Remote Method Invocation in JAVA

Philippe Laroque
Philippe.Laroque@dept-info.u-cergy.fr

$Id: rmi.lyx,v 1.2 2003/10/23 07:10:46 root Exp $

Abstract

This small document describes the mechanisms involved in remote method invocation (RMI) in the JAVA language.

1 Introduction

The aim of the RMI architecture is to allow the programmer to invoke services from remote objects (almost) the same way as from local objects.

To achieve this, RMI is designed into three independent layers. The programmer only explicitly deals with the topmost one (stubs and skeletons).

To illustrate RMI usage, we detail the server-side tasks to offer services to remote clients, then the client-side tasks to invoke those services form the remote, server objects.

2 Architecture layers

2.1 The role of interfaces

As usual in modern OO languages, most of the complexity is hidden behind the interface paradigm, which allows to separate behavior *description* and *implementation*. The fig. 1 shows how the RMI stuff is out of the client scope, who only deals with the exposed interface of the remote server object.

2.2 Stubs (& skeletons)

Of course, having only an interface on the client side of the application is not sufficient. One needs some way to "talk" with the server object. This is achieved via a (transparent) proxy object called the "*stub*" (see fig. 2). This object is generated by a JDK tool, `rmic`, from the server object compiled code, and distributed to the client. In older versions of RMI, there was a need for one more object, called the skeleton, which was responsible for the communication between the stub and the remote object, but it is no longer necessary since JDK

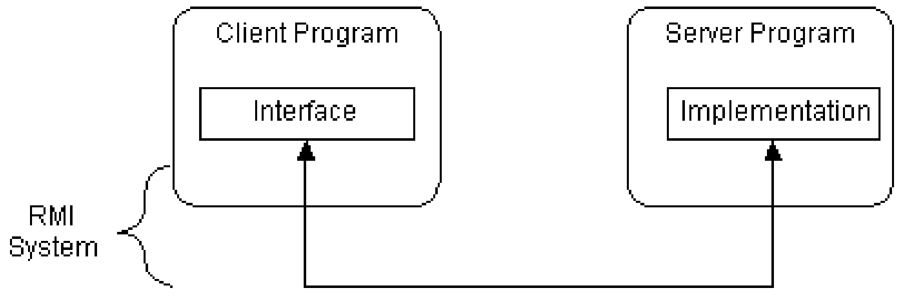

Figure 1: The interface paradigm is the heart of RMI system

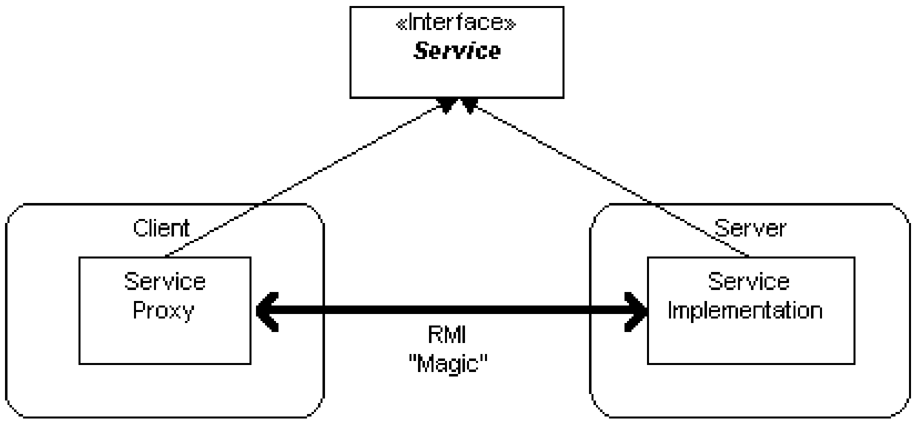

Figure 2: The role of the stub in the RMI communication

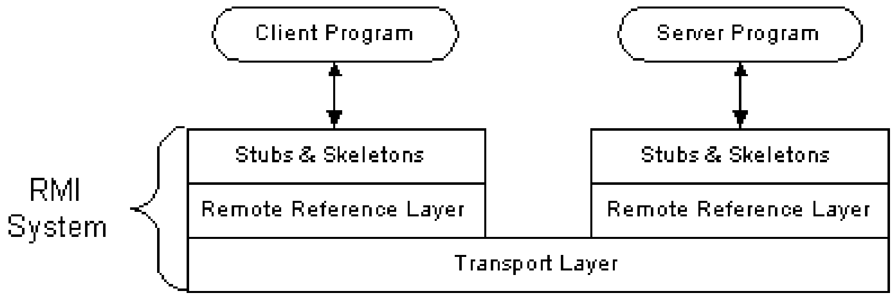

Figure 3: The RMI architecture layers

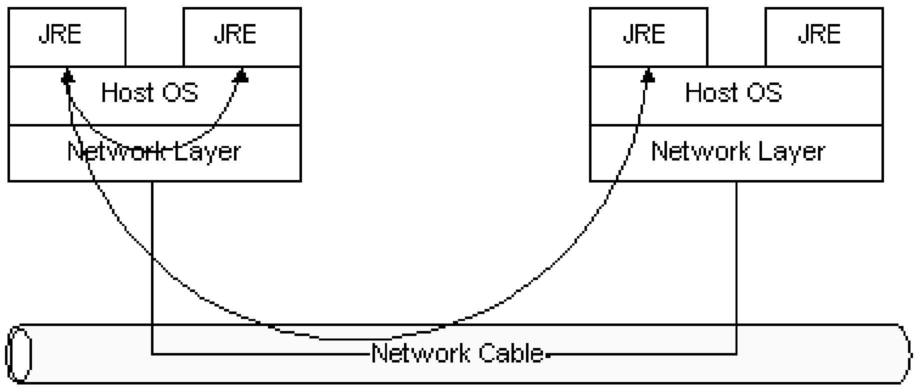

Figure 4: The RMI transport layer

1.3. The way this stub can be brought to the client is discussed at the end of this document (see 3.4).

2.3 Remote references

The second layer of the RMI architecture (see fig. 3)deals with the interpretation of references made from client to the server, remote objects. Since JDK 1.2, it is able to wake up a "dormant" remote object as needed by the client, *via* what is known as *Remote Object Activation*. The communication is made in a point-to-point (unicast) basis

2.4 Transport

This layer is responsible for connecting the two JVMs (see fig. 4). It uses TCP/IP, even if both client and server are on the same machine. In previous versions, only one protocol (JRMP, Java Remote Method Protocol) was available above TCP/IP. Since JDK 1.3, IIOP (Internet Inter-Orb Protocol, the basis for CORBA communications) is also available, which allows for RMI applications to use CORBA objects too.

3 Usage

The main question when it turns to distributed objects is "where can clients find an RMI remote service?". The solution is to use a *naming service*, which is available on a publicly-defined server/port. RMI can use any naming service (for exeample JNDI), but defines its own service, the RMI *registry*. The standard port for this service is 1099. The registry is handled by a standard JDK tool, `rmiregistry`.

The typical steps involved in building a distributed application with RMI include:

- Interface definitions for the remote services
- Implementations of the remote services
- Stub (and Skeleton) files
- A server to host the remote services
- An RMI Naming service that allows clients to find the remote services
- A class file provider (an HTTP or FTP server)
- A client program that needs the remote services

We illustrate those steps with a simple example of a Counter class. A Counter is an object that exposes an integer, non-negative value which can only be incremented or decremented by one.

3.1 Server side

3.1.1 Interface definition and compilation: file `Counter.java`

The interfaces only have to extend `java.rmi.Remote`, and the declared methods must throw `java.rmi.RemoteException`:

```
// $Id: Counter.java,v 1.1 2003/10/23 06:45:25 root Exp $

package pl.rmi;
import java.rmi.*;

/**
 * The Counter interface exposes the object's *behaviour*.
 * As such, it must be present on both sides of the RMI channel.
 * <p>
 * Every method has to be declared as throwing RemoteException
 * @author Philippe.Laroque@dept-info.u-cergy.fr
 * @version $Id: Counter.java,v 1.1 2003/10/23 06:45:25 root Exp $
 */
public interface Counter extends java.rmi.Remote {

    /**
     * read-only access to the counter value
     */
    public int value() throws RemoteException;

    /**
     * increment the counter value by one
     */
    public Counter incr() throws RemoteException;

    /**
     * decrement the counter value by one (if non-null)
     */
    public Counter decr() throws RemoteException;
}
```

3.1.2 Implementation of the remote services: file `CounterImpl.java`

The remote object class must implement the `java.rmi.server.UnicastRemoteObject` interface, or define an `exportObject()` method:

```
// $Id: CounterImpl.java,v 1.1 2003/10/23 06:45:25 root Exp $

package pl.rmi;
import java.rmi.*;

/**
 * Implementation of the interface published by Counter.
 * <p>
 * As a Remote Object, CounterImpl must:
 * <ul>
 * <li> extend the UnicastRemoteObject class
 * <li> declare, for every method, that it throws RemoteException
 * </ul>
 * N.B. : to be used by remote client, the stub must be placed in the correct
 * subdir of the http/ftp server, relatively to package name
 *  (here for instance, in http://theServer/rootStubDir/pl/rmi)
 * @version  $Id: CounterImpl.java,v 1.1 2003/10/23 06:45:25 root Exp $
 * @author  Philippe.Laroque@dept-info.u-cergy.fr
 */
public class CounterImpl extends java.rmi.server.UnicastRemoteObject implements Counter {
    /**
     * actual counter value
     */
    protected int value;

/////////////////////////////////////////////////////////////
/**
 * The unique constructor.
 */
public CounterImpl () throws RemoteException { value = 0; }

/////////////////////////////////////////////////////////////
/**
 * increment counter value by one and return the receiver
 * @return this the counter
 */
public Counter incr () throws RemoteException { value++; return this; }


/////////////////////////////////////////////////////////////
/**
 * decrement counter value by one and return the receiver
 * @return this the counter
 */
public Counter decr () throws RemoteException { if (value > 0) value--; return this; }


/////////////////////////////////////////////////////////////
/**
 * read-only access to the counter value
 * @return the current counter value (>= 0)
 */
public int value () throws RemoteException { return value; }

}
```

Once this file is compiled, use the `rmic` compiler to generate the stub class (see 3.3).

3.1.3 Server process for the remote object(s)

For the client to connect to the remote object via the registry, there must be such a remote object available. This is the task devoted to the server process, illustrated by the simple code below: file `CounterServer.java`

This process registers a `Counter` object for the registry to give to client programs. It must only instanciate an object and register it via the `rebind()` method of the `Naming` class, which handles the RMI registry process.

```
// $Id: CounterServer.java,v 1.1 2003/10/23 06:45:25 root Exp $

package pl.rmi;
import java.rmi.Naming;

/**
 * This program 1/ instanciates and 2/ publish a Counter to be used as a
 * remote object by client programs.
 * <hr>
 * WARNING: the instanciated object is multi-client. That is, when the client
 * program dies, the remote object still "lives" on the server:
 * subsequent runs of the client program will use the SAME remote object
 * <hr>
 * @author Philippe.Laroque@dept-info.u-cergy.fr
 * @version $Id: CounterServer.java,v 1.1 2003/10/23 06:45:25 root Exp $
 */
public class CounterServer {

    public CounterServer(String name) {
 try {
     // step 1 : instanciate the remote object
     Counter c = new CounterImpl();

     // step 2 : register the remote object, associated
     // with a published name (here "CounterService")
     // N.B. : server is localhost
     Naming.rebind("rmi://localhost:1099/" + name, c);
     System.err.println("Published a Counter via " + name);
 } catch (Throwable e) {
     System.out.println("Trouble: " + e);
 }
    }

    public static void main(String args[]) {
 // default service name. Can be overriden on command-line
 String name = "CounterService";
 try { name = args[0]; }
 catch (Exception e){}
 new CounterServer(name);
    }
}
```

3.2 Client side

The client needs both the interface code for the remote object and the generated stub class. With those two elements, it can use remote objects as if they were local to its own JVM: file `CounterClient.java`

```
// $Id: CounterClient.java,v 1.1 2003/10/23 06:45:25 root Exp $

package pl.rmi;
import java.rmi.*;
import java.net.*;

/**
 * Sample client application using a remote Counter object.
 * Typical steps include:
 * <ul>
 * <li>Get a reference to a remote object by using a naming service.
 *     This service is available via a well-known address/port
 * <li> use that reference as if it were a local object.
 * </ul>
 * usage: java [-Djava.rmi.server.codebase="http://serverName/classDir/"] CounterClient [se
 * (localhost by default). This way, client code must only include the interface
 *  and this file.
 * @author Philippe.Laroque@dept-info.u-cergy.fr
 * @version $Id: CounterClient.java,v 1.1 2003/10/23 06:45:25 root Exp $
 */
public class CounterClient {

    public static void main(String[] args) {

 // try to get server name. If none, take localhost
 String hostname="localhost";
 try { hostname = args[0]; }
 catch (Exception e) {}

 // try to get service name. If none, take "CounterService"
 String serviceName = "CounterService";
 try { serviceName = args[1]; }
 catch (Exception e2) {}
 // MANDATORY: install an RMISecurityManager.
 // Otherwise, no stub class can be remotely found and loaded
 System.setSecurityManager(new RMISecurityManager());


 try {
     Object o = Naming.lookup("rmi://"+ hostname + "/" + serviceName);
     System.out.println(o.getClass().getName());
     Counter c = (Counter)
  // Retrieve the object via a (published) name
  Naming.lookup("rmi://"+ hostname + "/" + serviceName);
     System.out.println( c.value());
     c.incr().incr(); // actual remote object incrementation
     System.out.println( c.value());
 }

 // typical errors when RMI does not work:
 /////////////////////////////////////////

 catch (MalformedURLException murle) {
     // incorrect host/port
     System.out.println();
     System.out.println("MalformedURLException");
     System.out.println(murle);
 }
 catch (RemoteException re) {
     // rmiregistry not running on server side
     System.out.println();
     System.out.println("RemoteException");
     System.out.println(re);
 }
 catch (NotBoundException nbe) {
     // server object not available
     System.out.println();
     System.out.println("NotBoundException");
     System.out.println(nbe);
 }
    }
}
```

3.3 Application generation

The subsequent operations work fine in an environment configured as follows:

```
export JLIBDIR="$HOME/src.d/java.d/lib.d"
export CLASSPATH=$JLIBDIR
export JAVA_HOME=/persist/local/jdk1.4.2_02
export PATH=$PATH:/usr/sbin:$HOME/bin:${JAVA_HOME}/bin:.
alias javac="javac -d ${JLIBDIR} "
alias javadoc="javadoc -version -author -noindex -notree -d doc "
alias rmic="rmic -d ${JLIBDIR} "
alias java="java -Djava.security.policy=${HOME}/src.d/java.d/policy.all "
```

3.3.1 Server-side operations

```
$ javac Counter.java
$ javac CounterImpl.java
$ rmic CounterImpl
$ javac CounterServer.java
$ rmiregistry&
$ java pl.rmi.CounterServer [serviceName]
```

3.3.2 Client-side operations

```
$ ... # getting stub code and interface
$ javac Counter.java # only if interface is local
$ javac CounterClient.java
$ java [option] pl.rmi.CounterClient # option if stub got from server
```

3.3.3 Example policy file

```
grant {
  permission java.security.AllPermission ‘“’,’”’;
};
```

3.4 Distributed usage of remote interfaces & stubs

A RMI version of the JAVA class loader, `RMIClassLoader`, has been designed to handle the problem of loading code needed by the client, even when this code is not local to the client JVM. Via the `java.rmi.server.codebase` property, the client JVM can be told where to find such code. The URL defined as the value of this property can point to one of the file://, ftp:// or http:// location types.

The client JVM only has to be given this information, as in the example below:

```
$ java \
-Djava.rmi.server.codebase=’’http://myCodeServer/myDir/’’\
pl.rmi.CounterClient [serverName [serviceName]]
```