Java RMI

José M. Vidal

Tue Feb 17 12:15:47 EST 2004

This talk summarizes material from

- Getting Started Using RMI[1]
- Dynamic Code Downloading using RMI[2]
- Creating an Activatable Object[3], SUN Guide, 2002.
- Making a UnicastRemoteObject Activatable[4], SUN Guide, 2002.
- Activating an object that does not extend java.rmi.activation.Activatable[5], SUN Guide, 2002.
- Using a Custom RMI Socket Factory[6], SUN Guide, 2002.

1 Introduction

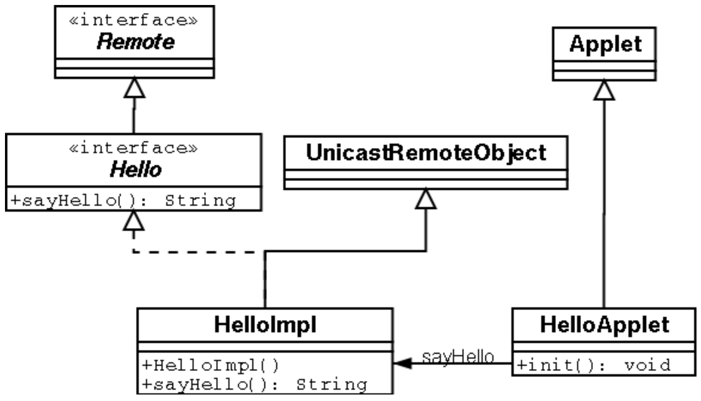
- Java's Remote Method Invocation is an extension of Java which allows you to distribute your objects.

Client		Server
Stub		Skeleton
Remote Reference		Remote Reference
	Transport	

- RMI provide dynamic method invocation. A client can invoke a server without knowing which methods are contained in server.
- The Remote Reference is responsible for determining the nature of the object. Is it local or remote? If remote, will it start automatically or does it need to be initialized?
- Java remote objects are platform independent, but you must use Java at both ends (if you ignore rmi-to-corba-dcom bridges).
- Remote objects are garbage collected using reference counting.
- The RMI **registry** provides access to remote objects.

Note:
The purpose of this presenation is to provide the step-by-step instructions and low-level implementation details needed to understand Java's RMI implementation, so you can get started on your programming. As always, these slides only serve to highlight the most important points in the readings. You should read the cited tutorials while sitting in front of a computers and taking the steps they indicate. It is the best way to get comfortable with this (and any) new technology.

1.1 Hello World Example



- First define the `Hello` interface.
- Then implement your interface, in this case by `HelloImpl`.

1.2 Hello.java

package examples.hello;

import java.rmi.**Remote**;
import java.rmi.**RemoteException**;

public **interface Hello extends Remote** {
  String sayHello() **throws RemoteException**;
}

- All methods must throw `RemoteException` since remote calls can fail due to network problems.
- It must be declared `public`.
- The data type of the remote object must be of the interface type (i.e., `Hello`), **not** the implementation type (i.e., `HelloImpl`).

1.3 Exporting and Binding Remote Objects

- Remote objects need to be **exported**. It makes the object listen on anonymous port for method requests from stub.
- Exporting happens automatically if you extend `RemoteObject` or `Activatable`. Otherwise you must call `UnicastRemoteObject.exportObject` or `Activatable.exportObject`.
- You must define a constructor that throws `RemoteException` (because the base class does).
- The object server must have a `SecurityManager`.
- The object must get bound (`rebind()`).
- These last two must be done by a when the server starts.

1.4 HelloImpl.java

```
package examples.hello;

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

/** This class serves both as the remote object implementation and the
    object server main. These two functions could be separated. */
public class HelloImpl extends UnicastRemoteObject
  implements Hello {

  public HelloImpl() throws RemoteException {
    super();
  }

  public String sayHello() {
    return "Hello World!";
  }

  public static void main(String args[]) {

    // Create and install a security manager
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    try {
      HelloImpl obj = new HelloImpl();
      // Bind this object instance to the name "HelloServer"
      Naming.rebind("HelloServer", obj);
      System.out.println("HelloServer bound in registry");
    } catch (Exception e) {
      System.out.println("HelloImpl err: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

- The implementation must implement all methods in the interface otherwise it will be abstract.
- It extends `UnicastRemoteObject` so it uses RMI's default socket-based transport and runs all the time.
  - We could have made it run only when activated (more later).
- Constructor is the same as for a non-remote class.
- Arguments and return values can be anything.
- But, if argument or return value is an object then this object must implement `java.io.Serializable`
- Remote objects are passed just like local objects, by reference, but it's actually a reference to a stub).

- You can implement other methods but these can only be called from within the JVM running the service.

1.5 HelloImpl.java Security and Registration

- You must (should) create and install a `SecurityManager`.
- If the client is an application (and not an applet) it would also require one.
- The `SecurityManager` guarantees that loaded classes do not do bad things.
- When the server starts running (`main`) it creates one (or more) instances of the remote object. Since this construction automatically exports the object, the object is immediately ready to accept incoming calls.
- We then register the remote object with `Naming.rebind("//host/objectname", obj)` which allows the caller to find the object by name. If host is omitted then localhost is assumed.
- Once the object is exported RMI substitutes the stub for the actual object (`obj`). The registry return this stub.
- For security reasons, an application can bind and unbind only to the registry running on local machine. Lookups can be done from any host.

Note:
The requirement that the registry run on the same machine as the server is a heavy-handed way of eliminating a man-in-the-middle attack. In this attack, a hostile client could find a registry, ask it to `list()` all its registered services, then proceed to `rebind()` all those services so they point to him. By forcing the registry to run on the same machine as the server this possibility is eliminated assuming, of course, that we trust all the users of the machine!

A better solution to this problem would be for the registry to only service those clients that have authenticated with it using, for example, a public key encryption algorithm. In fact, we could use HTTP's TLS for this purpose. (Has anyone done this already?).

1.6 HelloApplet.java

```
package examples.hello;

import java.applet.Applet;
import java.awt.Graphics;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloApplet extends Applet {

  String message = "blank";

  // "obj" is the identifier that we'll use to refer
  // to the remote object that implements the "Hello"
  // interface
  Hello obj = null;

  public void init() {
    try {
      obj = (Hello)Naming.lookup("//" +
                          getCodeBase().getHost() + "/HelloServer");
      message = obj.sayHello();
    } catch (Exception e) {
      System.out.println("HelloApplet exception: " +
                         e.getMessage());
      e.printStackTrace();
    }
  }

  public void paint(Graphics g) {
    g.drawString(message, 25, 50);
  }
}
```

- The `lookup` function does two things:
  1. Constructs a registry stub instance in order to access the registry.
  2. Uses this stub to call `lookup` method on registry.
- When the call is made to `sayHello()` then:
  1. A remote call is sent to server, by stub.
  2. RMI serializes "Hello world" string and returns it.
  3. RMI de-serializes it at the client and places it in message.

1.7 hello.html

```
<html>
<head>
<title>Hello World</title>
</head>
<body>
<center> <h1>Hello World</h1> </center>

The message from the HelloServer is:
<p>
<applet codebase="."
        code="examples.hello.HelloApplet"
        width=500 height=120>
</applet>
</body>
</html>
```

- The codebase is the directory where the applet will download the .class files.

1.8 Running

1. Compile sources:

   ```
   javac -d . *.java
   ```

2. Use `rmic` (not in JDK 1.5) to generate stub and skeleton .class files from the implementation and interface:

```
rmic -d . examples.hello.HelloImpl
```

3. Move hello.html to a webserver (along with the .class files), or use the appletviewer.
4. Start the rmi registry on the server:

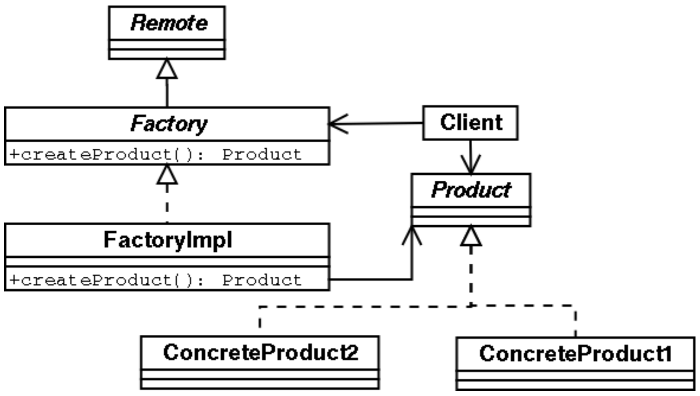
```
rmiregistry &
```

5. Start the server. Specify the codebase and policy to the stub class can be dynamically downloaded to registry.

```
java  -Djava.rmi.server.codebase=http://myhost/~myusrname/myclasses/
    -Djava.security.policy=$HOME/mysrc/policy  examples.hello.HelloImpl
```

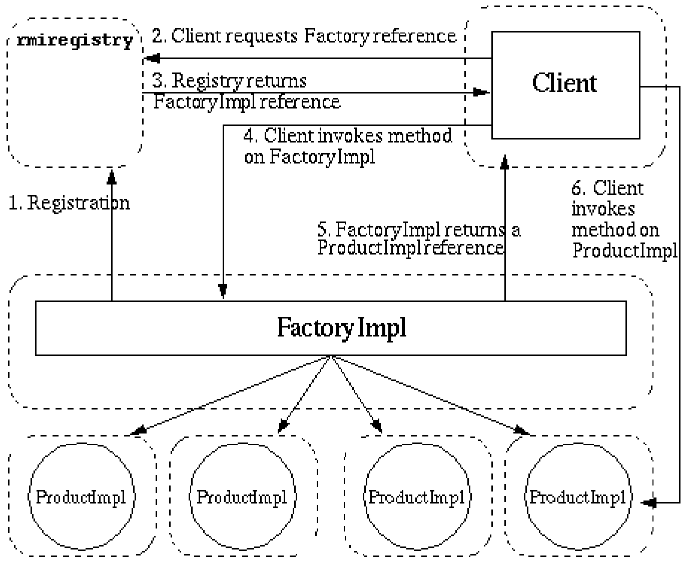
6. Run the applet (from a browser or with the appletviewer).

2 Using Factories in RMI



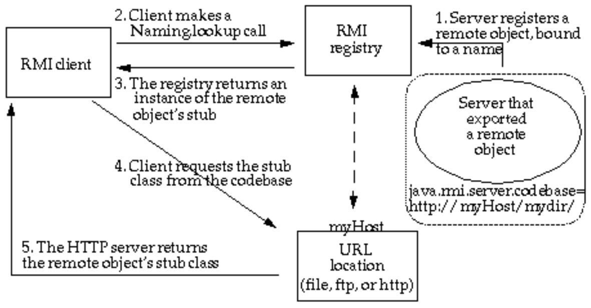
- In the Factory pattern[7] [GoF[8]] one class, which is usually a Singleton, is used to create instances of another class (or classes).
- It is often used to create different subclasses of a particular class, or different implementations of a particular interface (as the URL class does in Java).

2.1 RMI Factory


rmiregistry
2. Client requests Factory reference
3. Registry returns FactoryImpl reference
Client
4. Client invokes method on FactoryImpl
1. Registration
5. FactoryImpl returns a ProductImpl reference
6. Client invokes method on ProductImpl
Factory Impl
ProductImpl
ProductImpl
ProductImpl
ProductImpl

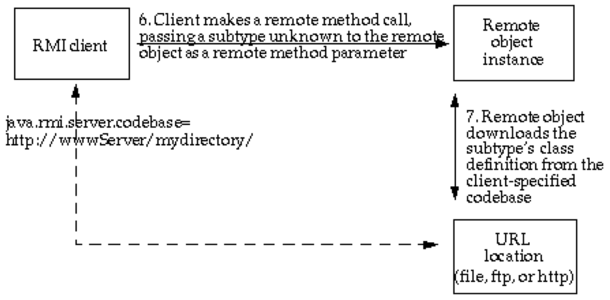
- We use a factory to obtain multiple instances of a remote object since the registry will only hold one instance of an object.

3 Dynamic Code Downloading


2. Client makes a Naming.lookup call
RMI registry
1. Server registers a remote object, bound to a name
RMI client
3. The registry returns an instance of the remote object's stub
Server that exported a remote object
java.rmi.server.codebase= http://myHost/mydir/
4. Client requests the stub class from the codebase
myHost
5. The HTTP server returns the remote object's stub class
URL location (file, ftp, or http)

- In the Hello example the object server and web server were running in the same machine. They need not.
- The codebase property value represents one or more URL locations from which code can be downloaded.
- The stub instance is annotated with the codebase.

3.1 Dynamic Code Downloading of Arguments


RMI client
6. Client makes a remote method call, passing a subtype unknown to the remote object as a remote method parameter
Remote object instance
java.rmi.server.codebase=
http://wwwServer/mydirectory/
7. Remote object downloads the subtype's class definition from the client-specified codebase
URL location (file, ftp, or http)

- When a remote object gets passed an argument:

1. If its a primitive type, there is nothing to do.
2. If its an object whose class in on the remote object's CLASSPATH, then it loads it from there.
3. If its not on the CLASSPATH then the object must be either an implementation of the interface that is declared as the method parameter or a subclass of the class that is declared as method parameter. (Why? Otherwise it would not have compiled.) It gets downloaded from the codebase.

3.2 Setting the Codebase

- If you want the client to download class code for your client you must first set the codebase:

  ```
  java -Djava.rmi.server.codebase="http://mymachine.com/" ...
  ```

  and make sure that the code can be downloaded from that URL.

- SUN provides a 2-page webserver known as ClassServer.java[9] for you to use.

4 The Registry

- The rmiregistry also supports:
- `bind()`.
- `rebind()`: to rebind to an existing name. Can be done by anyone!
- `list()`: enables a client to find the URLs of all the servers bound to the registry.
  - Along with reflection, this can be used to discover new interfaces.

5 Activatable Remote Objects

- Before Java 2, the `UnicastRemoteObject` could be accessed only from a server program that created instances of the object and ran all the time.
- With Java 2 we got the `Activatable` and the `rmid` daemon.
- An activatable class needs only to be registered with the `rmid`.
- This is an advantage for systems that have many remote object classes but only a few of them are active at any one time. They save memory (and so gain performance).

5.1 ActivatableImplementation.java

```
package examples.activation;

import java.rmi.*;
import java.rmi.activation.*;

public class ActivatableImplementation extends Activatable
  implements examples.activation.MyRemoteInterface {

  // The constructor for activation and export; this constructor is
  // called by the method ActivationInstantiator.newInstance during
  // activation, to construct the object.
  //
  public ActivatableImplementation(ActivationID id, MarshalledObject data)
    throws RemoteException {

    // Register the object with the activation system
    // then export it on an anonymous port
    //
    super(id, 0);
  }

  // Implement the method declared in MyRemoteInterface
  //
  public Object callMeRemotely() throws RemoteException {

    return "Success";
  }
}
```

- It needs a two argument constructor.
- It implements the remote interface.

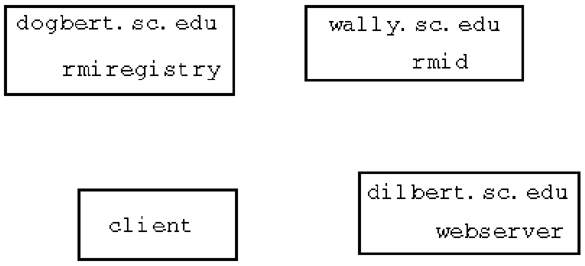
5.2 Setup.java

```
package examples.activation;

import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;

public class Setup {

  // This class registers information about the ActivatableImplementation
  // class with rmid and the rmiregistry
  //
  public static void main(String[] args) throws Exception {

    System.setSecurityManager(new RMISecurityManager());

    // Because of the 1.2 security model, a security policy should
    // be specified for the ActivationGroup VM. The first argument
    // to the Properties put method, inherited from Hashtable, is
    // the key and the second is the value
    //
    Properties props = new Properties();
    props.put("java.security.policy",
            "/home/rmi_tutorial/activation/policy");

    ActivationGroupDesc.CommandEnvironment ace = null;
    ActivationGroupDesc exampleGroup = new ActivationGroupDesc(props, ace);

    // Once the ActivationGroupDesc has been created, register it
    // with the activation system to obtain its ID
    //
    ActivationGroupID agi =
      ActivationGroup.getSystem().registerGroup(exampleGroup);

    // The "location" String specifies a URL from where the class
    // definition will come when this object is requested (activated).
    // Don't forget the trailing slash at the end of the URL
    // or your classes won't be found.
    //
    String location = "file:/home/rmi_tutorial/activation/";

    // Create the rest of the parameters that will be passed to
    // the ActivationDesc constructor
    //
    MarshalledObject data = null;

    // The location argument to the ActivationDesc constructor will be used
    // to uniquely identify this class; it's location is relative to the
    // URL-formatted String, location.
    //
    ActivationDesc desc = new ActivationDesc
      (agi, "examples.activation.ActivatableImplementation",
       location, data);

    // Register with rmid
    //
    MyRemoteInterface mri = (MyRemoteInterface)Activatable.register(desc);
    System.out.println("Got the stub for the ActivatableImplementation");

    // Bind the stub to a name in the registry running on 1099
    //
    Naming.rebind("ActivatableImplementation", mri);
    System.out.println("Exported ActivatableImplementation");

    System.exit(0);
  }

}
```

- Creates all the information necessary for the activatable class without creating an instance.
- Passes the information about the class to the `rmid`.

- Passes the identifier (name) to the `rmiregistry`.
- The job of the `ActivationDesc` is to provide all the information that `rmid` will require to create a new instance of the implementation class.

5.3 Compile and Run Activatable



1. Compile

   ```
   javac -d . *.java
   ```

2. Run rmic

   ```
   rmic -d . examples.activation.ActivatableImplementation
   ```

3. Start the registry. Make sure the CLASSPATH does not contain any classes that you want to download to client.

   ```
   rmiregistry &
   ```

4. Start the activation daemon.

   ```
   rmid -J -Djava.security.policy=rmid.policy &
   ```

5. Run the setup program. The codebase is a URL indicating where the stub code lives. Usually you will want this to be an http.

   ```
   java  -Djava.security.policy=/home/rmi_tutorial/activation/policy
    -Djava.rmi.server.codebase=http://dilbert.usc.edu/rmi_tutorial/activation/
    examples.activation.Setup
   ```

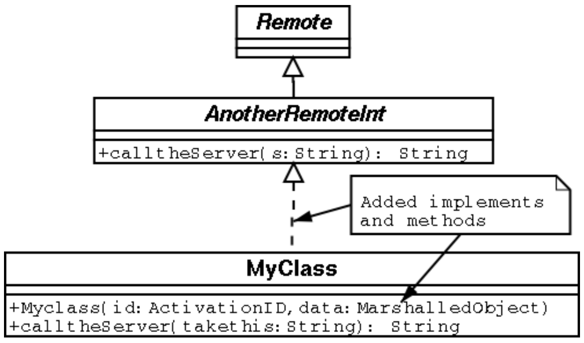
6. Run the client.

   ```
   java -Djava.security.policy=/home/rmi_tutorial/activation/policy
    examples.activation.Client vector
   ```

5.4 Converting UnicastRemoteObject to Activatable

- If you already have a `UnicastRemoteObject` and want to make it `Activatable` you just have to make some small changes to the implementation class.
  1. Make the appropriate imports.
  2. Make the class extend `Activatable`.
  3. Remove the no-argument constructor.
  4. Declare a two argument constructor.
- You then need to create a setup program just like before.

5.5 Object Into Activatable


Remote
AnotherRemoteInt
+calltheServer(s:String): String
Added implements and methods
MyClass
+Myclass(id:ActivationID,data:MarshalledObject)
+calltheServer(takethis:String): String

- To make `MyClass` into an activatable class we could also just make it implement an interface which extends Remote.

6 Custom RMI Socket Factory

- RMI uses plain sockets. If you want to encrypt or compress that data you should create and use a custom RMI Socket factory.
- You can create a custom `RMISocketFactory` that is used by all remote object, or you can associate it with only some objects.
- The steps are:
  1. Implement a custom `ServerSocket` and `Socket` classes (by extending the existing ones) which implement the encryption or compression you want.
  2. Implement a custom `RMIClientFactory` which returns one of your custom sockets. Basically, just redefine the `createSocket` function. This class must be serializable.
  3. Implement a custom `RMIServerSocketFactory`, as before. It need not be serializable.

Notes

[1]http://java.sun.com/j2se/1.4/docs/guide/rmi/getstart.doc.html
[2]http://java.sun.com/j2se/1.4/docs/guide/rmi/codebase.html
[3]http://java.sun.com/j2se/1.4/docs/guide/rmi/activation/activation.1.html
[4]http://java.sun.com/j2se/1.4/docs/guide/rmi/activation/activation.2.html
[5]http://java.sun.com/j2se/1.4/docs/guide/rmi/activation/activation.3.html
[6]http://java.sun.com/j2se/1.4/docs/guide/rmi/socketfactory/index.html
[7]http://www.wikipedia.org/wiki/Factory_method_pattern
[8]http://www.amazon.com/exec/obidos/ASIN/0201633612/multiagentcom
[9]http://jmvidal.cse.sc.edu/talks/javarmi/ClassServer.java

This talk is available at `http://jmvidal.cse.sc.edu/talks/javarmi`