CHAPTER 17

LEGO MINDSTORMS

LEGO MINDSTORMS IS A "ROBOTICS INVENTION SYSTEM" that allows you to build LEGO toys with a programmable computer. This chapter looks at the issues involved in interfacing with a specialized hardware device, using MINDSTORMS as an example.

Making Hardware into Jini Services

Hardware devices and preexisting software applications can equally be turned into Jini services. A legacy piece of software can have a "wrapper" placed around it, and this wrapper can act as a Jini service. Remote method calls into this service can then make calls into the application. Hardware devices are a little more complex because they are defined at a lower level, and often have resource constraints that do not apply to software.

There are two major categories of hardware services: those that can run a Java virtual machine, and those that do not have enough memory or an adequate processor. For example, an 8086 with 20-bit addressing and only 1 MB of addressable memory would not be an adequate processor, while the owner of a Palm handheld might not wish to squander too many of its limited resources running a JVM. Devices capable of running a JVM may be further subdivided into those that are capable of running a standard JDK 1.2 JVM and core libraries, and those that have to run some stripped-down version. At the time of writing, the lightweight JVM under development by Sun Microsystems called KVM does not support the features of JDK 1.2 required to run Jini.

Jini does not require all the core Java classes to run a service. For example, for a service that engages in discovery and registration does not require the AWT. However, it does require support for the newer RMI features found in JDK 1.2, and it does require enough of the standard language features. Again, this is not inclusive of all parts of Java; for example, floating point numbers are not required. Because many of the current embedded or small JVMs have removed features and standard core libraries, at present none of them have enough support for JDK 1.2 features to run Jini.

The current developments for embedded or small JVMs start with a minimal set of features and classes and incrementally allow more to be added, up to the level of a full JDK 1.2 with Jini. In any case, a device capable of running Jini will have 8 MB of RAM or more, with networking capabilities, on a 32-bit processor.

If the device cannot run a JVM, then something else must run the JVM and act as a proxy for the device. Your blender is unlikely to have 32 MB of RAM, but your home control center (possibly located on the front of the fridge) may have this capability. In that case, the blender service would be located in this JVM, and the fridge would have some means of sending commands to the blender.

MINDSTORMS

LEGO MINDSTORMS (`http://www.LEGOMINDSTORMS.com`) is a Robotics Invention System, which consists of a number of LEGO parts, a microcomputer called the RCX, plus an infrared transmitter (connected to the serial port of an ordinary computer), and various sensors and motors. Using this system, one can build an almost infinite variety of LEGO robots that can be controlled by the RCX. This RCX computer can be sent “immediate” commands, or can have a (small) program downloaded and then run.

MINDSTORMS is a pretty cool system that can be driven at a number of levels. A primary audience for programming this system is children, and there is a visual programming environment to help in this. This visual environment only runs on Windows or Macintosh machines, which are connected to the RCX by their serial port and the infrared transmitter. Behind this environment is a Visual Basic set of procedures captured in an OCX, and behind that is the machine code of the RCX, which can be sent as byte codes on the serial port.

The RCX computer is completely incapable of running Jini. It is a 16-bit processor with a mere 32 K of RAM, and the default firmware will only allow 32 variables. It can only be driven by a service running on, say, an ordinary PC.

MINDSTORMS as a Jini Service

As previously mentioned, a MINDSTORMS robot can be programmed and run from an infrared transmitter attached to the serial port of a computer. There is no security or real location for the RCX—it will accept commands from any transmitter in range. We will assume that a robot is controlled by a single computer, and that it always stays in range of this computer.

There must be a way of communicating with any hardware device. For a MINDSTORMS robot, this is done via the serial port, but other devices may have different mechanisms. Communication may be by Java code or by the native code of the device. Even if Java code is used, at some stage it must drop down to the native code level in order to communicate with the device—the only question is whether you write the native code or someone else does it for you and wraps it up in Java object methods.

For the serial port, Sun has an extension package—the `commAPI`–to talk to serial and parallel ports (`http://java.sun.com/products/javacomm/index.html`). This package includes platform-independent Java code, and also platform-specific native code libraries supplied as DLLs for Windows and Solaris. I am running Linux on my laptop, so I am using a Linux version of the DLL. This has been made by Trent Jarvi (`trentjarvi@yahoo.com`), and can be found at `http://www.frii.com/~jarvi/rxtx/`. The native code part of communicating with the device has been done for us, and it is all wrapped up in a set of portable Java classes.

The RCX expects particular message formats that start with standard headers, and so on. A Java package that makes generating messages in the correct format easier has been created by Dario Laverde and is available at `http://www.escape.com/~dario/java/rcx`. There are other packages that will do the same thing—see the "LEGO MINDSTORMS Internals" Web page by Russell Nelson at `http://www.crynwr.com/LEGO-robotics/`.

With this as background, we can look at how to make an RCX into a Jini service. It will involve constructing an RCX program on a client and sending this program back to the server where it can be sent on to the RCX via the serial port. This program will then allow a client to control a MINDSTORMS robot remotely.

The Jini part is pretty easy—the hard part was tracking down all the bits and pieces needed to drive the RCX from Java. With your own lumps of hardware, the hard part will be writing the low-level code (probably using the Java Native Interface, JNI) and Java code to drive it.

RCXPort

Version 1.1 of the `rcx` package by Dario Laverde defines various classes, of which the most important is `RCXPort`:

```
package rcx;

public class RCXPort {
    public RCXPort(String port);
    public void addRCXListener(RCXListener rl);
    public boolean open();
    public void close();
    public boolean isOpen();
    public OutputStream getOutputStream();
    public InputStream getInputStream();
    public synchronized boolean write(byte[] bArray);
    public String getLastError();
}
```

The `RCXOpcode` class has a useful static method for creating byte code:

```
package rcx;

public class RCXOpcode {
    public static byte[] parseString(String str);
}
```

The relevant methods for this project are the following:

- The constructor `RCXPort()`. This takes the name of a port as parameter, which should be something like `COM1` for Windows and `/dev/ttyS0` for Linux.

- The `write()` method is used to send an array of opcodes and their arguments to the RCX. This is machine code, and you can only read it with a disassembler or a Unix tool like octal dump (`od -t xC`).

- The static `parseString()` method of `RCXOpcode` can be used to translate a string of instructions in readable form to an array of bytes for sending to the RCX. It isn't as good as an assembler, because you have to give strings such as `"21 81"` to start the A motor. To use this method for Jini, we will have to use a non-static method in our interface, because static methods are not allowed.

- To handle responses from the RCX, a listener may be added with `addRCXListener()`. The listener must implement this interface:

  ```
  package rcx;

  import java.util.*;

  /*
   * RCXListener
   * @author Dario Laverde
   * @version 1.1
   * Copyright 1999 Dario Laverde, under terms of GNU LGPL
   */
  public interface RCXListener extends EventListener {
      public void receivedMessage(byte[] message);
      public void receivedError(String error);
  }
  ```

RCX Programs

At the lowest level, the RCX is controlled by machine-code programs sent via the infrared link. It will respond to these programs by stopping and starting motors, changing speed, and so on. As it completes commands or receives information from sensors, it can send replies back to the host computer. The RCX can handle instructions sent directly or have a program downloaded into firmware and run from there.

Kekoa Proudfoot has produced a list of the opcodes understood by the RCX, and it is available at `http://graphics.stanford.edu/~kekoa/rcx/`. Using these and the `rcx` package from Dario Laverde, we can control the RCX from a computer by standalone programs such as this:

```
/**
 * TestRCX.java
 */

package standalone;

import rcx.*;

public class TestRCX implements RCXListener {
    static final String PORT_NAME = "/dev/ttyS0"; // Linux

    public TestRCX() {
        RCXPort port = new RCXPort(PORT_NAME);

        port.addRCXListener(this);

        byte[] byteArray;

        // send ping message, reply should be e7 or ef
        byteArray = RCXOpcode.parseString("10"); // Alive
        port.write(byteArray);

        // beep twice
        byteArray = RCXOpcode.parseString("51 01"); // Play sound
        port.write(byteArray);

        // turn motor A on (forwards)
        byteArray = RCXOpcode.parseString("e1 81"); // Set motor direction
        port.write(byteArray);
        byteArray = RCXOpcode.parseString("21 81"); // Set motor on
        port.write(byteArray);
        try {
            Thread.currentThread().sleep(1000);
        } catch(Exception e) {
        }

        // turn motor A off
        byteArray = RCXOpcode.parseString("21 41"); // Set motor off
        port.write(byteArray);

        // turn motor A on (backwards)
        byteArray = RCXOpcode.parseString("e1 41"); // Set motor direction
        port.write(byteArray);
        byteArray = RCXOpcode.parseString("21 81"); // Set motor on
        port.write(byteArray);
        try {
            Thread.currentThread().sleep(1000);
        } catch(Exception e) {
        }

        // turn motor A off
        byteArray = RCXOpcode.parseString("21 41"); // Set motor off
        port.write(byteArray);
    }

    /**
     * listener method for messages from the RCX
     */
    public void receivedMessage(byte[] message) {
        if (message == null) {
            return;
        }
        StringBuffer sbuffer = new StringBuffer();
        for(int n = 0; n < message.length; n++) {
            int newbyte = (int) message[n];
            if (newbyte < 0) {
                newbyte += 256;
            }
            sbuffer.append(Integer.toHexString(newbyte) + " ");
        }
        System.out.println("response: " + sbuffer.toString());
    }

    /**
     * listener method for error messages from the RCX
     */
    public void receivedError(String error) {
        System.err.println("Error: " + error);
    }

    public static void main(String[] args) {
        new TestRCX();
    }

} // TestRCX
```

Jini Classes

A simple Jini service can use an RMI proxy, where the service just remains in the server and the client makes remote method calls on it. The service will hold an `RCXPort` and will feed the messages through it. This involves constructing the hierarchy of classes shown in Figure 17-1.

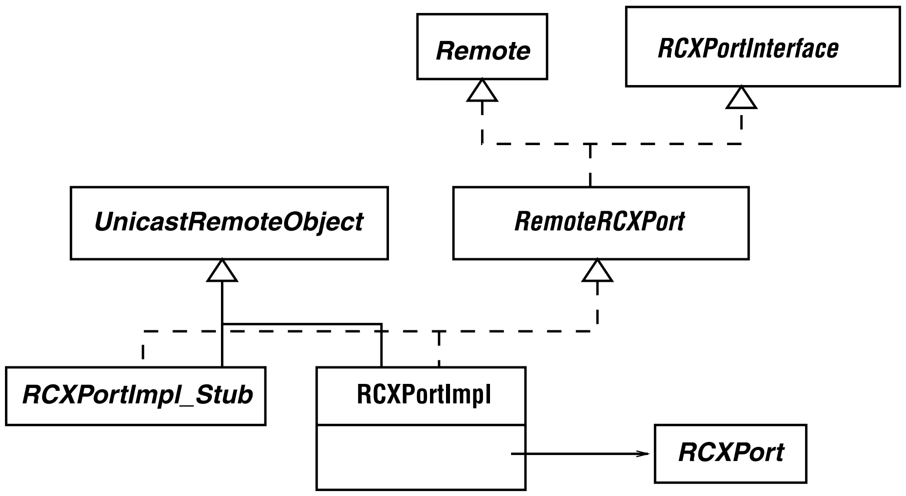

Figure 17-1. Class diagram for MINDSTORMS with RMI proxies

The `RCXPortInterface` just defines the methods we will be making available from the Jini service. It doesn't have to follow the `RCXPort` methods completely, because these will be wrapped up in implementation classes, such as `RCXPortImpl`. The interface is defined as follows:

```
/**
 * RCXPortInterface.java
 */

package rcx.jini;

import net.jini.core.event.RemoteEventListener;

public interface RCXPortInterface extends java.io.Serializable {

    /**
     * constants to distinguish message types
     */
    public final long ERROR_EVENT = 1;
    public final long MESSAGE_EVENT = 2;

    /**
     * Write an array of bytes that are RCX commands
     * to the remote RCX.
     */
    public boolean write(byte[] byteCommand) throws java.rmi.RemoteException;

    /**
     * Parse a string into a set of RCX command bytes
     */
    public byte[] parseString(String command) throws java.rmi.RemoteException;

    /**
     * Add a RemoteEvent listener to the RCX for messages and errors
     */
    public void addListener(RemoteEventListener listener)
        throws java.rmi.RemoteException;

    /**
     * The last message from the RCX
     */
    public byte[] getMessage(long seqNo)
        throws java.rmi.RemoteException;

    /**
     * The error message from the RCX
     */
    public String getError(long seqNo)
        throws java.rmi.RemoteException;

} // RCXPortInterface
```

We have chosen to make a subpackage of the `rcx` package and to place the preceding class in this package to make its role clearer. Note that the `RCXPortInterface` has no static methods, but makes `parseString()` into an ordinary instance method.

This interface contains two types of methods: those used to prepare and send messages to the RCX (`write()` and `parseString()`), and those used to handle messages sent from the RCX (`addListener()`, `getMessage()`, and `getError()`). Any listener that is added will be informed of events generated by implementations of this interface by having the listener's `notify()` method called. However, a `RemoteEvent` does not contain detailed information about what has happened, as it only contains an event type (`MESSAGE_EVENT` or `ERROR_EVENT`). It is up to the listener to make queries back into the object to discover what the event meant, which it does with `getMessage()` and `getError()`.

The `RemoteRCXPort` interface just adds the `Remote` interface:

```
/**
 * RemoteRCXPort.java
 */

package rcx.jini;

import java.rmi.Remote;

public interface RemoteRCXPort extends RCXPortInterface, Remote {

} // RemoteRCXPort
```

The `RCXPortImpl` constructs its own `RCXPort` object and feeds methods, such as `write()`, through to it. Since it extends `UnicastRemoteObject`, it also adds exceptions to each method, which cannot be done to the original `RCXPort` class. In addition, it picks up the value of the port name from the `port` property. (This follows the example of the `RCXLoader` in the `rcx` package, which provides a GUI interface for driving the RCX.) It looks for this `port` property in the `parameters.txt` file, which should have lines such as this:

```
port=/dev/ttyS0
```

Note that the parameters file exists on the server side—no client would know this information!

The `RCXPortImpl` also acts as a listener for "ordinary" RCX events signaling messages from the RCX. It uses the callback methods `receivedMessage()` and `receivedError()` to create a new `RemoteEvent` object and send it to the implementation's listener object (if there is one) by calling its `notify()` method.

The implementation looks like this:

```
/**
 * RCXPortImpl.java
 */

package rcx.jini;

import java.rmi.server.UnicastRemoteObject;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import rcx.*;
import java.io.*;
import java.util.*;

public class RCXPortImpl extends UnicastRemoteObject
    implements RemoteRCXPort, RCXListener {

    protected String error = null;
    protected byte[] message = null;
    protected RCXPort port = null;
    protected RemoteEventListener listener = null;
    protected long messageSeqNo, errorSeqNo;

    public RCXPortImpl()
        throws java.rmi.RemoteException {

        Properties parameters;
        String portName = null;
        File f = new File("parameters.txt");
        if (!f.exists()) {
            f = new File(System.getProperty("user.dir")
                         + System.getProperty("path.separator")
                         + "parameters.txt");
        }
        if (f.exists()) {
            try {
                FileInputStream fis = new FileInputStream(f);
                parameters = new Properties();
                parameters.load(fis);
                fis.close();
                portName = parameters.getProperty("port");
            } catch (IOException e) { }
        } else {
            System.err.println("Can't find parameters.txt with \"port=...\"
specified");
            System.exit(1);
        }

        port = new RCXPort(portName);
        port.addRCXListener(this);

    }

    public boolean write(byte[] byteCommands)
        throws java.rmi.RemoteException {
        return port.write(byteCommands);
    }

    public byte[] parseString(String command)
        throws java.rmi.RemoteException {
        return RCXOpcode.parseString(command);
    }

    /**
     * Received a message from the RCX.
     * Send it to the listener
     */
    public void receivedMessage(byte[] message) {

        this.message = message;

        // Send it out to listener
        if (listener == null) {
            return;
        }

        RemoteEvent evt = new RemoteEvent(this, MESSAGE_EVENT, messageSeqNo++,
null);
        try {
            listener.notify(evt);
        } catch(net.jini.core.event.UnknownEventException e) {
            e.printStackTrace();
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
        }
    }
```

```
/**
 * Received an error message from the RCX.
 * Send it to the listener
 */
public void receivedError(String error) {
    // System.err.println(error);

    // Send it out to listener
    if (listener == null) {
        return;
    }
    this.error = error;
    RemoteEvent evt = new RemoteEvent(this, ERROR_EVENT, errorSeqNo, null);
    try {
        listener.notify(evt);
    } catch(net.jini.core.event.UnknownEventException e) {
        e.printStackTrace();
    } catch(java.rmi.RemoteException e) {
        e.printStackTrace();
    }
}

/**
 * Expected use: the RCX has returned a message,
 * and we have informed the listeners. They query
 * this method to find the message for the message
 * sequence number they were given in the RemoteEvent.
 * We could use this as an index into a table of messages.
 */
public byte[] getMessage(long msgSeqNo) {
    return message;
}

/**
 * Expected use: the RCX has returned an error message,
 * and we have informed the listeners. They query
 * this method to find the error message for the error message
 * sequence number they were given in the RemoteEvent.
 * We could use this as an index into a table of messages.
 */
public String getError(long errSeqNo) {
    return error;
}
```

```
    /**
     * Add a listener for RCX messages.
     * Should allow more than one, or throw
     * TooManyListeners if more than one registers
     */
    public void addListener(RemoteEventListener listener) {
        this.listener = listener;
        messageSeqNo = 0;
        errorSeqNo = 0;
    }
} // RCXPortImpl
```

Getting It Running

To make use of these classes, we need to provide a server to get the service put onto the network, and we need some clients to make use of the service. This section will just look at a simple way of doing this, and later sections in this chapter will put in more structure.

The following is a simple server that follows the earlier examples of servers using RMI proxies (such as in Chapter 9), just substituting `RCXPort` for `FileClassifier` and using a `JoinManager`. It creates an `RCXPortImpl` object and registers it (or rather, the RMI proxy) with lookup services:

```
package rcx.jini;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.core.lookup.ServiceID;
// import com.sun.jini.lease.LeaseRenewalManager;
// import com.sun.jini.lookup.JoinManager;
// import com.sun.jini.lookup.ServiceIDListener;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.JoinManager;
import net.jini.lookup.ServiceIDListener;

/**
 * RCXServer.java
 */

public class RCXServer implements ServiceIDListener {

    protected RCXPortImpl impl;
    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

    public static void main(String argv[]) {
        new RCXServer();
        // remember to keepalive
    }

    public RCXServer() {
        try {
            impl = new RCXPortImpl();
        } catch(Exception e) {
            System.err.println("New impl: " + e.toString());
            System.exit(1);
        }

        // set RMI security manager
        System.setSecurityManager(new RMISecurityManager());

        // find, register, lease, etc
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            JoinManager joinMgr = new JoinManager(impl,
                                                  null,
                                                  this,
                                                  mgr,
                                                  new LeaseRenewalManager());
        } catch(java.io.IOException e) {
            e.printStackTrace();
        }
    }

    public void serviceIDNotify(ServiceID serviceID) {
        System.out.println("Got service ID " + serviceID.toString());
    }
} // RCXServer
```

Why is this example simplistic as a service? Well, it doesn't contain any information to allow a client to distinguish one LEGO MINDSTORMS robot from another, so that if there are many robots on the network, then a client could ask the wrong one to do things!

An equally simplistic client that makes the RCX perform a few actions is given below. In addition to sending a set of commands to the RCX, the client must also listen for replies from the RCX. I have separated out this listener as an `EventHandler` for readability. The listener will act as a remote event listener, with its `notify()` method called from the server. This can be done by letting it run an RMI stub on the server, so I have subclassed it from `UnicastRemoteObject`.

This particular client is designed to drive a particular robot: the "RoverBot" described in the LEGO MINDSTORMS "Constructopedia" (the instruction manual that comes with each MINDSTORMS set) and pictured in Figure 17-2.

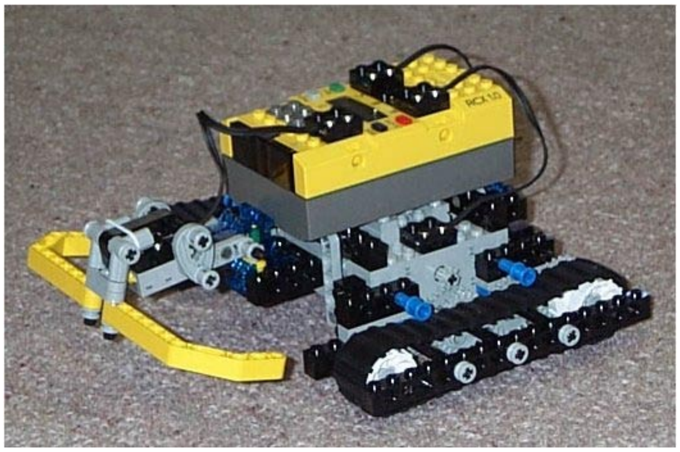

Figure 17-2. RoverBot MINDSTORMS robot

The RoverBot has motors to drive tracks or wheels on either side. The client can send instructions to make the RoverBot move forward or backward, stop, or turn to the left or right. The set of commands (and their implementation as RCX instructions) depends on the robot, and on what you want to do with it.

Here is the client code:

```
package client;

import rcx.jini.*;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.UnknownEventException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
/**
 * TestRCX.java
 */

public class TestRCX implements DiscoveryListener {

    public static final int STOPPED = 1;
    public static final int FORWARDS = 2;
    public static final int BACKWARDS = 4;

    protected int state = STOPPED;

    public static void main(String argv[]) {
        new TestRCX();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(10000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestRCX() {
        System.setSecurityManager(new RMISecurityManager());

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }
        discover.addDiscoveryListener(this);

    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();
        Class [] classes = new Class[] {RCXPortInterface.class};
        RCXPortInterface port = null;
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Service found");
            ServiceRegistrar registrar = registrars[n];
            try {
                port = (RCXPortInterface) registrar.lookup(template);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(2);
            }
            if (port == null) {
                System.out.println("port null");
                continue;
            }

            // add an EventHandler as an RCX Port listener
            try {
                port.addListener(new EventHandler(port));
            } catch(Exception e) {
                e.printStackTrace();
            }
        }
    }

    public void discarded(DiscoveryEvent evt) {
        // empty
    }

    class EventHandler extends UnicastRemoteObject
                       implements RemoteEventListener, ActionListener {

        protected RCXPortInterface port = null;
```

```
JFrame frame;
JTextArea text;

public EventHandler(RCXPortInterface port) throws RemoteException {
    super() ;
    this.port = port;

    frame = new JFrame("LEGO MINDSTORMS");
    Container content = frame.getContentPane();
    JLabel label = new JLabel(new ImageIcon("images/MINDSTORMS.ps"));
    JPanel pane = new JPanel();
    pane.setLayout(new GridLayout(2, 3));

    content.add(label, "North");
    content.add(pane, "Center");

    JButton btn = new JButton("Forward");
    pane.add(btn);
    btn.addActionListener(this);

    btn = new JButton("Stop");
    pane.add(btn);
    btn.addActionListener(this);

    btn = new JButton("Back");
    pane.add(btn);
    btn.addActionListener(this);

    btn = new JButton("Left");
    pane.add(btn);
    btn.addActionListener(this);

    label = new JLabel("");
    pane.add(label);

    btn = new JButton("Right");
    pane.add(btn);
    btn.addActionListener(this);

    frame.pack();
    frame.setVisible(true);
}

public void sendCommand(String comm) {
    byte[] command;
    try {
        command = port.parseString(comm);
        if (! port.write(command)) {
            System.err.println("command failed");
        }
    } catch(RemoteException e) {
        e.printStackTrace();
    }
}

public void forwards() {
    sendCommand("e1 85");
    sendCommand("21 85");
    state = FORWARDS;
}

public void backwards() {
    sendCommand("e1 45");
    sendCommand("21 85");
    state = BACKWARDS;
}

public void stop() {
    sendCommand("21 45");
    state = STOPPED;
}

public void restoreState() {
    if (state == FORWARDS)
        forwards();
    else if (state == BACKWARDS)
        backwards();
    else
        stop();
}

public void actionPerformed(ActionEvent evt) {
    String name = evt.getActionCommand();

    if (name.equals("Forward")) {
        forwards();
    } else  if (name.equals("Stop")) {
        stop();
    } else  if (name.equals("Back")) {
        backwards();
    } else  if (name.equals("Left")) {
        sendCommand("e1 84");
        sendCommand("21 84");
        sendCommand("21 41");
        try {
            Thread.sleep(100);
        } catch(InterruptedException e) {
        }
        restoreState();

    } else  if (name.equals("Right")) {
        sendCommand("e1 81");
        sendCommand("21 81");
        sendCommand("21 44");
        try {
            Thread.sleep(100);
        } catch(InterruptedException e) {
        }
        restoreState();
    }
}

public void notify(RemoteEvent evt) throws UnknownEventException,
                                         java.rmi.RemoteException {
    // System.out.println(evt.toString());

    long id = evt.getID();
    long seqNo = evt.getSequenceNumber();
    if (id == RCXPortInterface.MESSAGE_EVENT) {
        byte[] message = port.getMessage(seqNo);
        StringBuffer sbuffer = new StringBuffer();
        for(int n = 0; n < message.length; n++) {
            int newbyte = (int) message[n];
            if (newbyte < 0) {
                newbyte += 256;
            }
            sbuffer.append(Integer.toHexString(newbyte) + " ");
        }
        System.out.println("MESSAGE: " + sbuffer.toString());
    } else if (id == RCXPortInterface.ERROR_EVENT) {
        System.out.println("ERROR: " + port.getError(seqNo));
    } else {
```

```
                throw new UnknownEventException("Unknown message " + evt.getID());
            }
        }
    }

} // TestRCX
```

Why is this a simplistic *client*? It tries to find all robots on the local network, and creates a top-level window for each of them. If a robot has registered with, say, half-a-dozen service locators, and the client finds all of these, then it will create six top-level windows, one for each copy of the same robot. Some smarts are needed here, such as using the `ClientLookupManager` of Chapter 15.

Entry Objects for a Robot

The RCX was not designed for network visibility. It has no concept of identity or location. The closest it comes to this is when it communicates to other RCXs by the infrared transmitter—then one RCX may have to decide whether it is the master, which it does by setting a local variable to "master" if it broadcasts before it receives, and the other RCXs will set the variable to "slave" if they receive before broadcasting. Then each waits for a random amount of time before broadcasting. Crude, but it works.

In a Jini environment, there may be many RCX devices. These devices are not tied to any particular computer, as they will respond to any infrared transmitter on the correct frequency talking the right protocol. All the devices within range of a transmitter will accept signals from the transmitter, although this can cause problems, because the source computers tend to assume that there is only one target at a time, and they can get confused by responses from multiple RCXs. The solution is to turn off all but one RCX when a program is being downloaded, to avoid this confusion. Then turn on the next, and download to it, and so on. Not very elegant, but it works.

An RCX may also be mobile—it can control motors, so if it is placed in a mobile robot, it can drive itself out of the range of one PC and (maybe) into the range of another. There are no mechanisms to signal either passing out of range or coming into range.

The RCX is a poorly behaved animal from a network viewpoint. However, we will need to distinguish between different RCXs in order to drive the correct ones. An `Entry` class for distinguishing them should contain information such as this:

- An identifier for robot type, such as "Robo 1", "Acrobot 1", etc. This will allow the robot that the RCX is built into to be identified. The RCX will have no knowledge of its identifier—it must be externally supplied.

- The RCX can be driven by direct commands or by executing a program already downloaded (there may be up to five of these). An identifier for each downloaded program should be available.

- The RCX will have some sort of location, although it may move around to a limited extent. This location information may be available from the controlling computer, using the Jini `Location` or `Address` classes.

There may be other useful attributes, and there are certainly issues to be resolved about how the information could be stored and accessed from an RCX. However, they stray beyond the bounds of this chapter.

A Client-Side RCX Class

In the simplistic client given earlier, there were many steps that will be the same for all clients that can drive the RCX. Just as `JoinManager` simplifies repetitive code on the server side, we can define a "convenience" class for the RCX that will do the same on the client side. The aim is to supply a class that will make remote RCX programming as easy as local RCX programming.

A class that encapsulates client-side behavior may as well look as much as possible like the local `RCXPort` class. We define its (public) methods as follows:

```
public class JiniRCXPort {
    public JiniRCXPort();
    public void addRCXListener(RCXListener l);
    public boolean write(byte[] bArray);
    public byte[] parseString(String str);
}
```

This class should have some control over how it looks for services by including entry information, group information about locators, and any specific locators it should try. There are a variety of possible constructors, all ending up calling a constructor that looks like this:

```
public JiniRCXPort(Entry[] entries,
                   java.lang.String[] groups,
                   LookupLocator[] locators)
```

The class is also concerned with uniqueness issues, as it should not attempt to send the same instructions to an RCX more than once. However, it could send the same instructions to more than one RCX if they match the search criteria. Therefore, this class maintains a list of RCXs and does not add to the list if it has already seen the RCX from another service locator. This requires that a single RCX should be registered using the same `ServiceID` with all locators, which will be the case because the RCX server uses `JoinManager`.

Higher-Level Mechanisms: Not Quite C

"Not Quite C" (`nqc`) is a language and a compiler from David Baum, designed for the RCX. It defines a language with C-like syntax that defines tasks that can be executed concurrently. The RCX API also defines a number of constants, functions, and macros targeted specifically to the RCX. These include constants such as `OUT_A` (for output A) and functions such as `OnFwd` to turn a motor on forwards.

The following is a trivial `nqc` program to turn motor A on for 1 second (units are 1/100th of a second):

```
task main() {
        OnFwd(OUT_A);
        Wait(100);
        Off(OUT_A);
}
```

Writing programs using a higher-level language such as this is clearly preferable to writing in Assembler!

`nqc` is not the only higher-level language for programming the RCX. There are links to many others on the alternative MINDSTORMS site (`http://www.crynwr.com/LEGO-robotics/`). It is one of the earliest and more popular ones, though, and it is a typical example of a standalone, non-GUI program written in a language other than Java that can still be used as a Jini service.

The `nqc` compiler is written in C++ and needs to be compiled for each platform that it will run on. Precompiled versions are available for a number of systems, such as Windows and Linux. Once compiled, it is tied to a particular computer (at least, to computers with a particular OS and shared library configuration). It is software, not hardware like the MINDSTORMS robots, but it is nevertheless not mobile. It cannot be moved around like Java code can. However, it can be turned into a Jini service in exactly the same way as MINDSTORMS, by wrapping it in a Java class that can be exported as a Jini service. This also fits the RMI proxy model, with the client side using a thin proxy that makes calls to a service that invokes the `nqc` compiler.

The class diagram follows other RMI proxy diagrams and is shown in Figure 17-3.

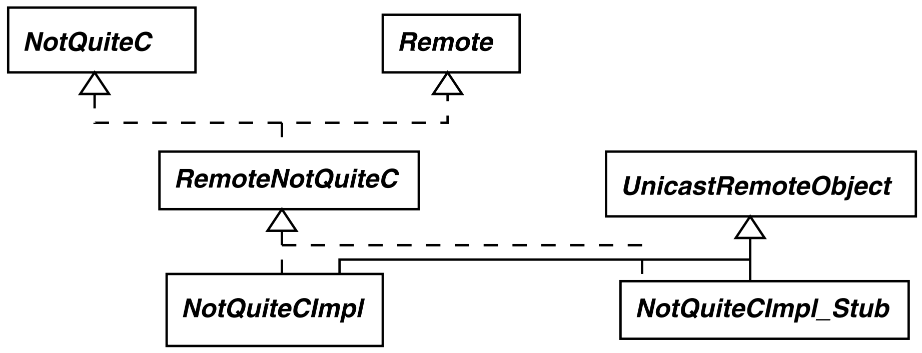

Figure 17-3. Class diagram for nqc with RMI proxy

The `NotQuiteC` and `RemoteNotQuiteC` interfaces are defined by

```
/**
 * NotQuiteC.java
 */

package rcx.jini;

import java.rmi.RemoteException;
import java.io.Serializable;

public interface NotQuiteC extends Serializable {

    public byte[] compile(String program)
        throws RemoteException, CompileException;

} // NotQuiteC
```

and by

```
/**
 * RemoteNotQuiteC.java
 */

package rcx.jini;

import java.rmi.Remote;

public interface RemoteNotQuiteC extends NotQuiteC, Remote {
} // RemoteNotQuiteC
```

The compile exception is thrown when things go wrong:

```
/**
 * CompileException.java
 */

package rcx.jini;

public class CompileException extends Exception {

    protected String error;

    public CompileException(String err) {
        error = err;
    }

    public String toString() {
        return error;
    }
} // CompileException
```

An implementation of the `RemoteNotQuiteC` interface needs to encapsulate a traditional application running in an environment of just reading and writing files. GUI applications, or those nuisance Unix ones that insist on using an interactive terminal (such as `telnet`), will need more complex encapsulation methods. The `nqc` type of application will read from standard input or from a file, often depending on command line flags. Similarly, it will write to a file or to standard output, again depending on command line flags. Applications either succeed or fail in their task; this should be indicated by what is known as an exit code, which by convention is 0 for success and some other integer value for failure. If a failure occurs, an application will usually write diagnostic output to the standard error channel.

The current version of `nqc` (version 2.0.2) is badly behaved for reading from standard input (it crashes) and writing to standard output (no way of doing this). So we can't create a `Process` to run `nqc` and feed into its input and output. Instead, we need to create temporary files and write to and read from these files so that the Jini wrapper can communicate with `nqc`. These files also need to be cleaned up on termination, whether the normal or exception routes are followed. On the other hand, if errors occur, they will be reported on the error channel of the process, and this needs to be captured in some way—in this example, we will do it via an exception constructor.

The hard part in this example is plowing your way through the Java I/O maze, and deciding exactly how to hook up I/O streams and/or files to the external process. The following code uses temporary files for ordinary I/O with the process (the current version I have of `nqc` has a bug with pipelines) and the standard error stream for compile errors.

```
/**
 * NotQuiteCImpl.java
 */

package rcx.jini;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import java.io.*;

public class NotQuiteCImpl extends UnicastRemoteObject
    implements RemoteNotQuiteC {

    protected final int SIZE = 1<<15; // 32k - the size of the RCX memory

    public NotQuiteCImpl() throws RemoteException {
    }

    public byte[] compile(String program)
        throws CompileException {

        // This is the input file we read from - it is the output from nqc
        File inFile = null;
        //  This is the output file that we write to - it is the input to nqc
        File outFile = null;
        byte[] buff = new byte[SIZE];

        try {
            outFile = File.createTempFile("jini", ".nqc");
            inFile = File.createTempFile("jini", ".rcx");
            OutputStreamWriter out = new OutputStreamWriter(
                                         new FileOutputStream(
                                             outFile));

            out.write(program);
            out.close();

            Process p = Runtime.getRuntime().exec("nqc -O" +
                                        inFile.getAbsolutePath() +
                                        " " + outFile.getAbsolutePath());

            int status = p.waitFor();
            if (status != 0) {
                BufferedReader err = new BufferedReader(
                                         new InputStreamReader(p.
                                              getErrorStream()));
                StringBuffer errBuff = new StringBuffer();
                String line;
                while ((line = err.readLine()) != null) {
                    errBuff.append(line + '\n');
                }
                throw new CompileException(errBuff.toString());
            }

            DataInputStream compiled = new DataInputStream(new
                                            FileInputStream(outFile));
            int nread = compiled.read(buff);
            byte[] result = new byte[nread];
            System.arraycopy(buff, 0, result, 0, nread);
            return result;
        } catch(IOException e) {
            throw new CompileException(e.toString());
        } catch(InterruptedException e) {
            throw new CompileException(e.toString());
        } finally {
            // clean up files even if exceptions thrown
            if (inFile != null) {
                inFile.delete();
            }
            if (outFile != null) {
                outFile.delete();
            }
        }
    }

    public static void main(String[] argv) {
        String program = "task main() {\n" +
                         "     OnFwd(OUT_A);\n" +
                         "     Wait(100);\n" +
                         "     Off(OUT_A);\n" +
                         "}";
        NotQuiteCImpl compiler = null;
        try {
            compiler = new NotQuiteCImpl();
            byte[] bytes = compiler.compile(program);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
} // NotQuiteCImpl
```

This section does not give server and client implementations—the server is the same as servers delivering other RMI services. A client will make a call on this service, specifying the program to be compiled. It can then write the byte stream to the RCX using the classes given earlier.

Summary

This chapter has considered some of the issues involved in using a piece of hardware with a Jini service. This was illustrated with LEGO MINDSTORMS, where a large part of the base work of native code libraries and encapsulation in Java classes has already been done. Even then, there is much work involved in making it a suitable Jini service, and these have been discussed. This work is not yet complete, and more remains to be done for LEGO MINDSTORMS.