Distributed Java Programming with RMI ~~and CORBA~~

Articles Index

Qusay H. Mahmoud
January 2002

The Java Remote Method Invocation (RMI) mechanism and the Common Object Request Broker Architecture (CORBA) are the two most important and widely used distributed object systems. Each system has its own features and shortcomings. Both are being used in the industry for various applications ranging from e-commerce to health care. Selecting which of these two distribution mechanisms to use for a project is a tough task. This article presents an overview of RMI ~~and CORBA~~, and more importantly it shows how to develop a useful application for downloading files from remote hosts. It then:

- Presents a brief overview of distributed object systems
- Provides a brief overview of RMI ~~and CORBA~~
- Gives you a flavor of the effort involved in developing applications in RMI ~~and CORBA~~
- Shows how to transfer files from remote machines using RMI ~~and CORBA~~
- Provides a brief comparison of RMI ~~and CORBA~~

The Client/Server Model

The client/server model is a form of distributed computing in which one program (the client) communicates with another program (the server) for the purpose of exchanging information. In this model, both the client and server usually speak the same language -- a protocol that both the client and server understand -- so they are able to communicate.

While the client/server model can be implemented in various ways, it is typically done using low-level sockets. Using sockets to develop client/server systems means that we must design a protocol, which is a set of commands agreed upon by the client and server through which they will be able to communicate. As an example, consider the HTTP protocol that provides a method called `GET`, which must be implemented by all web servers and used by web clients (browsers) in order to retrieve documents.

The Distributed Objects Model

A distributed object-based system is a collection of objects that isolates the requesters of services (clients) from the providers of services (servers) by a well-defined encapsulating interface. In other words, clients are isolated from the implementation of services as data representations and executable code. This is one of the main differences that distinguishes the distributed object-based model from the pure client/server model.

In the distributed object-based model, a client sends a message to an object, which in turns interprets the message to decide what service to perform. This service, or method, selection could be performed by either the object or a broker. The Java Remote Method Invocation (RMI) and the Common Object Request Broker Architecture (CORBA) are examples of this model.

RMI

RMI is a distributed object system that enables you to easily develop distributed Java applications. Developing distributed applications in RMI is simpler than developing with sockets since there is no need to design a protocol, which is an error-prone task. In RMI, the developer has the illusion of calling a local method from a local class file, when in fact the arguments are shipped to the remote target and interpreted, and the results are sent back to the callers.

The Genesis of an RMI Application

Developing a distributed application using RMI involves the following steps:

Define a remote interface
Implement the remote interface
Develop the server
Develop a client
Generate Stubs and Skeletons, start the RMI registry, server, and client
We will now examine these steps through the development of a file transfer application.

Example: File Transfer Application

This application allows a client to transfer (or download) any type of file (plain text or binary) from a remote machine. The first step is to define a remote interface that specifies the signatures of the methods to be provided by the server and invoked by clients.

Define a remote interface

The remote interface for the file download application is shown in Code Sample 1. The interface`FileInterface` provides one method `downloadFile` that takes a `String` argument (the name of the file) and returns the data of the file as an array of bytes.

Code Sample 1: FileInterface.java

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

public interface FileInterface extends Remote {
   public byte[] downloadFile(String fileName) throws
   RemoteException;
}
```

Note the following characteristics about the `FileInterface`:

It must be declared `public`, in order for clients to be able to load remote objects which implement the remote interface.

It must extend the `Remote` interface, to fulfill the requirement for making the object a remote one.

Each method in the interface must throw a `java.rmi.RemoteException`.

Implement the remote interface

The next step is to implement the interface `FileInterface`. A sample implementation is shown in Code Sample 2. Note that in addition to implementing the `FileInterface`, the `FileImpl` class is extending the `UnicastRemoteObject`. This indicates that the `FileImpl` class is used to create a single, non-replicated, remote object that uses RMI's default TCP-based transport for communication.

Code Sample 2: FileImpl.java

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class FileImpl extends UnicastRemoteObject
  implements FileInterface {

   private String name;

   public FileImpl(String s) throws RemoteException{
      super();
      name = s;
   }

   public byte[] downloadFile(String fileName){
      try {
         File file = new File(fileName);
         byte buffer[] = new byte[(int)file.length()];
         BufferedInputStream input = new
      BufferedInputStream(new FileInputStream(fileName));
         input.read(buffer,0,buffer.length);
         input.close();
         return(buffer);
      } catch(Exception e){
         System.out.println("FileImpl: "+e.getMessage());
         e.printStackTrace();
         return(null);
      }
   }
}
```

Develop the server

The third step is to develop a server. There are three things that the server needs to do:

Create an instance of the `RMISecurityManager` and install it
Create an instance of the remote object (`FileImpl` in this case)
Register the object created with the RMI registry. A sample implementation is shown in Code Sample 3.
Code Sample 3: FileServer.java

```
import java.io.*;
import java.rmi.*;

public class FileServer {
   public static void main(String argv[]) {
      if(System.getSecurityManager() == null) {
         System.setSecurityManager(new RMISecurityManager());
      }
      try {
         FileInterface fi = new FileImpl("FileServer");
         Naming.rebind("//127.0.0.1/FileServer", fi);
      } catch(Exception e) {
         System.out.println("FileServer: "+e.getMessage());
         e.printStackTrace();
      }
   }
}
```

The statement `Naming.rebind("//127.0.0.1/FileServer", fi)` assumes that the RMI registry is running on the default port number, which is 1099. However, if you run the RMI registry on a different port number it must be specified in that statement. For example, if the RMI registry is running on port 4500, then the statement becomes:

```
Naming.rebind("//127.0.0.1:4500/FileServer", fi)
```

Also, it is important to note here that we assume the rmi registry and the server will be running on the same machine. If they are not, then simply change the address in the `rebind` method.

Develop a client

The next step is to develop a client. The client remotely invokes any methods specified in the remote interface (`FileInterface`). To do so however, the client must first obtain a reference to the remote object from the RMI registry. Once a reference is obtained, the `downloadFile` method is invoked. A client implementation is shown in Code Sample 4. In this implementation, the client accepts two arguments at the command line: the first one is the name of the file to be downloaded and the second one is the address of the machine from which the file is to be downloaded, which is the machine that is running the file server.

Code Sample 4: FileClient.java

```
import java.io.*;
import java.rmi.*;

public class FileClient{
   public static void main(String argv[]) {
      if(argv.length != 2) {
        System.out.println("Usage: java FileClient fileName machineName");
        System.exit(0);
      }
      try {
         String name = "//" + argv[1] + "/FileServer";
         FileInterface fi = (FileInterface) Naming.lookup(name);
         byte[] filedata = fi.downloadFile(argv[0]);
         File file = new File(argv[0]);
         BufferedOutputStream output = new
           BufferedOutputStream(new FileOutputStream(file.getName()));
         output.write(filedata,0,filedata.length);
         output.flush();
         output.close();
      } catch(Exception e) {
         System.err.println("FileServer exception: "+ e.getMessage());
         e.printStackTrace();
      }
   }
}
```

Running the Application

In order to run the application, we need to ~~generate stubs and skeletons,~~ compile the server and the client, start the RMI registry, and finally start the server and the client.

~~To generate stubs and skeletons, use the `rmic` compiler:~~

~~**prompt>** `rmic FileImpl`~~

~~This will generate two files: `FileImpl_Stub.class` and `FileImpl_Skel.class`. The stub is a client proxy and the skeleton is a server skeleton.~~

The next step is to compile the server and the client. Use the `javac` compiler to do this. Note however, if the server and client are developed on two different machines, in order to compile the client you need a copy of the interface (`FileInterface`).

Finally, it is time to start the RMI registry and run the server and client. To start the RMI registry on the default port number, use the command `rmiregistry` or `start rmiregistry` on Windows. To start the RMI registry on a different port number, provide the port number as an argument to the RMI registry:

prompt> `rmiregistry portNumber`

Once the RMI registry is running, you can start the server `FileServer`. However, since the RMI security manager is being used in the server application, you need a security policy to go with it. Here is a sample security policy:

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

Note: this is just a sample policy. It allows anyone to do anything. For your mission critical applications, you need to specify more constraint security policies.

Now, in order to start the server you need a copy of all the classes (including stubs and skeletons) except the client class (`FileClient.class`). To start the server use the following command, assuming that the security policy is in a file named policy.txt:

prompt> `java -Djava.security.policy=policy.txt FileServer`

To start the client on a different machine, you need a copy of the remote interface (`FileInterface.class`) and stub (`FileImpl_Stub.class`). To start the client use the command:

prompt> `java FileClient fileName machineName`

where `fileName` is the file to be downloaded and `machineName` is the machine where the file is located (the same machine runs the file server). If everything goes ok then the client exists and the file downloaded is on the local machine.

To run the client we mentioned that you need a copy of the interface and stub. A more appropriate way to do this is to use RMI dynamic class loading. The idea is you do not need copies of the interface ~~and the stub~~. Instead, they can be located in a shared directory for the server and the client, and whenever a stub or a skeleton is needed, it is downloaded automatically by the RMI class loader. To do this you run the client, for example, using the following command: `java -Djava.rmi.server.codebase=http://hostname/locationOfClasses FileClient fileName machineName`. For more information on this, please see Dynamic Code Loading using RMI.