Programming with RMI – Reminder

(Sources: Gordon S Blair, Paul Grace)

Aims

After completing the following you should get a reminder of:

1. the fundamental concepts of Java Remote Method Invocation;
2. the steps involved in designing and implementing simple distributed applications using Java RMI.

Introducing RMI

The goal of the Java RMI package is to extend the Java object model to support programming with distributed objects. The intention is to make such distributed programming as easy as standard Java programming (cf *transparency* as introduced in Unit 1). In particular, using RMI it is possible to invoke methods on remote objects using exactly the same syntax as for local objects.

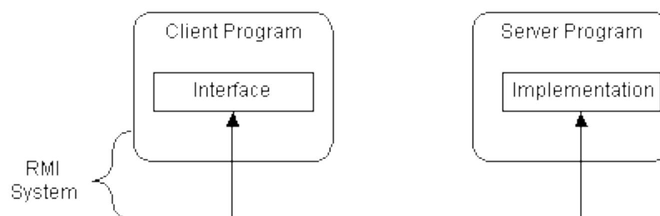
In practice, the use of RMI is visible to the programmer in a number of key ways:

- An object must be aware that it is making a remote call as it must handle RemoteExceptions;
- The implementor of a remote object is also aware of its status as the object must implement the Remote interface;
- The semantics of parameter passing are also different (see below).

Programming with Interfaces

The architecture of RMI builds on the concept of interfaces, again as introduced in Unit 1. In particular, the definition of a remote object is specified by its interface. Interfaces specify the services (methods) that should be made publicly available by an object, while classes define how these services are implemented. One advantage of RMI over other similar technologies such as CORBA is that RMI is based entirely on Java. This means that interfaces can be specified in Java, and there is no need to introduce a separate (language-independent IDL).

This reliance on interfaces is captured in the following diagram:



This approach has the important advantage of separating interface from implementation a key element of sound software engineering practice.

Parameter Passing in RMI

In Java RMI, any parameters of a method invocation equate to *input* parameters and the result of the method is the single *output* parameter (using the terminology introduced in the lecture). Input parameters are hence marshalled and sent to the remote object, with the result being marshalled and sent back to the calling object after the execution of the method.

Parameter passing in RMI is closely related to the Serializable interface in Java (http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html). In particular, any object that is serializable, i.e. implements the Serializable interface, is marshalled and copied by value. The original object remains at the host site. It is important to appreciate this when programming with Java RMI - a copy is made of the object, and the copy and the original may diverge.

If however the type of a parameter or the result is a remote interface, the corresponding argument or result is passed by reference. Subsequent method invocations on the object are then invoked using RMI.

If you want to learn more about the use of serialisation within RMI, you can look at Oracle's documentation on this topic:

http://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmi-objmodel.html

N.B. If an object is passed by value, and the destination site does not have a copy of the class, then this class will be *downloaded automatically*.

The Role of the Registry

In systems such as Java RMI, it is necessary to provide a naming service that enables the programmer to locate remote interfaces. RMI provides a simple service referred to as the RMIregistry. A copy of this service must run on any computer offering remote interfaces. The task of the RMIregistry is quite simple: it maintains a table mapping URL style names to interface references.

This raises the question of how RMI locates the RMIregistry. To achieve this, the registry must run on a well known port on the local machine, namely port 1099.

The URL for names is as follows:

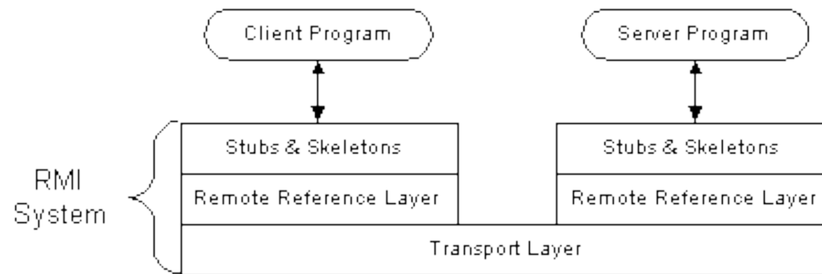
rmi://<host_name> [:<name_service_port>] / <service_name>

The <name_service_port> need only be specified if the registry runs on a different port from the default.

The remote interface for the registry can be found at http://docs.oracle.com/javase/7/docs/api/java/rmi/registry/Registry.html. You will see examples of the use of the registry below.

Implementation

The overall implementation of RMI is shown in the diagram below:



This is a fairly *classical* RPC style architecture, building on TCP as the transport protocol. The one interesting feature is that Java 1.2 and above (which you will be using) employs the concept of *reflection* to simplify the server side. In particular, reflection is used to implement a generic dispatcher at this end of the connection, replacing the need for individual skeletons.

Java 1.2 introduced the feature whereby remote objects that are not running can be automatically *activated* on invocation. Previously, it was necessary to ensure that such objects were already executing in order to be able to receive incoming invocations.

To find out more about remote activation of objects, read the following links:

http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/activation/overview.html

http://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmi-activation.html

A simple RMI example –The Distributed Calculator

Scenario

The task is to create a simple distributed system that performs the functionality of a remote calculator service. There is a single client and a single server. The server provides a set of arithmetic methods {add, subtract, multiply, divide & power} that can be remotely invoked by the client. Therefore, the server receives a request from the client, performs the arithmetic operation and then returns the result back to the client.

Using RMI

An RMI system must be composed of the following parts:

1. An interface definition of the remote services that are provided;
2. The implementations of the remote services;
3. A server to host the remote services;
4. An RMI Naming service that allows clients to find the remote services;
5. A client program that uses the remote services.

In the following sections, you will build a simple RMI system in a step-by-step fashion. At each step you will create one of the above components of the RMI system.

Creating the Interface

The first step in the process is to write and compile the interface for the remote service. In this case, the interface describes the set of methods provided by the remote calculator. This consists of the name of each method, the type of the value that is returned to the client and the list of parameters that each method takes:

- **add** describes the addition method. It returns a long value and takes two parameters of type long.
- **sub** describes the subtraction method. It returns a long value and takes two parameters of type long.
- **mul** describes the multiplication method. It returns a long value and takes two parameters of type long.
- **div** describes the division method. It returns a long value and takes two parameters of type long.
- **pow** describes the power method. It returns a long value and takes two parameters of type long.

We are only interested in providing a description of the available remote methods, so no implementation is included in the interface. For a reminder about programming with interfaces click here.

You will find the source of the calculator interface in the file calculator.java.

Notice that the interface extends Remote (http://docs.oracle.com/javase/7/docs/api/java/rmi/Remote.html) and each of the methods throws the RemoteException (http://docs.oracle.com/javase/7/docs/api/java/rmi/RemoteException.html). This allows the client to detect when an exception is generated due to a communication-related problem in the remote call. For example, the host may be unreachable.

1. Copy and paste the interface code into a new file using Your preferred editor.

2. Save the file as **calculator.java** in a new folder on your H:// drive.

(Alternatively copy the file directly to your H: folder from the link above)

3. Finally, compile the interface using the following command at the command prompt:

H:\yourfolder>javac calculator.java

Implementation of the remote service

The next step is to provide the implementation of the remote service as a remote object. In this case, a class called CalculatorImpl is defined. It contains the implementation code for each of the methods identified in the interface.

You will find the source of the calculator interface in the calculatorimpl.java file.

N.b. The CalculatorImpl class uses UnicastRemoteObject to link into the RMI system. That is, it states that this is a remote object whose references are only valid while the server hosting it is still alive. When a class extends UnicastRemoteObject, it must provide a constructor that declares that it may throw a RemoteException object. When this constructor calls super(), it activates code in UnicastRemoteObject that performs the RMI linking and remote object initialization.

4. Copy and paste the implementation code into a new file using Your preferred editor.

5. Save the file as **calculatorimpl.java** in the same folder on your H:// drive.

(Alternatively, save calculatorimpl.java directly to your H: folder from the link above.)

6. Finally compile the implementation class using the following command at the command prompt:

H:\yourfolder>javac calculatorimpl.java

Create the Host Server

We have created the implementation class for the remote object that provides the arithmetic methods, but we still need to create the server part of the distributed system to host this object. The following code provides a very simple server that will perform the hosting.

You will find the source of the calculator interface in the file calculatorserver.java.

The server first constructs a new instance of the CalculatorImpl object. Then it binds it to the naming service as described above. This is just a simple call of the Naming.rebind (http://docs.oracle.com/javase/7/docs/api/java/rmi/Naming.html) method with the URL of the remote service and the object reference. Note, in this demonstration we are only using a local machine to run both the client and the server. However, they can be run on separate machines. You can experiment, by using the machines' network addresses in the URL to achieve this if you wish.

9. Copy and paste the server code into a new file using Your preferred editor.

10. Save the file as **calculatorserver.java** in the same folder on your H:// drive.

(Or save it directly from the link above)

11. Finally compile the server using the following command at the command prompt:

H:\yourfolder> javac calculatorserver.java

Creating the Client

The final step in the implementation of this RMI system is to create a client that uses the remote methods that we have defined. To keep this as simple as possible, the client will simply call each of the methods once with a set of parameters and print the results of these calculations to the screen.

You will find the source of the calculator interface in the file calculatorclient.java.

The important points to note are:

- You must create a local reference to the remote object using the Naming.lookup (http://docs.oracle.com/javase/7/docs/api/java/rmi/Naming.html) method. In this case 'c' is the local reference.
- It is now possible to invoke methods in the same manner as a local call.
- However, all remote calls must be placed in a try/catch statement. This ensures that remote and network exceptions are dealt with.

12. Copy and paste

the client code into a new file using Your preferred editor.

13. Save the file as **calculatorclient.java** in the same folder on your H:// drive.

(Or save it directly to disk from the link above)

14.Finally compile the client using the following command at the command prompt:

H:\yourfolder> javac calculatorclient.java

Running the RMI system

All the individual pieces of the RMI system have now been created and they can be put together to run the distributed calculator service.

15. Open a new command prompt and go to the directory that contains the calculator files.

16. Start the RMI registry using the following command:

H:\yourfolder> rmiregistry

17. Open a new command prompt and go to the directory containing the calculator files.

18. Start the server process using the following command:

H:\yourfolder>java calculatorserver

19. Open a new command prompt and go to the directory containing the calculator files.

20. Start the client using the following command:

H:\yourfolder> java calculatorclient

You should see the following on the client's screen:

3+21 = 24

18-9 = 9

4*17 = 68

70/10 = 7

2^5 = 32

Instructions

2.1 Work through the steps described in the section above to create a fully functioning RMI-based remote calculator service.

2.2 Create a new client-server application for encrypting and decrypting messages. Therefore, you must provide a single server hosting a remote object that encrypts and decrypts message strings using the Caesar cipher. You must also implement a single client that requests messages to be encrypted or decrypted. If you know nothing about the Caesar cipher, go at http://acm.uva.es/p/v5/554.html. If you're feeling ambitious you can use any encryption method you know (Vigenere, transposition, Affine etc.).

To build the distributed system carry out the following steps:

1. Create the interface, defining two methods called encrypt and decrypt that both take a single String parameter (the message) and an integer parameter (the key) and return a String as the result. {*Use the interface – Calculator.java as your basis*}

2. Create the implementation of the remote service. Therefore, implement a remote class with two methods: one for encryption and one to decrypt messages.
3. Create the stub and skeleton files for your new RMI system.
4. Create a server process to host your remote service.
5. Create a client that requests messages to be encrypted and decrypted.

Hint: You may need to split the String into a char array to manipulate the letters. To do this use the toCharArray() method from java.lang.String:

char[] tempArray = nameofString.toCharArray();

For more information about String manipulation examine the (http://docs.oracle.com/javase/7/docs/api/java/lang/String.html) documentation.