Java RMI

OVERVIEW

Distributed systems are those where components running on multiple computers work together on a task. Java **Remote Method Invocation** (RMI) is a mechanism for distributed Java components to communicate with each other. RMI enables objects in one Java Virtual Machine (JVM) to be used by objects in a different JVM. These JVMs may even be on different computers spread across the Internet. In a normal Java program, all the objects reside in a single JVM and are all therefore ***local*** objects. Objects are called ***remote*** if they reside in a different JVM and normally (i.e. without RMI) they cannot interact with local objects. Distributed systems are usually divided into ***server*** components that make some functionality available and ***client*** components that call upon those server components for service. RMI specifies that server components are Java objects running on a server computer and client components are Java programs running on some number of client computers. An individual Java program (and the computer it runs on) may take on the role of client, server, or both.

Server objects make themselves available to clients by registering with the rmiregistry tool running on the same server computer (via `java.rmi.Naming.rebind`). Client programs obtain references to desired server objects by querying the rmiregistry on that server's computer (via `java.rmi.Naming.lookup`). Once a reference to a remote object is obtained, its methods may be called just as if it were a local object. In addition, if local objects are passed as parameters to a remote method, RMI sends a copy of those objects to the remote computer. NOTE: Objects that are only passed as parameters do not need to be registered with rmiregistry.

To use RMI, a server object must publish its applications programming interface (API) via some Java interface which, itself, must extend the `java.rmi.Remote` interface. Any server methods to be made available to clients must both, be declared in this interface, and be declared to throw `java.rmi.RemoteException`. Clients actually lookup references to this published interface rather than the server object itself. This allows RMI to substitute a *stub* implementation (a.k.a. *proxy*) of this interface which communicates with the real server object that is running on the server's JVM. The stub code (generated by the rmic tool) handles all the work of copying parameters back and forth between JVMs. The process of converting parameters to byte streams (via Java serialization), to be sent over a socket connection, is called ***marshalling.*** The the process of taking serial data and converting back to Java objects (on the receiving end of the socket) is called ***unmarshalling***. RMI takes care of all this via ***stubs*** running on the client and their partner ***skeletons*** running on the server side that in turn call the actual server objects. The actual server object must extend the `java.rmi.server.UnicastRemoteObject` class [or use the more advanced Java Activation Framework (JAF)] .

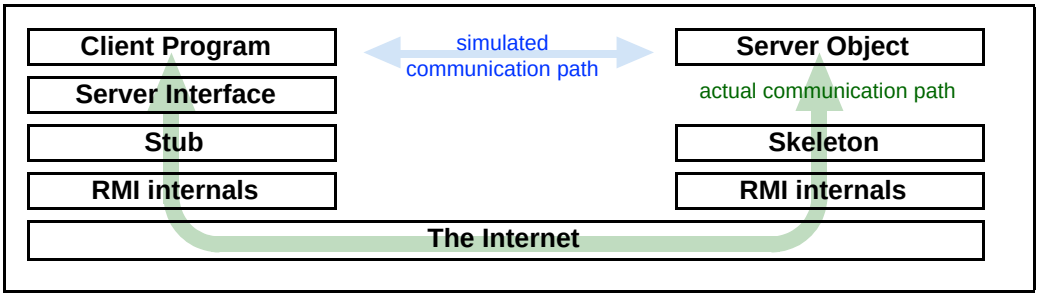

Figure 1. the RMI "protocol stack"

Standard web servers are used to copy local objects and interfaces to remote computers, in the same way that they can send Java Applet objects to remote browsers. So, all server computers must have a web server and any client computers that wish to pass local objects as parameters to remote methods must also have a web server. Java's ability to dynamically load bytecodes means that programs looking up remote objects may find and run them even though they were written on a different computer. In fact, as long as the server's public interface was published when the client program was written, the server objects may have actually been written at a later date [as long as each is registered via `rebind()` before the client calls `lookup()`]. Server objects are always registered with the rmiregistry running on the same computer as the server object itself. NOTE: the server objects do not have to "keep themselves alive" via thread waits, etc. because as long as either the rmiregistry or some client program contains a reference to a server object, it will not be garbage collected. Also, the rmiregistry process keeps the process of the JVM containing the server objects alive, so, the registering main program can exit as soon as it has finished doing rebinds.

COMPONENTS TO BUILD AND DEPLOY

With RMI, since multiple systems need to share Java class files, both at compile time and runtime, both locally via the CLASSPATH and also remotely via web servers, the build procedures are more complicated than normal Java programs. Multiple components must be developed, built, and deployed, and multiple programs must be run simultaneously for a distributed system using RMI to work. For dynamic code loading to work, certain class files will each need to be published via a web server on its parent computer. Even with dynamic class loading set up, some files (e.g. the Server API interface class files) must still be made available to all computers as local files. NOTE: It is possible for the same computer to be used as both the server and the client computer in the scenarios shown below.

The following are the components one must develop for even the simplest RMI system:
1) One or more Server Interfaces
2) One or more Server Objects
3) Server Stubs (that are generated by the **rmic** tool)
4) A program to register the Server Objects with the **rmiregistry** tool
5) A Client Program that looks up and calls one or more Server Objects

The following are the processes required to be running for even the simplest RMI system:
1) A web server running on the server computer
2) Each client computer that needs to send custom objects as RMI parameters requires its own web server
3) An **rmiregistry** running on the server computer
4) A JVM running on the server computer containing the registered Server Object(s)
5) A JVM running on the client computer running the Client Program that accesses Server Objects

The following steps are required to build even the simplest RMI system:
1) Compile the Server Interface(s) and make class file(s) available in the CLASSPATH on both client and server
2) Compile the Server program on the server computer
3) Compile the Client program on the client computer
4) Use the **rmic** tool on the server computer to generate Server Stub class files
5) copy the Server Interface and Stub classes to an area accessible via the Server computer's web server

The following steps are required to run even the simplest RMI system:
1) Run the **rmiregistry** tool as a process on the server computer
2) Run the web server as a process on the server computer (with shared class files in public area)
3) Run the program that registers the Server Objects on the server computer
4) Run the Client Program on the client computer

NOTE: A simple substitute for a full-blown web server is available from Sun that will serve classes as needed. It can be downloaded from:
ftp://ftp.javasoft.com/pub/jdk1.1/rmi/class-server.zip

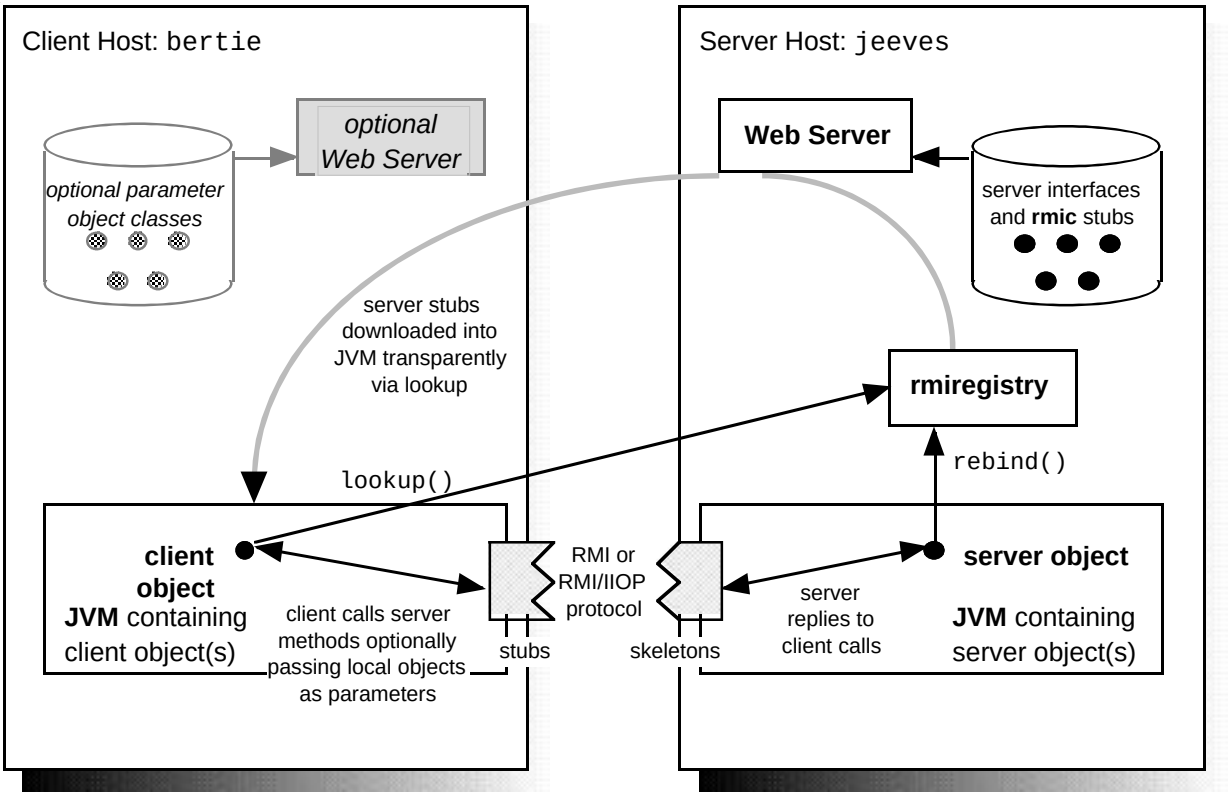

Figure 2. RMI Components, Processes, and Computers (aka Hosts)

Specific Example with Code

This small example RMI system will use the Sun class server and run everything on a single MS Windows computer. Three directories should be created for this example; c:\WEB_server, c:\RMI_server, and c:\RMI_client. Four Command (aka DOS) windows should be opened (3 server windows and 1 client window). A Java JDK should be installed and available via the default path. No CLASSPATH environment variable is needed for this example. There is a Java security policy file that can be used for both the client and the server. A copy should be placed in both c:\RMI_server and c:\RMI_client. Its contents are:

• **genericRMI.policy** - given in advance to both the client and server; defines access needed for RMI

```
grant
{
  permission java.net.SocketPermission "*:1024-65535", "connect,accept";
  permission java.net.SocketPermission "*:80", "connect";
};
```

SERVER WINDOW #1 will run the **rmiregistry** tool. It should have no CLASSPATH set and should run in a directory with no local .class files. There is no custom code for this window.

```
>cd c:\
>rmiregistry
```

SERVER WINDOW #2 will run the program registering the Server Object. The source for the Server Object, its main program that acts to register that object, and its public API interface are below. These .java files should be placed in c:\RMI_server.

```
>cd c:\RMI_server
>javac MyServer.java
>copy MyInterface.class c:\WEB_server
>rmic MyServer
>copy MyServer_*.class  c:\WEB_server
>REM wait until WINDOW #3 is set up and classServer is running, then continue
>java -Djava.rmi.server.codebase=http://localhost/ -Djava.security.policy=genericRMI.policy MyServer
```

• **MyInterface.java** - given in advance to both the client and server; defines server methods available remotely

```
public interface MyInterface extends java.rmi.Remote
{  public String getAnswer() throws java.rmi.RemoteException;
}
```

• **MyServer.java** - a server object and a utility main program to register it.

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class MyServer extends UnicastRemoteObject implements MyInterface
{
    public MyServer() throws RemoteException{ super(); }
    public String getAnswer(){ return "theAnswer"; }
    public static void main( String[] args )
    {    System.setSecurityManager( new RMISecurityManager() );
         try
         { MyServer me = new MyServer();
           Naming.rebind( "/MyServer", me );
         } catch( Exception x ){ System.out.println("Err["+x+"]"); }
    }

}
```

SERVER WINDOW #3 will run the Sun class server (acting as a web server) and its source should be downloaded and unzipped into the WEB_server\examples\classServer directory since it is defined in the examples.classServer package. The class files from Window #2 should be copied before running the classServer. The port and the directory to find the public class files are the two parameters to classServer.

```
>cd c:\WEB_server
>javac examples/classServer/*.java
>java examples.classServer.ClassFileServer 80 c:\WEB_server
```

CLIENT WINDOW will run the Client Program whose source is below.

```
>cd c:\RMI_client
>copy c:\RMI_server\MyInterface.class .
>javac MyClient.java
>java -Djava.security.policy=genericRMI.policy MyClient
```

• **MyClient.java** - a program remotely using a Server Object. NOTE: If no server hostname is specified to lookup then ***localhost*** is assumed.

```
import java.rmi.*;
public class MyClient
{
    public static void main( String[] args )
    {    System.setSecurityManager( new RMISecurityManager() );
         try
         { Remote   rObj = Naming.lookup("//localhost/MyServer");
           MyInterface x = (MyInterface) rObj;
           System.out.println("Answer is ["+ x.getAnswer() +"]");
         } catch( Exception x ){ System.out.println("Err["+x+"]"); }
    }
}
```