idle threads can be used to handle new client connections, minimizing resource wastage and maximizing throughput.

5. Scalability: Multithreaded servers are inherently scalable, as they can dynamically adjust the number of threads to handle increasing client loads. This scalability allows the server to accommodate a growing number of clients without significant degradation in performance.

## **2. With an example, explain the concept of multithreaded clients.**

Let's consider an example of a chat application where multiple clients can connect to a server and exchange messages with each other. In this scenario, a multithreaded client would allow each client to perform tasks concurrently, such as sending and receiving messages while also allowing the user to interact with the application's graphical user interface (GUI) without blocking.

Here's how the concept of multithreaded clients would work in this scenario:

1. Client Initialization: Each client instance is created as a separate thread. When a user launches the chat application, a new client thread is created to handle the client's interactions with the server.

2. User Interaction: The main thread of the client application is responsible for handling user interactions, such as typing messages, clicking buttons, or navigating the GUI. When a user sends a message or performs an action, the main thread captures the event and delegates the corresponding task to the appropriate worker thread.



3. Sending Messages: When a user types a message and clicks the "Send" button, the main thread extracts the message content and passes it to the client's worker thread responsible for sending messages. The worker thread then sends the message to the server over the network.

4. Receiving Messages: Meanwhile, another worker thread is continuously listening for incoming messages from the server. When a new message arrives, the worker thread receives it from the server and updates the client's message display area in the GUI to show the incoming message.

5. Concurrent Execution: Both the sending and receiving tasks can run concurrently in their respective worker threads without blocking the main thread or each other. This allows the client application to maintain responsiveness even while performing network I/O operations.