JOBSHEET 2

# Interoperabilitas “PEMROGRAMAN REMOTE PROCEDURE

**CALL (RPC)”**



Disusun oleh:

Ryvannio Satria Nugroho

4.33.22.0.25

TI-2A

**Program Studi Teknologi Rekayasa Komputer Jurusan Teknik Elektro**

# POLITEKNIK NEGERI SEMARANG 2024

### Tujuan Instruksional Khusus

Setelah menyelesaikan praktikkum ini, mahasiswa diharapkan :

* 1. Memahami konsep RPC
  2. Menjelaskan pengunaan RPC
  3. Mengimplementasikan RPC dengan Bahasa pemrograman Java

### Dasar Teori

RPC adalah sebuah metode yang memungkinkan kita untuk mengakses sebuah prosedur yang berada di komputer lain. Untuk dapat melakukan ini sebuah *server* harus menyediakan layanan *remote procedure.* Pendekatan yang dilakukan adalah sebuah *server* membuka *socket,* lalu menunggu *client* yang meminta prosedur yang disediakan *server.* RPC masih menggunakan cara primtif dalam pemrograman yaitu menggunakan paradigma *procedural programming*.

### Tujuan

RPC digunakan untuk administrasi sistem sehingga seorang administrator jaringan dapat mengakses sistemnya dan mengelola sistemnya darimna saja, selama sistemnya terhubung ke jaringan.

### Socket

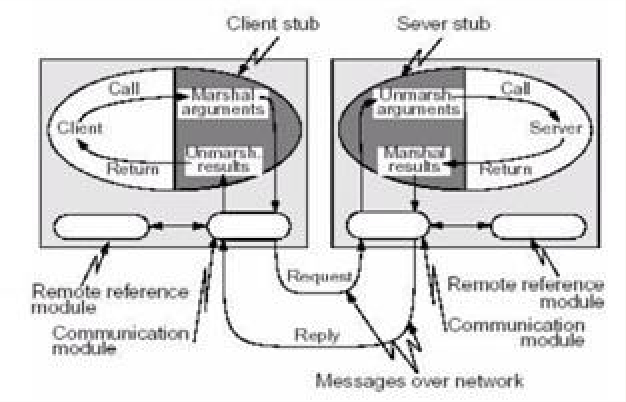
RPC menggunakan socket untuk berkomunikasi dengan proses lainnya.

### Cara Kerja RPC

Tiap prosedur yang dipanggil dalam RPC, maka proses ini harus berkoneksi dengan server remote dengan mengirimkan semua parameter yang dibutuhkan, menunggu balasan dari server dan melakukan proses kemudian selesai. Proses di atas disebut juga dengan stub pada sisi klien. Sedangkan Stub pada sisi server adalah proses menunggu tiap message yang berisi permintaan mengenai prosedur tertentu.

### Implementasi RPC

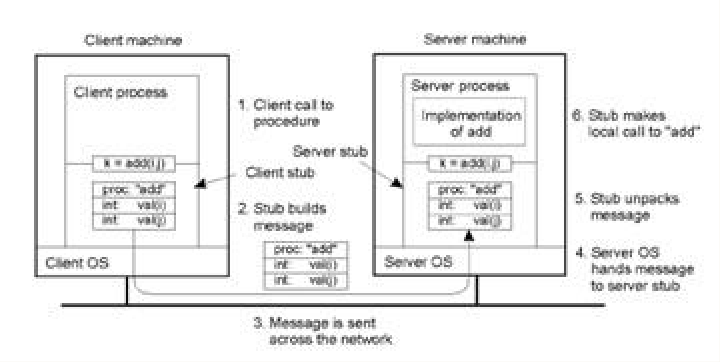
Untuk proses nya kurang lebih sama dengan RMI. Kalau RMI kita mengenal proxy dan skeleton, pada RPC dikenal dengan Stub( Client stub dan Server stub).



Gambar Ilustrasi Implementasi RPC

Remote Reference Modul dan Communication Modul berada pada tatanan sistem operasi. Contoh implementasi adalah Sun Microsystems Open Network Computing (ONC) : RPC specification, XDR (eXternal Data Representation) standard, UDP atau TCP transport protocol. Xerox Courier : RPC model, Data representation standard, XNS (Xerox Network Systems) SPP (Sequenced Packet Protocol) sbg transport protocol, Apollo s Network Computing Architecture (NCA), RPC protocol, NDR (Network Data Representation).

### Langkah-langkah dalam RPC



Prosedur client memanggil client stub

Client stub membuat pesan dan memanggil OS client OS client mengirim pesan ke OS server

OS server memberikan pesan ke server stub

Server stub meng-unpack parameter-parameter untuk memanggil server Server mengerjakan operasi, dan mengembalikan hasilnya ke server stub Server stub meng-pack hasil tersebut dan memanggil OS server

OS server mengirim pesan (hasil) ke OS client

OS client memberikan pesan tersebut ke client stub

Client stub meng-unpack hasil dan mengembalikan hasil tersebut ke client

## ALAT DAN BAHAN

Dalam penggunaan modul praktikum ini, anda harus menyediakan beberapa perangkat dan keperluan pendukung seperti:

* 1. Laptop atau Komputer Desktop
  2. Aplikasi Java Netbeans, Eclipse
  3. Akses Internet

## PRAKTIKUM

Pada praktikum kali ini, membuat dua aplikasi yang terdiri dari aplikasi server dan aplikasi client. Dua program tersebut yakni program client (RPCClient.java) dan server (RPCServer.java). Di bawah ini memperlihatkan bagaimana RPC (*Remote procedure call*) disimplementasikan di dalam Bahasa Pemrogaraman Java menggunakan pustaka bawaan Socket, SocketServer dan yang terkait.

#### RPCServer.java:

import java.io.BufferedReader; import java.io.IOException; import java.io.InputStreamReader; import

java.io.PrintStream; import java.net.InetAddress; import java.net.ServerSocket

; import java.net.Socket;

public class RPCServer {

private final ServerSocket serverSocket;

@SuppressWarnings("CallToThreadStartDuringObjectConstruction") public RPCServer(int port) throws IOException {

serverSocket = new ServerSocket(port);

String localIP = InetAddress.getLocalHost().getHostAddress(); System.out.println("Server is running on " + localIP + ":" + port); while (true)

{

Socket rpcClient = serverSocket.accept(); String address = rpcClient.getRemoteSocketAddress().toString(); System.out.println("New client connected : " + address);

new Thread(() -> { try {

addHook(rpcClient);

} catch (IOException ex) {

System.err.println("Client disconnected " + address);

}

}).

sta rt(

);

}

}

private void addHook(Socket rpcClient) throws IOException { BufferedReader reader = new

BufferedReader(new

InputStreamReader(rpcClient.getInputStr eam())); String line;

while ((line = reader.readLine()) != null) { System.out.println("Client request : " + line);

String[] commands = line.split(":", 3);

int result;

int operand1 =

Integer.parseInt(commands[1]); int operand2 = Integer.parseInt(commands[2]);

String message = "";

switch (commands[0]) {

case "add":

result = (operand1 + operand2);

message = operand1 + " + " + operand2 + " = " + result; break;

case "sub":

result = (operand1 - operand2);

message = operand1 + " - " + operand2 + " = " + result; break;

case "mul":

result = (operand1 \* operand2);

message = operand1 + " \* " + operand2 + " = " + result; break;

case "div":

result = (operand1 / operand2);

message = operand1 + " / " + operand2 + " = " + result; break;

case "mod":

result = (operand1 % operand2);

message = operand1 + " % " + operand2 + " = " + result; break;

}

PrintStream printStream = new PrintStream(rpcClient.getOutputStream(), true);

printStream.println(message);

}

}

public static void main(String[] args) throws Exception { RPCServer server = new RPCServer(3000);

}

}

#### RPCClient.java:

import java.io.BufferedReader; import java.io.IOException; import java.io.InputStreamReader; import

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

public class RPCClient {

private final PrintStream printStream; @SuppressWarnings("CallToThreadStartDuringObjectConstruction")

public RPCClient(String ipAddress, int port) throws IOException { Socket rpcClient = new Socket(ipAddress, port);

new Thread(()

-> {

try {

BufferedReader reader = new BufferedReader(new InputStreamReader(rpcClient.getInputStream(

))); String line;

while ((line = reader.readLine()) != null) { System.out.println("Server response : " + line);

System.out.print("\nCommands [add, sub, mul, div, mod, exit] : ");

}

} catch (IOException ex) { System.err.println("\nDisconnected!

!"); System.exit(0);

}

}).start();

printStream = new PrintStream(rpcClient.getOutputStream(), true);

}

public void sendMessage(String operation)

{ Scanner scan = new Scanner(System.in); System.out.print("\nEnter 1st number : "); int f1 = scan.nextInt();

System.out.print("Enter 2nd number : "); int s1 = scan.nextInt();

printStream.println(operation + ":" + f1 + ":" + s1);

}

public static void main(String[] args) {

try {

Scanner scan = new Scanner(System.in);

System.out.print("Enter server ip address :

"); String ipAddress = scan.nextLine();

System.out.print("Enter connection port : "); int port = scan.nextInt();

RPCClient client = new RPCClient(ipAddress, port); System.out.println("\nConnected to server\n");

System.out.print("Commands [add, sub, mul, div, mod, exit] : "); while

(true) {

scan = new Scanner(System.in);

String command = scan.nextLine();

if

(command.equals("exit ")) { System.exit(0);

}

client.sendMessage(comma nd); System.out.print("\n");

}

} catch (IOException ex) { System.err.println("\nUnable to connected!");

}

}

}

#### Server Output:

Server is running on 172.20.52.46:3000

New client connected : /172.20.52.46:52843 Client request : add:10:20

Client request : sub:50:20 Client request : mul:10:2 Client request : div:100:5 Client request : mod:1234:10

Client disconnected /172.20.52.46:52843

#### Client Output:

Enter server ip address : 172.20.52.42 Enter connection port : 3000

Connected to server

Commands [add, sub, mul, div, mod, exit] : add

Enter 1st number : 10

Enter 2nd number : 20

Server response : 10 + 20 = 30

sub

Enter 1st number : 50 Enter 2nd number : 20

Server response : 50 - 20 = 30

mul

Enter 1st number : 10 Enter 2nd number : 2

Server response : 10 \* 2 = 20

div

Enter 1st number : 100 Enter 2nd number : 5

Server response : 100 / 5 = 20

mod

Enter 1st number : 1234 Enter 2nd number : 10

Server response : 1234 % 10 = 4 exit

## TUGAS

Buat kelompok dengan 2 anggota didalamnya kemudian aplikasi RPC Client Server diatas diubauh menjadi Graphic User Interface (GUI). Kemudian ujicoba secara real dengan anggota kelompok anda yang bertindak sebagai server dan sebagai client.

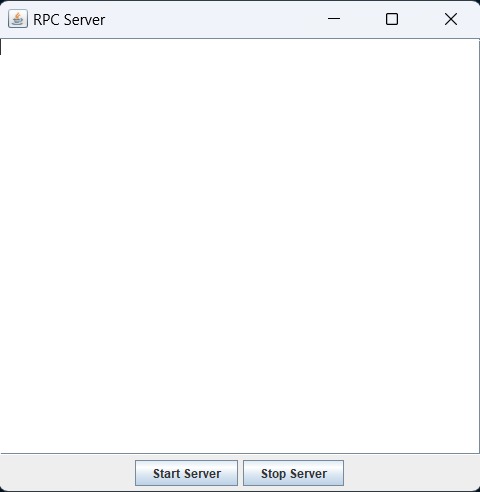
Bagian Server

1. **import** javax.swing.\*;
2. **import** java.awt.\*;
3. **import** java.awt.event.ActionEvent;
4. **import** java.awt.event.ActionListener;
5. **import** java.io.BufferedReader;
6. **import** java.io.IOException;
7. **import** java.io.PrintStream;
8. **import** java.io.InputStreamReader;
9. **import** java.net.ServerSocket;
10. **import** java.net.Socket;
11. **import** java.util.ArrayList;
12. **import** java.util.List;
14. **public** **class** RPCServerGUI {
15. **private** **final** ServerSocket serverSocket;
16. **private** **final** JTextArea textArea;
17. **private** JButton startButton;
18. **private** List<Socket> clientSockets = **new** ArrayList<>(); *// Declare and initialize the clientSockets variable here*
20. **public** RPCServerGUI(**int** port) **throws** IOException {
21. *// List<Socket> clientSockets = new ArrayList<>();*
23. serverSocket = **new** ServerSocket(port);
24. textArea = **new** JTextArea();
25. startButton = **new** JButton("Start Server");
26. JButton stopButton = **new** JButton("Stop Server"); *// Declare the stopButton variable*
28. JFrame frame = **new** JFrame("RPC Server");
29. frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);
30. frame.setSize(500, 500);
31. frame.setLayout(**new** BorderLayout());
32. frame.add(**new** JScrollPane(textArea), BorderLayout.CENTER);
34. JPanel buttonPanel = **new** JPanel(); *// Add this line to create the buttonPanel*
35. buttonPanel.add(startButton);
36. buttonPanel.add(stopButton); *// Use the stopButton variable*
38. frame.add(buttonPanel, BorderLayout.SOUTH);
40. frame.setVisible(**true**);
42. startButton.addActionListener(**new** ActionListener() {
43. @Override
44. **public** **void** actionPerformed(ActionEvent e) {
45. startButton.setEnabled(**false**);
46. stopButton.setEnabled(**true**);
47. textArea.append("Server is running on 192.168.117.54:5000**\n**");
48. **new** Thread(() -> {
49. **while** (!serverSocket.isClosed()) {
50. **try** {
51. Socket rpcClient = serverSocket.accept();
52. String address = rpcClient.getRemoteSocketAddress().toString();
53. textArea.append("New client connected : " + address + "**\n**");
54. **new** Thread(() -> {
55. **try** {
56. addHook(rpcClient);
57. } **catch** (IOException ex) {
58. textArea.append("Client disconnected " + address + "**\n**");
59. }
60. }).start();
61. } **catch** (IOException ex) {
62. **if** (!serverSocket.isClosed()) {
63. textArea.append("Error accepting client connection**\n**");
64. }
65. }
66. }
67. }).start();
68. }
69. });
71. *// JButton stopButton = new JButton("Stop Server");*
72. stopButton.addActionListener(**new** ActionListener() {
73. @Override
74. **public** **void** actionPerformed(ActionEvent e) {
75. stopButton.setEnabled(**false**);
76. startButton.setEnabled(**true**);
77. **try** {
78. **for** (Socket clientSocket : clientSockets) {
79. **if** (!clientSocket.isClosed()) {
80. clientSocket.close();
81. }
82. }
83. serverSocket.close();
84. textArea.append("Server stopped**\n**");
85. } **catch** (IOException ex) {
86. textArea.append("Error stopping server**\n**");
87. }
88. }
89. });
90. }
91. **private** **void** addHook(Socket rpcClient) **throws** IOException {
92. *// List<Socket> clientSockets = new ArrayList<>(); // Declare and initialize the clientSockets variable*
93. clientSockets.add(rpcClient);
94. BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(rpcClient.getInputStream()));
95. String line;
96. *// continue your code here*
97. **while** ((line = reader.readLine()) != **null**) {
98. **if** ("exit".equals(line)) {
99. **break**;
100. }
101. System.out.println("Client request : " + line);
102. String[] commands = line.split(":", 3);
103. **int** result;
104. **int** operand1 = Integer.parseInt(commands[1]);
105. **int** operand2 = Integer.parseInt(commands[2]);
106. String message = "";
108. **switch** (commands[0]) {
109. **case** "add":
110. result = (operand1 + operand2);
111. message = operand1 + " + " + operand2 + " = " + result;
112. **break**;
113. **case** "sub":
114. result = (operand1 - operand2);
115. message = operand1 + " - " + operand2 + " = " + result;
116. **break**;
117. **case** "mul":
118. result = (operand1 \* operand2);
119. message = operand1 + " \* " + operand2 + " = " + result;
120. **break**;
121. **case** "div":
122. result = (operand1 / operand2);
123. message = operand1 + " / " + operand2 + " = " + result;
124. **break**;
125. **case** "mod":
126. result = (operand1 % operand2);
127. message = operand1 + " % " + operand2 + " = " + result;
128. **break**;
129. **default**:
130. message = "Invalid operation";
131. **break**;
132. }
133. PrintStream printStream = **new** PrintStream(rpcClient.getOutputStream(), **true**);
134. printStream.println(message);
135. textArea.append("Client request : " + line + "**\n**");
136. textArea.append("Response : " + message + "**\n**");
137. }
138. }

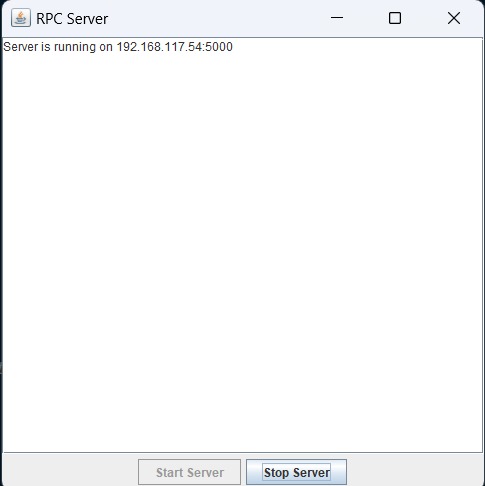
141. **public** **static** **void** main(String[] args) {
142. SwingUtilities.invokeLater(**new** Runnable() {
143. **public** **void** run() {
144. **try** {
145. **new** RPCServerGUI(5000);
146. } **catch** (IOException e) {
147. e.printStackTrace();
148. }
149. }
150. });
151. }
152. }

Output

* Ketika Server di Run lalu Klik Start Server



* Ketika sudah di Start

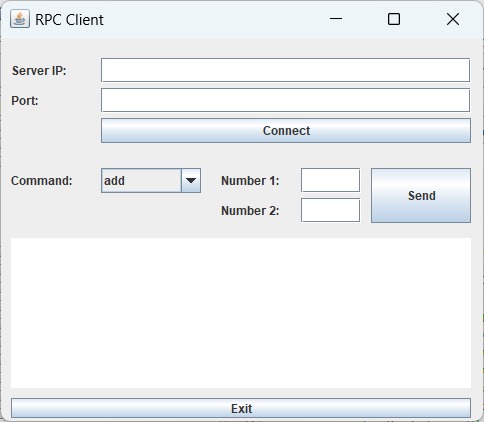


Bagian Client

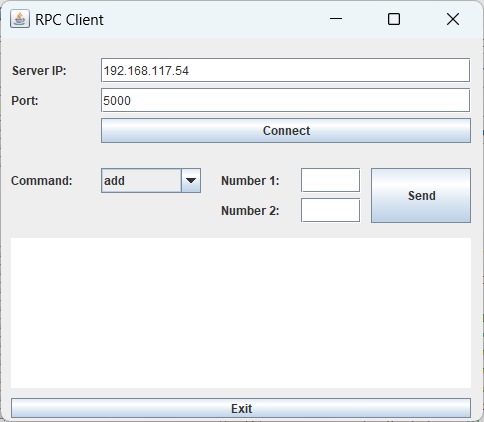
1. **import** javax.swing.\*;
2. **import** java.awt.\*;
3. **import** java.awt.event.\*;
4. **import** java.io.\*;
5. **import** java.net.Socket;
6. **import** java.util.Scanner;
8. **public** **class** RPCClientGUI {
9. **private** JFrame frame;
10. **private** JTextField ipField;
11. **private** JTextField portField;
12. **private** JTextField num1Field;
13. **private** JTextField num2Field;
14. **private** JTextArea responseArea;
15. **private** JComboBox<String> commandBox;
16. **private** Socket rpcClient;
17. **private** PrintWriter writer;
19. **public** RPCClientGUI() {
20. frame = **new** JFrame("RPC Client");
21. frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);
22. frame.setSize(500, 430);
24. JPanel panel = **new** JPanel();
25. frame.add(panel);
26. placeComponents(panel);
28. frame.setVisible(**true**);
29. }
31. **private** **void** placeComponents(JPanel panel) {
32. panel.setLayout(**null**);
34. JLabel ipLabel = **new** JLabel("Server IP:");
35. ipLabel.setBounds(10, 20, 80, 25);
36. panel.add(ipLabel);
38. ipField = **new** JTextField(20);
39. ipField.setBounds(100, 20, 370, 25); *// Adjust the size and position of the components*
40. panel.add(ipField);
42. JLabel portLabel = **new** JLabel("Port:");
43. portLabel.setBounds(10, 50, 80, 25);
44. panel.add(portLabel);
46. portField = **new** JTextField(20);
47. portField.setBounds(100, 50, 370, 25);
48. panel.add(portField);
50. JButton connectButton = **new** JButton("Connect");
51. connectButton.setBounds(100, 80, 370, 25);
52. panel.add(connectButton);
54. connectButton.addActionListener(**new** ActionListener() {
55. @Override
56. **public** **void** actionPerformed(ActionEvent e) {
57. connectToServer();
58. }
59. });
61. *// Add more components for commands and numbers here*
63. JLabel commandLabel = **new** JLabel("Command:");
64. commandLabel.setBounds(10, 130, 80, 25);
65. panel.add(commandLabel);
67. String[] commands = {"add", "sub", "mul", "div", "mod"};
68. commandBox = **new** JComboBox<>(commands);
69. commandBox.setBounds(100, 130, 100, 25);
70. panel.add(commandBox);
72. JLabel num1Label = **new** JLabel("Number 1:");
73. num1Label.setBounds(220, 130, 80, 25);
74. panel.add(num1Label);
76. num1Field = **new** JTextField(20);
77. num1Field.setBounds(300, 130, 60, 25);
78. panel.add(num1Field);
80. JLabel num2Label = **new** JLabel("Number 2:");
81. num2Label.setBounds(220, 160, 80, 25);
82. panel.add(num2Label);
84. num2Field = **new** JTextField(20);
85. num2Field.setBounds(300, 160, 60, 25);
86. panel.add(num2Field);
88. responseArea = **new** JTextArea();
89. responseArea.setBounds(10, 200, 460, 150); *// Adjust the size of the responseArea*
90. responseArea.setEditable(**false**);
91. panel.add(responseArea);
93. JButton exitButton = **new** JButton("Exit");
94. exitButton.setBounds(10, 360, 460, 20); *// Set the coordinates and size of the exit button*
95. panel.add(exitButton);
97. exitButton.addActionListener(**new** ActionListener() {
98. @Override
99. **public** **void** actionPerformed(ActionEvent e) {
100. System.exit(0);
101. }
102. });
104. JButton sendButton = **new** JButton("Send");
105. sendButton.setBounds(370, 130, 100, 55); *// Adjust the size and position of the sendButton*
106. panel.add(sendButton);
108. sendButton.addActionListener(**new** ActionListener() {
109. @Override
110. **public** **void** actionPerformed(ActionEvent e) {
111. sendCommand();
112. }
113. **private** **void** sendCommand() {
114. String command = (String) commandBox.getSelectedItem();
115. String num1 = num1Field.getText();
116. String num2 = num2Field.getText();
118. writer.println(command + ":" + num1 + ":" + num2);
120. **try** {
121. BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(rpcClient.getInputStream()));
122. String response = reader.readLine();
123. responseArea.append("Server response: " + response + "**\n**");
124. } **catch** (IOException e) {
125. responseArea.append("Error reading response from server**\n**");
126. }
127. }
128. });
130. }
132. **private** **void** connectToServer() {
133. String serverIp = ipField.getText();
134. **int** port = Integer.parseInt(portField.getText());
136. **try** {
137. rpcClient = **new** Socket(serverIp, port);
138. writer = **new** PrintWriter(rpcClient.getOutputStream(), **true**);
139. responseArea.append("Connected to server**\n**");
140. } **catch** (IOException e) {
141. responseArea.append("Error connecting to server**\n**");
142. }
143. }
145. **public** **static** **void** main(String[] args) {
146. **new** RPCClientGUI();
147. }
148. }

Output

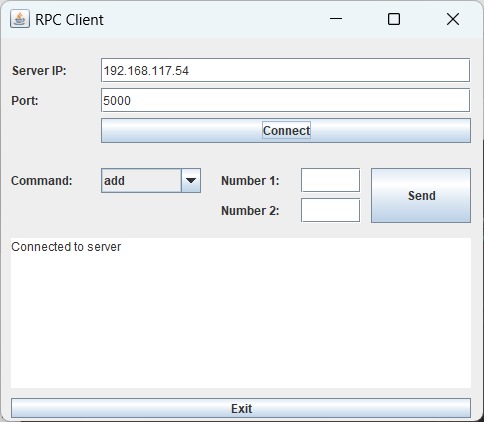
* Tampilan awal ketika di run



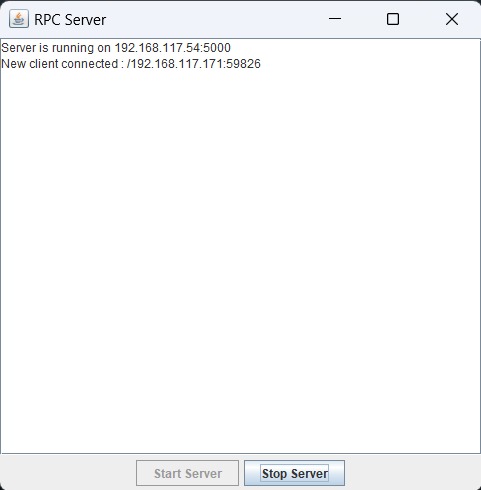
* Memasukan Ip Address dari server lalu klik Connect



* Tampilan Setelah Connect Di bagian Client

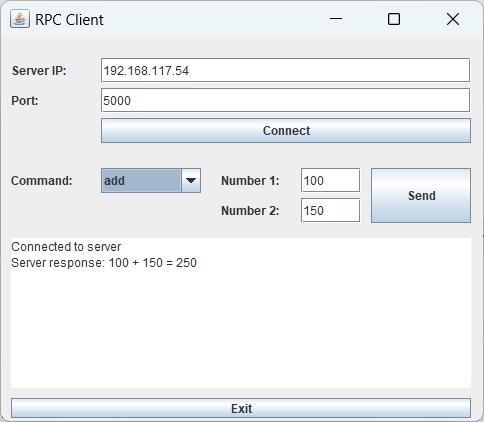


* Tampilan Setelah Connect Di bagian Server

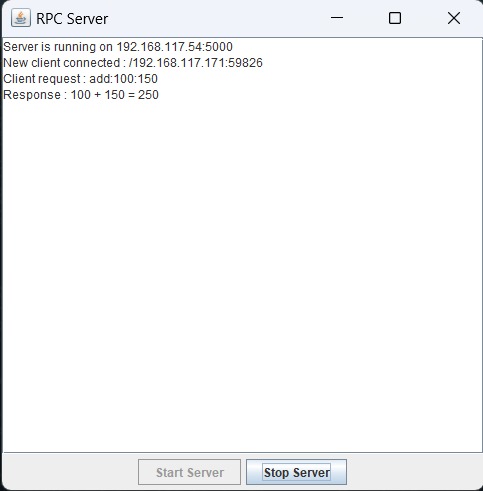


**Oprasi Perhitungan Aritmatika**

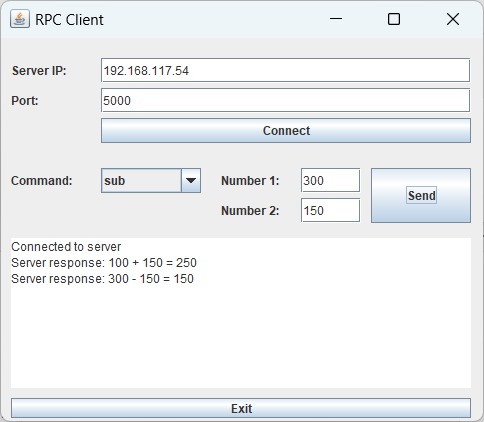
* Oprasi Add di Bagian Client



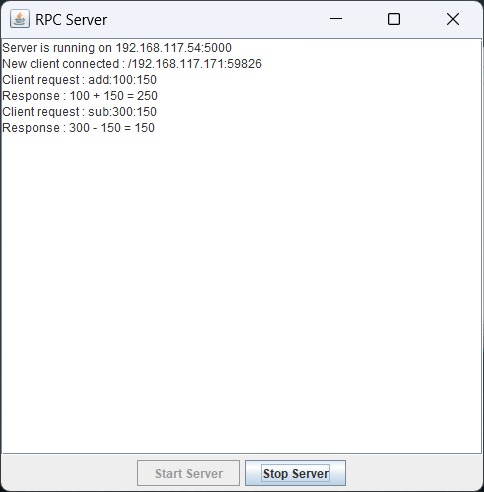
* Tampilan Di bagian Server



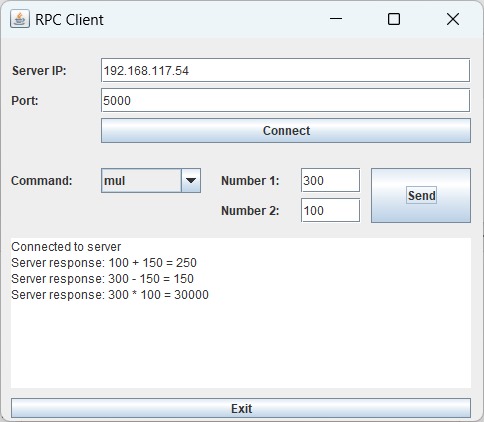
* Oprasi Add di Bagian Client



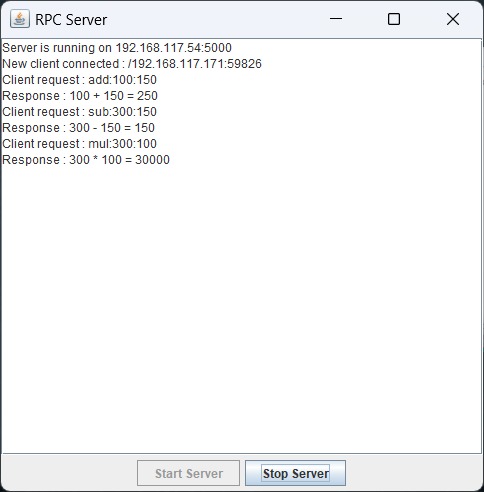
* Tampilan Di bagian Server



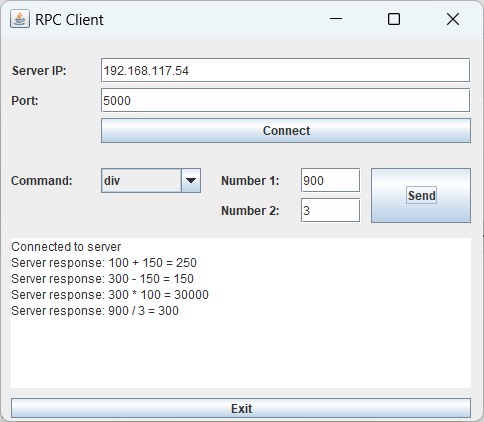
* Oprasi Mul di Bagian Client



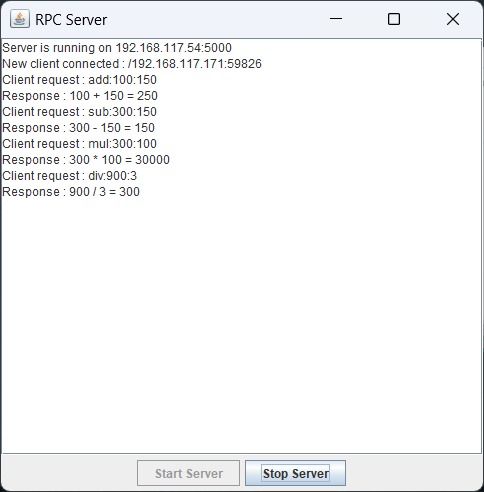
* Tampilan Di bagian Server



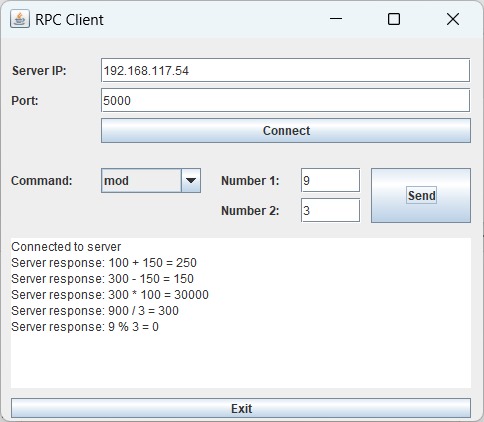
* Oprasi Div di Bagian Client



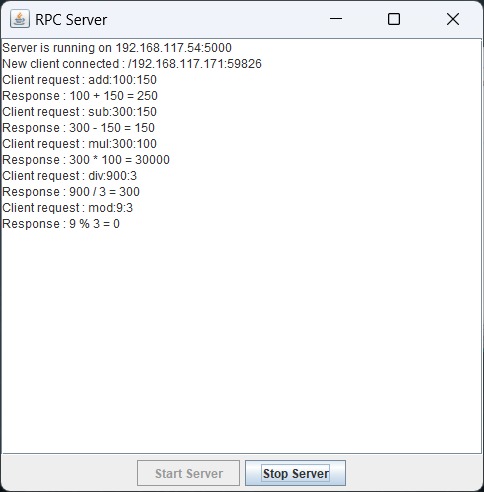
* Tampilan Di bagian Server



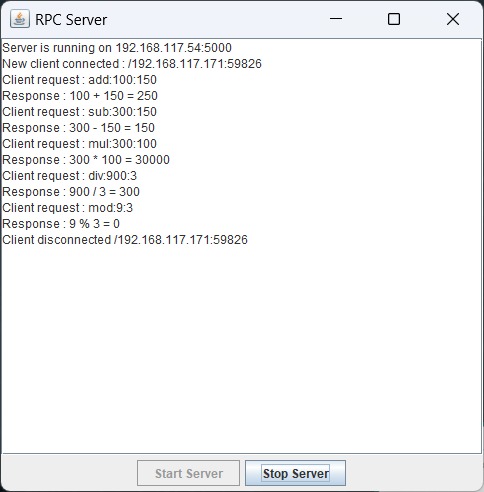
* Oprasi Mod di Bagian Client



* Tampilan Di bagian Server



* Tampilan Ketika Disconnect



1. **KESIMPULAN**

Pemrograman Remote Procedure Call (RPC) adalah sebuah paradigma komunikasi antara proses di jaringan komputer yang memungkinkan aplikasi untuk memanggil fungsi atau prosedur yang berjalan di komputer jarak jauh secara transparan. Dengan RPC, aplikasi dapat mengirim permintaan ke server jarak jauh dan menerima respons tanpa harus memikirkan detail komunikasi jaringan.