External Project Report on
Computer Networking (CSE3034)

Develop Console-based Application for a Scientific Calculator using Client-Server Architecture. (Socket Programming)

Submitted by

Piyush Kumar	Regd. No. 2141011136
Harsh Raj	Regd. No. 2141018030
Bibhukalyan Biswal	Regd. No. 2141002123
Mannat Mohapatra	Regd. No. 2141013263

B. Tech. CSE 5th Semester (Section 2141019)

INSTITUTE OF TECHNICAL EDUCATION AND RESEARCH

(FACULTY OF ENGINEERING)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE UNIVERSITY), BHUBANESWAR, ODISHA

Declaration

We, the undersigned students of B. Tech. of **Computer Science and Engineering** department hereby declare that we own the full responsibility for the information, results etc. provided in this PROJECT titled **Develop a Console-based Application for a Scientific Calculator using a Client-Server Architecture (Socket Programming)** submitted to **Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar** for the partial fulfillment of the subject **Computer Networking (CSE 3034)**. We have taken care in all respects to honor the intellectual property right and have acknowledged the contribution of others for using them in academic purpose and further declare that in case of any violation of intellectual property right or copyright we, as the candidate(s), will be fully responsible for the same.

Piyush Kumar **2141011136**	**Harsh Raj** **2141018030**
Bibhukalyan Biswal **2141002123**	**Mannat Mohapatra** **2141013263**

DATE: 13 JANUARY 2023

PLACE: BHUBANESWAR

Abstract

This project delves into the development of a robust and versatile scientific calculator utilizing the client-server architecture in Java. Moving beyond traditional desktop models, the application leverages the distributed power of this architecture, dividing functionalities between dedicated clientand server components. This design choice promises enhanced performance, improved resource allocation, and potential scalability for future expansion.

The client acts as the user interface, accepting expressions and mathematical operations through a console interface. It then transmits these requests to the server, acting as the computational engine. The server parses the received expression, performs the required calculations, and sends the results back to the client for display.

This report details the entire development process, encompassing problem identification, methodological approach, implementation specifics, result analysis with interpretations, and a conclusive remark on the project's achievements and potential future work.

Contents

1. Introduction

This project presents a novel approach to the classic scientific calculator, transcending the limitations of standalone applications by embracing the flexibility and power of a client-server architecture in Java. We move away from the traditional one-machine model and harness the distributed advantage of dedicated client and server components. This design philosophy offers key benefits:

- Enhanced performance: Calculations occur on the server, freeing up the client's resources for smoother user interaction and efficient input handling.
- Improved resource allocation: The server shoulders the computational burden, allowing for optimal resource utilization on both sides.
- Scalability potential: The architecture provides a solid foundation for future expansion, enabling the addition of new functionalities and accommodating increased user load.

The client acts as the user's portal, presenting a console interface for inputting complex expressions and mathematical operations. These requests are then relayed to the server, the brains of the operation. The server meticulously parses the received expression, applies the selected mathematical tools, and transmits thecomputed result back to the client for display.

This report meticulously documents the journey of this project, from identifying the problem and selecting the methodology to meticulously implementing the system, analyzing the results, and culminating in a conclusive evaluation of its success and potential for further development.

2. Problem Statement

The conventional scientific calculators available in the market often face limitations in handling complex mathematical computations due to their reliance on local processing resources. To address this, our project aims to develop a console-based scientific calculator with a client-server architecture, leveraging the power of distributed computing.

Users, through the console interface, will input complex mathematical expressions and scientific functions on the client side. The system will transmit these inputs to the server for computation. The server, equipped with an extensive set of scientific functionalities, processes the calculations and sends the results back to the client. The final output is then displayed on the client's console.

The key challenge lies in establishing efficient communication between the client and server components, ensuring seamless data exchange and timely response. Moreover, handling diverse mathematical operations, including trigonometric, logarithmic, and exponential functions, adds complexity to the server-side implementation.

Constraints include managing real-time communication between the client and server, ensuring data integrity during transmission, and addressing potential latency issues. Additionally, the server must efficiently handle concurrent requests from multiple clients. The project must adhere to these constraints to deliver a reliable and responsive scientific calculator that enhances computational capabilities beyond traditional local calculators.

3. Methodology

Algorithms Involved:

1. Communication Protocol Algorithm:

   Objective: Define the rules for communication between the client and server.
   Algorithm:

   1.1. Client-Side:
   - Accept user input for mathematical expressions.
   - Validate the input.
   - Create a message containing the operation and parameters.
   - Send the message to the server.

   1.2. Server-Side:
   - Receive the message from the client.
   - Validate the incoming message.
   - Extract the operation and parameters from the message.
   - Perform the specified operation.
   - Send the result back to the client.

2. Server-Side Scientific Functionalities Algorithm:

   Objective: Implement various scientific operations on the server.
   Algorithm:

   2.1. Perform Operation:
   - Accept the operation type and parameters.
   - Based on the operation type, execute the corresponding mathematical operation.
   - Return the result.

   2.2. Calculate Sine:
   - Accept an angle in radians as a parameter.
   - Use trigonometric algorithms to calculate the sine.
   - Return the result.

   2.3. Calculate Logarithm:
   - Accept a base and value as parameters.
   - Use logarithmic algorithms to calculate the logarithm.
   - Return the result.
     (Additional algorithms can be implemented for various scientific operations.)

3. Client Interface Algorithm:

   Objective: Develop a user-friendly console interface for interaction.

Algorithm:

3.1. Get User Input:
   - Prompt the user to input a mathematical expression.
   - Accept the input from the console.
   - Return the user input.

3.2. Display Result:
   - Accept the result from the server.
   - Print the result on the console.

4. Error Handling and Validation Algorithm:

Objective: Ensure data integrity and handle errors gracefully.
Algorithm:

4.1. Validate Input:
   - Check the validity of user input before sending it to the server.

4.2. Validate Message:
   - Verify the integrity of incoming messages from the client.

4.3. Handle Errors:
   - Implement mechanisms to handle errors, such as invalid input or communication failures.
   - Provide appropriate error messages to the user.

These algorithms collectively form the foundation for the development of a console-based scientific calculator with a client-server architecture in Java. The integration of these algorithms will result in a robust system capable of performing complex scientific calculations in a distributed computing environment.

The development of the console-based scientific calculator with a client-server architecture involves several key steps, including designing the communication protocol, implementing server-side scientific functionalities, and creating an interactive client interface. The following outlines the methodology along with pseudocode for critical components:

1. Communication Protocol Design:
   - Define the structure of messages exchanged between the client and server.
   - Establish rules for data validation and error handling.

Pseudocode:

```
// Client-Side Pseudocode
function sendRequest(operation, parameters):
    validateInput(parameters)
    message = createMessage(operation, parameters)
    sendToServer(message)

// Server-Side Pseudocode
function receiveRequest(message):
    validateMessage(message)
    operation, parameters = extractMessage(message)
    result = performOperation(operation, parameters)
    sendToClient(result)
```

2. Server-Side Scientific Functionalities:
   - Implement a comprehensive set of scientific operations, including trigonometric, logarithmic, and exponential functions.
   - Ensure efficient handling of complex mathematical expressions.

   Pseudocode:

```
// Server-Side Pseudocode
function performOperation(operation, parameters):
    switch operation:
        case "sin":
            return calculateSin(parameters)
        case "log":
            return calculateLog(parameters)
        // Additional cases for various operations

function calculateSin(parameters):
    // Perform sine calculation
    // Return result
```

3. Client Interface:

   - Develop a user-friendly console interface for inputting mathematical expressions.
   - Display results received from the server on the client's console.

   Pseudocode:

```
// Client-Side Pseudocode
function getUserInput():
    // Get user input from the console
    return userInput

function displayResult(result):
    // Display the result on the console
    print("Result:", result)
```

4. Error Handling and Validation:

   - Implement mechanisms for handling input validation and error reporting.
   - Ensure robust error handling for communication failures.

   Pseudocode:

```
// Common Pseudocode
function validateInput(parameters):
    // Validate user input before sending to the server

function validateMessage(message):
    // Validate incoming message from the client
```

By following this methodology and integrating the pseudocode into the actual Java implementation, we aim to create a reliable and efficient console-based scientific calculator with distributed computing capabilities.

4. Implementation

The complete implementation of a console-based scientific calculator using a client-server architecture in Java involves several components, including the server, client, and the communication protocol. Below is a simplified example illustrating the implementation of the key aspects. Note that this is a basic representation, and a production-ready system would require additional features, error handling, and optimization.

server/Server:

```java
package Calc.server;

import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

public class Server {
   public static void main(String[] args) throws IOException {
       ServerSocket serverSocket = new ServerSocket(5006);
       System.out.println("Server started. Listening on Port 5006");

       while (true) {
           try (Socket socket = serverSocket.accept();
                PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
                                                     BufferedReader  in  =  new  BufferedReader(new
InputStreamReader(socket.getInputStream()))) {

               String inputLine;
               while ((inputLine = in.readLine()) != null) {
                   System.out.println("Received message from client: " + inputLine);
                   // Add calculation logic here
                   String result = calculate(inputLine);
                   out.println("Echo from server: " + result);
               }
           } catch (IOException e) {
                 System.out.println("Exception caught when trying to listen on port 5000 or listening
for a connection");
               System.out.println(e.getMessage());
           }
       }
   }
```

```java
// Inside the Server class

public static String calculate(String input) {
    // Validate input
    if (input == null || input.trim().isEmpty()) {
        return "Invalid input";
    }

    String[] parts = input.split(" ");
    // if (parts.length != 3) {
    //      return "Invalid format. Expected format: operation operand1 operand2";
    // }

    String operation = parts[1];
    double num1, num2 = 0;
    try {
        if(parts.length == 2){
            operation = parts[0];
            num1 = Double.parseDouble(parts[1]);
        }
        else{
        num1 = Double.parseDouble(parts[0]);
        num2 = Double.parseDouble(parts[2]);
        }
    } catch (NumberFormatException e) {
        return "Invalid numbers: " + parts[0] + ", " + parts[2];
    }

    switch (operation) {
    case "+":
            return String.valueOf(num1 + num2);
    case "-":
            return String.valueOf(num1 - num2);
        // Add more cases for multiplication, division, etc.
    case "*":
        return String.valueOf(num1 * num2);
    case "%":
        if (num2 == 0) return "Error: Division by zero";
        return String.valueOf(num1 / num2);
    case "pow":
        return String.valueOf(Math.pow(num1, num2));
    case "sqrt":
        if (num1 < 0) return "Error: Square root of negative number";
        return String.valueOf(Math.sqrt(num1));
        case "log":
            if (num1 <= 0) return "Error: Log of non-positive number";
            return String.valueOf(Math.log(num1));
        case "log10":
            if (num1 <= 0) return "Error: Log10 of non-positive number";
            return String.valueOf(Math.log10(num1));
        case "sin":
            return String.valueOf(Math.sin(Math.toRadians(num1)));
        case "cos":
            return String.valueOf(Math.cos(Math.toRadians(num1)));
        case "tan":
            return String.valueOf(Math.tan(Math.toRadians(num1)));
        case "exp":
        return String.valueOf(Math.exp(num1));
        case "ln":
        if (num1 <= 0) return "Error: Ln of non-positive number";
        return String.valueOf(Math.log(num1));
        case "arcsin":
        if (num1 < -1 || num1 > 1) return "Error: Arcsin out of range";
        return String.valueOf(Math.toDegrees(Math.asin(num1)));
        case "arccos":
        if (num1 < -1 || num1 > 1) return "Error: Arccos out of range";
        return String.valueOf(Math.toDegrees(Math.acos(num1)));
        case "arctan":
        return String.valueOf(Math.toDegrees(Math.atan(num1)));
        case "sinh":
        return String.valueOf(Math.sinh(num1));
        case "cosh":
        return String.valueOf(Math.cosh(num1));
        case "tanh":
        return String.valueOf(Math.tanh(num1));
        case "percent":
        return String.valueOf(num1 / 100);
        case "abs":
        return String.valueOf(Math.abs(num1));
        default:
            return "Invalid operation";
        }
    }
}
```

client/Client:

```java
package Calc.Client;

import java.io.*;
import java.net.Socket;
import java.util.Scanner;

public class Client {
   public static void main(String[] args) throws IOException {
       try (Socket socket = new Socket("localhost", 5006);
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            Scanner scanner = new Scanner(System.in)) {

           String userInput;
           while (true) {
               System.out.println("Enter operation and numbers (e.g., '5 + 3' or 'sqrt 4'):");
               userInput = scanner.nextLine();
               if ("exit".equalsIgnoreCase(userInput)) {
                   break;
               }
               out.println(userInput);
               System.out.println("Server response: " + in.readLine());
           }
       } catch (IOException e) {
           System.out.println("An error occurred: " + e.getMessage());
           e.printStackTrace();
       }
   }
}
```

This basic example establishes a connection between a client and server, with the server echoing back the received message. To complete the implementation, you would extend the `Server` to process various mathematical operations and enhance the client to handle more sophisticated user inputs. Additionally, error handling and security measures should be incorporated for a robust implementation.

5. Results & Interpretation

Results and Interpretation:

2.1. Output - Server.java

```
Server started. Listening on Port 5006
Received message from client: sqrt 4
```

2.1. Interpretation:

- The server has started successfully and is now waiting for client connections on the specified port.

2.2. Output - Client.java

```
Enter operation and numbers (e.g., '5 + 3' or 'sqrt 4'):
sqrt 4
Server response: Echo from server: 2.0
Enter operation and numbers (e.g., '5 + 3' or 'sqrt 4'):
```

Interpretation:

- The client has successfully connected to the server.
- The user has entered the mathematical expression log(e).
- The server responds with an error, indicating that the expression contains an invalid input (log(e) is not a supported format in the current implementation).

Explanation:

The server is actively waiting for incoming connections, as indicated by the message "Server is running. Waiting for clients...". On the client side, after establishing a connection with the server, the user is prompted to enter mathematical expressions.
In the provided example, the user attempts to calculate the logarithm of the mathematical constant 'e' using the expression log(e). However, the server returns an error because the current implementation does not support this specific format. The server-side logic could be extended to handle more complex expressions, including constants like 'e' and mathematical functions like logarithms.
To improve the system, additional error handling and support for a broader range of mathematical expressions could be implemented on both the client and server sides. This would enhance the calculator's functionality and user experience.

6. Conclusion

In conclusion, the development of a console-based scientific calculator using a client-server architecture in Java has been successfully implemented, demonstrating the potential of distributed computing in enhancing computational capabilities. The project aimed to overcome the limitations of local calculators by offloading complex mathematical computations to a remote server, thereby enabling more efficient and scalable calculations.

The client-server model facilitated seamless communication between the user interface and the computational engine. While the initial implementation showcased basic functionality, such as connection establishment and message exchange, there is ample room for further refinement and feature enhancement. The server successfully received user input, processed requests, and returned responses to the client.

The project highlighted the importance of a well-defined communication protocol for effective data exchange between the client and server components. Additionally, the server-side implementation included rudimentary scientific functionalities, showcasing the potential for further expansion to support a broader range of mathematical expressions and operations.

Despite the success of the initial implementation, there are areas for improvement. Enhancements could include a more extensive set of supported mathematical functions, robust error handling to address various scenarios, and the ability to handle concurrent client connections efficiently. Future iterations of the project could also explore incorporating graphical user interfaces (GUIs) for a more user-friendly experience.

In summary, the console-based scientific calculator demonstrates the feasibility of utilizing distributed computing for scientific applications. This project lays the groundwork for future development, encouraging exploration of advanced mathematical operations and considerations for real-world deployment, ultimately contributing to the evolution of scientific calculators with enhanced computational capabilities.

References

[1] Tannenbaum, A. S. (2003). "Computer Networks." Pearson India.
[2] Harold, E. R. (2005). "Java Network Programming." O’Reilly (Shroff Publishers).
[3] Oracle Corporation. (2021). "Java™ Platform, Standard Edition 11 Documentation." [Online]. Available: https://docs.oracle.com/en/java/javase/11/
[4] Stevens, W. R., Fenner, B., & Rudoff, A. M. (2003). "UNIX Network Programming, Volume 1: The Sockets Networking API." Addison-Wesley.