Distributed Java: RMI Tutorial

Ian F. Darwin

Notice

RMI is rather a legacy technology today, having been substantially displaced by technologies like RESTful Web Services. This material was originally the RMI Chapter in my O'Reilly Java Cookbook, so it is in the standard *Problem/Solution/Discussion* format of the Cookbook series. It has been lightly updated from time to time since. The chapter was removed from the Third Edition of the book, and is reprinted here by permission of O'Reilly Media. There are several links that do not work as a result. The formatting of some of the code examples may be failsome due to tab expansion.

Please consider this as a snapshot of technology (and my coverage of it) from fairly early on in the life of Java.

The source code examples can be downloaded from https://github.com/IanDarwin/javasrcee, in the *src/main/java/rmi* subdirectory.

Introduction

A *distributed system* is a program or set of programs that runs on more than one computing resource. Distributed computing covers a wide spectrum, from intraprocess distributed applications (which Java calls threaded applications, discussed in [javacook-threads]), through intrasystem applications (such as a network client and server on the same machine), to applications where a client program and a server program run on machines far apart (such as a web application).

Distributed computing was around long before Java. Some traditional distributed mechanisms include RPC (remote procedure call) and CORBA. Java adds RMI (Remote Method Invocation), its own CORBA support, and EJB (Enterprise JavaBeans) to the mix. This chapter covers only RMI in detail, but these other technologies are discussed briefly.

At its simplest level, *remote procedure call* is the ability to run code on another machine and have it behave as much as possible like a local method call. Most versions of Unix use remote procedure calls extensively: Sun/Oracle NFS, YP/NIS, and NIS+ are all built on top of their RPC mechanism. Another RCP implementation for UNIX is called DCE; Microsoft Windows implements large parts of the Unix DCE Remote Procedure Call specification and can interoperate with it. Each of these defines its own slightly ad hoc method of specifying the interface to the remote call. Sun/Oracle RPC uses a program called *rpcgen*, which reads a protocol specification and writes both the client and server network code. These are both Unix-specific; they have their place but aren't as portable as Java.

Java *Remote Method Invocation* (RMI) is a type of remote procedure call that is network-independent, lightweight, and totally portable, as it's written in pure Java. I discuss RMI in this chapter in enough detail to get you started.

CORBA is the Object Management Group's (OMG) Common Object Request Broker Architecture, a sort of remote procedure call for programs written in C, C++, Java, Ada, Smalltalk, and others to call methods in objects written in any of those languages. It provides a transport service called

the Internet Inter-Orb Protocol (IIOP) that allows object implementations from different vendors to interoperate. One version of RMI runs over IIOP, making it possible to claim that RMI is CORBA-compliant.

Both RMI and CORBA should really be called "remote method calls," as they both emphasize remote *objects*.

Enterprise JavaBeans (EJB) is a distributed object mechanism used primarily for building reusable distributed objects that provide business logic and often database storage. There are several types of EJBs, but the ones we're concerned with can be accessed both locally (e.g, inside the same JVM from a Web Tier component or another EJB) and remotely. As you might expect, the remote access is over RMI!

CORBA and EJB are of interest primarily to enterprise developers; they are covered in O'Reilly's *Java EE 7 Essentials: Enterprise Developer Handbook* (see https://shop.oreilly.com/product/0636920030614.do). You can read more about EJB in the book *Enterprise JavaBeans* https://shop.oreilly.com/product/9780596158033.do.

To use RMI:

1. Define (or locate) the remote interface in agreement with the server.
2. Write your server.
3. (Optional) Run *rmic* (Java RMI stub compiler) to generate the network glue.
4. Write the client.
5. Ensure that the RMI registry is running.
6. Start the server.
7. Run one or more clients.

Defining the RMI Contract

Problem

You want to define the communications exchange between client and server.

Solution

Define a Java interface.

Discussion

The contract between an RMI server and its clients is defined using a standard Java mechanism: interfaces. An interface is similar to an abstract class, but a class can implement more than one interface. RMI remote interfaces must be subinterfaces of java.rmi.Remote. All parameters and return values must be either primitives (int, double, etc.), or implement Serializable (as do most of the standard types like String). Or, as we'll see in Program: RMI Callbacks, they can also be Remote - a Remote can pass another Remote across the RMI connection.

RMI overview shows the relationships between the important classes involved in an RMI implementation. The developer need only write the interface and two classes, the client application and the server object implementation. The RMI stub or proxy and the RMI skeleton

or adapter are generated for you by the RMI framework (see Creating an RMI Server), while the RMI Registry and other RMI classes at the bottom of the figure are provided as part of RMI itself.

RemoteDate.java is a simple RemoteDate getter interface, which lets us find out the date and time on a remote machine.

RemoteDate.java

```
package rmi;

import java.time.LocalDateTime;

// tag::main[]
/** A statement of what the client & server must agree upon. */
public interface RemoteDate extends java.rmi.Remote {

        /** The method used to get the current date on the remote */
        public LocalDateTime getRemoteDate() throws java.rmi.RemoteException;

        /** The name used in the RMI registry service. */
        public final static String LOOKUPNAME = "RemoteDate";
}
// end::main[]
```

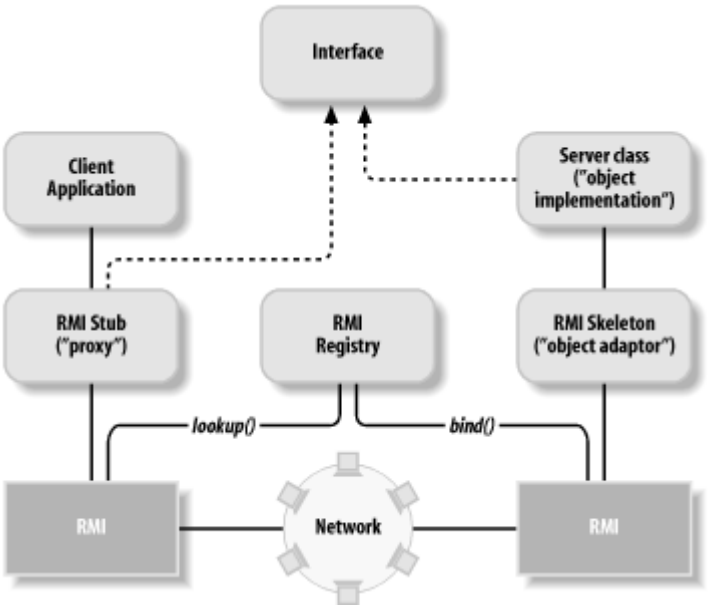

Figure 1. RMI overview

This file must list all the methods that will be callable from the server by the client. The lookup name is an arbitrary name that is registered by the server and looked up by the client to establish communications between the two processes (when looked up by the client it will normally be part of an *rmi*: URL). While most authors just hardcode this string in both programs, I find this error-prone, so I usually include the lookup name in the interface.

"So interfaces can contain variables?" you ask. No variables indeed, but interfaces may contain nonvariable (final) fields such as the field LOOKUPNAME in RemoteDate.java. Putting the lookup name here ensures that both server and client really agree, and that is what this interface is all about, after all. I've seen other developers waste a considerable amount of time tracking down spelling mistakes in the lookup names of various remote services, so I prefer doing it this way.

Creating an RMI Client

Problem

You want to write a client to use an RMI service.

Solution

Locate the object via JNDI and call its methods.

Discussion

Assume for now that the server object is running remotely. To locate it, you have to use the Java Naming and Directory Interface (JNDI, package javax.naming). JNDI ise a general-purpose binding and lookup mechanism that is supported both by Java SE and by Java EE. Here we only use the client parts of it to look up the server, which we'll actually create in Creating an RMI Server.

Use the JNDI static method Naming.lookup( ), passing in the lookup name. This gives you a reference to a *proxy object*, an object that, like the real server object, implements the remote interface but runs in the same Java Virtual Machine as your client application. Here we see the beauty of interfaces: the proxy object implements the interface so that your code can use it just as it would use a local object providing the given service. And the remote object also implements the interface so that the proxy object's remote counterpart can use it exactly as the proxy is used. DateClient.java shows the client for the RemoteDate service.

DateClient.java

```
package rmi;

import java.rmi.Naming;
import java.time.LocalDateTime;

// tag::main[]
/* A very simple client for the RemoteDate service. */
public class DateClient {

     /** The local proxy for the service. */
     protected static RemoteDate netConn = null;

     public static void main(String[] args) {
          try {
               System.out.println("DateClient starting lookup...");
               netConn = (RemoteDate)Naming.lookup(RemoteDate.LOOKUPNAME);
               LocalDateTime today = netConn.getRemoteDate();
               System.out.println("DateClient received date: " + today.toString());
// Could use a DateFormat...
          } catch (Exception e) {
               System.err.println("DateClient: RemoteDate exception: " +
e.getMessage());
               e.printStackTrace();
          }
     }
}
// end::main[]
```

Creating an RMI Server

Problem

The client looks good on paper, but it will be lonely without a server to talk to.

Solution

You need to write two parts for the server, an implementation class and a main method. These can sometimes be in the same class but are usually separated for clarity.

Discussion

To implement an RMI server:

1. Define (or locate) the remote interface in agreement with the client.
2. Define the constructor for the remote object.
3. Provide implementations for the methods that can be invoked remotely.
4. Create and install a security manager.
5. Create one or more instances of a remote object.
6. Register at least one of the remote objects with the RMI remote object registry.

This implementation divides the server into the traditional two parts—a main program and an implementation class. It is just as feasible to combine these in a single class. The main program shown in DateServer.java simply constructs an instance of the implementation and registers it with the lookup service.

DateServer.java

```
package rmi;

import java.rmi.Naming;

// tag::main[]
public class DateServer {
     public static void main(String[] args) {
          System.out.println("DateServer starting...");

          // You probably need a SecurityManager for downloading of classes:
          // System.setSecurityManager(new SecurityManager());

          System.out.println("Security Manager installed; setting up service");

          try {
               // Create an instance of the server object
               RemoteDateImpl im = new RemoteDateImpl();

               System.out.println("Publishing DateServer...");

               // Publish it in the RMI registry.
               // Of course you have to have rmiregistry or equivalent!
               Naming.rebind(RemoteDate.LOOKUPNAME, im);

               System.out.println("DateServer ready.");
          } catch (Exception e) {
               System.err.println(e);
               System.exit(1);
          }
     }
}
// end::main[]
```

The Naming.bind( ) method creates an association between the lookup name and the instance of the server object. This method fails if the server already has an instance of the given name, requiring you to call rebind( ) to overwrite it. But since that's exactly where you'll find yourself if the server crashes (or you kill it while debugging) and you restart it, most people just use rebind() all the time.

The implementation class must implement the given remote interface

RemoteDateImpl.java

```
package rmi;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.time.LocalDateTime;

// tag::main[]
public class RemoteDateImpl extends UnicastRemoteObject implements RemoteDate {

     /** Construct the object that implements the remote server.
      * Called from main, after it has the SecurityManager in place.
      */
     public RemoteDateImpl() throws RemoteException {
          super();  // sets up networking
     }

     /** The remote method that "does all the work". This won't get
      * called until the client starts up.
      */
     public LocalDateTime getRemoteDate() throws RemoteException {
          return LocalDateTime.now();
     }
}
// end::main[]
```

Using the server

Once you've compiled the implementation class, you can optionally run *rmic* (the RMI compiler) to build some glue files. You then install the class files (for the interface, the stub, and any data classes) into the client's CLASSPATH:

```
$ javac -d . RemoteDateImpl.java
$ ls darwinsys/distdate
DateApplet$1.class          DateClient.class            RemoteDate.class
DateApplet.class            DateServer.class            RemoteDateImpl.class
$ rmic -d . darwinsys.distdate.RemoteDateImpl
$ ls darwinsys/distdate
DateApplet$1.class          DateServer.class         RemoteDateImpl_Skel.class
DateApplet.class            RemoteDate.class         RemoteDateImpl_Stub.class
DateClient.class            RemoteDateImpl.class
$
```

If you don't run rmic, the RMI system will generate those helper classes dynamically, so most people don't bother these days.

You can then start the RMI registry program. If you're doing this by hand, just type the command **`rmiregistry`** in a separate window or start it in the background on systems that support this. For now, be sure to start it in the same directory as the generated classes.

Deploying RMI Across a Network

Problem

As shown so far, the server and the client must be on the same machine—not much of a distributed system!

Solution

Get the RMI registry to dish out the client stubs on demand.

Discussion

RMI does not provide true *location transparency*, which means that you must at some point know the network name of the machine the server is running on. The server machine must be running the RMI registry program as well. There's no need for the RMI registry to be running on the client.

The RMI registry needs to send the client stubs to the client. The easiest way to do this is to install the class files into a web server and provide their HTTP(S) URL to the RMI server's startup by defining it in the system properties:

```
java -Djava.rmi.server.codebase=https://serverhost/stubsdir/ ServerMain
```

In this example, serverhost is the TCP/IP network name of the host where the RMI server and registry are running, and stubsdir is some directory relative to the web server from which the stub files can be downloaded.

In this version, be careful to start the RMI registry in its own directory, i.e., **not** in the directory where you are have the RMI interface and impl classes. If RMI can find the stubs in its own CLASSPATH, it assumes they are universally available and won't download them! That's why in the previous recipe we said to start the registry where the classes are; that won't help us here.

Obviously the HTTP URL must be resolvable on and reachable from all machines that want to access the service.

The only other thing to do is to change the client's view of the RMI lookup name to something like *rmi://serverhost/foo_bar_name* And for security reasons, the installation of the RMI Security Manager, which was optional before, is now a requirement.

Program: RMI Callbacks

One major benefit of RMI is that almost any kind of object can be passed as a parameter or return value of a remote method. The recipient of the object will not know ahead of time the class of the actual object it will receive. If the object is of a class that implements Remote (java.rmi.Remote), the returned object will in fact be a proxy object that implements at least the declared interface. If the object is not remote, it must be serializable, and a copy of it is transmitted across the Net. The prime example of this is a String. It makes no sense to write an RMI proxy object for String. Why? Remember from [javacook-strings] that String objects are immutable! Once you have a String, you can copy it locally but never change it. So Strings, like most other core classes, can be copied across the RMI connection just as easily as they are copied locally. But Remote objects cause an RMI proxy to be delivered. So what stops the caller from passing an RMI object that is also itself a proxy? Nothing at all, and this is the basis of the powerful RMI callback mechanism.

An RMI callback occurs when the client of one service passes an object that is the proxy for another service. The recipient can then call methods in the object it received and be calling back

(hence the name) to where it came from. Think about a stock ticker service. You write a server that runs on your desktop and notifies you when your stock moves up or down. This server is also a remote object. You then pass this server object to the stock ticker service, which remembers it and calls its methods when the stock price changes. See RMI callback service for the big picture.

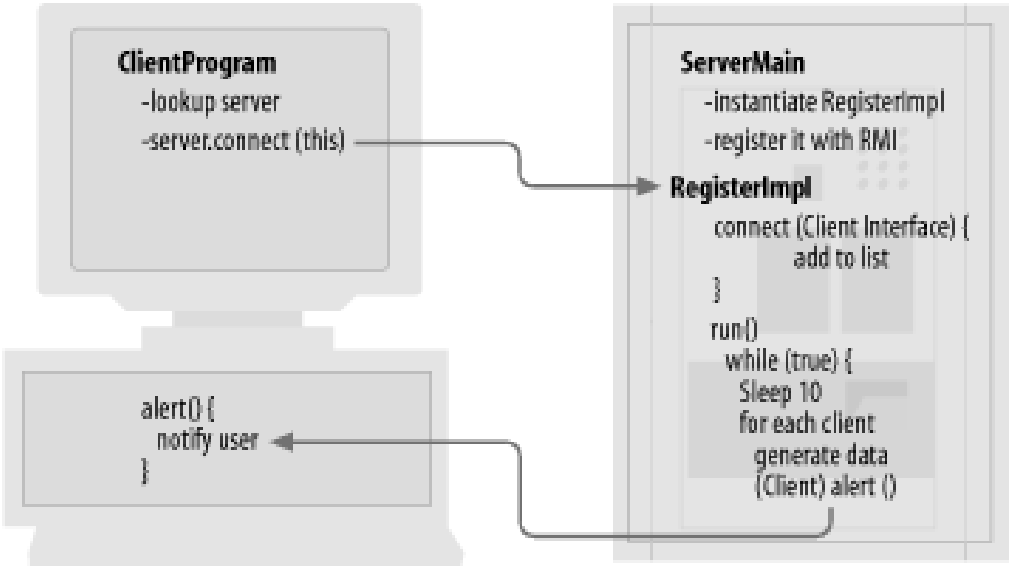

Figure 2. RMI callback service

The code for the callback service comes in several parts. Because there are two servers, there are also two interfaces. The first is the interface for the TickerServer service. There is only one method, connect( ), which takes one argument, a Client:

```
package rmi.callback;
import java.rmi.Remote;
import java.rmi.RemoteException;

// tag::main[]
public interface TickerServer extends Remote {
     public static final String LOOKUP_NAME = "Ticker_Service";
     public void connect(Client d) throws RemoteException;
}
// end::main[]
```

Client is the interface that displays a stock price change message on your desktop. It also has only one method, alert( ), which takes a String argument:

Client.java

```
package rmi.callback;
import java.rmi.Remote;
import java.rmi.RemoteException;

/** Client -- the interface for the client callback */
public interface Client extends Remote {
     public void alert(String mesg) throws RemoteException;
}
```

Now that you've seen both interfaces, let's look at the TickerServer implementation (TickerServerImpl.java). Its constructor starts a background thread to "track" stock prices; in fact, this implementation just calls a random number generator. A real implementation might use a third RMI service to track actual stock data. The connect( ) method is trivial; it just adds the given client (which is really an RMI proxy for the client server running on your desktop). The run method runs forever; on each iteration, after sleeping for a while, it picks a random stock movement and reports it to any and all registered clients. If there's an error on a given client, the client is removed from the list.

TickerServerImpl.java

```
package rmi.callback;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Random;

// tag::main[]
/** This is the main class of the server */
public class TickerServerImpl extends UnicastRemoteObject implements TickerServer,
Runnable
{
     private static final long serialVersionUID = -4641962773626590008L;
     List<Client> list = new ArrayList<Client>();

     /** Construct the object that implements the remote server.
      * Called from main, after it has the SecurityManager in place.
      */
     public TickerServerImpl() throws RemoteException {
          super(); // sets up networking
     }

     /** Start background thread to track stocks :-) and alert users. */
     public void start() {
          new Thread(this).start();
     }

     /** The remote method that "does all the work". This won't get
      * called until the client starts up.
      */
     public void connect(Client da) throws RemoteException {
          System.out.println("Adding client " + da);
          list.add(da);
     }

     boolean done = false;
     Random rand = new Random();

     public void run() {
          while (!done) {
               try {
                    Thread.sleep(10 * 1000);
                    System.out.println("Tick");
               } catch (InterruptedException unexpected) {
                    System.out.println("WAHHH!");
                    done = true;
               }
               Iterator it = list.iterator();
               while (it.hasNext()){
                    String mesg = ("Your stock price went " +
                         (rand.nextFloat() > 0.5 ? "up" : "down") + "!");
                    // Send the alert to the given user.
                    // If this fails, remove them from the list
                    try {
                         ((Client)it.next()).alert(mesg);
                    } catch (RemoteException re) {
                     System.out.println(
                              "Exception alerting client, removing it.");
                         System.out.println(re);
                         it.remove();
                    }
               }
          }
     }
} // end::main[]
```

As written, this code is not threadsafe; things might go bad if one client connects while we are running through the list of clients. How to fix this is discussed in the Threads chapter of the Java Cookbook.

This program's "server main" is trivial, so I don't include it here; it just creates an instance of the class we just saw and registers it. More interesting is the client application shown in Callback ClientProgram.java, which is both the RMI client to the connect() method and the RMI server to the alert() method in the server in TickerServerImpl.java.

Callback ClientProgram.java

```
package rmi.callback;
import java.io.IOException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

/** This class tries to be all things to all people:
 *   - main program for client to run.
 *   - "server" program for remote to use Client of
 */
public class ClientProgram extends UnicastRemoteObject implements Client
{
     protected final static String host = "localhost";

     /** No-argument constructor required as we are a Remote Object */
     public ClientProgram() throws RemoteException {
     }

     /** This is the main program, just to get things started. */
     public static void main(String[] argv) throws IOException, NotBoundException {
          new ClientProgram().do_the_work();
     }

     /** This is the server program part */
     private void do_the_work() throws IOException, NotBoundException {

          System.out.println("Client starting");

          // First, register us with the RMI registry
          // Naming.rebind("Client", this);

          // Now, find the server, and register with it
          System.out.println("Finding server");
          TickerServer server =
               (TickerServer)Naming.lookup("rmi://" + host + "/" +
               TickerServer.LOOKUP_NAME);

          // This should cause the server to call us back.
          System.out.println("Connecting to server");
          server.connect(this);

          System.out.println("Client program ready.");
     }

     /** This is the client callback */
     public void alert(String message) throws RemoteException {
          System.out.println(message);
     }
}
```

In this version, the client server alert() method simply prints the message in its console window. A more realistic version would receive an object containing the stock symbol, a timestamp, and the current price and relative price change; it could then consult a GUI control to decide whether the given price movement is considered noticeable and pop up a JOptionPane if so.

Program: NetWatch

Here's a program I put together while teaching Java courses for Learning Tree (https://www.learningtree.com). In one exercise, each student starts the RMI registry on his or her machine and uses Naming.rebind( ) (as in Creating an RMI Server) to register with it. Some students come up with interesting variations on the theme of registering. This program contacts the RMI registry on each of a batch of machines and shows the instructor graphically which machines have RMI running and what is registered. A red flag shows machines that don't even have the registry program running; a black flag shows machines that are dead to the (networked) world.

This program also uses many ideas from elsewhere in the book. A Swing GUI ([javacook-gui]) is used. The layout is a GridLayout (discussed briefly in [javacook-gui-SECT-2]). A default list of machines to watch is loaded from a Properties object ([javacook-structure-SECT-7]). For each host, an RMIPanel is constructed. This class is both a JComponent ([javacook-gui-SECT-13]) and a thread ([javacook-threads]). As a JComponent, it can be run in a panel; as a thread, it can run independently and then sleep for 30 seconds (by default; settable in the properties file) so that it isn't continually hammering away at the RMI registry on all the machines (the network traffic could be awesome). This program combines all these elements and comes out looking like the display in NetWatch watching the class (alas, we didn't have color pages in earlier editions of this book, so somebody converted the illustrations to grayscale).

Figure 3. NetWatch watching the class

NetWatch.java is the main class, NetWatch, which creates the JFrame and all the RMIPanels and puts them together.

NetWatch.java

```
package netwatch;

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

import javax.swing.JOptionPane;

/** Main program for the NetWatch program: watch the network status
```

```
 * on a bunch of machines (i.e., in a classroom or lab). Currently only
 * for RMI, but can be extended for TCP socket, CORBA ORB, etc.*/
// tag::main[]
public class NetWatch {
     public static void main(String[] argv) {
          Properties p = null;
          NetFrame f = new NetFrame("Network Watcher", p);

          try {
               FileInputStream is = new FileInputStream("NetWatch.properties");
               p = new Properties();
               p.load(is);
               is.close();
          } catch (IOException e) {
               JOptionPane.showMessageDialog(f,
                    e.toString(), "Properties error",
                    JOptionPane.ERROR_MESSAGE);
          }

          // NOW CONSTRUCT PANELS, ONE FOR EACH HOST.

          // If arguments, use them as hostnames.
          if (argv.length!=0) {
               for (int i=0; i<argv.length; i++) {
                    f.addHost(argv[i], p);
               }
          // No arguments. Can we use properties?
          } else if (p != null && p.size() > 0) {
               String net = p.getProperty("netwatch.net");
               int start = Integer.parseInt(p.getProperty("netwatch.start"));
               int end = Integer.parseInt(p.getProperty("netwatch.end"));
               for (int i=start; i<=end; i++) {
                    f.addHost(net + "." + i, p);
               }
               for (int i=0; ; i++) {
                    String nextHost = p.getProperty("nethost" + i);
                    if (nextHost == null)
                         break;
                    f.addHost(nextHost, p);
               }
          }
          // None of the above. Fall back to localhost
          else {
               f.addHost("localhost", p);
          }

          // All done. Pack the Frame and show it.
          f.pack();
          // UtilGUI.centre(f);
          f.setVisible(true);
          f.addWindowListener(new WindowAdapter() {
               public void windowClosing(WindowEvent e) {
                    System.exit(0);
               }
          });
     }
}
// end::main[]
```

The per-machine class, RMIPanel, is shown in RMIPanel.java. This class is instantiated once for each machine being monitored. Its run method loops, getting the list of registered objects from the given machine's RMI registry, and checks the contents to see if the expected string is present, setting the state to one of several integer values defined in the parent class NetPanel (EMPTY, DUBIOUS, FINE, etc.) based on what it finds. This state value is used to decide what color to paint this particular RMIPanel in the setState( ) method of the parent class NetPanel, which we have no reason to override.

RMIPanel.java

```
package netwatch;
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.util.Properties;

/** Displays one machines status, for RMI*/
// tag::main[]
public class RMIPanel extends NetPanel implements Runnable {
     private static final long serialVersionUID = 1L;

     public RMIPanel(String host, Properties p) {
          super(host, p);
     }

     /** Keep the screen updated forever, unless stop()ped. */
     public void run() {
          String thePort = props.getProperty("rmiwatch.port", "");
          String theURL = "rmi://" + hostName + ":" + thePort;
          while (!done) {
               try {
                    String[] names = Naming.list(theURL);
                    ta.setText("");
                    for (int i=0; i<names.length; i++) {
                         ta.append(i + ": " + names[i] + "\n");
                    }
                    // If we didnt get an exception, host is up.
                    String expect = props.getProperty("rmiwatch.expect");
                    String fullText = ta.getText();
                    if (fullText.length() == 0) {
                         ta.setText("(nothing registered!)");
                         setState(EMPTY);
                    } else if (expect != null && fullText.indexOf(expect)==-1) {
                         setState(DUBIOUS);
                    } else setState(FINE);
               } catch (java.rmi.ConnectIOException e) {
                    setState(DOWN);
                    ta.setText("Net error: " + e.detail.getClass());
               } catch (java.rmi.ConnectException e) {
                    setState(NOREG);
                    ta.setText("RMI error: " + e.getClass().getName() + "\n" +
                         "  " + e.detail.getClass());
                    // System.err.println(hostName + ":" + e);
               } catch (RemoteException e) {
                    setState(NOREG);
                    ta.setText("RMI error: " + e.getClass().getName() + "\n" +
                         "  " + e.detail.getClass());
               } catch (MalformedURLException e) {
                    setState(DOWN);
                    ta.setText("Invalid host: " + e.toString());
               } finally {
                    // sleep() in "finally" so common "down" states dont bypass.
                    // Randomize time so we dont make net load bursty.
                    try {
                         Thread.sleep((int)(sleepTime * MSEC * 2 * Math.random()));
                    } catch (InterruptedException e) {
                         /*CANTHAPPEN*/
                    }
               }
          }
     }
}
// end::main[]
```

The last part is NetPanel, shown in NetPanel.java. Notice the state variable definitions, and the setState() method that calls setBackground() to set the correct color given the state.

NetPanel.java

```
package netwatch;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Font;
import java.util.Properties;
import javax.swing.BorderFactory;
import javax.swing.JPanel;
import javax.swing.JTextArea;

/**
 * Displays one machine's status. Part of the NetWatch program: watch the network
 * on a bunch of machines (i.e., in a classroom or lab). <P>Each non-abstract
 * subclass just needs to implement run(), which must, in a while (!done) loop:
 * <UL><LI>Try to contact the host<LI>call setState(); (argument below)
 * <LI>call ta.setText();<LI>Thread.sleep(sleepTime * MSEC);</UL>
 * <P>The argument to setState() must be one of:
 * <UL><LI>FINE == Server has "expect"ed name registered.
 * <LI>DUBIOUS == Server does not have expected name registered.
 * <LI>EMPTY == Server has nothing registered.
 * <LI>NOREG == host is up but not running RMI
 * <LI>DOWN == host unreachable, not responding, ECONN, etc.</UL>
 */
// tag::main[]
public abstract class NetPanel extends JPanel implements Runnable {

    private static final long serialVersionUID = 1L;
    /** The name of this host */
    protected String hostName;
    /** The text area to display a list of stuff */
    protected JTextArea ta;
    /** Properties, passed in to constructor */
    protected Properties props;
    /** Default sleep time, in seconds. */
    protected static int DEFAULT_SLEEP = 30;
    /** Sleep time, in seconds. */
    protected int sleepTime = DEFAULT_SLEEP;
    /** Conversion */
    protected int MSEC = 1000;
    /** The constant-width font, shared by all instances. */
    protected static Font cwFont;
    /** The states */
    /** The state for: has "expect"ed name registered. */
    protected final static int FINE = 1;
    /** The state for: does not have expected name registered. */
    protected final static int DUBIOUS = 2;
    /** The state for: Server has nothing registered. */
    protected final static int EMPTY = 3;
    /** The state for: host is up but not running RMI */
    protected final static int NOREG = 4;
    /** The state for: host unreachable, not responding, ECONN, etc. */
    protected final static int DOWN = 5;
    /** The color for when a machine is FINE */
    protected static final Color COLOR_FINE = Color.green;
    /** The color for when a machine is DUBIOUS */
    protected static final Color COLOR_DUBIOUS = Color.yellow;
    /** The color for when a machine is EMPTY */
    protected static final Color COLOR_EMPTY = Color.white;
    /** The color for when a machine has NOREG */
    protected static final Color COLOR_NOREG = Color.red;
    /** The color for when a machine is NOREG */
    protected static final Color COLOR_DOWN = Color.black;
```

```
    /** State of the monitored hosts RMI registry, up or down.
     * Initially set 0, which isnt one of the named states, to
     * force the background color to be set on the first transition.*/
    protected int state = 0;

    public NetPanel(String host, Properties p) {
        hostName = host;
        props = p;
        String s = props.getProperty("rmiwatch.sleep");
        if (s != null)
            sleepTime = Integer.parseInt(s);
        // System.out.println("Sleep time now " + sleepTime);

        // Maybe get font name and size from props?
        if (cwFont == null)
            cwFont = new Font("lucidasansTypewriter", Font.PLAIN, 10);

        // Gooey gooey stuff.
        ta = new JTextArea(2, 26);
        ta.setEditable(false);
        ta.setFont(cwFont);
        add(BorderLayout.CENTER, ta);
        setBorder(BorderFactory.createTitledBorder(hostName));

        // Sparks. Ignition!
        new Thread(this).start();
    }

    boolean done = false;
    /** Stop this Thread */
    public void stop() {
        done = true;
    }

    /** Record the new state of the current machine.
     * If this machine has changed state, set its color
     * @param newState - one of the five valid states in the introduction.
     */
    protected void setState(int newState) {
        if (state /*already*/ == newState)
            return;       // nothing to do.
        switch(newState) {
            case FINE:         // Server has "expect"ed name registered.
                ta.setBackground(COLOR_FINE);
                ta.setForeground(Color.black);
                break;
            case DUBIOUS: // Server does not have expected name registered.
                ta.setBackground(COLOR_DUBIOUS);
                ta.setForeground(Color.black);
                break;
            case EMPTY:        // Server has nothing registered.
                ta.setBackground(COLOR_EMPTY);
                ta.setForeground(Color.black);
                break;
            case NOREG:        // host is up but not running RMI
                ta.setBackground(COLOR_NOREG);
                ta.setForeground(Color.white);
                break;
            case DOWN:         // host unreachable, not responding, ECONN, etc.
                ta.setBackground(COLOR_DOWN);
                ta.setForeground(Color.white);
                break;
            default:
                throw new IllegalStateException("setState("+state+") invalid");
            }
        state = newState;
    }
}
// end::main[]
```

See Also

The term *distributed computing* covers a lot of terrain. Here I've shown only the basics of RMI. For more on RMI, see the Oracle Tutorial.

See especially the Oracle information on security and permissions.

It is possible to use RMI to access CORBA objects, or vice versa, using a mechanism called RMI-IIOP. See https://docs.oracle.com/javase/7/docs/technotes/guides/rmi-iiop/.

Potentially the most important distributed mechanism for large-scale computing projects is Enterprise JavaBeans, part of the Java Enterprise Edition. See the O'Reilly book *Java EE 7 Essentials*.

Since almost every web interaction involves at least two computers, you can also think of servlets and JSPs/JSF as a kind of distributed computing, used primarily as the gateway into these other distributed object mechanisms.

Last updated 2020-03-11 09:25:33 -0400