Universidad Politécnica de Cartagena

Escuela Técnica Superior de Ingeniería de Telecomunicación

SISTEMAS Y SERVICIOS DISTRIBUIDOS

LAB *5*: PROGRAMMING OF DISTRIBUTED SYSTEMS (3)

JAVA REMOTE METHOD INVOCATION (RMI)

Profesores:

Antonio Guillén Pérez
Esteban Egea López

INDICE

1. Goals.

- Understanding RPC mechanisms and operation
- Introducing RMI Java libraries
- Programming distributed applications with Java RMI

Remote procedure call (RPC) mechanisms are explained in detail in the theory classes. The idea is to invoke methods (or functions) running on a machine or process different from the one that makes the invocation. Normally the process is executing on a different machine and the caller needs to send information across the network. And sometimes the devices involved have different characteristics (processor architecture, OS, etc.) so it is necessary to convert data to the appropriate format. The goal is to use a neutral communication mechanism (neutral with respect to platform and programming language).

The ultimate goal of RPC is to implement sharing and conversion of information in a standard manner, with the help of standard programming libraries to achieve transparency, ie, the programmer would work with a remote function as a function local.

The Java Remote Method Invocation, RMI, is conceptually similar to a generic RPC, albeit with some nuances. RMI works only between Java applications. That is, it is not independent of language. In contrast, RPC supports the use of different programming languages.

In addition, RPC is based on remote procedure invocation, while the use of RMI remote objects, ie, allows the exchange of generic objects as well as Java primitive types. By contrast, in a generic RPC, the data types that can be used are only those that can be represented by the XDR language, which does not contemplate the passage of objects. There are in fact other distributed systems technologies that allow the passage of generic objects and even support the use of multiple programming languages, such as CORBA. Java can be used with CORBA with other libraries.

In short, Java RMI is a system of remote procedure call for applications developed entirely in Java.

2. Operation of RPC and RMI.

RPC operation is to generate delegated functions for both the client and the server, called stub and skeleton respectively. These functions are invoked on the client as a normal function is invoked, but internally they create a connection to the server and send the function parameters to the server, properly formatted for delivery over the network, which, for remote objects is called serialization data.

Figure. 1. RMI operation

On the server side, the corresponding function (the skeleton) is responsible for receiving the data, converting them to the format of the server platform, and invoking the function on the server, passing parameters. Once the server response is available, the skeleton sends that response back through the network to the client. Upon receipt, the stub will convert the data to the format of the client platform and return them to the application that invoked it.

For this process to work it is necessary to define a service interface. The interface declares what functions can be invoked remotely on the server, which data types are allowed as parameters and the return types.

For a generic RPC mechanism, it is necessary to declare the service interface in a language independent programming language. Later a special compiler (interface compiler) automatically generates from the service interface the code stubs or skeletons in a given language. For example, a server can be implemented in C++ while the client is written in Java. The interface compiler generates the Java stub that would be used by a Java application to call remote invocation.

In the case of Java RMI, since both the server and client are implemented in Java, the operation is slightly different. Traditionally, we used a compiler interface that generates the stub code, but in the latest versions of Java this step is not even necessary.

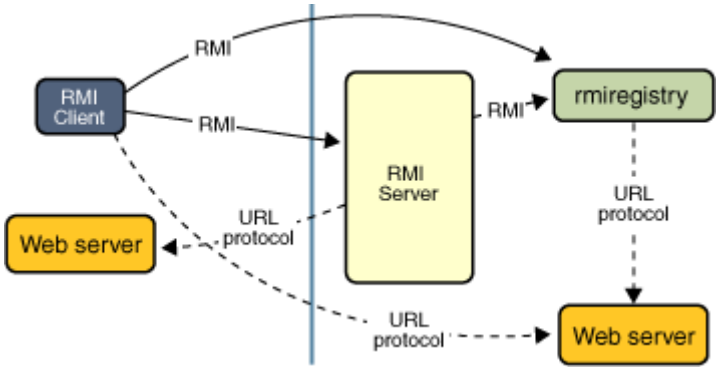

Figure. 2 RMI and Registry operation. The client obtains a reference to the remote object registry. The server previously registers a service.

Additionally, to separate the service of the machine, an intermediate element, called the registry, is used. Clients contact the registry and obtain what is called a reference to the remote object. Servers, meanwhile, register their services in the registry, so they can be found by the client.

3. Step-by-step development of a RMI Application

To clarify the operation of the RMI, we will implement a Hello World with RMI step by step. Subsequently you will implement a more sophisticated application.

4.1 Interface declaration

First step is always to declare the service interface. In our case:

```
import java.rmi.*;

public interface HelloRMI extends Remote {
      String sayHello(String name) throws RemoteException;
      long getTime() throws RemoteException;
}
```

Note that it is mandatory that all functions throw a RemoteException. Otherwise it is declared as usual except that it derives from the Remote interface.

Note it allows the remote invocation of two methods, the sayHello and getTime methods

Declaration of objects to be remotely exchanged

In the declaration of the previous interface, a primitive type (long) as well as an object of the String class are exchanged. **To allow a class to be exchanged remotely, it must be able to be serialized. For that, it needs to implement the interface java.io.Serializable**. For example, if you define an Invoice class, which can be used in a remote interface, ie, can be passed as a parameter and can be obtained as a result of the invocation, its declaration would have to be like this: `public class Invoice implements java.io.Serializable {}`. The java.io.Serializable interface does not define any method, there is no need to implement any additional method. It is used as a tag that tells the compiler that the object can be serialized. Check in the documentation that the String class implements that interface.

4.2 Interface implementation

The next step is to implement the interface, ie, implement the functions that actually run on the server. In our case:

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

public class HelloRMIImplementation extends UnicastRemoteObject
implements HelloRMI {
      private static final long serialVersionUID = 1L;

      public HelloRMIImplementation() throws RemoteException {
            super();
      }
      @Override
      public long getTime() throws RemoteException {
           return System.currentTimeMillis();
      }

      @Override
      public String sayHello(String name) throws RemoteException {
            return ("Hello "+name);
      }
        public static void main(String[] args)throws Exception {
          HelloRMIImplementation pt = new HelloRMIImplementation();
          Naming.rebind("rmi://localhost/HelloRMI", pt);
          System.out.println("Ready to say Hello");
        }
}
```

The implementation class must extend UnicastRemoteObject and implement the service interface as well. Additionally it includes the `main` method with the server implementation that launches the service (HelloRMIImplementation instance) and registers it in the Registry (Naming.rebind). In general, you should separate the service implementation of the server that executes it, ie, functionality that has been included in main should be included in its own class. As a simple example, we have implemented them together.

4.3 Run the registry

Java JDK includes a Registry that can be run in different platforms.

In Linux, open a console, **go to the directory where the classes are** and execute: `rmiregistry`

In Windows, **do the same steps** but execute: `start miregistry`

You can use a parameter to specify the port that will listen. If not indicated port 1099 is used by default. Once the server has registered the HelloRMIImplementation object this will remain active on the register until it is closed, even if the server has finished its execution, as can be seen in the main method.

Please note that the registration should be able to find the classes to be loaded, ie, the CLASSPATH must be set correctly. In general, **if the above command is executed in the same directory where the classes are**, **the Registry is able to find them**. Otherwise, the process is more complex and you should read the documentation.

4.5 Client implementation

The next step is the implementation of the client application that use remote invocations. In our case:

```
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;

public class HelloRMIClient {
      /**
       * @param args
       * @throws NotBoundException
       * @throws MalformedURLException
       */
      public static void main(String[] args) throws RemoteException,
MalformedURLException, NotBoundException {
            HelloRMI t =
            (HelloRMI)Naming.lookup("rmi://localhost/HelloRMI");
            System.out.println("Hola server...= " +
                        t.sayHello("Pepe")+ "Hora actual="+t.getTime());

                  }
      }
```

Note that the client gets an instance of **HelloRMI interface**, not the implementation class, through the Registry. In fact, the client does not need to know the implementation of the methods. All it needs is to know the shared interface, ie, the declaration of the functions offered. Once it has the reference to the instance, it can invoke its methods as if they were a local call.

4.5 Run the service [Optional]

Although with the previous steps apparently we have concluded the development, the truth is that it is still necessary to properly configure the service. This is because it is necessary to establish a series of security and access control mechanisms among other issues. In fact, this is in many cases the most problematic step in the deployment of RMI applications. It requires careful study of the documentation and quite practical in real cases.

We must **invoke the Java Virtual Machine (JVM)** with the following parameters:

" -Djava.rmi.server.codebase=file:///home/student/workspace/p1/bin" [Linux]

" -Djava.rmi.server.codebase=file:/C:/Users/student/workspace/p1/bin" [Windows]

REPLACE THE PATHS IN THIS EXAMPLE WITH THE **ACTUAL PATH IF** APPLICABLE, YOUR OPERATING SYSTEM AND **REMOVE THE QUOTES]**

The route is to where the .class of the implementation is.

To pass parameters to the JVM or application, select the file class and right-click Run As> Run Configurations ... and select the Arguments tab.

In addition, make sure that machines in both client and server have an entry in the /etc/hosts with a valid IP assigned to the computer.

4. Exercises

Use Eclipse to implement the following programs:

a) Develop and execute the HelloRMI application described in the previous sections.

- To do this, first create the *HelloRMI* class, which will be the interface that will extend Remote and will contain the skeletons (the name of the method, the necessary parameters, the type of object it returns) of the methods that can be invoked. It is mandatory that all functions launch a *RemoteException*
- Create the implementation of the interface, for it create the class *HelloRMIImplementation*, where in this class are defined the methods that can be invoked remotely. The class should extend *UnicastRemoteObject*.
- Implement a *HelloRMIClient* client, where you connect to the registry, and make use of the shared functions.
- Start the registration. To do this, depending on your OS, you must run *rmiregistry* or *start rmiregistry* in the same folder where you have the .class implemented (eg **F:\practica5\bin**). Also, if you are on Windows, you must first save in the system environment variables the path to the folder where the program *rmiregistry* is hosted. That is to say, look for where you have java installed, and inside the bin folder, it is that path the one you must add to the Windows PATH. (i.e.: **C:\Program Files\Java\jre1.8.0_261\bin**)
- Optional], add the parameters for the Java Virtual Machine (JVM).

b) Modify your client to pass the Registry IP address as a parameter to the program. Provide your client to a partner and ask him to run. Run wireshark and describe the exchange of messages between client and server.

c) Develop an RMI application for matrix calculation. Clients will pass matrices to the server to perform operations on them. The operations that the server provides are addition, subtraction and multiplication of matrices. To do this, first implement a Matrix class that represents a generic NxM matrix of real numbers. The dimensions of the matrix will be indicated in the constructor. The matrix includes a method to display it on the screen. Provide methods to give values to the matrix elements.

One of the objectives of this exercise is to verify that indeed Java RMI allows generic object exchange between client and server.

- Create the *Matrix* class where you will have basic matrix methods implemented. The constructor will allow us to create an empty matrix, indicating the size of the matrix (square, or indicating the number of rows and the number of columns, can be one or several constructors). With a set method it will allow us to indicate a row and a column and a value, and the method adds the value to the array. With get we get the value of a row and column. A couple of methods to know how many rows and columns an array has (*rowLength* and *colLength*). Finally, a *toString* method that will go through the array by rows and columns, printing on screen the value of each cell.
- The *MatrixServer* class will be where all matrix operations are implemented. As a first exercise, develop the addition and subtraction of matrices (*add(Matrix A, Matrix B)* and *substract(Matrix A, Matrix B*)). For this, remember how they were operated with matrices. As an extension, you can implement matrix multiplication, determinant, convolution, and anything else you can think of.
- The *MatrixAlgebra* interface will indicate the methods implemented by *MatrixServer*.
- Finally, implement the *MatrixClient* class, which will connect to the register, create some empty Matrix objects, fill them with random values (either requested by keyboard, or determined by the row/column), print the matrices on the screen, and send them to operate. The result is also printed on the screen.

d) Verify that the application works correctly between different machines and capture the exchanges with wireshark.

e) There are more interesting RMI applications, for example, it can be used to run a generic task in a server, ie, any Java program. Do the tutorial at

http://docs.oracle.com/javase/tutorial/rmi/overview.html

Bibliografía

1. Bruce Eckel, Thinking in Java, 4th Edition, Prentice Hall, 2006

(Las versiones anteriores están disponibles gratuitamente en formato electrónico en http://www.mindviewinc.com/Books/downloads.html)

2. David Reilly, Java Network Programming and Distributed Computing, Addison-Wesley, 2002.

Al Williams, Java 2 network protocols black book, Coriolis Groups Books, 2001.