Java RMI

Introduction

This is a brief introduction to Java Remote Method Invocation (RMI). Java RMI is a mechanism that allows one to invoke a method on an object that exists in another address space. The other address space could be on the same machine or a different one. The RMI mechanism is basically an object-oriented RPC mechanism. CORBA is another object-oriented RPC mechanism. CORBA differs from Java RMI in a number of ways:

1. CORBA is a language-independent standard.
2. CORBA includes many other mechanisms in its standard (such as a standard for TP monitors) none of which are part of Java RMI.
3. There is also no notion of an "object request broker" in Java RMI.

Java RMI has recently been evolving toward becoming more compatible with CORBA. In particular, there is now a form of RMI called RMI/IIOP ("RMI over IIOP") that uses the Internet Inter-ORB Protocol (IIOP) of CORBA as the underlying protocol for RMI communication.

This tutorial attempts to show the essence of RMI, without discussing any extraneous features. Sun has provided a Guide to RMI, but it includes a lot of material that is not relevant to RMI itself. For example, it discusses how to incorporate RMI into an Applet, how to use packages and how to place compiled classes in a different directory than the source code. All of these are interesting in themselves, but they have nothing at all to do with RMI. As a result, Sun's guide is unnecessarily confusing. Moreover, Sun's guide and examples omit a number of details that are important for RMI.

There are three processes that participate in supporting remote method invocation.

1. The *Client* is the process that is invoking a method on a remote object.
2. The *Server* is the process that owns the remote object. The remote object is an ordinary object in the address space of the server process.
3. The *Object Registry* is a name server that relates objects with names. Objects are *registered* with the Object Registry. Once an object has been registered, one can use the Object Registry to obtain access to a remote object using the name of the object.

In this tutorial, we will give an example of a Client and a Server that solve the classical "Hello, world!" problem. You should try extracting the code that is presented and running it on your own computer.

There are two kinds of classes that can be used in Java RMI.

1. A *Remote* class is one whose instances can be used remotely. An object of such a class can be referenced in two different ways:
    1. Within the address space where the object was constructed, the object is an ordinary object which can be used like any other object.
    2. Within other address spaces, the object can be referenced using an *object handle*. While there are limitations on how one can use an object handle compared to an object, for the most part one can use object handles in the same way as an ordinary object.

    For simplicity, an instance of a Remote class will be called a *remote object*.
2. A *Serializable* class is one whose instances can be copied from one address space to another. An instance of a Serializable class will be called a *serializable object*. In other words, a serializable object is one that can be marshaled. Note that this concept has no connection to the concept of serializability in database management systems.

If a serializable object is passed as a parameter (or return value) of a remote method invocation, then the value of the object will be copied from one address space to the other. By contrast if a remote object is passed as a parameter (or return value), then the object handle will be copied from one address space to the other.

One might naturally wonder what would happen if a class were both Remote and Serializable. While this might be possible in theory, it is a poor design to mix these two notions as it makes the design difficult to understand.

Serializable Classes

We now consider how to design Remote and Serializable classes. The easier of the two is a Serializable class. A class is Serializable if it implements the java.io.Serializable interface. Subclasses of a Serializable class are also Serializable. Many of the standard classes are Serializable, so a subclass of one of these is automatically also Serializable. Normally, any data within a Serializable class should also be Serializable. Although there are ways to include non-serializable objects within a serializable objects, it is awkward to do so. See the documentation of java.io.Serializable for more information about this.

Using a serializable object in a remote method invocation is straightforward. One simply passes the object using a parameter or as the return value. The type of the parameter or return value is the Serializable class. Note that both the Client and Server programs must have access to the definition of any Serializable class that is being used. If the Client and Server programs are on different machines, then class definitions of Serializable classes may have to be downloaded from one machine to the other. Such a download could violate system security. This problem is discussed in the Security section.

The only Serializable class that will be used in the "Hello, world!" example is the String class, so no problems with security arise.

Remote Classes and Interfaces

Next consider how to define a Remote class. This is more difficult than defining a Serializable class. A Remote class has two parts: the interface and the class itself. The Remote interface must have the following properties:

1. The interface must be public.
2. The interface must extend the interface java.rmi.Remote.
3. Every method in the interface must declare that it throws java.rmi.RemoteException. Other exceptions may also be thrown.

The Remote class itself has the following properties:

1. It must implement a Remote interface.
2. It should extend the java.rmi.server.UnicastRemoteObject class. Objects of such a class exist in the address space of the server and can be invoked remotely. While there are other ways to define a Remote class, this is the simplest way to ensure that objects of a class can be used as remote objects. See the documentation of the java.rmi.server package for more information.
3. It can have methods that are not in its Remote interface. These can only be invoked locally.

Unlike the case of a Serializable class, it is not necessary for both the Client and the Server to have access to the definition of the Remote class. The Server requires the definition of both the Remote class and the Remote interface, but the Client only uses the Remote interface. Roughly speaking, the

Remote interface represents the type of an object handle, while the Remote class represents the type of an object. If a remote object is being used remotely, its type must be declared to be the type of the Remote interface, not the type of the Remote class.

In the example program, we need a Remote class and its corresponding Remote interface. We call these Hello and HelloInterface, respectively. Here is the file HelloInterface.java:

```
import java.rmi.*;
/**
 * Remote Interface for the "Hello, world!" example.
 */
public interface HelloInterface extends Remote {
  /**
   * Remotely invocable method.
   * @return the message of the remote object, such as "Hello, world!".
   * @exception RemoteException if the remote invocation fails.
   */
  public String say() throws RemoteException;
}
```

Here is the file Hello.java:

```
import java.rmi.*;
import java.rmi.server.*;
/**
 * Remote Class for the "Hello, world!" example.
 */
public class Hello extends UnicastRemoteObject implements HelloInterface {
  private String message;
  /**
   * Construct a remote object
   * @param msg the message of the remote object, such as "Hello, world!".
   * @exception RemoteException if the object handle cannot be constructed.
   */
  public Hello (String msg) throws RemoteException {
    message = msg;
  }
  /**
   * Implementation of the remotely invocable method.
   * @return the message of the remote object, such as "Hello, world!".
   * @exception RemoteException if the remote invocation fails.
   */
  public String say() throws RemoteException {
    return message;
  }
}
```

All of the Remote interfaces and classes should be compiled using javac. Once this has been completed, the stubs and skeletons for the Remote interfaces should be compiled by using the rmic stub compiler. The stub and skeleton of the example Remote interface are compiled with the command:

```
rmic Hello
```

The only problem one might encounter with this command is that rmic might not be able to find the files Hello.class and HelloInterface.class even though they are in the same directory where rmic is being executed. If this happens to you, then try setting the CLASSPATH environment variable to the current directory, as in the following command:

```
setenv CLASSPATH .
```

If your CLASSPATH variable already has some directories in it, then you might want to add the current directory to the others.

Programming a Client

Having described how to define Remote and Serializable classes, we now discuss how to program the Client and Server. The Client itself is just a Java program. It need not be part of a Remote or Serializable class, although it will use Remote and Serializable classes.

A remote method invocation can return a remote object as its return value, but one must have a remote object in order to perform a remote method invocation. So to obtain a remote object one must already have one. Accordingly, there must be a separate mechanism for obtaining the first remote object. The Object Registry fulfills this requirement. It allows one to obtain a remote object using only the name of the remote object.

The name of a remote object includes the following information:

1. The Internet name (or address) of the machine that is running the Object Registry with which the remote object is being registered. If the Object Registry is running on the same machine as the one that is making the request, then the name of the machine can be omitted.
2. The port to which the Object Registry is listening. If the Object Registry is listening to the default port, 1099, then this does not have to be included in the name.
3. The local name of the remote object within the Object Registry.

Here is the example Client program:

```
/**
 * Client program for the "Hello, world!" example.
 * @param argv The command line arguments which are ignored.
 */
public static void main (String[] argv) {
  try {
    HelloInterface hello =
      (HelloInterface) Naming.lookup ("//ortles.ccs.neu.edu/Hello");
    System.out.println (hello.say());
  } catch (Exception e) {
    System.out.println ("HelloClient exception: " + e);
  }
}
```

The Naming.lookup method obtains an object handle from the Object Registry running on ortles.ccs.neu.edu and listening to the default port. Note that the result of Naming.lookup must be cast to the type of the Remote interface.

The remote method invocation in the example Client is hello.say(). It returns a String which is then printed. A remote method invocation can return a String object because String is a Serializable class.

The code for the Client can be placed in any convenient class. In the example Client, it was placed in a class HelloClient that contains only the program above.

Programming a Server

The Server itself is just a Java program. It need not be a Remote or Serializable class, although it will use them. The Server does have some responsibilities:

1. If class definitions for Serializable classes need to be downloaded from another machine, then the security policy of your program must be modified. Java provides a security manager class called RMISecurityManager for this purpose. The RMISecurityManager defines a security policy that allows the downloading of Serializable classes from another machine. The "Hello, World!" example does not need such downloads, since the only Serializable class it uses is String. As a result it isn't necessary to modify the security policy for the example program. If your program defines Serializable classes that need to be downloaded to another machine, then insert the statement System.setSecurityManager (new RMISecurityManager()); as the first statement in the main program below. If this does not work for your program, then you should consult the Security section below.
2. At least one remote object must be registered with the Object Registry. The statement for this is: Naming.rebind (objectName, object); where object is the remote object being registered, and objectName is the String that names the remote object.

Here is the example Server:

```
/**
 * Server program for the "Hello, world!" example.
 * @param argv The command line arguments which are ignored.
 */
public static void main (String[] argv) {
  try {
    Naming.rebind ("Hello", new Hello ("Hello, world!"));
    System.out.println ("Hello Server is ready.");
  } catch (Exception e) {
    System.out.println ("Hello Server failed: " + e);
  }
}
```

The rmiregistry Object Registry only accepts requests to bind and unbind objects running on the same machine, so it is never necessary to specify the name of the machine when one is registering an object.

The code for the Server can be placed in any convenient class. In the example Server, it was placed in a class HelloServer that contains only the program above.

Starting the Server

Before starting the Server, one should first start the Object Registry, and leave it running in the background. One performs this by using the command:

```
rmiregistry &
```

It takes a second or so for the Object Registry to start running and to start listening on its socket. If one is using a script, then one should program a pause after starting the Object Registry. If one is typing at the command line, it is unlikely that one could type fast enough to get ahead of the Object Registry.

The Server should then be started; and, like the Object Registry, left running in the background. The example Server is started using the command:

```
java HelloServer &
```

The Server will take a few seconds to start running, and to construct and register remote objects. So one should wait a few seconds before running any Clients. Printing a suitable message, as in the example Server, is helpful for determining when the Server is ready.

Running a Client

Th Client is run like any other java program. The example Client is executed using:

```
java HelloClient
```

Security

One of the most common problems one encounters with RMI is a failure due to security constraints. This section gives a very brief introduction to the Java security model as it relates to RMI. For a more complete treatment, one should read the documentation for the Java SecurityManager and Policy classes and their related classes. Note that this section assumes that one is using Java 1.2 or later. Some of the statements are not true for earlier versions.

A Java program may specify a security manager that determines its security policy. A program will not have any security manager unless one is specified. One sets the security policy by constructing a SecurityManager object and calling the setSecurityManager method of the System class. Certain operations require that there be a security manager. For example, RMI will download a Serializable class from another machine only if there is a security manager and the security manager permits the downloading of the class from that machine. The RMISecurityManager class defines an example of a security manager that normally permits such downloads.

However, many Java installations have instituted security policies that are more restrictive than the default. There are good reasons for instituting such policies, and one should not override them carelessly. The rest of this section discusses some ways that can be used for overriding security policies that prevent RMI from functioning properly.

The SecurityManager class has a large number of methods whose name begins with check. For example, checkConnect (String host, int port). If a check method returns, then the permission was granted. For example, if a call to checkConnect returns normally, then the current security policy allows the program to establish a socket connection to the server socket at the specified host and port. If the current security policy does not allow one to connect to this host and port, then the call throws an exception. This usually causes your program to terminate with a message such as:

```
java.security.AccessControlException: access denied
(java.net.SocketPermission 127.0.0.1:1099 connect,resolve)
```

The message above would occur when an RMI server or client was not allowed to connect to the RMI registry running on the same machine as the server or client.

As discussed above, one sets the security policy by passing an object of type SecurityManager to the setSecurityManager method of the System class. There are several ways to modify the security policy of a program. The simplest technique is to define a subclass of SecurityManager and to call System.setSecurityManager on an object of this subclass. In the definition of this subclass, you should override those check methods for which you want a different policy. For example, if you find that your "Hello, World!" program refuses to connect to the registry, then you should override the checkConnect methods. There are two checkConnect methods. The first was discussed above, and the second checkConnect method has a third parameter that specifies the security context of the request.

The following code illustrates how to do this:

```
System.setSecurityManager (new RMISecurityManager() {
  public void checkConnect (String host, int port) {}
  public void checkConnect (String host, int port, Object context) {}
});
```

The code above uses an *anonymous inner class*. Such a class is convenient when the class will only be used to construct an object in one place, as in this example. Of course, one could also define the subclass of RMISecurityManager in the usual way.

Defining and installing a security manager was the original technique for specifying a security policy in Java. Unfortunately, it is very difficult to design such a class so that it does not leave any security holes. For this reason, a new technique was introduced in Java 1.2, which is backward compatible with the old technique. In the default security manager, all check methods (except checkPermission) are implemented by calling the checkPermission method. The type of permission being checked is specified by the parameter of type Permission passed to the checkPermission method. For example, the checkConnect method calls checkPermission with a SocketPermission object. The default implementation of checkPermission is to call the checkPermission method of the AccessController class. This method checks whether the specified permission is implied by a list of granted permissions. The Permissions class is used for maintaining lists of granted permissions and for checking whether a particular permission has been granted.

This is the mechanism whereby the security manager checks permissions, but it does not explain how one specifies or changes the security policy. For this purpose there is yet another class, named Policy. Like SecurityManager, each program has a current security policy that can be obtained by calling Policy.getPolicy(), and one can set the current security policy using Policy.setPolicy, if one has permission to do so. The security policy is typically specified by a policy configuration file (or "policy file" for short) which is read when the program starts and any time that a request is made to refresh the security policy. The policy file defines the permissions contained in a Policy object. It is not inaccurate to think of the policy file a kind of serialization of a Policy object (except that a policy file is intended to be readable by humans as well as by machines). As an example, the following will grant all permissions of any kind to code residing in the RMI directory on the C: drive:

```
grant codeBase "file:C:/RMI/-" {
  permission java.security.AllPermission;
};
```

The default security manager uses a policy that is defined in a collection of policy files. For the locations of these files see the documentation of the policytool program. If one wishes to grant additional permissions, then one can specify them in a policy file and then request that they be loaded using options such as the following:

```
java -Djava.security.manager -Djava.security.policy=policy-file MyClass
```

Both of the "-D" options specify system properties. The first system property has the same effect as executing the following statement as the first statement in your program:

```
System.setSecurityManager (new SecurityManager());
```

The second system property above causes the specified policy-file (which is specified with a URL) to be added to the other policy files when defining the entire security policy. The policytool can be used to construct the policy file, but one can also use any text editor.

As if this wasn't already complicated enough, there is yet another way to deal with the problem of downloading Serializable classes. The command-line option -Djava.rmi.server.codebase=code-base specifies a location from which Serializable classes may be downloaded. Of course, your security manager must recognize this system property, and not all of them will do so. Furthermore, as mentioned earlier, this is only necessary if you actually need to download Serializable classes.