

Assignment 3

RMI principles - develop a simple distributed computing environment consisting of a multiple Clients and a Server

Submitted By-

Nahida Sultana Chowdhury

Student ID: 2000189256

Date: 11/30/2017

1. Introduction

In distributed computing, Java Remote Method Invocation (Java RMI) is a Java API that performs remote method invocation, the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java classes and distributed garbage-collection [1]. Remote Method Invocation (RMI) can also be seen as the process of activating a method on a remotely running object. RMI offers location transparency because a user feels that a method is executed on a locally running object. Check some RMI Tips here.

The RMI architecture is based on a very important principle which states that the definition of the behavior and the implementation of that behavior, are separate concepts. RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs [2].

In this assignment, we supposed to implement the principles of remote method invocations using the Java-RMI model of distributed-object computing that supports multiple clients and server. The system supports the following functions by server to multi clients and they will concurrently request various functions:

1. Returns the current path
2. Accepts an unsorted list and returns its sorted version
3. Returns whatever a Client sends as an input
4. Accepts a file name and checks if a file is available in the current directory or not and displays a message accordingly
5. Accepts two integer matrices and returns their multiplication

2. System Overview

The RMI architecture consists of the following layers [2]:

- **Stub and Skeleton layer:** This layer lies just beneath the view of the developer. This layer is responsible for intercepting method calls made by the client to the interface and redirect these calls to a remote RMI Service.
- **Remote Reference Layer:** The second layer of the RMI architecture deals with the interpretation of references made from the client to the server's remote objects. This layer interprets and manages references made from clients to the remote service objects. The connection is a one-to-one (unicast) link.
- **Transport layer:** This layer is responsible for connecting the two JVM participating in the service. This layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.

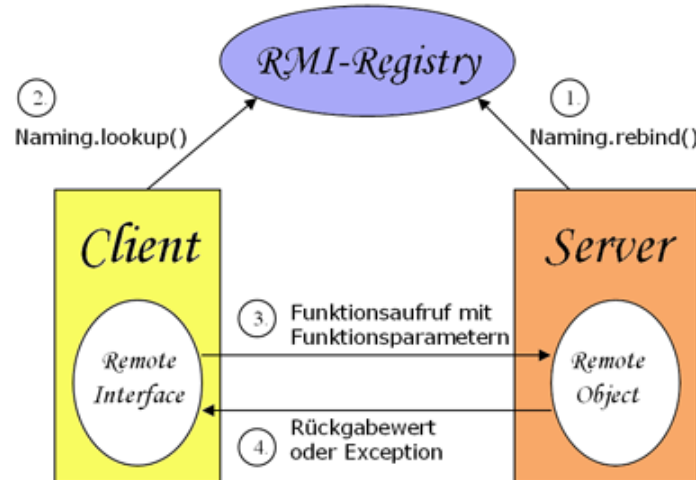


Figure 1: RMI Structure [3]

The following steps must be involved in order for a RMI program to work properly:

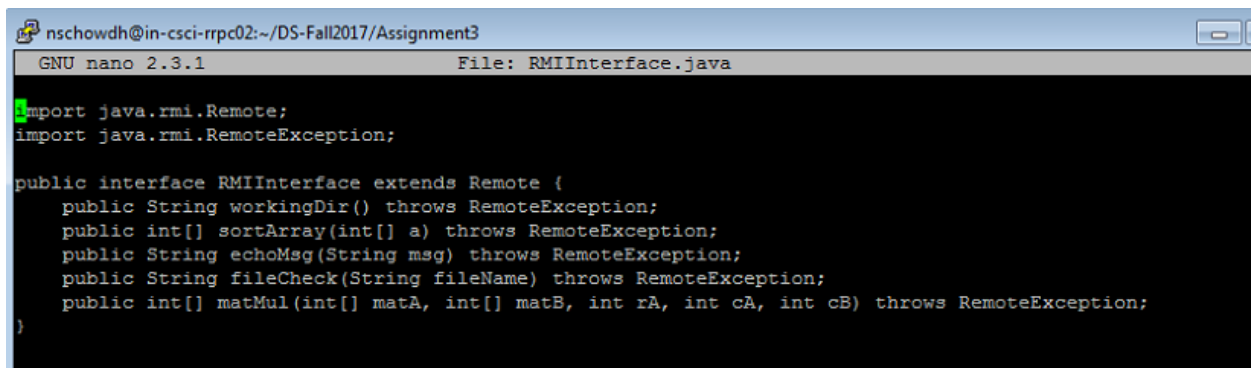
- Compilation of all source files.
- Generation of the stubs using `rmic`.
- Start the `rmiregistry`.
- Start the `RMIServer`.
- Run the client program.

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes and includes the object's data, as well as information about the object's type, and the types of data stored in the object. Thus, serialization can be seen as a way of flattening objects, in order to be

stored on disk, and later, read back and reconstituted. Deserialisation is the reverse process of converting an object from its flattened state to a live object.

3. Implementation and Results

Distributed application development, using *rmi*, starts with designing the Interface Definition Language file. This file contains the details about the services that run on different autonomous computers in the distributed system. The file has to be saved with a **.java** extension like **RMIInterface.java**, as in this case.

A screenshot of a terminal window with a blue title bar. The title bar text is 'nschowdh@in-csci-rpc02:~/DS-Fall2017/Assignment3'. Below the title bar, the terminal shows the GNU nano 2.3.1 editor interface. The file being edited is 'RMIInterface.java'. The code content is as follows:

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

public interface RMIInterface extends Remote {
    public String workingDir() throws RemoteException;
    public int[] sortArray(int[] a) throws RemoteException;
    public String echoMsg(String msg) throws RemoteException;
    public String fileCheck(String fileName) throws RemoteException;
    public int[] matMul(int[] matA, int[] matB, int rA, int cA, int cB) throws RemoteException;
}
```

Figure 2: Interface definition file RMIInterface.java

The next step is to implement the client and server interface and bind server with specific IP address. In this case we bind server with “10.234.136.55”.

Naming.rebind("//10.234.136.55/MyServer", new ServerOperation());

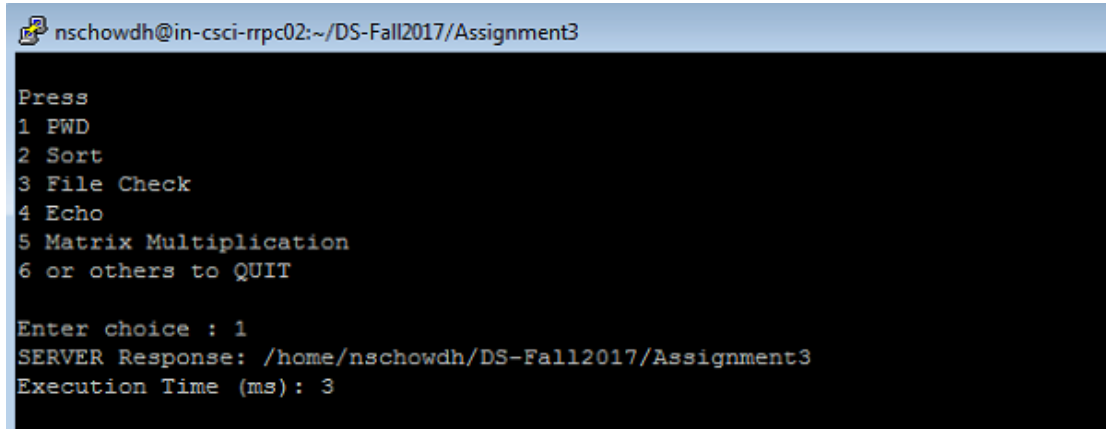
Then finally compile the three java files. Finally run the “*rmiregistry*” first on server machine. Afterwards run the server execution file where server will wait for client to connect. Client will use Naming lookup by passing server IP address to connect as follows:

(RMIInterface) Naming.lookup("//10.234.136.55/MyServer");

From the client side clients are free to request any function for any number of times to execute to server. The loop will continue until the client doesn’t want to terminate. In this application to **quit** client should provide 6 or any other integer number excluding 1 to 5. The server will listen always, will never terminate. To force terminate server please press **ctrl+c**.

a) Function 1: Returns the current path

Client will request to server to get the current working directory and in return server will return the full pathname of the current directory. The client side screenshot is given below:

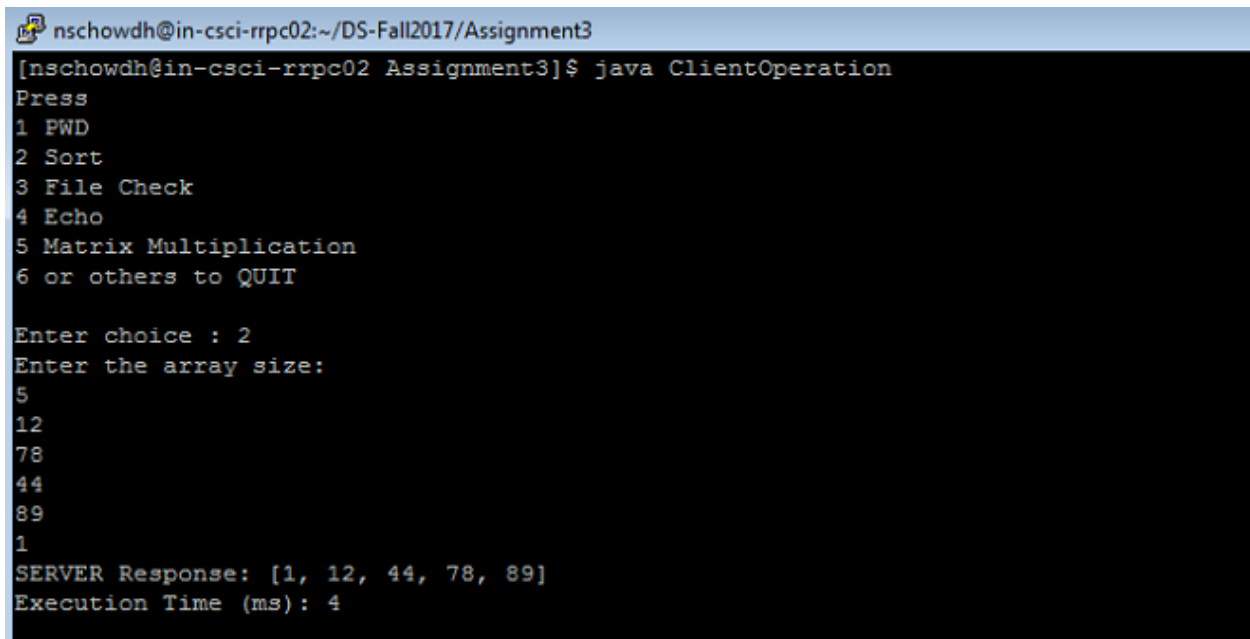


```
nschowdh@in-csci-rrpc02:~/DS-Fall2017/Assignment3
Press
1 PWD
2 Sort
3 File Check
4 Echo
5 Matrix Multiplication
6 or others to QUIT

Enter choice : 1
SERVER Response: /home/nschowdh/DS-Fall2017/Assignment3
Execution Time (ms): 3
```

b) Function 2: Accepts an unsorted list and returns its sorted version

Form the client side user will provide the input array that will ask user to provide the length of array and the elements. Then the array sends to server. Server accepts the unsorted array and after applying sorting mechanism then again sends back to client. Here bubble sort is implemented for sorting on server side. The client side screenshot is given below:

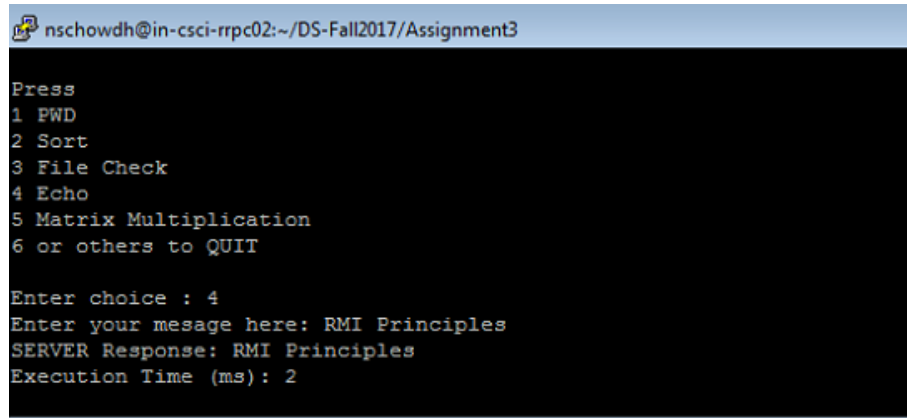


```
nschowdh@in-csci-rrpc02:~/DS-Fall2017/Assignment3
[nschowdh@in-csci-rrpc02 Assignment3]$ java ClientOperation
Press
1 PWD
2 Sort
3 File Check
4 Echo
5 Matrix Multiplication
6 or others to QUIT

Enter choice : 2
Enter the array size:
5
12
78
44
89
1
SERVER Response: [1, 12, 44, 78, 89]
Execution Time (ms): 4
```

c) Function 3: Returns whatever a Client sends as an input

Take input from user and send it to server and server response the same sent by client side. The client side screenshot is given below:

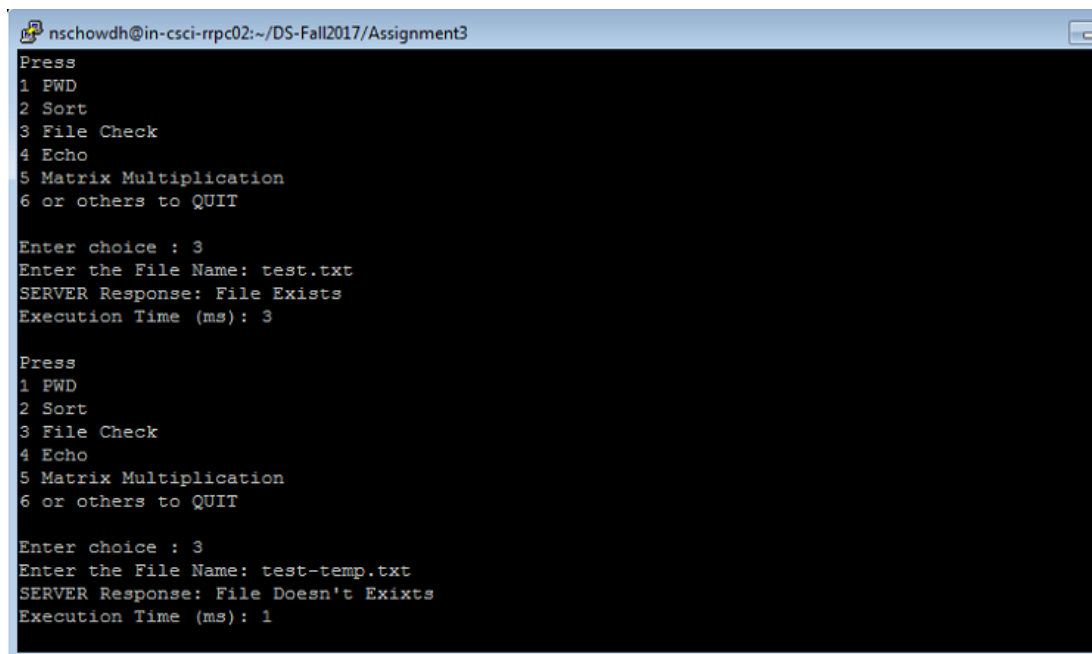


```
nschowdh@in-csci-rrpc02:~/DS-Fall2017/Assignment3
Press
1 PWD
2 Sort
3 File Check
4 Echo
5 Matrix Multiplication
6 or others to QUIT

Enter choice : 4
Enter your message here: RMI Principles
SERVER Response: RMI Principles
Execution Time (ms): 2
```

d) Function 4: Accepts a file name and checks if a file is available in the current directory

From the client side user will provide a file name to server and server will check if that file is available in the current directory or not and displays a message accordingly. Such if the file exists then server will return "File Exists" and otherwise "File Doesn't Exist". The client side screenshot is given below for both cases:



```
nschowdh@in-csci-rrpc02:~/DS-Fall2017/Assignment3
Press
1 PWD
2 Sort
3 File Check
4 Echo
5 Matrix Multiplication
6 or others to QUIT

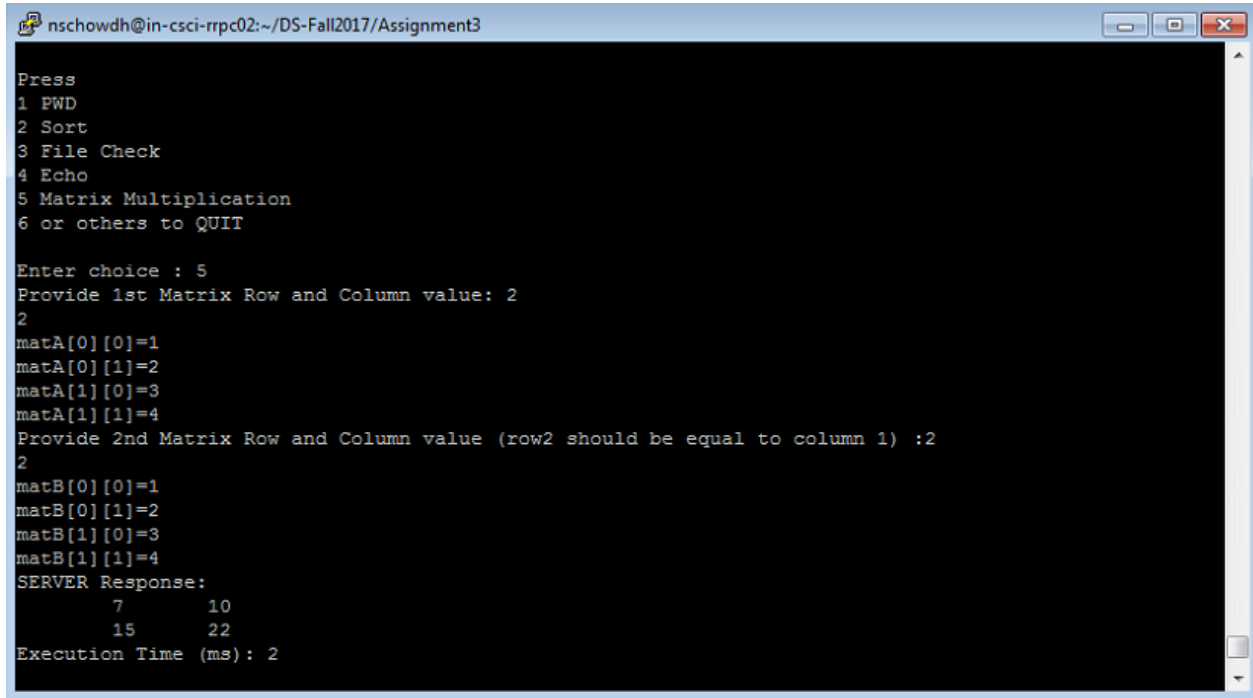
Enter choice : 3
Enter the File Name: test.txt
SERVER Response: File Exists
Execution Time (ms): 3

Press
1 PWD
2 Sort
3 File Check
4 Echo
5 Matrix Multiplication
6 or others to QUIT

Enter choice : 3
Enter the File Name: test-temp.txt
SERVER Response: File Doesn't Exist
Execution Time (ms): 1
```

e) Function 5: Accepts two integer matrices and returns their multiplication

Take two matrices from user and send it to server and then server response the result of multiplication. The client side screenshot is given below:



```
nschowdh@in-csci-rrpc02:~/DS-Fall2017/Assignment3
Press
1 FWD
2 Sort
3 File Check
4 Echo
5 Matrix Multiplication
6 or others to QUIT

Enter choice : 5
Provide 1st Matrix Row and Column value: 2
2
matA[0][0]=1
matA[0][1]=2
matA[1][0]=3
matA[1][1]=4
Provide 2nd Matrix Row and Column value (row2 should be equal to column 1) :2
2
matB[0][0]=1
matB[0][1]=2
matB[1][0]=3
matB[1][1]=4
SERVER Response:
    7    10
   15    22
Execution Time (ms): 2
```

4. Comparison between RPC and RMI

For the implemented five functionalities the response time from server side to receive a request and reply accordingly for both RPC and RMI are given below:

Table I: Roundtrip latency (RPC vs RMI)

Function Name	RPC (execution time in micro-sec)	RMI (execution time in ms)
Working Directory	2192.6	3.2
Sorting	439	3
Echo	267	1.6
File Check	942.8	2.6
Matrix Multiplication	268.7	3.4

Roundtrip Latency: Measure of latency using Java RMI and RPC is given in Table 1. In case of Java RMI, the daemon is responsible for all communications. It provides flexibility but it eats up the processing power of the machine, due to this reason Java RMI take more time to pass the message (high latency). The Java RMI has higher round trip latency when compared to RPC as the communication is done in terms of objects. When communication is done over the network using Java RMI, the objects are

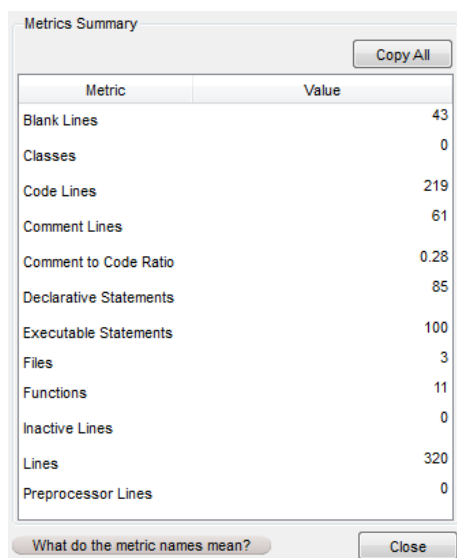
converted into bytes and then these bytes are transmitted over the network. The conversion of objects into bytes is a time consuming job. For these reasons, the Java RMI has the highest latency.

Because of its easy implementation nature RMI has definite advantages over RPC. Because it is implemented using Java it gets all the advantages like object oriented, parallel computing (Multi-threading), design pattern, easy to write and reuse, safe and secure, Write once and run anywhere. Also one can integrate RMI based application with any other application because of its platform independent ability.

RPC handles the complexities involved with passing the call from the local to the remote computer. RMI does the very same thing; handling the complexities of passing along the invocation from the local to the remote computer. But instead of passing a procedural call, RMI passes a reference to the object and the method that is being called.

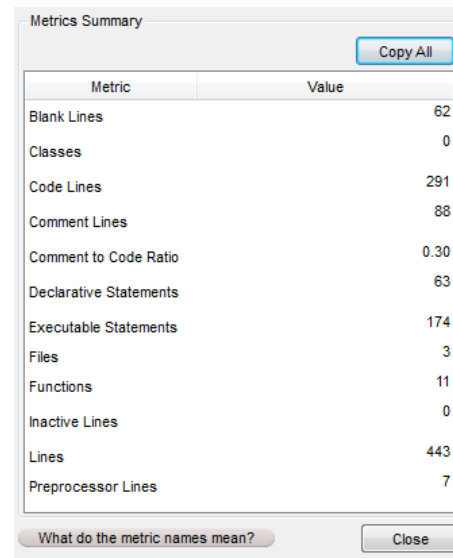
Code Size: In RMI principle it has the wide library to use so the code size becomes less than compared to the implemented code in RPC.

Table III: Code Size - RMI



Metric	Value
Blank Lines	43
Classes	0
Code Lines	219
Comment Lines	61
Comment to Code Ratio	0.28
Declarative Statements	85
Executable Statements	100
Files	3
Functions	11
Inactive Lines	0
Lines	320
Preprocessor Lines	0

Table II: Code Size - RPC



Metric	Value
Blank Lines	62
Classes	0
Code Lines	291
Comment Lines	88
Comment to Code Ratio	0.30
Declarative Statements	63
Executable Statements	174
Files	3
Functions	11
Inactive Lines	0
Lines	443
Preprocessor Lines	7

References:

1. Java RMI: https://en.wikipedia.org/wiki/Java_remote_method_invocation
2. <https://snowdream.github.io/115-Java-Interview-Questions-and-Answers/115-Java-Interview-Questions-and-Answers/en/rmi.html>
3. https://commons.wikimedia.org/wiki/File:Remote_Method_Invocation.png