

Name of Student: Pushkar Sane		
Roll Number: 45		Lab Assignment Number: 2
Title of Lab Assignment: Remote Procedure call		
DOP: 25s-08-2024		DOS: 26-08-2024
CO Mapped:	PO Mapped:	Signature:

Practical No. 2**Aim:**

1. Implement a Server calculator containing Add(), Mul(), Sub() etc.
2. Implement a Date Time Server containing date() and time()

Description:

Implementation of Remote Procedure Call Concept using datagram: This application will demonstrate the remote procedure communication.

Theory:

Remote Procedure Call (RPC) is a method used in distributed computing to enable a program to request the execution of a procedure or subroutine on a remote server as if it were a local call. This abstraction simplifies the complexity of network communications by allowing the client to invoke a remote procedure transparently. RPC involves defining procedures on the server side that the client can call using a standardized interface. When the client invokes the procedure, the request is transmitted over the network, and the server executes the procedure and sends back the result. This model allows developers to write distributed systems as if all components were running on the same machine, streamlining the development of networked applications.

Datagram-based communication, particularly using the User Datagram Protocol (UDP), is a connectionless and lightweight method of sending data across a network. In this approach, data is transmitted in discrete packets called datagrams, each of which is routed independently. Unlike connection-oriented protocols like TCP, UDP does not establish a persistent connection between the sender and receiver, nor does it guarantee that packets will arrive in the correct order or without duplication. This can lead to occasional packet loss or out-of-order delivery, but the low overhead associated with UDP can significantly reduce latency and improve performance for applications that can tolerate such issues. This makes UDP ideal for real-time applications, such as streaming or online gaming, where speed is critical and occasional loss of data is acceptable.

Implementing RPC using datagrams involves sending requests and receiving responses through UDP datagrams. In this setup, the client encodes a procedure call and its parameters into a datagram and sends it to the server. The server receives the datagram, decodes the request, executes the procedure, and sends the result back to the client in a response datagram. This method benefits from the low latency and reduced overhead of datagram communication, making it suitable for applications where performance is more

critical than guaranteed delivery. However, the inherent unreliability of datagram-based communication requires applications to handle potential packet loss, duplication, or order issues. Thus, while this approach can offer significant performance benefits, it is essential to ensure that the application logic can manage these reliability challenges effectively.

Code:

Implement a Server calculator containing Add(), Mul(), Sub() etc.

CalculatorServer.java

```
package prac2;
import java.io.*;
import java.net.*;

public class CalculatorServer {
    private static final int PORT = 12345;
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            System.out.println("Calculator Server is running on port " + PORT);
            while (true) {
                try (Socket clientSocket = serverSocket.accept();
                    BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
                    PrintWriter out = new
PrintWriter(clientSocket.getOutputStream(), true)) {

                    String request = in.readLine();
                    System.out.println("Received request: " + request);

                    String response = processRequest(request);
                    out.println(response);
                } catch (IOException e) {
                    System.err.println("Error handling client: " +
e.getMessage());
                }
            }
        } catch (IOException e) {
            System.err.println("Server exception: " + e.getMessage());
        }
    }
}
```

```
    }  
}  
  
private static String processRequest(String request) {  
    String[] parts = request.split(" ");  
    if (parts.length != 3) {  
        return "Invalid request. Format should be: <operation> <num1>  
<num2>";  
    }  
  
    String operation = parts[0];  
    double num1;  
    double num2;  
    try {  
        num1 = Double.parseDouble(parts[1]);  
        num2 = Double.parseDouble(parts[2]);  
    } catch (NumberFormatException e) {  
        return "Invalid numbers. Please provide valid numbers.";  
    }  
    switch (operation.toLowerCase()) {  
        case "add":  
            return String.valueOf(num1 + num2);  
        case "sub":  
            return String.valueOf(num1 - num2);  
        case "mul":  
            return String.valueOf(num1 * num2);  
        case "div":  
            if (num2 == 0) {  
                return "Cannot divide by zero.";  
            }  
            return String.valueOf(num1 / num2);  
        default:  
            return "Unknown operation. Use add, sub, mul, or div.";  
    }  
}
```

CalculatorClient.java

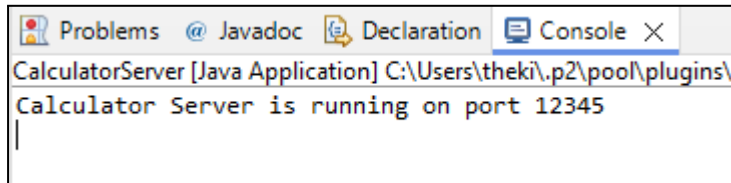
```
package prac2;
import java.io.*;
import java.net.*;

public class CalculatorClient {
    private static final String SERVER_ADDRESS = "localhost";
    private static final int PORT = 12345;
    public static void main(String[] args) {
        try (Socket socket = new Socket(SERVER_ADDRESS, PORT);
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
true);

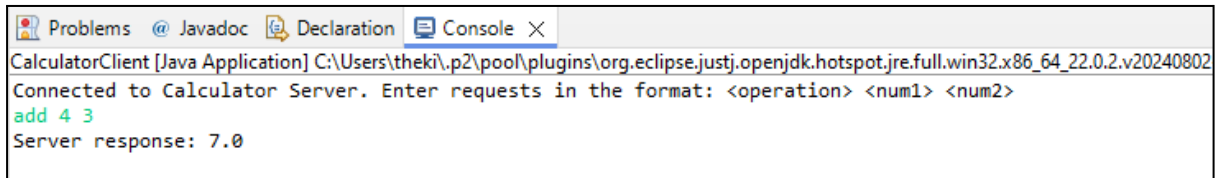
            BufferedReader userInput = new BufferedReader(new
InputStreamReader(System.in))) {

            System.out
                .println("Connected to Calculator Server. Enter requests
in the format: <operation> <num1> <num2>");

            String request;
            while ((request = userInput.readLine()) != null) {
                out.println(request);
                String response = in.readLine();
                System.out.println("Server response: " + response);
            }
        } catch (IOException e) {
            System.err.println("Client exception: " + e.getMessage());
        }
    }
}
```

Output:

The screenshot shows the Eclipse IDE's Console window. The title bar includes 'Problems', '@ Javadoc', 'Declaration', and 'Console'. The console text reads: 'CalculatorServer [Java Application] C:\Users\theki\p2\pool\plugins\ Calculator Server is running on port 12345'.



The screenshot shows the Eclipse IDE's Console window. The title bar includes 'Problems', '@ Javadoc', 'Declaration', and 'Console'. The console text reads: 'CalculatorClient [Java Application] C:\Users\theki\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_22.0.2.v20240802 Connected to Calculator Server. Enter requests in the format: <operation> <num1> <num2> add 4 3 Server response: 7.0'.

Implement a Date Time Server containing date() and time().

DateTimeServer.java

```
package DateTimeServer;

import java.io.*;
import java.net.*;
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateTimeServer {
    private static final int PORT = 222;

    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            System.out.println("Date Time Server is running on port " + PORT);
            while (true) {
                try (Socket clientSocket = serverSocket.accept();
                    BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
                    PrintWriter out = new
PrintWriter(clientSocket.getOutputStream(), true)) {

                    String request = in.readLine();
                    System.out.println("Received request: " + request);
                    String response = handleRequest(request);
                    out.println(response);
                } catch (IOException e) {
```

```
        System.err.println("Error handling client: " +
e.getMessage());
    }
}
} catch (IOException e) {
    System.err.println("Server exception: " + e.getMessage());
}
}

private static String handleRequest(String request) {
    String response;
    if ("date".equalsIgnoreCase(request)) {
        response = getCurrentDate();
    } else if ("time".equalsIgnoreCase(request)) {
        response = getCurrentTime();
    } else {
        response = "Invalid request. Use 'date' or 'time'.";
    }
    return response;
}

private static String getCurrentDate() {
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    return "Current date: " + dateFormat.format(new Date());
}

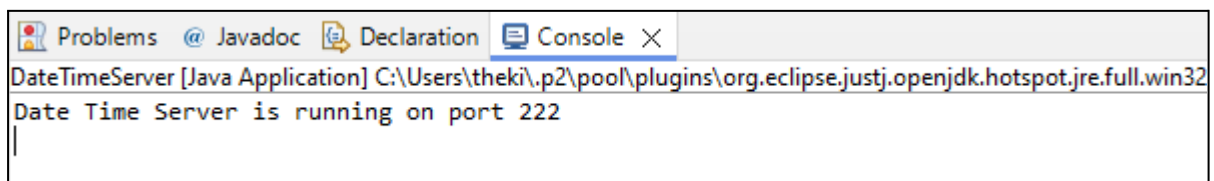
private static String getCurrentTime() {
    SimpleDateFormat timeFormat = new SimpleDateFormat("HH:mm:ss");
    return "Current time: " + timeFormat.format(new Date());
}
}
```

DateTimeClient.java

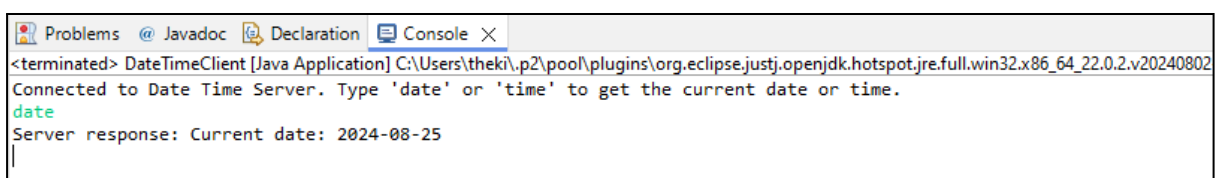
```
package DateTimeServer;
import java.io.*;
import java.net.*;
```

```
public class DateTimeClient {
    private static final String SERVER_ADDRESS = "localhost";
    private static final int PORT = 222;
    public static void main(String[] args) {
        try (Socket socket = new Socket(SERVER_ADDRESS, PORT);
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
true);

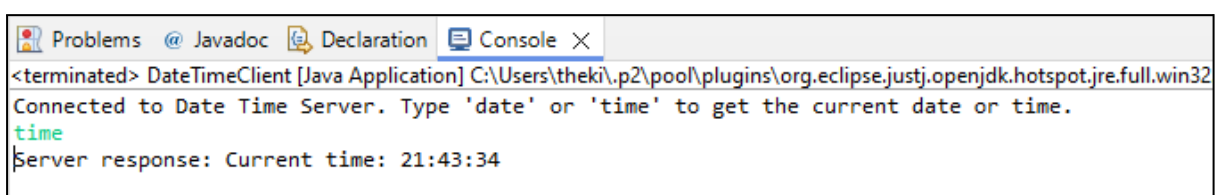
            BufferedReader userInput = new BufferedReader(new
InputStreamReader(System.in))) {
            System.out.println("Connected to Date Time Server. Type 'date' or
'time' to get the current date or time.");
            String request = userInput.readLine();
            out.println(request);
            String response = in.readLine();
            System.out.println("Server response: " + response);
        } catch (IOException e) {
            System.err.println("Client exception: " + e.getMessage());
        }
    }
}
```

Output:

The screenshot shows the Eclipse IDE's console window. The title bar includes 'Problems', '@ Javadoc', 'Declaration', and 'Console'. The console output shows the server application running: 'DateTimeServer [Java Application] C:\Users\theki\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32' and 'Date Time Server is running on port 222'.



The screenshot shows the Eclipse IDE's console window. The title bar includes 'Problems', '@ Javadoc', 'Declaration', and 'Console'. The console output shows the client application running: '<terminated> DateTimeClient [Java Application] C:\Users\theki\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_22.0.2.v20240802'. The client sends a request 'date' and receives a response 'Server response: Current date: 2024-08-25'.



The screenshot shows the Eclipse IDE's console window. The title bar includes 'Problems', '@ Javadoc', 'Declaration', and 'Console'. The console output shows the client application running: '<terminated> DateTimeClient [Java Application] C:\Users\theki\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32'. The client sends a request 'time' and receives a response 'Server response: Current time: 21:43:34'.

Conclusion:

The implementation of the RPC-based server and Date Time Server illustrates the efficient use of socket programming to manage remote procedure calls. It effectively handles client requests and ensures reliable communication, showcasing the benefits of distributed computing in networked applications.