How to create an RMI system

In this article, I lead you through the process of creating a very simple RMI system. (This example was inspired by the RMI demo in Orfali and Harkey's book on CORBA; I thought it was still too complicated, so I've taken the simplification a bit further). I've tried to make this example as bare-bones as possible in order to keep the focus on the steps needed to make an RMI program work. I've also tried to avoid skipping any steps that might throw a first-time user, and have tried to ensure that all steps are performed in the proper order (there are a couple of steps in which order matters). That being said, let's do some RMI!

Steps to creation of an RMI system:

The short version

```
1) Create an interface. (in this case, the interface is myRMIInterface.java).
2) Create a class that implements the interface. (in this case, myRMIImpl.java).
3) Create a server that creates an instance of this class
4) Create a client that connects to the server object using Naming.lookup()
5) Compile these classes.
6) Run the RMI interface compiler on the .class file of the implementation
   class (in this case, you'd say "rmic myRMIImpl").
7) Start the RMI registry (on Windows NT/95, say "start rmiregistry").
8) Start the server class ("start java myRMIServer").
9) Run the client program ("java myRMIClient").
```

The long version

1. Create an interface
   An interface is similar to a pure virtual class in C++; it defines the methods (and the arguments to the methods) that will be available in a class that implements it; however, it doesn't actually implement any of the methods.
   The interface I created for this example is myRMIInterface.java. It contains only one method; the method takes no arguments and returns an object of type java.util.Date. Note 2 things about this interface: 1) it extends the java.rmi.Remote interface (all interfaces used in RMI must do this). 2) the method throws a java.rmi.RemoteException (every method in a remote object's interface must specify this exception in its "throws" clause; this exception is a superclass of all RMI exceptions that can be thrown. See the JDK 1.1 final docs (in the java.rmi section) for a complete list of exceptions that can be thrown.
2. Create a class that implements the interface
   In this example, the implementation is found in myRMIImpl.java. This class must extend java.rmi.UnicastRemoteObject and must implement the interface you created in step 1. In my example, the only method that needs to be implemented is getDate(), which returns the current date and time on the system. Note 2 things about the constructor: 1) the call to super(). 2) the call to Naming.rebind(name, this). This call informs the RMI registry that this object is available with the name given in "String name". Other than that, this object simply implements all the methods declared in the interface.
3. Create a server that creates an instance of the "impl" class
   In this example, the server class is myRMIServer.java. In this case, the server is pretty simple. It does 2 things: 1) Installs a new RMISecurityManager (Note that RMI uses a different security manager from the security manager used for applets). 2) Creates an instance of the myRMIImpl class, and gives it the name "myRMIImplInstance". The myRMIImpl object takes care of registering the object with the RMI registry. After this code is run, the object will be available to remote clients as "rmi://
4. Create a client that connects to the server object using Naming.lookup().
   In this example, the client class is myRMIClient.java. The client first installs a new RMI Security Manager (see previous step), then uses the static method Naming.lookup() to get a reference to the remote object. Note that the client is using the *interface* to hold the reference and make method calls. You should make sure you've created your interface before you try to build the client, or you'll get "class not found" errors when you try to compile your client.
5. Compile these classes
   Just do "javac *.java". Piece of cake. :-)
6. Run the RMI interface compiler on your implementation class.
   This step generates some additional Java classes. They're stubs and skeletons used by RMI; you don't have to worro about what's in them. *Note:* You only need to run rmic on the class that implements your RMI interface. In this

case, you'd do "rmic myRMIImpl". *Also note* that the rmic compiler runs on a .class file, not a .java file.

7. Start the RMI registry
   OK. You're done with development at this point; you've built all the code you need to run this example. Now you're setting up the environment so that you can run it. rmiregistry is a program that comes with the JDK 1.1 final; you can find it in the "bin" directory of your JDK installation. Under Windows 95 or NT, you can simply say "start rmiregistry" on the command line, which will cause the RMI registry to be started in its own DOS window. The RMI registry must be started before you can start your server.
8. Start your RMI server program
   Under Windows 95 or NT, say "start java myRMIServer" on the command line. This starts the server running. As we discussed earlier, the server then creates an instance of myRMIImpl and makes it known to the RMI server as "myRMIImplInstance".
9. Run your client program
   Say "java myRMIClient". The program will ask the RMI registry for a reference to "myRMIImplInstance". After it has this reference, the client can invoke any methods declared in the myRMIInterface interface as if the object were a local object.

Problems that you may run into

problem: You get a "class not found" error when running rmic
solution: Add the current directory to your classpath.

problem: You get the following error when running the client

```
Exception occured: java.rmi.UnmarshalException: Return value class not found; nested exception is:
        java.lang.ClassNotFoundException: myRMIImpl_Stub
```

solution: The file myRMIImpl_Stub.class must be deployed with your client application (that is, you must place it somewhere in the classpath of the client machine; If you were using an applet as a client, you would place it in the directory specified in the CODEBASE parameter to achieve the same effect).

problem: You get the following error when running the client

```
C:\test3>java myRMIClient 127.0.0.1
Exception occured: java.security.AccessControlException: access denied (java.net
.SocketPermission 127.0.0.1:1099 connect,resolve)
```

or the following when running the server

```
C:\test3>java myRMIServer
Exception occurred: java.security.AccessControlException: access denied (java.ne
t.SocketPermission 127.0.0.1:1099 connect,resolve)
```

solution: I experienced this problem under Java2 with the default security policy in place. You'll need to modify your security policy to allow these activities to take place. A full writeup on this is available at The Sun RMI tutorial page.
In summary, you'll need to do the following:

1. Create a new security policy file. See your JDK docs or the links referenced from the Sun RMI tutorial for more information on this.
2. When you run the client or the server, pass the location of your new security policy file in as an argument. This allows you to run under a new policy without having to modify your system policy files. Here is a .policy file that grants all permissions to everybody. *DO NOT* install this policy file in a production system. However, you can use it in trivial testing. You can then run the server with the command line
   java -Djava.security.policy=c:\test3\wideopen.policy myRMIServer
   or the client with
   java -Djava.security.policy=c:\test3\wideopen.policy myRMIClient 127.0.0.1

Of course, you'd replace c:\test3\wideopen.policy with the full path to your own properties file.

Here is another policy file that includes only the permissions necessary to run this app.

And that's the whole story of how to create and run an RMI program. It's really not that bad, is it? :-)
Here's the source code:
myRMIInterface.java
myRMIImpl.java
myRMIServer.java
myRMIClient.java