RMI Programming

Ph. Truillet

October 2021 – v. 2.3

0. Preamble

RMI (**R**emote **M**ethod **I**nvocation) is a technology developed and provided by Sun from JDK 1.1 onwards to allow easy implementation of distributed objects using TCP/IP.
In this tutorial, you will have to:

- execute an example illustrating the architecture of RMI
- develop a Yellow Page service using RMI

1. introduction

RMI allows a remote method call belonging to a remote object, i.e. managed by another JVM (Java Virtual Machine). This, it becomes possible for so-called client applications (running locally) to invoke methods on remote objects located in an application called "*server*".
The objectives of RMI are to allow the call, execution and management of the result of a method executed in a virtual machine different from that of the object calling it. This virtual machine can be launched on a different machine as long as it is accessible via the network.
The client-side call of a remote method is a little more complicated than calling a method of a local object but it is still simple. It consists in obtaining a reference on the remote object and then simply calling the method from this reference.

Principle:
The RMI mechanism allows an application running on an M1 machine to create an object and make it accessible to other applications running in another JVM: this application and the M1 machine thus act as a server.

Other applications that manipulate such an object are clients. **To manipulate a remote object, a client retrieves on its machine a representation of the object** also called a *stub*: this stub implements the interface (the Java meaning) of the remote object and it is via this stub that the client will be able to invoke methods on the remote object. Such an invocation will be transmitted to the server (thanks to the TCP protocol) in order to execute it (cf. figure 1).

On the server side, a skeleton is in charge of receiving remote invocations, performing them and returning the results. The RMI technology thus makes the location of the remote object, its call and the return of the result transparent.

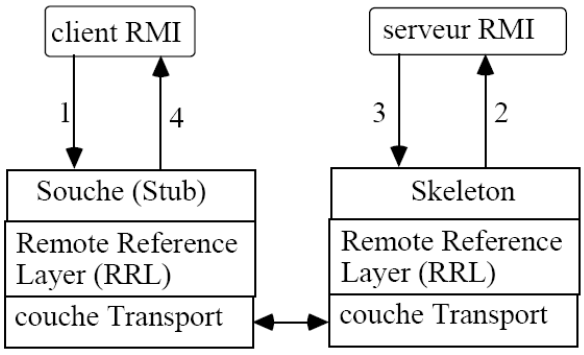


Figure 1: how RMI works

The two particular classes, *stub* and *skeleton*, are generated with the `rmic` tool provided with the JDK. As explained above, the *stub* is a *client-side* class and the *skeleton* is its *server-side* counterpart. These two classes take care of all the mechanisms of calling, communicating, executing, returning and receiving the result.

2. Steps to create a remote object and call it with RMI

The server side development consists of:

- the definition of an interface that contains the methods that can be called remotely

- writing a class that implements this interface
- writing a class that will instantiate the object and register it with a name in the RMI Registry

Finally, the *stub* and *skeleton* classes must be generated by running the **`rmic`** program with the source file of the remote object.

Client-side development consists of:

- **obtaining a reference to the remote object from its name**
- **and the call to the method from this reference**

3. server-side development

3.1. definition of an interface that contains the methods of the remote object

The interface to be defined must inherit from the `java.rmi.Remote` interface. This interface does not contain any methods but simply indicates that the interface can be called remotely.

The interface must contain all the methods that are likely to be called remotely.
The communication between the client and the server during the invocation of the remote method can fail for various reasons such as a server crash, a broken link, etc.

Thus, each method called remotely must declare that it is able to raise the exception `java.rmi.RemoteException`.

3.2. writing a class that implements this interface

This class corresponds to the remote object. It must therefore implement the defined interface and contain the necessary code.
This class must necessarily inherit from the `UnicastRemoteObject` class which contains the various elementary treatments for a remote object whose call by the client stub is unique. The stub can only obtain a single reference on a remote object inheriting from `UnicastRemoteObject`.

The hierarchy of the `UnicastRemoteObject` class is:

- `java.lang.Object`
- `java.rmi.Server.RemoteObject`
- `java.rmi.Server.RemoteServer`
- `java.rmi.Server.UnicastRemoteObject`

As stated in the interface, all remote methods must indicate that they can raise the `RemoteException` as well as the class's constructor. Thus, even if the constructor does not contain any code, it must be redefined to inhibit the generation of the default constructor which does not raise this exception.

3.3. writing a class to instantiate the object and register it in the registry

These operations can be performed in the `main` method of a dedicated class or in the main method of the remote object's class. The advantage of a dedicated class is that it allows all these operations to be grouped together for a set of remote objects.
The procedure contains three steps:

- setting up of a dedicated security manager (optional)
- instantiation of an object of the remote class
- registering the class in the RMI name register by giving it a name

3.3.1. security manager

This operation is not mandatory but it is recommended in particular if the server needs to load classes which are not on the server. Without a security manager, it is mandatory to provide the server with all the classes it will need (they must be in the server's `CLASSPATH`). With a security manager, the server can dynamically load certain classes. However, the dynamic loading of these classes can cause security problems because the server will execute code from another machine. This aspect can thus lead to not using a security manager.

You can specify a security policy at server startup using the **-D** option:

```
java -Djava.security.policy= policyfilename serveur
```

The syntax of the security file is described in **docs/technotes/guides/security/PolicyFiles.html** in the JDK distribution.
Here is an example of a security file that allows all actions!!

```
grant {
        permission java.security.AllPermission;
};
```

It is also possible to allow only certain operations, such as:

```
permission java.net.SocketPermission "*:8080", "connect, accept";
```

3.3.2. instantiation of an object of the remote class

This operation is very simple as it simply consists of creating an object of the remote object's class.

3.3.3. registration in the RMI registry by giving it a name

The last operation consists of registering the object created in the name register by assigning it a name. This name is provided to the registry in the form of a URL consisting of the prefix `rmi://`, the name of the server (*hostname*) and the name associated with the object preceded by a slash.
The server name can be supplied "hard" as a constant string or can be dynamically obtained using the `InetAddress` class for local use.

It is this name that will be used in a URL by the client to obtain a reference to the remote object.
The registration is done using the `rebind` method of the `Naming` class. It expects the URL of the object's name and the object itself as parameters.

3.3.4. Dynamic launch of the RMI registry

On the server, the RMI name register must run before an object can be registered or a reference obtained.
This registry can be launched as an application provided by SUN in the JDK (**rmiregistry**) as shown below or be launched dynamically in the class that registers the object (see below).

```
/* Start rmiRegistry at the server level */
try {
        java.rmi.registry.LocateRegistry.createRegistry(1099); // port 1099
        System.out.println("RMI registruy is started.");
}
catch (Exception e) {
        System.out.println("RMI registry could not be launched.");
}
```

This launch must only take place once. It can be interesting to use this code if you create a class dedicated to registering remote objects. The code to run the registry is the `createRegistry` method of the `java.rmi.registry.LocateRegistry` class. This method expects a port number as a parameter.

4. Client-side development

The call of a remote method can be done in an application or in an applet.

4.1. security manager

As for the server side, this operation is optional. The choice of setting up a security manager on the client side follows the same rules as on the server side. Without its use, it is necessary to put in the `CLASSPATH` of the client all the necessary classes including the *stub* class.

4.2. obtaining a reference to the remote object from its name

To obtain a reference to the remote object from its name, you must use the static `lookup`() method of the `Naming` class.
This method expects as a parameter a URL indicating the name that references the remote object. This URL is composed of several elements: the prefix `rmi://`, the *hostname* and the name of the object as registered in the registry preceded by a slash.
It is preferable to provide the server name as a parameter of the application or applet for more flexibility.
The `lookup()` method will search the server registry for the object and return a stub object. The returned object is of class `Remote` (this class is the parent class of all remote objects).
If the name provided in the URL is not referenced in the registry, the method throws the `NotBoundException`.

4.3. Calling the method from the reference on the remote object

The returned object being of type `Remote`, it is necessary to carry out a casting towards the interface which defines the methods of the remote object. For more security, we check that the returned object is an instance of this interface.
Once the casting is done, you simply call the method.

4.4. calling a remote method in an applet

Calling a remote method is the same in an application as in an applet.
Only the implementation of a dedicated *security manager* in applets is unnecessary as they already use a security manager (`AppletSecurityManager`) which allows the loading of remote classes.

5. generation of stub and skeleton classes

To generate these classes, simply use the `rmic` tool provided with the JDK, giving it the name of the class as a parameter. The class must have been compiled; `rmic needs` the `. class` file.

Note: under Eclipse, this step is done automatically!

Exemple:

```
rmic -cp . -keepgenerated HelloServeur
REM keeps the generated stub and skeleton files
```

6. Implementation of RMI objects

Implementing and using remote objects with RMI requires several steps:

- start the RMI registry on the server either by using the **`rmiregistry`** program that comes with the JDK or by running a class that performs the launch.
- run the class that instantiates the remote object and registers it in the RMI name server
- launch the application or applet to test it.

6.1. launch RMI registry

The `rmiregistry` command is provided with the JDK. It must be run in the background:

- Under Unix: **`rmiregistry&`**
- On Windows: **`start rmiregistry`**

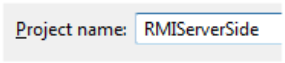
This register allows an object to be matched to a name and vice versa. It is called when the `Naming.bind()` and `Naming.lookup()` methods are called. The register is launched on port `1099` by default.

6.2. instantiation and registration of the remote object

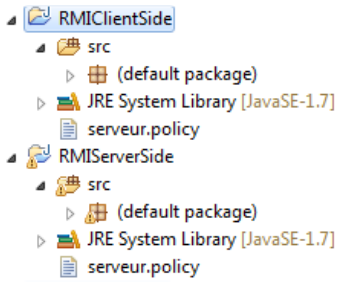
The class that will instantiate the remote object must be executed and registered under its name in the previously launched registry.
In order not to have any problems, it is necessary to ensure that all the useful classes (the class of the remote object, the interface which defines the methods, the *skeleton*) are present in a directory defined in the **`CLASSPATH`** variable.

7. exercices

Get `RMI.zip` file from:
`https://github.com/truillet/upssitech/blob/master/SRI/3A/ID/TP/Code/RMI.zip`



Create a project in Eclipse (**File** | **New** | **Java Project**)
Add the files located in the **`RMIServerSide/src`** folder of RMI.zip to your project (**src** directory). Copy the **`server.policy`** file into your project..



Launch your project: the server is operational.

Do the same with (the new) **`RMIClientSide`** project
What is going on? Then analyse the source code of the server and client parts.

Note: The port used may already be **in use**: consider changing it if necessary.

Based on the previous example, develop a remote method calling system that **allows the management of a shared "Yellow Pages" type directory** using RMI.
The directory will be composed of a set of people.
A person will be defined at least by a **number,** a **surname/first name** and a **telephone number** (or even a **photo**).
The operations (*methods*) **of initialising, adding, deleting** and **viewing** from the name will be available.
Write the java interface file as well as the server part and an example of a client that uses the directory and performs the 4 operations defined above.

Note: remember to serialize objects!

8. bibliography

- `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html`
- `http://cedric.cnam.fr/~farinone/IAGL/rmi.pdf`
- `http://www.ejbtutorial.com/java-rmi/a-step-by-step-implementation-tutorial-for-java-rmi`