Academic Year – 2021-22

Class: T.Y.MCA Semester: V Subject: Distributed computing and Cloud Computing Lab

Subject In charge: Nikhita Mangaonkar Course Code**:** MCAL51

ROLL NO. : 2019450046 **BATCH: D** **DATE: 09/09/2021**

NAME: Ayush Sah

EXPERIMENT NO: 02

EXPERIMENT TITLE: Implement RPC or RMI for given scenario

Objective

Design, implement and test a client-server distributed system that uses RPC/RMI to compute your semester grade, result and Performance status.

Problem statement

Suppose we are requested to create a distributed application for computing your grade card using a computational expensive algorithm that cannot be run on any client machine. Thus, the algorithm is run on a remote physical machine having more resources (the server). The customers (remote clients) want to use the algorithm to compute the grade for the students by sending data to the server and receiving the computation results to be displayed.

The client application sends the data regarding the subject marks to the server. The clients contains the following fields:

- *int sub1 ,sub2 ……marks (depending on no. of subjects)*
- *int total marks*

Based on this data, the server will compute the Grade value and performance status using the formula. (behind grade card)

Where *grade value depends* on the total marks from following table: Relation between grade and performance.

% Marks	Grade	GP	Performance
80.00 and Above	O	10	Outstanding
75.00 - 79.99	A	9	Excellent
70.00 - 74.99	B	8	Very Good
60.00 - 69.99	C	7	Good
55.00 - 59.99	D	6	Fair
50.00 - 54.99	E	5	Average

45.00 - 49.99	P	4	Pass
Less than 45.00	F	0	Fail
--	X	0	Defaulter
--	AB	0	Absent

NOTE: *Usually, the method from the server is a computational intensive calculus that requires more physical resources than are available on the client.*

Application analysis and design

From the problem requirements we notice an important aspect: the algorithm used to compute the calculation of marks is computational intensive, thus being unsuited for the clients to run it locally on their physical machines. Consequently, the chosen solution will be a distributed application having client-server architecture. The server, having more physical resources, will run the computational intensive algorithm. The server will expose a method that must be executed remotely by the client, leading to a remote procedure call technique or through remote method invocation.

The solution can be decomposed into the following subsystems:

- Communication protocol –RPC /RMI between client and server
- The server application
  - RPC /RMI
  - Communication layer over the network
- The client application
  - Communication layer over the network
  - RPC /RMI

Communication mechanism

This section defines the message structure that will allow a remote method invocation, or Remote Procedure Call (RPC).

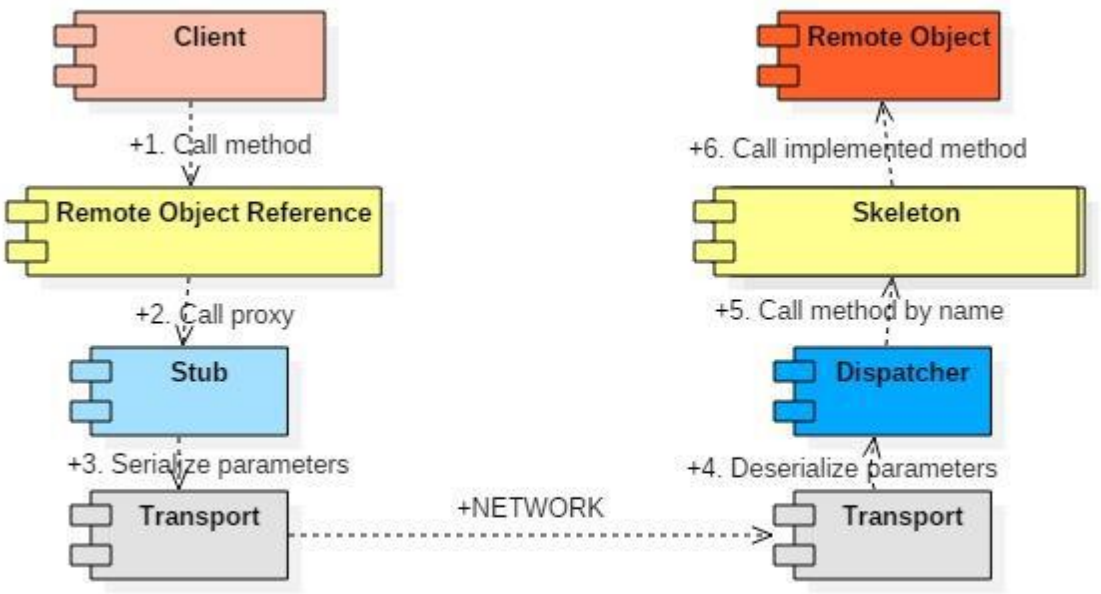
During the invocation of a method, the parameters are stored on the stack and the control is passed to the code section located at the address mapped to the procedure name. What is important to notice is that a procedure is defined by its name (that maps to an address in the memory where the actual code is located) and its parameters.

In an Object-Oriented Programming (OOP) environment, we have a Remote Method Invocation (RMI) technique that allows invoking a method from a remote object. In this case, we must know the object address (or name), the method name and its parameters. Furthermore, in a distributed

environment, to identify a remote object, besides knowing the object name (and implicitly its memory address) we must also know the address of the server where it is located.

Basically, this RPC/RMI technique introduces an intermediate layer between the method call and its actual execution, mainly because the method call happens on the client and the execution on the server.

For the client to make the call, it must know the signature of the method (name, parameters and return type). The signature of a method is defined in OOP languages in an interface. Thus, we might assume that the methods from the server are defined in an interface. This client has also a reference to this interface, thus knowing the method signature that will be called. Considering the above aspects, the system communication flow is shown in following figure:



The steps involved in calling a remote method are described below:

Client calls the method: The client application makes a call to a special proxy object that implements the remote interface. The client handles this object as it was a local object implementing the interface. The client calls the desired method.

Call forwarded to the proxy: The method call is forwarded to a proxy that has a special implementation of the interface. Instead of implementing the functionality of the methods, this proxy creates a communication mechanism that takes the method's name and parameters and serializes them to be sent over the network.

Data sent over the network: The data is packed and sent over the network. The following information is serialized: remote object name (address space), remote object method and method parameters.

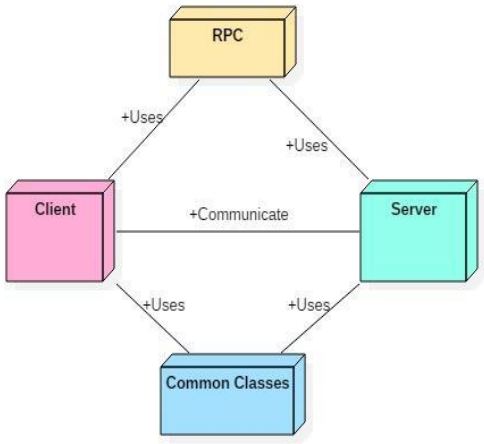
Server receives data: The server receives the data, de-serializes it and sends it to the Dispatcher.
Server calls method: The Dispatcher is responsible for calling the method from the Skeleton that is the interface exposed by the remote object.

Server executes method: The server executes the method with the parameters send from the client. It computes the return value of the method and serializes the result for the client.

Result returned to the client: The result is returned to the client, which de-serializes it and returns it to the Stub as it has been computed locally.

Application structure and implementation

The solution is implemented in 4 different modules: Client application, Server application, RPC package that contains the classes for remote communication and the Common Classes for both client and server application. The relation between the modules is presented in following figure



Academic Year – 2021-22

Class: T.Y.MCA Semester: V Subject: Distributed computing and Cloud Computing Lab

Subject In charge: Nikhita Mangaonkar Course Code**:** MCAL51

Each module has the following components:

- Client application - contains one package (Communication) with two classes:
  - *ClientStart* - Class which contains the main method. Here, the remote object is invoked after a reference is created.
  - *ServerConnection* – class that contains the sockets connecting the client with the server
- Server Application – contains two packages:
  - *Communication* - contains the server-side communication
  - *Services* – contains the implementation of the remote object
- Common Classes – contain two packages:
  - *Entities* - contains the entities (Car)
  - *ServiceInterface* - contains the definition of the interface exposed by the remote object (Skeleton)
- RPC – library that contains the protocol definition. Contains one package with five classes:
  - *Connection* - interface specifying the connection of a client to the server. Such a connection must provide a method to send a message to the server and retrieve the message response.
  - *Dispatcher* - dispatches the call received from the client. It interprets the given *Message*, gets the correct object from the registry, calls the required method of that object and then bundles and returns a response *Message*.
  - *Message* - represents the object of communication between the client and the server. It contains all the necessary fields for communication. For example, when the client sends the message to the server, the message contains:
    - the endpoint from the *Registry*, which is associated to the remote object
    - the name of the method to be called
    - the arguments of the method, in order
    - when the server replies, it adds the result (return value of the method, or a status message, or an exception) in the arguments array, on the first position.
    - *Naming* - provides a static method to look up for a remote object on the server.
    - *Registry* - provides a mapping of endpoint-object. It is used by the server to specify which object can be remotely used by a client. The client must identify the object at the endpoint.

Implementation technologies: Choose between JAVA RMI or RPC

Academic Year – 2021-22

Class: T.Y.MCA Semester: V Subject: Distributed computing and Cloud Computing Lab

Subject In charge: Nikhita Mangaonkar Course Code**:** MCAL51

Deliverables:

Code –

RPC –

Server.java

```
package Exp2.RPC.StudentMarks;

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

public class Server {
   public static void main(String[] args) throws Exception {
      ServerSocket sersock = new ServerSocket(5002);
      System.out.println("Server ready");
      Socket sock = sersock.accept();
      BufferedReader keyRead = new BufferedReader(new InputStreamReader(System.in));
      OutputStream ostream = sock.getOutputStream();
      PrintWriter pwrite = new PrintWriter(ostream, true);
      InputStream istream = sock.getInputStream();
      BufferedReader receiveRead = new BufferedReader(new InputStreamReader(istream));
      String receiveMessage, sendMessage, fun;
      int countOfSubjects, sum, temp;

      while (true) {
         sum = 0;
         receiveMessage = receiveRead.readLine();
         if (receiveMessage != null) {
            if (receiveMessage.compareTo("end") == 0) {
               System.out.println("Closing connection");
               sock.close();
               receiveRead.close();
               return;
            }
         }

         countOfSubjects = Integer.parseInt(receiveMessage);
         int count = 0;
         while (count++ < countOfSubjects) {
```

```
            temp = Integer.parseInt(receiveRead.readLine());
            sum += temp;
            System.out.println("Marks received for subject " + count + ": " + temp);
         }
         System.out.println("Total Marks Obtained = " + sum);
         pwrite.println("Total Marks Obtained = " + sum);
         double percent = (double) (sum * 100) / (countOfSubjects * 100);
         System.out.println("Percentage = " + percent + "%");
         pwrite.println("Percentage = " + percent + "%");
         String grade[] = { "O", "A", "B", "C", "D", "E", "P", "F" }, performance[] = { "Outstandi
ng", "Excellent",
                  "Very Good", "Good", "Fair", "Average", "Pass", "Fail" };
         int gp[] = { 10, 9, 8, 7, 6, 5, 4, 0 }, i = 7;
         if (percent >= 80)
            i = 0;
         else if (percent >= 75)
            i = 1;
         else if (percent >= 70)
            i = 2;
         else if (percent >= 60)
            i = 3;
         else if (percent >= 55)
            i = 4;
         else if (percent >= 50)
            i = 5;
         else if (percent >= 45)
            i = 6;
         String grading = "Grade: " + grade[i] + "\nGrade Point: " + gp[i] + "\nPerformance: " + p
erformance[i];
         System.out.println(grading);
         pwrite.println("Grade: " + grade[i]);
         pwrite.println("Grade Point: " + gp[i]);
         pwrite.println("Performance: " + performance[i]);
         System.out.flush();
      }
   }
}
```

Client.java

```
package Exp2.RPC.StudentMarks;

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

public class Client {
    public static void main(String[] args) throws Exception {
        Socket sock = new Socket("127.0.0.1", 5002);
        BufferedReader keyRead = new BufferedReader(new InputStreamReader(System.in));
        OutputStream ostream = sock.getOutputStream();
        PrintWriter pwrite = new PrintWriter(ostream, true);
        InputStream istream = sock.getInputStream();
        BufferedReader receiveRead = new BufferedReader(new InputStreamReader(istream));
        System.out.println("Client ready, type and press Enter key");
        String receiveMessage, sendMessage, temp;

        while (true) {

            System.out.print("Enter the number of Subjects you want to calculater marks for: ");
            temp = keyRead.readLine();
            sendMessage = temp.toLowerCase();
            pwrite.println(sendMessage);

            int countOfSubjects = Integer.parseInt(temp);
            int count = 0;
            while (count++ < countOfSubjects) {
                System.out.print("Enter the marks scored in Subject " + count + ": ");
                sendMessage = keyRead.readLine();
                pwrite.println(sendMessage);
            }
            System.out.flush();
            System.out.println(receiveRead.readLine());
            System.out.println(receiveRead.readLine());
            System.out.println(receiveRead.readLine());
            System.out.println(receiveRead.readLine());
            System.out.println(receiveRead.readLine() + "\n\n");
            System.out.print("Do you want to continue? ");
            temp = keyRead.readLine();
```

```
        if (Objects.equals(temp, "no")) {
            pwrite.println("end");
            System.out.println("Connection Closed!");
            keyRead.close();
            pwrite.close();
            sock.close();
            return;
        }
    }
  }
}
```

Output –

Server

```
PS D:\Code\DCCC\src\Exp2\RPC\StudentMarks> java .\Server.java
Server ready
Marks received for subject 1: 84
Marks received for subject 2: 75
Marks received for subject 3: 89
Marks received for subject 4: 71
Marks received for subject 5: 95
Total Marks Obtained = 414
Percentage = 82.8%
Grade: O
Grade Point: 10
Performance: Outstanding
Closing connection
```

Client

```
PS D:\Code\DCCC\src\Exp2\RPC\StudentMarks> java .\Client.java
Client ready, type and press Enter key
Enter the number of Subjects you want to calculater marks for: 5
Enter the marks scored in Subject 1: 84
Enter the marks scored in Subject 2: 75
Enter the marks scored in Subject 3: 89
Enter the marks scored in Subject 4: 71
Enter the marks scored in Subject 5: 95
Total Marks Obtained = 414
Percentage = 82.8%
Grade: O
Grade Point: 10
Performance: Outstanding


Do you want to continue? no
Connection Closed!
```

RMI –

RemoteInterfaceMarks.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteInterfaceMarks extends Remote {
    public int totalMarks(int countOfSubjects, int marks[]) throws RemoteException;

    public double percentage(int countOfSubjects, int totalMarks) throws RemoteException;

    public String grading(double percent) throws RemoteException;
}
```

ImplremetnataionMarks.java

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ImplementationMarks extends UnicastRemoteObject implements RemoteInterfaceM
arks {
    public ImplementationMarks() throws RemoteException {
        super();
```

```
        System.setProperty("java.rmi.server.hostname", "192.168.1.2");
        System.out.println("Implementation Class for Marks");
    }

    @Override
    public int totalMarks(int countOfSubjects, int marks[]) throws RemoteException {
        int total = 0;
        for (int i = 0; i < countOfSubjects; i++)
            total += marks[i];
        return total;
    }

    @Override
    public double percentage(int countOfSubjects, int totalScored) throws RemoteException {
        return (double) (totalScored) * 100 / (countOfSubjects * 100);
    }

    @Override
    public String grading(double percent) throws RemoteException {
        String grade[] = { "O", "A", "B", "C", "D", "E", "P", "F" },
                performance[] = { "Outstanding", "Excellent", "Very Good", "Good", "Fair", "Average
", "Pass", "Fail" };
        int gp[] = { 10, 9, 8, 7, 6, 5, 4, 0 }, i = 7;
        if (percent >= 80)
            i = 0;
        else if (percent >= 75)
            i = 1;
        else if (percent >= 70)
            i = 2;
        else if (percent >= 60)
            i = 3;
        else if (percent >= 55)
            i = 4;
        else if (percent >= 50)
            i = 5;
        else if (percent >= 45)
            i = 6;
        return "Grade: " + grade[i] + "\nGrade Point: " + gp[i] + "\nPerformance: " + performance[i
];
    }
```

```
}
```

Server.java

```
import java.rmi.Naming;

public class Server {
   public static void main(String[] args) {
      try {
         ImplementationMarks marks = new ImplementationMarks();
         Naming.rebind("marks", marks);
      } catch (Exception e) {
         System.err.println("Server exception: " + e.toString());
         e.printStackTrace();
      }
   }
}
```

Client.java

```
import java.rmi.*;
import java.time.temporal.Temporal;
import java.util.Objects;
import java.util.Scanner;

public class Client {
   public static void main(String[] args) {
      try {
         Scanner sc = new Scanner(System.in);
         String URL = "rmi://localhost/marks";
         boolean run = true;
         RemoteInterfaceMarks inf = (RemoteInterfaceMarks) Naming.lookup(URL);
         while (run) {
            int countOfSubjects;
            System.out.print("Enter the total count of the subjects: ");
            countOfSubjects = sc.nextInt();
            int marks[] = new int[countOfSubjects];
            int count = 0;
            while (count < countOfSubjects) {
               System.out.print("Enter the marks scored in subject " + (count + 1) + ": ");
```

```
                marks[count++] = sc.nextInt();
            }
            int totalMarks = inf.totalMarks(countOfSubjects, marks);
            System.out.println("Total marks scored: " + totalMarks);
            double percent = inf.percentage(countOfSubjects, totalMarks);
            System.out.println("Percentage: " + percent);
            System.out.println(inf.grading(percent));

            System.out.print("Do you want to continue: ");
            String temp = sc.next();

            if (Objects.equals(temp, "no"))
                run = false;
        }
        sc.close();
    } catch (Exception e) {
    }
  }

}
```

Output

```
PS D:\Code\DCCC\src\Exp2\RMI\StudentMarks> javac *.java
PS D:\Code\DCCC\src\Exp2\RMI\StudentMarks> start rmiregistry
PS D:\Code\DCCC\src\Exp2\RMI\StudentMarks> java .\Server.java
Implementation Class for Marks
```

```
D:\Applications\java\Oracle\bin\rmiregistry.exe
WARNING: A terminally deprecated method in java.lang.System has been called
WARNING: System::setSecurityManager has been called by sun.rmi.registry.RegistryImpl
WARNING: Please consider reporting this to the maintainers of sun.rmi.registry.RegistryImpl
WARNING: System::setSecurityManager will be removed in a future release
```

```
PS D:\Code\DCCC\src\Exp2\RMI\StudentMarks> java .\Client.java
Enter the total count of the subjects: 5
Enter the marks scored in subject 1: 84
Enter the marks scored in subject 2: 72
Enter the marks scored in subject 3: 91
Enter the marks scored in subject 4: 86
Enter the marks scored in subject 5: 75
Total marks scored: 408
Percentage: 81.6
Grade: O
Grade Point: 10
Performance: Outstanding
Do you want to continue: no
PS D:\Code\DCCC\src\Exp2\RMI\StudentMarks>
```

RPC/RMI Call Steps

1. Run RMI Registry
2. Compile all files
3. Run the server file
4. Run the client file
5. Enter the count of subject
6. Enter the marks of each subject
7. Client stub is called by client.
8. Client stub prepares a call by packing parameters to the procedure call. (This process is called marshalling)
9. Client's local machine sends the message to the server machine.
10. Server stub receives the sent message.
11. Server stub unpacks the parameters from the message. (This process is called unmarshalling)
12. Server stub calls the server procedure and returns the results to the client machine after marshalling the data.
13. Print result to the client.

Academic Year – 2021-22

Class: T.Y.MCA Semester: V Subject: Distributed computing and Cloud Computing Lab

Subject In charge: Nikhita Mangaonkar Course Code**:** MCAL51

Test Procedure for objective validation :.

Description	**Action**	**Expected Output**	**Actual Output**	**Status**
Start RMI Registry	Start the RMI Registry from command line	RMI Registry started	RMI Registry starts with some exception	Partial
Start Server	Run Server.java	"Server Started" will be printed on Server Terminal	"Server Started" is printed on Server Terminal	PASS
Start Client	Run Client	Client terminal displays "Connected".	Client terminal displays "Connected".	PASS
Send Count of subjects	Enter count of subjects	Server received the count.	Count is displayed on server side	PASS
Send marks of all subjects	Clients enters marks of subjects	Marks is send to the server	Marks are displayed on the server side	PASS
Server computes and sends the results	Server send the result to client	Results are displayed on client side	Client is able to see results.	PASS