Deploying Java

or

How many ways can you write "Hello, World"?

Jay Danielsen
Systems Engineer
Sun Microsystems
January 2000

Table of Contents

1. INTRODUCTION 3
2. DEFINITIONS 3
3. JAVA DEPLOYMENT ALTERNATIVES 4
3.1. APPLICATION 4
3.2. HTML 5
3.3. APPLET 6
3.4. SERVLET 7
3.5. JAVA SERVER PAGE 9
3.6. JSP/JAVABEAN INTEGRATION 10
3.7. RMI 11
3.8. COMMON OBJECT REQUEST BROKER ARCHITECTURE (CORBA) 13
3.9. JINI 16
3.10. ENTERPRISE JAVA BEAN 20
4. APPENDIX 23
5. REFERENCES 37

1. Introduction

There's more to Java than meets the eye. By my count, there are dozens of ways to deploy a Java technology solution. As the number of alternative Java technologies continues to grow, it may be difficult to decide which type of Java deployment mechanism works best for a given software architecture. This paper attempts to define how best to deploy a Java solution.

In this paper, I provide a brief discussion of the following Java deployment APIs: Application, Applet, Servlet, Java Server Page, JavaBean, RMI, Common Object Request Broker Integration, Jini, Enterprise Java Bean, SmartCard/JavaRing, Java Database Connectivity, and Java Dynamic Management (Managed Bean). And because Java builds on and extends the capabilities of both HTML, I also include a brief discussion of the HTML protocol.

To learn the ins-and-outs of implementing these various Java technologies, I have found that experimenting with simple sample code to be the best teacher. In keeping with the tradition of Computer Science 101, each of my examples demonstrate the Java technology being examined with an implementation of a 'Hello, World' program to demonstrate the technology.

For each deployment technique, I include the following information:
- A Protocol description
- Distinguishing characteristics (How to you know what you're looking at)
- How it is deployed and where it runs
- How does an end user interact with it
- Advantages and Disadvantages
- Sample Code
- How the Code is compiled, deployed, and invoked

2. Definitions

Before we get started, I expect it will be useful to define some Java terms related to the technologies we will be discussing.

Java Virtual Machine (JVM)
The Java Virtual Machine provides an operating system abstraction layer implementing the following services:
- A Class Loader that locates and loads Java objects, also known as classes.
- A Byte Code Verifier which verifies the integrity of the loaded Java class, eliminating the possibility of harmful 'virus' code.
- An Execution Engine used to execute the Java code.
- And finally, a Garbage Collector used to manage system memory and automatically clean up and remove Java code that is no longer in use.

Java
Java is a portable object-oriented platform and computing environment. The familiar motto associated with Java - 'Write Once, Run Anywhere' – refers to the fact that the object code written in the Java language can be executed on any computing device that supports a Java Virtual Machine, regardless of processor architecture and operating system. Java is also the command supplied with the JDK and used to load and execute Java objects.

Java Compiler
Software developers can download the Java Developer's Kit (JDK) from http://java.sun.com/jdk/. The JDK includes a Java Virtual Machine for executing Java classes and a compiler for converting your Java software into the executable byte code objects. The java compiler provided with the JDK is known as javac.

Classpath
CLASSPATH is an important environment variable used by the JVM to locate Java classes not explicitly identified.

3. Java Deployment Alternatives

3.1. Application

A Java Application is a standalone piece of Java code. What that means is that all Java class files required to run the Java program reside locally on the system which will be executing the Java code. An application does not require any additional Java class files be downloaded.

The Java application is self-contained, requiring no networking or other connectivity in order to run. Everything needed to execute the program is already contained on the system.

As you would expect, an application is executed on your local desktop computer.

Applications can be implemented with command line interfaces or graphical interfaces, it's'up to the developer to decide which form makes sense for the provided service.

Advantages to this form of Java are that, compared to the other forms, applications have the shortest load and start-up times, since all the code required to run is already resident on the system.

The disadvantage to this approach is that when you implement an application you forego an important benefit of Java: the ability to dynamically download objects when they are required. Applications thus require the highest degree of administration; software distribution solutions must be architected to support this type of Java deployment just as they must be architected for non-Java (C, C++, other) binary software distribution.

The following code demonstrates the simplest form of Java application. Note that this code contains a `main()` function. The `main()` declaration indicates that this code is a Java Application.

File HelloWorld.java:
```
public class HelloWorld {
     public static void main(String[] args) {
          System.out.println("Hello, World");
     }
}
```

Compile
- `javac HelloWorld.java`

Deploy
- `on the client`

Execute
- `java HelloWorld`

3.2. HTML

Although not a Java Technology per se, much of Java's capability depends on and leverages the existence and capabilities of HyperText Markup Language (HTML). HTML is the language of the internet; internet browsers such as netscape navigator, microsoft internet explorer convert the HTML content downloaded from your favorite web site into the text and graphics displayed on your browser.

HTML is static information accessed via the HTTP protocol. The HTML language itself is rather uninteresting. Extensions embedded within special HTML tags are what make the web interesting by enabling dynamic web content to be included within the HTML.

HTML runs on the client inside an HTML-compatible web browser. HTML content is downloaded to the browser by entering a URL address, such as http://www.sun.com/Hello.html

HTML advantages include centralized administration and deployment; administration and deployment associated with HTML is the overhead associated with creating and maintaining a web site.

The disadvantage of an HTML-only solution is that all that can be delivered is static information. Fortunately, HTML provides for extensions that increase the usefulness of this deployment framework.

File: Hello.html

```
<HTML>
<BODY>
<H1>
Hello, World
</H1>
</BODY>
</HTML>
```

Compile

- `N/A`

Deploy

- `web server`

Execute

- `http://hostname/Hello.html` `(from web browser)`

3.3. Applet

Building on the HTML framework, the possibilities start to get interesting. A Java Applet is a downloadable piece of Java code. Instructions for downloading this form of Java are embedded in an HTML file. Applets are another form of client-side Java, which is to say that this form of Java is downloaded from a web server and executed on the client workstation inside a java-enabled web browser.

The instructions to download a Java Applet are contained in the HTML <APPLET> tag. This tag, which is included in an html file, provides instructions defining where to locate the java code and the width and height (in pixels) to allocate for display of the applet's graphical output.

The following applet tag:

```
<Applet CODE=Hello.class WIDTH=300 HEIGHT=200 ></Applet>
```

tells the browser to download a java class named 'Hello.class' and create a 300x200 pixel graphical panel inside the browser for displaying the applet's output.

Java applets have the advantage of centralizing administration and deployment of dynamic code, but a disadvantage of the applet form is that delivery of large amounts of Java code via this mechanism requires a high-bandwidth network.

From this point on, more than one file is required to deploy the java class code. In this case, both the html file (containing an applet tag), and the Java applet itself are required.

HTML File: HelloApplet.html

```
<HTML>
<BODY>
<APPLET CODE=Hello.class WIDTH=300 HEIGHT=100>
<PARAM NAME="name" VALUE="Jay">
<PARAM NAME="size" VALUE="28">
</APPLET>
</BODY>
</HTML>
```

Applet File: Hello.java

```
import java.awt.Label;
import java.awt.Font;
import java.awt.Applet;

public class Hello extends Applet {
     init() {
          String name = getParameter("name");
          String size = getParameter("size");
          Label label = new Label("Hello," + name);
          label.setFont(new Font("dialog", Font.BOLD,
          Integer.parseInt(size)));
          add(label);
     }
}
```

Compile

- `javac Hello.java`

Deploy

- `install on web server`

Execute

- `http://hostname/HelloApplet.html (from web browser)`

Comparing Java applet code to Java application code, you will notice that the applet has no `main()` method, but instead contains an `init()` method. The `init()` method is required for all applets, and is called by the browser's JVM to instantiate the java object.

3.4. Servlet

Java objects do not have to be downloaded to a client to deliver dynamic content to the desktop. Servlets, as you may deduce, are server-side java objects. Once again, you communicate with this form of Java via your trusty web browser. In this case, the java code remains on and is executed on the web server. The servlet output is delivered to the browser, and can take the form of any application type known by your browser; you may

have heard of the term 'mime type' – these types are the forms of output that you can deliver by a servlet.

The servlet form is steadily gaining popularity as a better alternative to Perl and the Common Gateway Interface (CGI) on the web server. In addition, servlets are one of the key technologies associated with the Java 2 Enterprise Edition (J2EE). The J2EE framework provides the additional benefit of load balancing and clustering your java classes for high availability. For additional information on J2EE, see http://java.sun.com/j2ee.

Your browser communicates with a servlet via the HTTP protocol, supporting a small number of commands – GET, POST, HEAD, PUT, TRACE, OPTIONS. Most servlets can get by supporting only the GET request and letting the servlet framework handle the rest of the protocol.

One unique feature of servlets is the ability to pass a parameter via the URL. Thus a URL formed as follows:

`http://www.sun.com/servlet/HelloWorld?name=foo`

contacts the servlet named `HelloWorld` on host `www.sun.com`, and additionally assigns the value `foo` to the variable `name`.

Advantages to this form of Java are that the servlet, while similar in function to its CGI equivalent, is much more scalable – servlets are handle simultaneous client requests via the threads interface, which is a more efficient mechanism for handling high volumes of web access. Disadvantages to the servlet framework are few – but servlets don't provide the complete solution. Servlets have no provision for a sophisticated GUI – complex client GUIs will require some form of applet combined with the servlet framework. With the servlet, you are now able to segment your application into multiple tiers – HTML and Applet for presentation logic and the servlet for mid-tier business logic.

File: HelloServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet {
     public void doGet(… req, … res) {
          res.setContentType("text/html");
          PrintWriter out = res.getWriter();

          String name = req.getParameter("name");
          if ( name == null )
               name = "World";

          out.println("<html><body><h1>Hello, " + name);
```

```
        out.println("</h1></body></html>");
    }
}
```

Compile

- `javac HelloServlet.java`

Deploy

- `install on web server`

Execute

- `http://hostname/servlet/HelloServlet`

3.5. Java Server Page

The Java Server Page (JSP) builds on the servlet technology. JSPs are snippets of servlet code embedded directly within an HTML file. JSPs are identified in an HTML file via special <jsp:> tags which cause the embedded java code to be compiled on the web server into an executable servlet.

Embedded JSP script looks just like servlet code and it runs just like servlet code on your web server. The unique feature of a JSP is its ability to be constructed on the fly on a web server. JSP files are named with a `.jsp` extension which triggers a web server to pre-process the HTML, looking for embedded JSP code to compile into Servlet code.

JSP technology adds a great deal of flexibility to a web page. The JSP framework provides the servlet request and response classes along with input and output capability to the JSP framework, so as a developer you only include your java logic. But JSPs must be converted to servlets on the web server – the first time a JSP is accessed, the web server compiles it into a running servlet, which takes a finite amount of time. And consider this - any servlet engine can also become a JSP engine – the JSP engine is itself a servlet running on the web server!

JSPs – like Servlets – are more suited for the business logic tier of your application framework than they are for the presentation tier. JSPs do not provide for delivery of sophisticated GUIs.

File: Hello.jsp

`Request, response, in`, and `out` are built-in variables. This bit of JSP code uses the built-in `request` and `out` objects.

```
<HTML>
<BODY>
<%
    String name = request.getParameter("name");
    if ( name == null )
        name = "World";
```

```
     out.println("Hello, "+name);
%>
</BODY>
</HTML>
```

Compile
- `N/A`

Deploy
- `install on web server`

Execute
- `on web server via URL http://hostname/Hello.jsp`

3.6. JSP/JavaBean Integration

JSPs provide the useful feature of being able to integrate with JavaBeans components. JavaBeans are self-contained Java classes that are neither Applications nor Applets (which is to say they don't have a main() or an init() method). JavaBeans are their own type of class, with minimal restrictions associated with them. If you want to expose a Beans methods to any number of integration frameworks, including JSP integration, you include `set` and `get` methods. The JSP provides for instantiating a JavaBean and calling its get and set methods.

Since JavaBeans are separate pieces of Java Code floating out on the network somewhere, it would be nice to be able to control how long the Bean objects are active and when they can be garbage collected. The JSP protocol provides for this 'time-to-live' capability through – surprise! – a <jsp:> tag. The Bean integration tags provide for various levels of persistence: per page hit, per connection, or per user session.

File: HelloBean.jsp
```
<jsp:useBean id="helloB" class="HelloBean"
scope="session"/>
<html>
<body>
<% helloB.getString() %>
</body>
</html>
```

File: HelloBean.java
```
package hello;
import java.util.*;

public class HelloBean {
     int i=0;
     String[] s = { "so Long, World",
```

```
                    "Adios, Mundo",
                    "auf Wiedersen, Welt" };
      public HelloBean() {
      }

      public String getMessage() {
            return s[(i++)%s.length];
      }
}
```

In the file HelloBean.jsp, note the jsp:useBean tag, which defines the object helloB as an instantiation of the class HelloBean.class. HelloBean.class object will remain activated for the entire user session, which means it won't be garbage collected until the user exits his web browser. Note also that I grew tired of the same old 'Hello, World' message with this Bean. Since the Bean remains active on my Web server, when I reload my web page I cycle through all of the greetings I have defined in my String array `s[]`.

Compile

- `javac HelloBean.java`

Deploy

- `install HelloBean.jsp and HelloBean.class on web server`

Execute

- `http://hostname/Hello.jsp`

3.7. RMI

RMI stands for Remote Method Invocation. RMI is client-server Java. A web server is not required for this type of Java, but it can still add value to the RMI infrastructure. When you think of traditional client-server development, this form of Java deployment comes closest to that particular paradigm. There are some unique features associated with RMI that make it much more useful than traditional client-server.

RMI allows you to deploy distributed objects across your network infrastructure for true distributed computing. This capability requires an extra process, known as the RMI registry, to be running somewhere in your compute environment. The RMI registry allows Java server code to register either its location or the entire object with the registry service. When a Java client wishes to access the RMI server's service, it first contacts the RMI registry to either locate the remote Java object and communicate with it, or in the second form of RMI, to download the entire RMI object and access the methods locally.

RMI implements the Java Security Management framework, thus you must explicitly grant access for remote Java object code to be executed on your local machine.

Once the remote service has been identified through the RMI registry, the reference returned from the registry allows the client to access the remote methods as through they were associated with local objects. The software plumbing to make this happen is taken care of in an initial Stub/Skeleton pre-compilation step.

RMI is not tied to a specific form of GUI. The client portion of the RMI infrastructure can be implemented with either a command line Graphical User interface.

Advantages to this form of Java are an additional level of flexibility for Enterprise deployments; a disadvantage to this mechanism is that RMI requires the use of arbitrary network port numbers, which make it difficult to deploy through firewall architectures.

From this point on, the Java code becomes more complex. Beginning with the RMI example, I will highlight only the relevant code sections. The complete sample code listing can be found in the **Appendix**.

File: `HelloRMI.java` defines an RMI interface

```
import java.rmi.*;
public interface HelloRMI implements Remote {
     public String getSalutation() throws RemoteException;
}
```

Step 1: Create and compile an RMI interface file containing the prototype for each method provided by the RMI server service. The Java interface will be used to generate the stub/skeleton code for the client and server tiers of this solution.

```
javac HelloRMI.java
```

Step 2: Implement the Client and Server classes.

HelloClient.java – See RMI Sample Code in Appendix
It is the Client's responsibility to contact the RMI registry for a handle to the remote object:

```
String name = "rmi://localhost/HelloRMI";
```

If this returns successfully, the client creates a new SecurityManager (required when we're dealing with remote objects), performs a lookup on the object, then can begin accessing the remote object's methods.

```
System.setSecurityManager(new RMISecurityManager());
HelloRMI HelloObj = (HelloRMI)Naming.lookup(name);
String results = HelloObj.getSalutation();
```

File: HelloServer.java - See RMI Sample Code in Appendix

The RMI server implements the interface defined in HelloRMI.java. The server implements the methods defined in the interface file and registers itself with the RMI

registry service. This example defines the registry service on the localhost. In real-life implementations, the registry would most likely be 'somewhere else' in the network.

```
System.setSecurityManager(new RMISecurityManager());
HelloServer server = new HelloServer();
String host = InetAddress.getLocalHost().getHostName();
String url = "rmi://" + host + "/HelloRMI";
Naming.rebind(url, server);
```

In addition to binding with the registry, the RMI server code also places its stub code on a web server for client download.

Compile
- Interface javac HelloRMI.java
- Client javac HelloClient.java
- Server javac HelloServer.java
  rmic HelloServer

Deploy
Interface N/A
- Client install on client
- Server install on server (doesn't have to be web server)
- Stub install on web server
- Registry Start it up on known network host

Execute
- Client java -Djava.security.policy=… HelloClient <hostname>
- Server java -Djava.rmi.server.codebase=http://… -Djava.security.policy=… HelloServer

Note the special command line options associated with the client and server startup. `java.security.policy` defines the permissions granted to downloaded code. `java.rmi.server.codebase` defines the location of the web server containing the stub class.

3.8. Common Object Request Broker Architecture (Corba)

You've probably noticed that the complexity is increasing with each Java technology we've added to this point. Keep in mind that each of these technologies builds on the previous technology, so don't be intimidated by all this code!

The Common Object Request Broker Architecture (Corba) is a Non-Java enterprise framework allowing heterogeneous client-server code to communicate via a common API. The Java developers Kit (JDK) provides for the client-server communication of Java-to-Corba services. Both Java-as-Client and Java-as-Server are supported, although it's more likely you will use this capability to communicate with non-Java server objects.

Corba is similar to RMI in that you first define a model of the services that will be provided over the network. In the case of RMI, the model is known as an Interface. In the Corba world, the interface model is defined using the Interface Definition Language (IDL). The IDL interface is compiled with the JDK's idl2java compiler. This tool generates both the client and the server stubs, which can be used as starting points for your Java-to-Corba services. Generally, you would implement either the client stub or the server stub – you're most likely creating a java object to integrate with a pre-existing Corba client or server. In our example, I implement both client and server – so I have a Java client-server solution implemented to communicate via the Corba Internet Inter-ORB Protocol (IIOP) protocol (in place of the Java native communication protocol, Java Remote Method Protocol (JRMP)).

The Corba equivalent of the RMI registry is the Object Request Broker or ORB. The ORB provides the same services – the Corba server registers its service with the ORB, the Corba client contacts the ORB to locate and access the service.

The obvious advantage to this Java solution is the ability to interoperate in a heterogeneous distributed computing environment. Disadvantages include the requirement for the additional ORB Service daemon. And as was the case with RMI, the corba infrastructure also utilizes arbitrary network ports to deliver its services which make internet deployment through firewalls difficult to accommodate.

Now that we've gone over the basics, let's create our Java-Corba Hello, World solution.

Step 1. define the IDL for our application and pre-process with `idltojava` compiler.
Interface File: Hello.idl

```
module HelloApp {
     interface Hello {
          string sayHello();
     };
};
```

Pre-process with idltojava compiler

```
idltojava Hello.idl
```

This step creates the HelloApp subdirectory containing stubs for the Corba client and server.

Step 3. Implement the Server

```
// Create and initialize the ORB
      ORB orb = ORB.init(args, null);

      // Create the servant and register it with the ORB
      HelloServant helloRef = new HelloServant();
      orb.connect(helloRef);

      // Get the root naming context
```

```
	org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
	NamingContext ncRef = NamingContextHelper.narrow(objRef);

	// Bind the object reference in naming
	NameComponent nc = new NameComponent("Hello", "");
	NameComponent path[] = {nc};
	ncRef.rebind(path, helloRef);
```

The classes have different names, but the process is essentially the same. The server first contacts the ORB, then registers and binds its HelloServant() proxy with the ORB.

The server requires some of the stub files generated by the `idltojava` command and contained in the `HelloApp` directory. Compile the client as follows:

```
javac HelloServer.java HelloApp/*.java
```

Step 4. Implement the Client

```
// Create and initialize the ORB
	ORB orb = ORB.init(args, null);

	// Get the root naming context
	org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
	NamingContext ncRef = NamingContextHelper.narrow(objRef);

	// Resolve the object reference in naming
	NameComponent nc = new NameComponent("Hello", "");
	NameComponent path[] = {nc};
	Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));

	// Call the Hello server object and print results
	String hello = helloRef.sayHello();
	System.out.println(hello);
```

Here the client contacts the ORB, then requests the Hello service. Once a reference to the Hello service is obtained, the helloRef object can call methods contained in the remote object.

The client requires some of the stub files generated by the `idltojava` command and contained in the `HelloApp` directory. Compile the client as follows:

```
javac HelloClient.java HelloApp/*.java
```

Step 5. Start the Object Request Broker (ORB) and Server

The Java Developers Kit (JDK) includes a development ORB which can be used to test your CORBA solution. The included ORB is included in the `tnameserv` program and is invoked as follows:

```
tnameserv -ORBInitialPort 1050

To start the server:
java HelloServer -ORBInitialPort 1050
```

Step 6. Start the Client
`java HelloClient -ORBInitialPort 1050`

Voila! You're finished!

Compile
- Interface `idltojava Hello.idl`
- Client `javac HelloClient.java HelloApp/*.java`
- Server `javac HelloServer.java HelloApp/*.java`

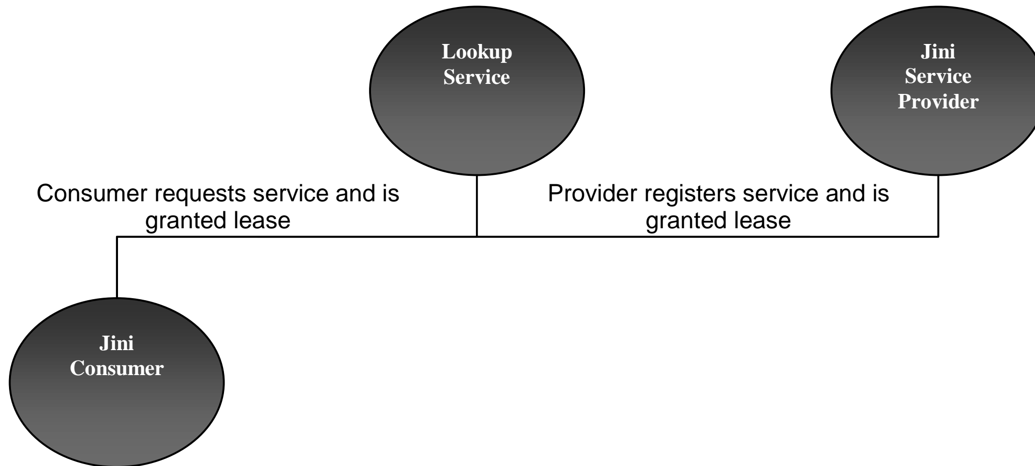
Deploy
- ORB `anywhere`
- Client `anywhere`
- Server `anywhere`

Execute
- ORB `tnameserv -ORBInitialPort 1050`
- Client `java HelloClient -ORBInitialPort 1050`
- Server `java HelloServer -ORBInitialPort 1050`

3.9. Jini

Chances are you have already read something about Jini. The Jini technology provides a mechanism to automatically identify network services through a computing resource known as the Lookup service. The Lookup Service is similar to the registry services provided with RMI and CORBA; the difference is that the Jini infrastructure improves on the concept of a service registry by providing for service discovery. The Jini Discovery service is essentially a broadcast mechanism that allows a client to find a particular service without prior knowledge of the location of the service.

Jini builds on the RMI communication infrastructure, but it hides the implementation details in utility classes. The Jini framework takes care of the RMI plumbing for you. Jini is a 'self-healing' infrastructure, based on the concept of service leases. When a client requests a service it is granted a lease providing exclusive access to the service for a fixed amount of time. Thus if the client should die or exit from a Jini community unexpectedly, the Lookup service will detect the loss of the client on lease expiration, garbage collect any dangling objects, and re-advertise the availability of the service. Likewise, the Lookup service also cleans up after a service provider. The service provider also is assigned a lease with the Lookup service, thus should a service be depleted, crash or otherwise come up missing in action, the Lookup service can clean up its infrastructure on lease expiration.



Advantages to the Jini model are simple deployment, late binding, distributed computing, and URL-style access to services. There's a lot of interest in Jini coming from the embedded devices market, for which this technology typically solves late binding integration problems. Jini APIs are defined up front, thus Jini allows you as a developer to push integration out later into the development lifecycle, enabling late binding. But Jini is still a little top heavy for embedded solutions. Sun is actively working on optimizing the Jini framework for embedded application deployment.

The following steps will help you get started with the development of a Jini community.

Jini Infrastructure Requirements

In addition to the Lookup Service, the Jini Architecture requires RMI and Web services for service registration and delivery. The Jini toolkit provides these services and a GUI to start them. Modify the definition of JINIHOME in the startup script to reflect your Jini toolkit installation directory.

```
#!/bin/sh
JINIHOME=/files/jini1_0/lib
java -cp $JINIHOME/jini-examples.jar
com.sun.jini.example.service.StartService
```

When the `StartService` GUI is displayed, select the `Jini Lookup Service` radio button on the Registry services tab. Select the `Run it All!` tab and start up the Web, RMI, and Lookup services. The `StartService` GUI also supports Transaction and JavaSpaces sevices beyond the scope of this document. Those interested in these additional Jini Services should refer to the `Core Jini` book referenced at the end of this paper.

Jini Service Implementation

To demonstrate the distributed nature of Jini objects, a separate Web server is instantiated to provide the services communications stub.

```
#!/bin/sh
JINIDIR=/download/java/jini
JINIHOME=/files/jini1_0_1
java -jar $JINIHOME/lib/tools.jar -dir $JINIDIR/service-dl -port 8085 &
```

The next step is to establish a Java CLASSPATH that pulls in the necessary service libraries.

```
 #!/bin/sh
JINIDIR=/download/java/jini
JINIHOME=/files/jini1_0_1
CLASSPATH=$JINIHOME/lib/jini-core.jar:$JINIHOME/lib/jini-
ext.jar:$JINIHOME/lib/sun-util.jar:$JINIDIR/service:$JINIDIR/service-dl
export CLASSPATH
```

The Jini `ServiceWrapper` service implements Discovery, Service Registration, and Lease Renewal. The source code for the `ServiceWrapper` example is included in the Appendix. For additional detail on these examples, refer to the Core Jini, by W. Keith Edwards.

Compile

```
javac -d $JINIDIR/service
$JINIDIR/corejini/chapter8/ServiceWrapper.java
```

Deploy

`MyProxy.class` deployed to service-dl directory associated with service web server. `ServiceWrapper` Service can be deployed on any host on subnet.

Execute

```
java -Djava.rmi.server.codebase=http://$HOST:8085/ -
Djava.security.policy=$JINIDIR/policy.all
corejini.chapter8.ServiceWrapper
```

Jini Client

The interesting feature of this Jini client is the inclusion of the callback mechanism. Thus when the client contacts the Lookup service, if the service is not available, the client registers an interest in the service through an event callback. This feature allows clients to join the Jini community ahead of the Jini services, and through the callback be notified when he service comes on-line in the future. The client implements Discovery, Template-based service, Lease Renewal, and service event callback.

To demonstrate the distributed nature of Jini objects, another Web server is instantiated to provide the client callback stub.

```
#!/bin/sh
JINIDIR=/download/java/jini
JINIHOME=/files/jini1_0_1
java -jar $JINIHOME/lib/tools.jar -dir $JINIDIR/client-dl -port 8086 &
```

The next step is to establish a Java CLASSPATH that pulls in the necessary client libraries.

```
#!/bin/sh
JINIDIR=/download/java/jini
JINILIB=/files/jini1_0_1/lib
CLASSPATH=$JINILIB/jini-core.jar:$JINILIB/jini-ext.jar$JINILIB/sun-util.jar:$JINIDIR/client:$JINIDIR/client-dl
export CLASSPATH
```

Compile

- `HelloWorldClientWithLeases` extends `HelloWorldClientWithEvents` which extends `HelloWorldClient – all three required for this example. You can examine each individually to gain undestanding of each of the key Jini concepts: HelloWorldClient – Discovery, HelloWorldClientWithEvents – callbacks, HelloWorldClientWithLeases – leasing.`

```
javac -d $JINIDIR/client
$JINIDIR/corejini/chapter5/HelloWorldClient.java
javac -d $JINIDIR/client
$JINIDIR/corejini/chapter5/HelloWorldClientWithEvents.java
javac -d $JINIDIR/client
$JINIDIR/corejini/chapter5/HelloWorldClientWithLeases.java
```

Deploy

`HelloWorldCLientWithEvents$EvtListener_Stub.class` deployed to client-dl directory associated with client web server.
`HelloWorldClient`, `HelloWorldClientWithEvents`, and `HelloWorldClientWithLeases` can be deployed together on any host on subnet.

Execute

```
java -Djava.rmi.server.codebase=http://$HOST:8086/ -
Djava.security.policy=$JINIDIR/policy.all
corejini.chapter5.HelloWorldClientWithLeases
```

Setup Specifics

- For this example, both the client and the service components are running on the same host. Normally these components would be distributed on different hosts across the network.
- Separate Web servers are configured for the Lookup Service (port 8080), for the Hello Service (port 8085), and for the client callback registration (port 8086) to demonstrate the distributed nature of the Jini framework.
- The distributed object framework requires a security infrastructure. The Java security policy file used for this example provides a wide-open resource access to the systems running the java/jini components. Typically the security policy file would be more restrictive.

  `policy.all:`

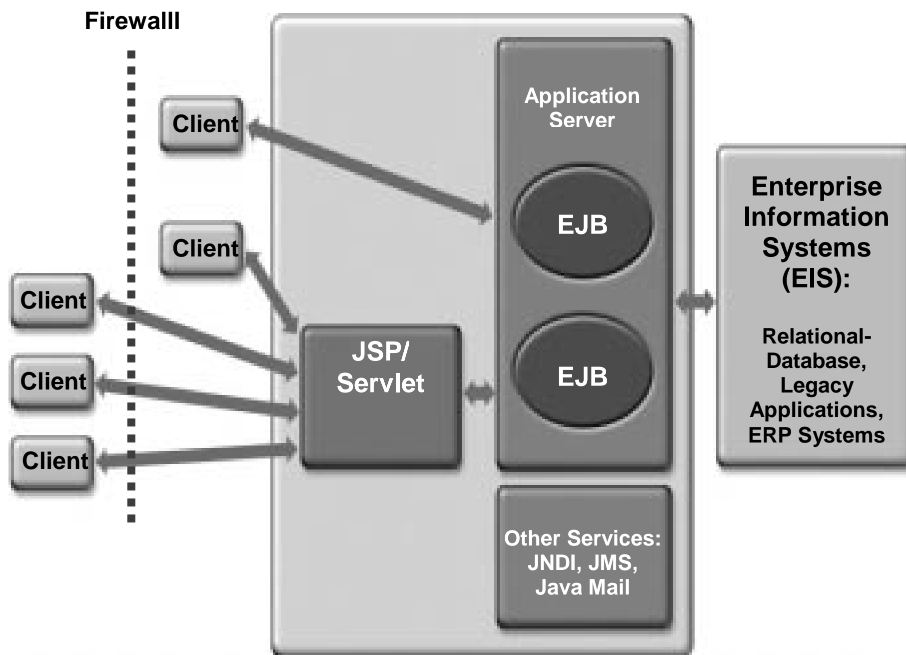
  ```
  grant {
     // Allow everyone access to everything
     permission java.security.AllPermission;
  };
  ```

3.10. Enterprise Java Bean

Enterprise Java Bean (EJB) is the final Java deployment technology we review. EJBs are part of the Java 2 Enterprise Edition (J2EE) framework and are most often used to implement integration with legacy databases and client state and session management in an enterprise application environment. The J2EE framework leverages legacy data sources to integrate the back-end data souces to middle-tier enterprise business logic. As you may have guessed, the J2EE framework provides the middle-tier plumbing for the enterprise, which enables enterprise framework architects and developers to focus on business logic, not communications infrastructure.

EJB/J2EE is a client-server deployment architecture which consists of client, Servlet/JSP, EJB, and Enterprise Integration tiers (see figure). The J2EE framework manages state, session, transaction management, lifecycle, and persistence – capabilities that formerly required extensive knowledge and programming effort to implement. The J2EE framework simplifies the deployment of enterprise applications through interposition – EJBs are deployed in an application server container which isolates and intercepts client communications and applies behavior (state, session, persistence, lifecycle, transaction management) across the wrapper. Thus the EJB framework provides flexibility to re-use EJB java objects and change the behavior of the object my modifying the behavior of the container. For example, an object can be deployed in one container with container-managed persistence. In another container, the same object methods are required, but persistence is not supported. Thus the same EJB can be deployed across multiple containers with different behaviors.

Because the J2EE framework is based on the RMI communications infrastructure, we have difficulty deploying these solutions through firewalls. Typically, J2EE delivers thin client to the internet via a servlet/jsp web tier, which manages communications with the



EJB tier.

EJB Deployment Scenario

EJBs can take several forms – stateless session, stateful session, and entity. For detailed information regarding the various forms of EJB, please refer to the J2EE tutorial contained in the J2EE reference implementation documentation at http://java.sun.com/j2ee. This tutorial offers detailed examples of each form of EJB.

We'll step through the simplest form of EJB, the stateless session bean. A stateless session bean does not maintain a conversational state for any particular client. Thus this form of EJB is highly scalable, since any stateless bean is capable of servicing any client request.

To communicate with an EJB, a client first finds and contacts the EJB's `home` interface. The container maps the `home` interface request to a `remote` interface object, which is used by the client to access the true EJB. Thus when we implement an EJB we must provide the EJB container a `home` interface, a `remote` interface, and the EJB class – stateful or stateless session, or entity. We will implement our `Hello` service as a stateless session bean.

- `Hello` service `home` Interface – `HelloHome.java`. The `home` interface defines the methods that allow a client to create, find, or remove an enterprise bean.

```
import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface HelloHome extends EJBHome {

    Hello create() throws RemoteException, CreateException;
}
```

- `Hello` service `remote` Interface – `Hello.java`. The `remote` interface defines the business methods that a client may call. The business methods are implemented in the enterprise bean code. Our Bean supports a single method – `getMessage`().

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Hello extends EJBObject {

   public String getMessage() throws RemoteException;
}
```

- `Hello` service SessionBean – `HelloEJB.java`. This Bean defines all of the required EJB methods, overriding default methods when necessary, and implementing the `getMessage()` method.

```
import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class HelloEJB implements SessionBean {

   public String getMessage() {

      return "Hello, EJB!";
   }

   public HelloEJB() {}
   public void ejbCreate() {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}
   public void setSessionContext(SessionContext sc) {}

} // HelloEJB
```

- HelloClient - the following code implements a command line client. The important methods to understand are:
  - obtain an object reference to MyHello via the `initial.lookup()` method
  - map the reference to a `HelloHome` type, using the `home` interface
  - call the `home` interface's create method to gain access to remote `getMessage()` method
  - call the EJB methods via `remote` interface – `getMessage()` , in our example

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

import Hello;
import HelloHome;

public class HelloClient {

   public static void main(String[] args) {
       try {
           Context initial = new InitialContext();
           Object objref = initial.lookup("MyHello");

           HelloHome home =
               (HelloHome)PortableRemoteObject.narrow(objref,
                                             HelloHome.class);

           Hello hello = home.create();
         for(int i=0; i < 50; i++) {
            System.out.println(hello.getMessage());
```

```
            }

        } catch (Exception ex) {
            System.err.println("Caught an unexpected exception!");
            ex.printStackTrace();
        }
    }
}
```

Compile

- use shell script to automate the compilation process -

#!/bin/sh
J2EE_HOME=<installation-location>
CPATH=.:$J2EE_HOME/lib/j2ee.jar
javac -classpath "$CPATH" HelloEJB.java HelloHome.java Hello.java

Deploy

Use J2EE reference implementation's `deploytool` to package the EJB files into a .jar file for deployment. Specifics on the use of `deploytool` are included with the J2EE reference implementation available at http://java.sun.com/j2ee.

Execute

- Start the Server – adjust to reflect directories containing J2SDKEE and JDK

```
#!/bin/sh
J2EE_HOME=/j2sdkee1.2
JAVA_HOME=/jdk1.2.2/jre
/j2sdkee1.2/bin/j2ee
```

- Run the Client
  `java HelloClient`

4. *Appendix*

Application Sample Code

File: HelloWorld.java

```
public class HelloWorld {
      public static void main(String[] args) {
            System.out.println("Hello, World");
      }
}
```

HTML Sample Code

File: Hello.html

```
<HTML>
<BODY>
<H1>
Hello, World
</H1>
</BODY>
</HTML>
```

Applet Sample Code

HTML File: helloapplet.html

```
<HTML>
<BODY>
<APPLET CODE=Hello.class WIDTH=300 HEIGHT=100>
<PARAM NAME="name" VALUE="Jay">
<PARAM NAME="size" VALUE="28">
</APPLET>
</BODY>
</HTML>
```

Applet File: Hello.java

```
import java.awt.Label;
import java.awt.Font;
import java.awt.Applet;

public class Hello extends Applet {
      init() {
            String name = getParameter("name");
            String size = getParameter("size");
            Label label = new Label("Hello," + name);
            label.setFont(new Font("dialog", Font.BOLD,
            Integer.parseInt(size)));
            add(label);
      }
}
```

Servlet Sample Code

File: HelloServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet {
      public void doGet(… req, … res) {
            res.setContentType("text/html");
            PrintWriter out = res.getWriter();

            String name = req.getParameter("name");
            if ( name == null )
                  name = "World";

            out.println("<html><body><h1>Hello, " + name);
            out.println("</h1></body></html>");
      }
}
```

JSP Sample Code

File: Hello.jsp

request, response, in, out are built-in variables

```
<HTML>
<BODY>
<%
      String name = request.getParameter("name");
```

```
	if ( name == null )
		name = "World";
	out.println("Hello, "+name);
%>
</BODY>
</HTML>
```

JavaBean Integration Sample Code

File: HelloBean.jsp

```
<jsp:useBean id="helloB" class="HelloBean" scope="session"/>
<html>
<body>
<% helloB.getString() %>
</body>
</html>
```

File: HelloBean.java

```
package hello;
import java.util.*;

public class HelloBean {
	int i=0;
	String[] s = { "so Long, World",
			"Adios, Mundo",
			"auf Wiedersen, Welt" };
	public HelloBean() {
	}

	public String getMessage() {
		return s[(i++)%s.length];
	}
}
```

RMI Sample Code

File: HelloRMI.java

```
import java.rmi.*;
public interface HelloRMI extends Remote {
	public String getSalutation() throws RemoteException;
}
```

File: HelloClient.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.MalformedURLException;
public class HelloClient {
	public static void main(String args[])
	{
		if ( args.length != 1 ) {
			System.out.println("Usage: HelloClient <remote
host>");
			System.exit(1);
		}

		String name = "rmi://" + args[0] + "/HelloRMI";

		try {
```

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

                HelloRMI HelloObj = (HelloRMI)Naming.lookup(name);

                while ( true ) {
                      String results = HelloObj.getSalutation();

                      if (results == null)
                            System.err.println("** not found **");
                      else
                            System.out.println(results);

                      Thread.sleep(1000);
                }

          } catch (NotBoundException ex) {
                System.out.println("There is no object bound to: "+
                      name);
                System.exit(1);

          } catch (MalformedURLException ex) {
                System.out.println("The string: "+ name +
                      " is not a valid rmi url");
                System.exit(1);

          } catch (Throwable ex) {
                System.err.println("exception: " + ex);
                System.exit(1);
          }
    }
}
```

File: HelloServer.java

```
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.Naming;
import java.rmi.server.UnicastRemoteObject;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.net.MalformedURLException;

public class HelloServer extends UnicastRemoteObject implements
HelloRMI {

    int i=0;
    String[] helloString = {
          "Hello, World",
          "Hola, Mundo",
          "Guten Tag, Welt",
          "Bonjour, Monde",
          "Ciao, Mondo"
    };

    public HelloServer() throws RemoteException {
    }
```

```
      public String getSalutation()
      {
            return helloString[(i++)%helloString.length];
      }

      public static void main(String args[])
      {
            try {
                  System.setSecurityManager(new RMISecurityManager());

                  HelloServer server = new HelloServer();
                  String host =
InetAddress.getLocalHost().getHostName();
                  String url = "rmi://" + host + "/HelloRMI";
                  Naming.rebind(url, server);
                  System.err.println("HelloServer ready...");
            }
            catch (Throwable e) {
                  System.err.println("exception: " + e);
                  e.printStackTrace();
                  System.exit(1);
            }
      }
}
```

Corba Sample Code

Interface File: Hello.idl

```
module HelloApp {
      interface Hello {
            string sayHello();
      };
};
```

File: HelloClient.java

```
import HelloApp.*;            // The package containing our stubs.
import org.omg.CosNaming.*;   // HelloClient will use the naming
service.
import org.omg.CORBA.*;       // All CORBA applications need these
classes.


public class HelloClient
{
  public static void main(String args[])
  {
    try{

      // Create and initialize the ORB
      ORB orb = ORB.init(args, null);

      // Get the root naming context
      org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

```
      // Resolve the object reference in naming
      NameComponent nc = new NameComponent("Hello", "");
      NameComponent path[] = {nc};
      Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));

      // Call the Hello server object and print results
      String hello = helloRef.sayHello();
      System.out.println(hello);

    } catch(Exception e) {
        System.out.println("ERROR : " + e);
        e.printStackTrace(System.out);
      }
  }
}
```

File: HelloServer.java

```
// The package containing our stubs.
import HelloApp.*;
// HelloServer will use the naming service.
import org.omg.CosNaming.*;
// The package containing special exceptions thrown by the name
service.
import org.omg.CosNaming.NamingContextPackage.*;b
// All CORBA applications need these classes.
import org.omg.CORBA.*;
public class HelloServer
{
  public static void main(String args[])
  {
    try{

      // Create and initialize the ORB
      ORB orb = ORB.init(args, null);

      // Create the servant and register it with the ORB
      HelloServant helloRef = new HelloServant();
      orb.connect(helloRef);

      // Get the root naming context
      org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(objRef);

      // Bind the object reference in naming
      NameComponent nc = new NameComponent("Hello", "");
      NameComponent path[] = {nc};
      ncRef.rebind(path, helloRef);

      // Wait for invocations from clients
      java.lang.Object sync = new java.lang.Object();

      System.out.println("Waiting for client connection");

      synchronized(sync){
```

```
        sync.wait();
      }

    } catch(Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
      }
  }
}

class HelloServant extends _HelloImplBase
{
  public String sayHello()
  {
    return "\nHello world!!\n";

  }
}
```

Jini Sample Code

File: HelloWorldClient.java

```
// A simple Client to exercise the HelloWorldService

package corejini.chapter5;

import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import java.util.Vector;
import java.io.IOException;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;

public class HelloWorldClient implements Runnable {
	protected ServiceTemplate template;
	protected LookupDiscovery disco;

	// An inner class to implement DiscoveryListener
	class Listener implements DiscoveryListener {
		public void discovered(DiscoveryEvent ev) {
			ServiceRegistrar[] newregs = ev.getRegistrars();
			for(int i=0; i < newregs.length; i++) {
				lookForService(newregs[i]);
			}
		}

		public void discarded(DiscoveryEvent ev) {
		}
	}

	public HelloWorldClient() throws IOException {
		Class[] types = { HelloWorldServiceInterface.class };

		template = new ServiceTemplate(null, types, null);

		// Set a security manager
		if ( System.getSecurityManager() == null) {
			System.setSecurityManager(new RMISecurityManager());
		}

		// Only search the public group
		disco = new LookupDiscovery(new String[] { "" });
```

```
        // Install a listener
        disco.addDiscoveryListener(new Listener());
    }

    // Once we've found a new lookup service, search
    // for proxies that implement HelloWorldServiceInterface
    protected Object lookForService(ServiceRegistrar lusvc) {
        Object o = null;

        try {
            o = lusvc.lookup(template);
        } catch(RemoteException ex) {
            System.err.println("Error doing Lookup: "+
                ex.getMessage());

            return null;
        }

        if ( o == null ) {
            System.err.println("No matching service.");
            return null;
        }

        System.out.println("Got a matching service.");
        for( int ix=0; ix < 50; ix++ ) {
            System.out.println(
                ((HelloWorldServiceInterface)o).getMessage());
        }
        System.out.println("Called Service 50 times.");

        return o;
    }

    // This thread does nothing -- it simply keeps the
    // VM from exiting while we do discovery.
    public void run() {

        while(true) {
            try {
                Thread.sleep(1000000);
            } catch (InterruptedException ex) {
            }
        }
    }

    // Create a HelloWorldClient and start its thread.
    public static void main(String[] args) {
        try {
            HelloWorldClient hwc = new HelloWorldClient();
            new Thread(hwc).start();
        } catch (IOException ex) {
            System.out.println("Couldn't create client: "+
                ex.getMessage());
        }
    }
}
```

File: HelloWorldClientWithEvents.java

- Extends HelloWorldClient and implements inner class MyEventListener() for callback

```
// Extend the client so that it can receive events
// when new services appear.

package corejini.chapter5;

import net.jini.core.lookup.ServiceEvent;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.event.RemoteEvent;
```

```
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.UnknownEventException;
import java.util.Vector;
import java.io.IOException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class HelloWorldClientWithEvents extends HelloWorldClient {

        // 10 minute leases
        protected final int LEASE_TIME = 10 * 60 * 1000;

        // An inner class to listen for events.
        class MyEventListener extends UnicastRemoteObject
                implements RemoteEventListener {
                public MyEventListener() throws RemoteException {
                }

                // Called when an event is received.
                public void notify(RemoteEvent ev)
                        throws RemoteException, UnknownEventException {

                        System.out.println("Got an Event from: " +
                                ev.getSource());

                        if ( ev instanceof ServiceEvent ) {
                                ServiceEvent sev = (ServiceEvent) ev;
                                ServiceItem item = sev.getServiceItem();
                                HelloWorldServiceInterface hws =
                                        (HelloWorldServiceInterface) item.service;

                                System.out.println("Got a matching service.");
                                System.out.println("Its message is: "+
                                        hws.getMessage());
                        }

                        else {
                                System.out.println("Not a service event, "+
                                        "ignoring");
                        }
                }
        }

        protected MyEventListener eventCatcher;

        // Same as superclass, only create an event
        // listener

        public HelloWorldClientWithEvents() throws RemoteException,IOException {
                eventCatcher = new MyEventListener();

        }

        protected Object lookForService(ServiceRegistrar lu) {
                Object o = super.lookForService(lu);

                // Register for Events even if Service has been found.
                //if ( o != null ) {
                //      return o;
                //} else {

                // register even if
                        try {
                                registerForEvents(lu);
                        } catch(RemoteException ex) {
                                System.err.println("Can't solicit events: "+
                                        ex.getMessage());

                                // Discard it, so we can find it again
                                disco.discard(lu);
                        } //finally {
```

```
                                //return null;
                        //}
                //}
                return o;
        }

        // Ask for Events from the lookup service
        protected void registerForEvents(ServiceRegistrar lu)
                throws RemoteException {

                lu.notify(template, ServiceRegistrar.TRANSITION_NOMATCH_MATCH,
                        eventCatcher, null, LEASE_TIME);
        }

        // Start the Client.
        public static void main(String[] args) {
                try {
                        HelloWorldClientWithEvents hwc =
                                new HelloWorldClientWithEvents();
                        new Thread(hwc).start();
                } catch (IOException ex) {
                        System.out.println("Couldn't create client: " +
                                ex.getMessage());
                }
        }
}
```

File: HelloWorldClientWithLeases.java – extends HelloWorldClientWithEvents

```
// Extend the client to renew its event registration leases.
package corejini.chapter5;

import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.EventRegistration;
import net.jini.core.lease.Lease;
import net.jini.core.lease.UnknownLeaseException;
import java.util.Vector;
import java.io.IOException;
import java.rmi.RemoteException;

public class HelloWorldClientWithLeases extends HelloWorldClientWithEvents {
        protected Vector eventRegs = new Vector();
        protected Thread leaseThread = null;

        public HelloWorldClientWithLeases()
                throws RemoteException, IOException {
        }

        // When we register for events, add the event's
        // registration to the set of managed registrations.
        protected void registerForEvents(ServiceRegistrar lu)
                throws RemoteException {
                EventRegistration evreg;

                evreg = lu.notify(template,
                                ServiceRegistrar.TRANSITION_NOMATCH_MATCH,
                                eventCatcher, null, LEASE_TIME);

                eventRegs.addElement(evreg);
                leaseThread.interrupt();
        }

        // run maintains our leases
        public void run() {

                while (true) {
                        try {
                                long sleepTime = computeSleepTime();
                                Thread.sleep(sleepTime);
```

```
                renewLeases();
            } catch (InterruptedException ex) {
            }
        }
    }

    // Figure out how long to sleep.
    protected synchronized long computeSleepTime() {
        long soonestExpiration = Long.MAX_VALUE;
        for (int i=0, size=eventRegs.size(); i < size; i++) {
            Lease l = ((EventRegistration)
                eventRegs.elementAt(i)).getLease();

            if ( l.getExpiration() - (20 * 1000) <
                soonestExpiration) {
                soonestExpiration =
                    l.getExpiration() - (20*1000);
            }
        }

        long now = System.currentTimeMillis();

        if ( now>=soonestExpiration) {
            return 0;
        } else {
            return soonestExpiration - now;
        }
    }

    // Do the Lease renewal work.
    protected synchronized void renewLeases() {
        long now = System.currentTimeMillis();
        Vector deadLeases = new Vector();

        for ( int i=0, size=eventRegs.size(); i<size; i++) {
            Lease l = ((EventRegistration)
                eventRegs.elementAt(i)).getLease();

            if ( now <= l.getExpiration() &&
                now>=l.getExpiration() - (20*1000) ) {
                try {
                    System.out.println("Renewing Lease");
                    l.renew(LEASE_TIME);
                } catch (Exception ex) {
                    System.err.println("Couldn't renew: "+
                        ex.getMessage());

                    deadLeases.addElement(
                        eventRegs.elementAt(i));
                }
            }
        }

        // Clean up after any leases that died.
        for (int i=0, size=deadLeases.size(); i<size; i++) {
            eventRegs.removeElement(deadLeases.elementAt(i));
        }

    }

    // Start the Service
    public static void main(String[] args) {
        try {
            HelloWorldClientWithLeases hwc =
                new HelloWorldClientWithLeases();

            hwc.leaseThread = new Thread(hwc);
            hwc.leaseThread.start();
        } catch (IOException ex) {
            System.out.println("Couldn't create client: " +
                ex.getMessage());
```

```
                }
        }
}
```

File: ServiceWrapper.java

- MyProxy() Implements HelloWorldServiceInterface which is deployed via service web server

```
// a basic wrapper that uses JoinManager

package corejini.chapter8;

import java.io.*;
import net.jini.core.lookup.ServiceID;
import net.jini.core.discovery.LookupLocator;
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.ServiceInfo;
import com.sun.jini.lookup.JoinManager;
import com.sun.jini.lookup.ServiceIDListener;
import java.rmi.RMISecurityManager;

class MyProxy implements Serializable,
        corejini.chapter5.HelloWorldServiceInterface {

        public MyProxy() {}

        private int i=0;
        public String getMessage() {
                String[] s = { "Hasta Lluega, Mundo",
                               "Auf Wiedersen, Welt",
                               "Au Revoir, Monde",
                               "Aloha, World",
                               "Ciao, Mondo" };
                return s[i++%s.length];
        }
}

public class ServiceWrapper implements Runnable {
        protected JoinManager join = null;
        protected File serFile = null;
        protected Object proxy = new MyProxy();

        // Note: STATIC
        static class PersistentData implements Serializable {
                ServiceID serviceID;
                Entry[] attrs;
                String[] groups;
                LookupLocator[] locators;

                public PersistentData() {
                }
        }

        class IDListener implements ServiceIDListener {

                public void serviceIDNotify(ServiceID serviceID) {
                        System.out.println("Got ServiceID " +
                                serviceID);
                        PersistentData state = new PersistentData();
                        state.serviceID = serviceID;
                        state.attrs = join.getAttributes();
                        state.groups = join.getGroups();
                        state.locators = join.getLocators();

                        try {
                                writeState(state);
                        } catch (IOException ex) {
                                System.err.println("Couldn't write: "+
                                        ex.getMessage());
                                ex.printStackTrace();
                                join.terminate();
```

```
                        System.exit(1);
                }
        }
}

public ServiceWrapper(File serFile, boolean firsttime)
        throws IOException, ClassNotFoundException {
        this.serFile = serFile;

        if ( System.getSecurityManager() == null ) {
                System.setSecurityManager(new RMISecurityManager());
        }

        if ( firsttime)
                register();
        else
                reregister();

}

public void run() {
        while (true) {
                try {
                        Thread.sleep(Long.MAX_VALUE);
                } catch (InterruptedException ex) {
                }
        }
}

protected void register() throws IOException {
        if ( join != null ) {
                throw new IllegalStateException(
                        "Wrapper already started.");
        }
        System.out.println("Starting ...");

        // Set attributes
        Entry[] attrTemplate = new Entry[1];
        attrTemplate[0] = new ServiceInfo("HelloWorld", "JRD",
                        "Jay", "1.0", "One", "1");
        join = new JoinManager(proxy, attrTemplate,
                        new IDListener(), null);
}

protected void reregister() throws
        IOException, ClassNotFoundException {
        if ( join != null ) {
                throw new IllegalStateException(
                        "Wrapper already started.");
        }

        PersistentData state = readState();
        System.out.println("Restarting: old ID is " +
                state.serviceID);

        join = new JoinManager(state.serviceID,
                        proxy, state.attrs,
                        state.groups,
                        state.locators, null);
}

protected void writeState(PersistentData state)
        throws IOException {

        ObjectOutputStream out =
                new ObjectOutputStream(new FileOutputStream(serFile));

        out.writeObject(state);
        out.flush();
        out.close();
}
```

```
        protected PersistentData readState()
                throws IOException, ClassNotFoundException {

                ObjectInputStream in =
                        new ObjectInputStream(new FileInputStream(serFile));

                PersistentData state = (PersistentData)in.readObject();

                in.close();
                return state;
        }

        static void usage() {
                System.out.println("Usage: ServiceWrapper "+
                        "[-f] serialization_file");
                System.exit(1);
        }

        public static void main(String[] args) {

                boolean firsttime = false;
                String serFileName = null;
                File serFile = null;

                if (args.length < 1 || args.length > 2) {
                        usage();
                }

                if ( args.length == 2 ) {
                        if ( args[0].equals("-f")) {
                                firsttime = true;
                                serFileName = args[1];
                        }

                        else {
                                usage();
                        }
                } else {
                        serFileName = args[0];
                }

                serFile = new File(serFileName);

                try {
                        ServiceWrapper wrapper =
                                new ServiceWrapper(serFile, firsttime);
                        new Thread(wrapper).start();
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
        }
}
```

Enterprise JavaBeans Sample Code

File `HelloEJB.java`

- A stateless session bean

```
import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class HelloEJB implements SessionBean {

   public String getMessage() {

      return "Hello, EJB!";
   }

   public HelloEJB() {}
```

```
   public void ejbCreate() {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}
   public void setSessionContext(SessionContext sc) {}

} // HelloEJB
```

File `HelloHome.java`

- EJB Home Interface – Client finds and creates EJB through this interface

```
import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface HelloHome extends EJBHome {

    Hello create() throws RemoteException, CreateException;
}
```

File `Hello.java`

- EJB Remote Interface – EJB Container returns reference to this interface to EJB object

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Hello extends EJBObject {

   public String getMessage() throws RemoteException;
}
```

5. References

Web Sites of Interest:

- http://java.sun.com, Sun Microsystems, Inc. Current information on Java and related topics, including Java releases, security issues, and online documentation.
- http://java.sun.com/docs/books/tutorial, Sun Microsystems, Inc. Online tutorial of Java APIs. For virtually every Java API available for download from java.sun.com there exists a corresponding Java tutorial with implementation guidelines and code samples.
- http://java/sun/com/j2ee, Java2 Enterprise Edition Tutorial, Sun Microsystems, Inc. Online tutorial covering the Java 2 Enterprise Edition is contained in the J2SDKEE documentation download and can be downloaded with the Java 2 Software Developers Kit Enterprise Edition.

Java Reference Manuals

The Java Programming Language, Second Edition, Arnold, Ken and Gosling, James, Addison-Wesley, 1998.

Java in a Nutshell, Flanagan, Timothy, O'Reilly and Associates, 1997

Advanced Techniques for Java Developers, Berg, Daniel and Fritzinger, J. Steven, John Wiley & Sons, Inc. 1999

Java Servlet Programming, Hunter, Jason, O'Reilly and Associates, Inc., 1998

Core Jini, Edwards, W. Keith, Sun Microsystems Press, 1999.