Leased FileClassifier

A dynamically extensible version of a file classifier will have methods to add and remove MIME mappings:

```
package common;

import java.io.Serializable;

/**
 * LeaseFileClassifier.java
 */

import net.jini.core.lease.Lease;

public interface LeaseFileClassifier extends Serializable {

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException;

    /*
     * Add the MIME type for the given suffix.
     * The suffix does not contain '.' e.g. "gif".
     * @exception net.jini.core.lease.LeaseDeniedException
     * a previous MIME type for that suffix exists.
     * This type is removed on expiration or cancellation
     * of the lease.
     */
    public Lease addType(String suffix, MIMEType type)
        throws java.rmi.RemoteException,
               net.jini.core.lease.LeaseDeniedException;

    /**
     * Remove the MIME type for the suffix.

     */
    public void removeType(String suffix)
        throws java.rmi.RemoteException;
} // LeaseFileClasssifier
```

The addType() method returns a lease. We shall use the landlord leasing system discussed in Chapter 7. The client and the service will be in different Java VMs, probably on different machines. Figure 13-5 gives the object structure on the service side.

This should be compared to Figure 7-3 where we considered the "foo" implementation of landlord leasing.

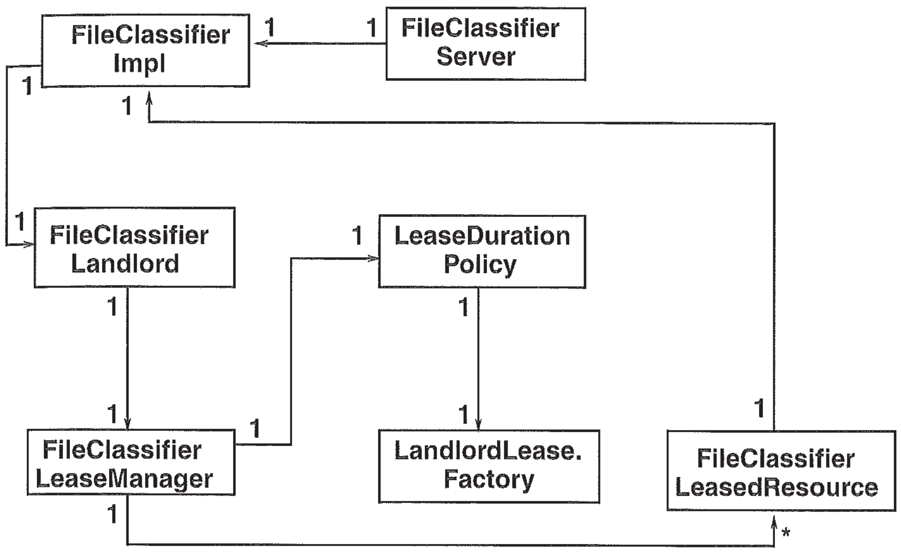


Figure 13-5. Class diagram for leasing on the server

On the client side, the lease object will be a copy of the lease created on the server (normally RMI semantics), but the other objects from the service will be stubs that call into the real objects on the service. This is shown in Figure 13-6.

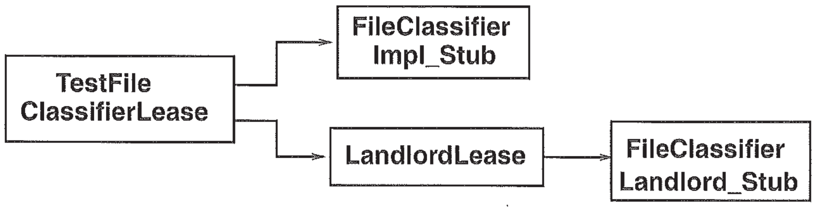


Figure 13-6. Class diagram for leasing on the client

The FileClassifierLeasedResource Class

The FileClassifierLeasedResource class acts as a wrapper around the actual resource, adding cookie and time expiration fields around the resource. It adds a unique cookie mechanism, in addition to making the wrapped resource visible.

```
/**
 * FileClassifierLeasedResource.java
 */
package lease;

import common.LeaseFileClassifier;
import com.sun.jini.lease.landlord.LeasedResource;

public class FileClassifierLeasedResource implements LeasedResource  {

    static protected int cookie = 0;
    protected int thisCookie;
    protected LeaseFileClassifier fileClassifier;
    protected long expiration = 0;
    protected String suffix = null;

    public FileClassifierLeasedResource(LeaseFileClassifier fileClassifier,
                                        String suffix) {
        this.fileClassifier = fileClassifier;
        this.suffix = suffix;
        thisCookie = cookie++;
    }

    public void setExpiration(long newExpiration) {
        this.expiration = newExpiration;
    }

    public long getExpiration() {
        return expiration;
    }
    public Object getCookie() {
        return new Integer(thisCookie);
    }

    public LeaseFileClassifier getFileClassifier() {
        return fileClassifier;
    }
    public String getSuffix() {
        return suffix;
    }
} // FileClassifierLeasedResource
```

The FileClassifierLeaseManager Class

The `FileClassifierLeaseManager` class is very similar to the code given for the `FooLeaseManager` in Chapter 7:

```
/**
 * FileClassifierLeaseManager.java
 */
package lease;

import java.util.*;
import common.LeaseFileClassifier;

import net.jini.core.lease.Lease;
import com.sun.jini.lease.landlord.LeaseManager;
import com.sun.jini.lease.landlord.LeasedResource;
import com.sun.jini.lease.landlord.LeaseDurationPolicy;
import com.sun.jini.lease.landlord.Landlord;
import com.sun.jini.lease.landlord.LandlordLease;
import com.sun.jini.lease.landlord.LeasePolicy;

public class FileClassifierLeaseManager implements LeaseManager {

    protected static long DEFAULT_TIME = 30*1000L;

    protected Vector fileClassifierResources = new Vector();
    protected LeaseDurationPolicy policy;

    public FileClassifierLeaseManager(Landlord landlord) {
        policy = new LeaseDurationPolicy(Lease.FOREVER,
                                         DEFAULT_TIME,
                                         landlord,
                                         this,
                                         new LandlordLease.Factory());
        new LeaseReaper().start();
    }

    public void register(LeasedResource r, long duration) {
        fileClassifierResources.add(r);
    }

    public void renewed(LeasedResource r, long duration, long olddur) {
        // no smarts in the scheduling, so do nothing
    }

    public void cancelAll(Object[] cookies) {
        for (int n = cookies.length; --n >= 0; ) {
            cancel(cookies[n]);
        }
    }

    public void cancel(Object cookie) {
        for (int n = fileClassifierResources.size(); --n >= 0; ) {
            FileClassifierLeasedResource r = (FileClassifierLeasedResource)
                                      fileClassifierResources.elementAt(n);
            if (!policy.ensureCurrent(r)) {
                System.out.println("Lease expired for cookie = " +
                                   r.getCookie());
                try {
                    r.getFileClassifier().removeType(r.getSuffix());
                } catch(java.rmi.RemoteException e) {
                    e.printStackTrace();
                }
                fileClassifierResources.removeElementAt(n);
            }
        }
    }

    public LeasePolicy getPolicy() {
        return policy;
    }

    public LeasedResource getResource(Object cookie) {
        for (int n = fileClassifierResources.size(); --n >= 0; ) {
            FileClassifierLeasedResource r = (FileClassifierLeasedResource)
                                      fileClassifierResources.elementAt(n);
            if (r.getCookie().equals(cookie)) {
                return r;
            }
        }
        return null;
    }
    class LeaseReaper extends Thread {
        public void run() {
            while (true) {
                try {
                    Thread.sleep(DEFAULT_TIME) ;
                }
                catch (InterruptedException e) {
                }
                for (int n = fileClassifierResources.size()-1; n >= 0; n--) {
                   FileClassifierLeasedResource r = (FileClassifierLeasedResource)
                                           fileClassifierResources.elementAt(n)
;
                    if (!policy.ensureCurrent(r)) {
                        System.out.println("Lease expired for cookie = " +
                                           r.getCookie()) ;
                        try {
                            r.getFileClassifier().removeType(r.getSuffix());
                        } catch(java.rmi.RemoteException e) {
                            e.printStackTrace();
                        }
                        fileClassifierResources.removeElementAt(n);

                    }
                }
            }
        }
    }

} // FileClassifierLeaseManager
```

The FileClassifierLandlord Class

The FileClassifierLandlord class is very similar to the FooLandlord in Chapter 7:

```
/**
 * FileClassifierLandlord.java
 */

package lease;

import common.LeaseFileClassifier;
import com.sun.jini.lease.landlord.*;
import net.jini.core.lease.LeaseDeniedException;
import net.jini.core.lease.Lease;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.Remote;

public class FileClassifierLandlord extends UnicastRemoteObject implements Land-
lord, Remote {

    FileClassifierLeaseManager manager = null;

    public FileClassifierLandlord() throws java.rmi.RemoteException {
        manager = new FileClassifierLeaseManager(this);
    }

    public void cancel(Object cookie) {
        manager.cancel(cookie);
    }

    public void cancelAll(Object[] cookies) {
        manager.cancelAll(cookies);
    }

    public long renew(java.lang.Object cookie,
                      long extension)
        throws net.jini.core.lease.LeaseDeniedException,
               net.jini.core.lease.UnknownLeaseException {
        LeasedResource resource = manager.getResource(cookie);
        if (resource != null) {
            return manager.getPolicy().renew(resource, extension);
        }
        return -1;
    }

    public Lease newFileClassifierLease(LeaseFileClassifier fileClassifier,
                                        String suffixKey, long duration)
        throws LeaseDeniedException {
        FileClassifierLeasedResource r = new
        FileClassifierLeasedResource(fileClassifier,
                                                                   suffixKey);
        return manager.getPolicy().leaseFor(r, duration);
    }

    public Landlord.RenewResults renewAll(java.lang.Object[] cookie,
                                            long[] extension) {
        return null;
    }
} // FileClassifierLandlord
```

Summary

Jini provides a framework for building distributed applications. Nevertheless, there is still room for variation in how services and clients are written, and some of these are better than others. This chapter has looked at some of the variations that can occur and how to deal with them.

CHAPTER 14

Remote Events

COMPONENTS OF A SYSTEM CAN CHANGE STATE and may need to inform other components that this change has happened. Java Beans and user-interface elements such as AWT or Swing objects use events to signal these changes. Jini also has an event mechanism, and this chapter looks at the distributed event model that is part of Jini. It looks at how remote event listeners are registered with objects, and how these objects notify their listeners of changes. Event listeners may disappear, and so the Jini event mechanism uses leases to manage listener lists.

This chapter also looks at how leases are managed by event sources. Finally, we'll look at how events can be used by applications to monitor when services are registered or discarded from service locators.

Event Models

Java has a number of event models, differing in various subtle ways. All of these involve an object (an *event source*) generating an event in response to some change of state, either in the object itself (for example, if someone has changed a field), or in the external environment (such as when a user moves the mouse). At some earlier stage, a listener (or set of listeners) will have registered interest in this event. When the event source generates an event, it will call suitable methods on the listeners with the event as parameter. The event models all have their origin in the `Observer` pattern from *Design Patterns*, by Eric Gamma et al., but this is modified by other pressures, such as Java Beans.

There are low-level input events, which are generated by user actions when they control an application with a graphical user interface. These events—of type `KeyEvent` and `MouseEvent`–are placed in an event queue. They are removed from the queue by a separate thread and dispatched to the relevant objects. In this case, the object that is responsible for generating the event is not responsible for dispatching it to listeners, and the creation and dispatch of events occurs in different threads.

Input events are a special case caused by the need to listen to user interactions and always deal with them without losing response time. Most events are dealt with in a simpler manner: an object maintains its own list of listeners, generates its own events, and dispatches them directly to its listeners. In this category fall all the semantic events generated by the AWT and Swing toolkits, such as `ActionEvent`, `ListSelectionEvent`, etc. There is a large range of these event types, and they all call different methods in the listeners, based on the event name. For example, an ActionEvent is used in a listener's actionPerformed() method of an ActionListener. There are naming conventions involved in this, specified by Java Beans.

Java Beans is also the influence behind PropertyChange events, which get delivered whenever a Bean changes a "bound" or "constrained" property value. These are delivered by the event source calling the listener's PropertyChangeListener's propertyChange() method or the VetoableChangeListener's vetoableChange() method. These are usually used to signal a change in a field of an object, where this change may be of interest to the listeners either for information or for vetoing.

Jini objects may also be interested in changes in other Jini objects, and might like to be listeners for such changes. The networked nature of Jini has led to a particular event model that differs slightly from the other models already in Java. The differences are caused by several factors:

- Network delivery is unreliable—messages may be lost. Synchronous methods requiring a reply may not work here.
- Network delivery is time-dependent—messages may arrive at different times to different listeners. As a result, the state of an object as perceived by a listener at any time may be inconsistent with the state of that object as perceived by others. Passing complex object state across the network may be more complex to manage than passing simpler information.
- A remote listener may have disappeared by the time the event occurs. Listeners have to be allowed to time out, like services do.
- Java Beans can require method names and event types that vary and can use many classes. This requires a large number of classes to be available across the network, which is more complex than a single class with a single method with a single event type as parameter (the original Observer pattern used a single class with only one method, for simplicity).

Remote Events

Unlike the large number of event classes in the AWT and Swing, for example, Jini uses events of one type, the RemoteEvent, or a small number of subclasses of RemoteEvent. The RemoteEvent class has these public methods (and some inherited methods):

```
package net.jini.core.event;

public class RemoteEvent implements java.io.Serializable {
    public long getID();
    public long getSequenceNumber();
    public java.rmi.MarshalledObject getRegistrationObject();
}
```

Events in Beans and AWT convey complex object state information, and this is enough for the listeners to act with full knowledge of the changes that have caused the event to be generated. Jini events avoid this, and convey just enough information to allow state information to be found if needed. A remote event is serializable and is moved around the network to its listeners. The listeners then have to decide whether or not they need more detailed information than the simple information in each remote event. If they do need more information, they will have to contact the event source to get it.

AWT events, such as `MouseEvent`, contain an `id` field that is set to values such as `MOUSE_PRESSED` or `MOUSE_RELEASED`. These are not seen by the AWT programmer because the AWT event dispatch system uses the `id` field to choose appropriate methods, such as `mousePressed()` or `mouseReleased()`. Jini does not make these assumptions about event dispatch, and just gives you the identifier. Either the source or the listener (or both) will know what this value means. For example, a file classifier that can update its knowledge of MIME types could have message types `ADD_TYPE` and `REMOVE_TYPE` to reflect the sort of changes it is going through.

In a synchronous system with no losses, both sides of an interaction can keep consistent ideas of state and order of events. In a network system this is not so easy. Jini makes no assumptions about guarantees of delivery and does not even assume that events are delivered in order. The Jini event mechanism does not specify how events get from producer to listener—it could be by RMI calls, but it may be through an unreliable third party. The event source supplies a sequence number that could be used to construct state and ordering information if needed, and this generalizes things such as time-stamps on mouse events. For example, a message with `id` of `ADD_TYPE` and sequence number of `10` could correspond to the state change "added MIME type text/xml for files with suffix .xml." Another event with `id` of `REMOVE_TYPE` and sequence number of `11` would be taken as a later event, even if it arrived earlier. The listener will receive the event with id and sequence number only. Either this will be meaningful to the listener, or it will need to contact the event source and ask for more information about that sequence number. The event source should be able to supply state information upon request, given the sequence number.

An idea borrowed from systems such as the Xt Intrinsics and Motif is called *handback* data. This is a piece of data that is given by the listener to the event source at the time it registers itself for events. The event source records this handback and then returns it to the listener with each event. This handback can be a reminder of listener state at the time of registration.

This can be a little difficult to understand at first. The listener is basically saying to the event source that it wants to be told whenever something interesting happens, but when that does happen, the listener may have forgotten why it was interested in the first place, or what it intended to do with the information. So the listener also the gives the event source some extra information that it wants returned as a "reminder."

For example, a Jini taxi-driver might register interest in taxi-booking events from the base station while passing through a geographical area. It registers itself as a listener for booking events, and as part of its registration, it could include its current location. Then, when it receives a booking event, it is told its old location, and it could check to see if it is still interested in events from that old location. A more novel possibility is that one object could register a different object for events, so your stockbroker could register you for events about stock movements, and when you receive an event, you would also get a reminder about who registered your interest (plus a request for commission...).

Event Registration

Jini does not say how to register listeners with objects that can generate events. This is unlike other event models in Java that specify methods, like this

```
public void addActionListener(ActionListener listener);
```

for ActionEvent generators. What Jini does do is to specify a convenience class as a return value from this registration. This is the convenience class EventRegistration:

```
package net.jini.core.event;
import net.jini.core.lease.Lease;

public class EventRegistration implements java.io.Serializable {
    public EventRegistration(long eventID, Object source,
                             Lease lease, long seqNum);
    public long getID();
    public Object getSource();
    public Lease getLease();
    public long getSequenceNumber();
}
```

This return object contains information that may *be* of value to the object that registered a listener. Each registration will typically only be for a limited amount of time, and this information may be returned in the Lease object. If the event registration was for a particular type, this may be returned in the id field. A sequence number may also be given. The meaning of these values may depend on the particular system—in other words, Jini gives you a class that is optional in use, and whose fields are not tightly specified. This gives you the freedom to choose your own meanings to some extent.

This means that as the programmer of a event producer, you have to define (and implement) methods such as these:

```
public EventRegistration addRemoteEventListener(RemoteEventListener listener);
```

There is no standard interface for this.

Listener List

Each listener for remote events must implement the RemoteEventListener interface:

```
public interface RemoteEventListener
                extends java.rmi.Remote, java.util.EventListener {
    public void notify(RemoteEvent theEvent)
               throws UnknownEventException,
                      java.rmi.RemoteException;
}
```

Because it extends Remote, the listener will most likely be something like an RMI stub for a remote object, so that calling notify() will result in a call on the remote object, with the event being passed across to it.

In event generators, there are multiple implementations for handling lists of event listeners all the way through the Java core and extensions. There is no public API for dealing with event-listener lists, and so the programmer has to reinvent (or copy) code to pass events to listeners. There are basically two cases:

- Only one listener can be in the list.
- Any number of listeners can be in the list.

Single Listener

The case where there is only one listener allowed in the list can be implemented by using a single-valued variable, as shown in Figure 14-1.

This is the simplest case of event registration:

```
protected RemoteEventListener listener = null;

public EventRegistration addRemoteListener(RemoteEventListener listener)
       throws java.util.TooManyListenersException {
```

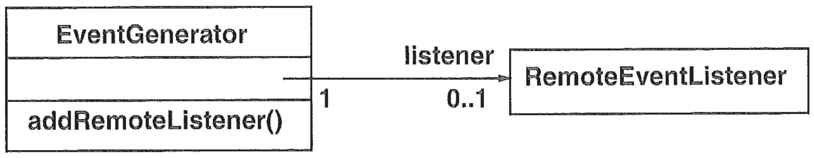


Figure 14-1. A single listener

```
    if (this.listener == null {
        this.listener = listener;
    } else {
        throw new java.util.TooManyListenersException();
    }
    return new EventRegistration(0L, this, null, 0L);
}
```

This is close to the ordinary Java event registration—no really useful information is returned that wasn't known before. In particular, there is no lease object, so you could probably assume that the lease is being granted "forever," as would be the case with non-networked objects.

When an event occurs, the listener can be informed by the event generator calling `fireNotify()`:

```
protected void fireNotify(long eventID,
                          long seqNum) {
    if (listener == null) {
        return;
    }

    RemoteEvent remoteEvent = new RemoteEvent(this, eventID,
                                              seqNum, null);
    listener.notify(remoteEvent);
}
```

It is easy to add a handback to this: just add another field to the object, and set and return this object in the registration and notify methods. Far more complex is adding a non-null lease. Firstly, the event source has to decide on a lease policy, that is, for what periods of time it will grant leases. Then it has to implement a timeout mechanism to discard listeners when their leases expire. And finally, it has to handle lease renewal and cancellation requests, possibly using its lease policy again to make decisions. The landlord package would be of use here.

Multiple Listeners

For the case where there can be any number of listeners, the convenience class `javax.swing.event.EventListenerList` can be used. The object delegates some of the list handling to the convenience class, as shown in Figure 14-2.

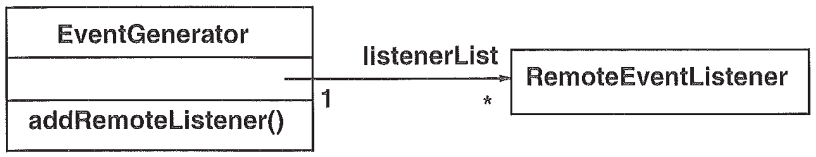


Figure 14-2. Multiple listeners

A version of event registration suitable for ordinary events is as follows:

```
import javax.swing.event.EventListenerList;

EventListenerList listenerList = new EventListenerList();

public EventRegistration addRemoteListener(RemoteEventListener l) {
    listenerList.add(RemoteListener.class, l);
    return new EventRegistration(0L, this, null, 0L);
}

public void removeRemoteListener(RemoteEventListener l) {
    listenerList.remove(RemoteListener.class, l);
}

// Notify all listeners that have registered interest for
// notification on this event type.  The event instance
// is lazily created using the parameters passed into
// the fire method.

protected void fireNotify(long eventID,
                          long seqNum) {
    RemoteEvent remoteEvent = null;

    // Guaranteed to return a non-null array
    Object[] listeners = listenerList.getListenerList();

    // Process the listeners last to first, notifying
    // those that are interested in this event
```

```
        for (int n = listeners.length - 2; n >= 0; n -= 2) {
            if (listeners[n] == RemoteEventListener.class) {
                RemoteEventListener listener =
                                (RemoteEventListener) listeners[n+1];
                if (remoteEvent == null) {
                    remoteEvent = new RemoteEvent(this, eventID,
                                                  seqNum, null);
                }
                try {
                    listener.notify(remoteEvent);
                } catch(UnknownEventException e) {
                    e.printStackTrace();
                } catch(java.rmi.RemoteException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

In this case, a source object need only call `fireNotify()` to send the event to all listeners. (You may decide that it is easier to simply use a `Vector` of listeners.)

It is again straightforward to add handbacks to this. The only tricky point is that each listener can have its own handback, so they will need to be stored in some kind of map (say a `HashMap`) keyed on the listener. Then, before `notify()` is called for each listener, the handback will need to be retrieved for the listener and a new remote event created with that handback.

Listener Source

The ordinary Java event model has all objects in a single address space, so that registration of event listeners and notifying these listeners all takes place using objects in the one space. We have already seen that this is not the case with Jini. Jini is a networked federation of objects, and in many cases one is dealing with proxy objects, not the real objects.

This is the same with remote events, except that in this case we often have the direction of proxies reversed. To see what I mean by this, consider what happens if a client wants to monitor any changes in the service. The client will already have a proxy object for the service, and it will use this proxy to register itself as a listener. However, the service proxy will most likely just hand this listener back off to the service itself (that is what proxies, such as RMI proxies, do). So we need to get a proxy for the client over to the service.

Consider the file classification problems we looked at in earlier chapters. The file classifier had a hard-coded set of filename extensions built in. However, it would be possible to extend these, if applications come along that know how to define (and maybe handle) such extensions. For example, an application would locate the file classification server, and using an exported method from the file classification interface would add the new MIME type and file extension. This is no departure from any standard Java or earlier Jini stuff. It only affects the implementation level of the file classifier, changing it from a static list of filename extensions to a more dynamic one.

What it does affect is the poor application that has been blocked (and is probably sleeping) on an unknown filename extension. When the classifier installs a new file type, it can send an event saying so. The blocked application could then try again to see if the extension is now known. If so, it uses it, and if not, it blocks again. Note that we don't bother with identifying the actual state change, since it is just as easy to make another query once you know that the state has changed. More complex situations may require more information to be maintained. However, in order to get to this situation, the application must have registered its interest in events, and the event producer must be able to find the listener.

How this gets resolved is for the client to first find the service in the same way as we discussed in Chapter 6. The client ends up with a proxy object for the service in the client's address space. One of the methods on the proxy will add an event listener, and this method will be called by the client.

For simplicity, assume that the client is being added as a listener to the service. The client will call the add listener method of the proxy, with the client as parameter. The proxy will then call the real object's add listener method, back on its server side. But in doing this, we have made a remote call across the network, and the client, which was local to the call on the proxy, is now remote to the real object, so what the real object is getting is a proxy to the client. When the service makes notification calls to the proxy listeners, the client's proxy can make a remote call back to the client itself. These proxies are shown in Figure 14-3.

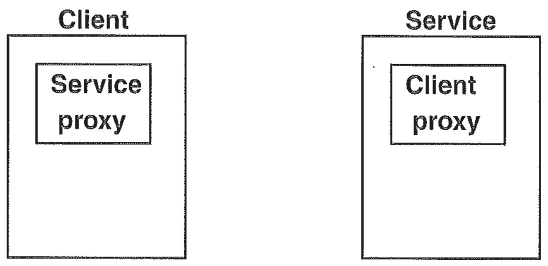


Figure 14-3. Proxies for services and listeners

File Classifier with Events

Let's make this discussion more concrete by looking at a new file classifier application that can have its set of mappings dynamically updated.

The first thing to be modified is the `FileClassifier` interface. This needs to be extended to a `MutableFileClassifier` interface, known to all objects. This new interface adds methods that will add and remove types, and that will also register listeners for events. The event types are labeled with two constants. The listener model is simple, and does not include handbacks or leases. The sequence identifier must be increasing, so we just add 1 on each event generation, although we don't really need it here: it is easy for a listener to just make MIME type queries again.

```
package common;

import java.io.Serializable;

/**
 * MutableFileClassifier.java
 */

import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.EventRegistration;

public interface MutableFileClassifier extends FileClassifier {

    static final public long ADD_TYPE = 1;
    static final public long REMOVE_TYPE = 2;

    /*
     * Add the MIME type for the given suffix.
     * The suffix does not contain '.' e.g. "gif".
     * Overrides any previous MIME type for that suffix
     */
    public void addType(String suffix, MIMEType type)
        throws java.rmi.RemoteException;

    /*
     * Delete the MIME type for the given suffix.
     * The suffix does not contain '.' e.g. "gif".
     * Does nothing if the suffix is not known
     */
    public void removeMIMEType(String suffix, MIMEType type)
        throws java.rmi.RemoteException;
    public EventRegistration addRemoteListener(RemoteEventListener listener)
        throws java.rmi.RemoteException;


} // MutableFileClasssifier
```

The RemoteFileClassifier interface is known only to services, and it just changes its package and inheritance for any service implementation:

```
package mutable;

import common.MutableFileClassifier;
import java.rmi.Remote;

/**
 * RemoteFileClassifier.java
 */

public interface RemoteFileClassifier extends MutableFileClassifier, Remote {

} // RemoteFileClasssifier
```

Previous implementations of file classifier services (such as in Chapter 8) use a static list of if...then statements because they deal with a fixed set of types. For this implementation, where the set of mappings can change, we change the implementation to a dynamic map keyed on file suffixes. It manages the event listener list for multiple listeners in the simple way discussed earlier in this chapter, and it generates events whenever a new suffix/type is added or successfully removed. The following code is an implementation of the file classifier service with this alternative implementation and an event list:

```
package mutable;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.MarshalledObject;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.EventRegistration;
import java.rmi.RemoteException;
import net.jini.core.event.UnknownEventException ;

import javax.swing.event.EventListenerList;

import common.MIMEType;
import common.MutableFileClassifier;
import java.util.Map;
import java.util.HashMap;

/**
 * FileClassifierImpl.java
 */

public class FileClassifierImpl extends  UnicastRemoteObject
                                implements RemoteFileClassifier {

    /**
     * Map of String extensions to MIME types
     */
    protected Map map = new HashMap();

    /**
     * Listeners for change events
     */
    protected EventListenerList listenerList = new EventListenerList();

    protected long seqNum = 0L;

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        System.out.println("Called with " + fileName);

        MIMEType type;
        String fileExtension;
        int dotIndex = fileName.lastIndexOf('.');

        if (dotIndex == -1 || dotIndex + 1 == fileName.length()) {
            // can't find suitable suffix
            return null;
        }

        fileExtension= fileName.substring(dotIndex + 1);
        type = (MIMEType) map.get(fileExtension);
        return type;

    }

    public void addType(String suffix, MIMEType type)
        throws java.rmi.RemoteException {
```

```
        map.put(suffix, type);
        fireNotify(ADD_TYPE);
    }

    public void removeMIMEType(String suffix, MIMEType type)
        throws java.rmi.RemoteException {
        if (map.remove(suffix) != null) {
            fireNotify(REMOVE_TYPE);
        }
    }

    public EventRegistration addRemoteListener(RemoteEventListener listener)
        throws java.rmi.RemoteException {
        listenerList.add(RemoteEventListener.class, listener);

        return new EventRegistration(0, this, null, 0);
    }

    // Notify all listeners that have registered interest for
    // notification on this event type.  The event instance
    // is lazily created using the parameters passed into
    // the fire method.

    protected void fireNotify(long eventID) {
        RemoteEvent remoteEvent = null;

        // Guaranteed to return a non-null array
        Object[] listeners = listenerList.getListenerList();

        // Process the listeners last to first, notifying
        // those that are interested in this event
        for (int i = listeners.length - 2; i >= 0; i -= 2) {
            if (listeners[i] == RemoteEventListener.class) {
                RemoteEventListener listener = (RemoteEventListener) listeners[i+1];
                if (remoteEvent == null) {
                    remoteEvent = new RemoteEvent(this, eventID,
                                                  seqNum++, null);
                }
                try {
                    listener.notify(remoteEvent);
                } catch(UnknownEventException e) {
                    e.printStackTrace();
                } catch(RemoteException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public FileClassifierImpl()  throws java.rmi.RemoteException {
        // load a predefined set of MIME type mappings
        map.put("gif", new MIMEType("image", "gif"));
        map.put("jpeg", new MIMEType("image", "jpeg"));
        map.put("mpg", new MIMEType("video", "mpeg"));
        map.put("txt", new MIMEType("text", "plain"));
        map.put("html", new MIMEType("text", "html"));
    }
} // FileClassifierImpl
```

The proxy changes its inheritance, and as a result has more methods to implement, which it just delegates to its server object. The following class is for the proxy:

```
package mutable;

import common.MutableFileClassifier;
import common.MIMEType;

import java.io.Serializable;
import java.io.IOException;
import java.rmi.Naming;

import net.jini.core.event.EventRegistration;
import net.jini.core.event.RemoteEventListener;

/**
 * FileClassifierProxy
 */

public class FileClassifierProxy implements MutableFileClassifier, Serializable {

    RemoteFileClassifier server = null;

    public FileClassifierProxy(FileClassifierImpl serv) {
        this.server = serv;
        if (serv==null) System.err.println("server is null");
    }

    public MIMEType getMIMEType(String fileName)
```

```
        throws java.rmi.RemoteException {
        return server.getMIMEType(fileName);
    }

    public void addType(String suffix, MIMEType type)
        throws java.rmi.RemoteException {
        server.addType(suffix, type);
    }

    public void removeMIMEType(String suffix, MIMEType type)
        throws java.rmi.RemoteException {
        server.removeMIMEType(suffix, type);
    }

    public EventRegistration addRemoteListener(RemoteEventListener listener)
        throws java.rmi.RemoteException {
        return server.addRemoteListener(listener);
    }

} // FileClassifierProxy
```

Monitoring Changes in Services

Services will start and stop. When they start, they will inform the lookup services, and sometime after they stop, they will be removed from the lookup services. However, there are a lot of times when other services or clients will want to know when services start or are removed. For example, an editor may want to know if a disk service has started so that it can save its file; a graphics display program may want to know when printer services start up; the user interface for a camera may want to track changes in disk and printer services so that it can update the Save and Print buttons.

A service registrar acts as a generator of `ServiceEvent` type events, which subclass from `RemoteEvent`. These events are generated in response to changes in the state of services that match (or fail to match) a template pattern for services. This event type has three categories from the `ServiceEvent.getTransition()` method:

- `TRANSITION_NOMATCH_MATCH`: A service has changed state so that whereas it previously did not match the template, now it does. In particular, if it didn't exist before, now it does. This transition type can be used to spot new services starting or to spot wanted changes in the attributes of an existing registered service; for example, an offline printer can change attributes to being online, which now makes it a useful service.

- TRANSITION_MATCH_NOMATCH: A service has changed state so that whereas it previously did match the template, now it doesn't. This can be used to detect when services are removed from a lookup service. This transition can also be used to spot changes in the attributes of an existing registered service that are not wanted; for example, an online printer can change attributes to being offline.

- TRANSITION_MATCH_MATCH: A service has changed state, but it matched both before and after. This typically happens when an Entry value changes, and it is used to monitor changes of state, such as a printer running out of paper, or a piece of hardware signaling that it is due for maintenance work.

A client that wants to monitor changes of services on a lookup service must first create a template for the types of services it is interested in. A client that wants to monitor all changes could prepare a template such as this:

```
ServiceTemplate templ = new ServiceTemplate(null, null, null); // or
ServiceTemplate templ = new ServiceTemplate(null, new Class[] {}, new Entry[] {});
// or
ServiceTemplate templ = new ServiceTemplate(null, new Class[] {Object.class},
null);
```

It then sets up a transition mask as a bit-wise OR of the three service transitions, and then calls notify() on the ServiceRegistrar object. The following is a program to monitor all changes.

```
/**
 * RegistrarObserver.java
 */

package observer;

import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.lookup.ServiceEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lease.Lease;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceID;
import net.jini.core.event.EventRegistration;
// import com.sun.jini.lease.LeaseRenewalManager; // Jini 1.0
import net.jini.lease.LeaseRenewalManager;        // Jini 1.1
import net.jini.core.lookup.ServiceMatches;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import net.jini.core.entry.Entry;
import net.jini.core.event.UnknownEventException;

public class RegistrarObserver extends UnicastRemoteObject implements
                              RemoteEventListener {

    protected static LeaseRenewalManager leaseManager = new LeaseRenewalManager();
    protected ServiceRegistrar registrar;

    protected final int transitions = ServiceRegistrar.TRANSITION_MATCH_NOMATCH |
                                  ServiceRegistrar.TRANSITION_NOMATCH_MATCH |
                                  ServiceRegistrar.TRANSITION_MATCH_MATCH;

    public RegistrarObserver() throws RemoteException {
    }

    public RegistrarObserver(ServiceRegistrar registrar) throws RemoteException {
        this.registrar = registrar;
        ServiceTemplate templ = new ServiceTemplate(null, null, null);
        EventRegistration reg = null;
        try {
            // eventCatcher = new MyEventListener();
            reg = registrar.notify(templ,
                               transitions,
                               this,
                               null,
                               Lease.ANY);
            System.out.println("notified id " + reg.getID());
        } catch(RemoteException e) {
            e.printStackTrace();
        }
        leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, null);
    }

    public void notify(RemoteEvent evt)
        throws RemoteException, UnknownEventException {
        try {
            ServiceEvent sevt = (ServiceEvent) evt;
            int transition = sevt.getTransition();
            System.out.println("transition " + transition);
            switch (transition) {
            case ServiceRegistrar.TRANSITION_NOMATCH_MATCH:
                System.out.println("nomatch -> match");
                break;
            case ServiceRegistrar.TRANSITION_MATCH_MATCH:
                System.out.println("match -> match");
                break;
            case ServiceRegistrar.TRANSITION_MATCH_NOMATCH:
                System.out.println("match -> nomatch");
                break;
            }
            System.out.println(sevt.toString());
            if (sevt.getServiceItem() == null) {
                System.out.println("now null");
            } else {
                Object service = sevt.getServiceItem().service;
                System.out.println("Service is " + service.toString());
            }
        } catch(Exception e) {
            e.printStackTrace();
        }

    }

} // RegistrarObserver
```

The following is a suitable driver for the preceding observer class:

```
package client;

import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
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.lookup.ServiceMatches;

import java.util.Vector;
import observer.RegistrarObserver;

/**
 * ReggieMonitor.java
 */

public class ReggieMonitor implements DiscoveryListener {
```

```
protected Vector observers = new Vector();

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

    // stay around long enough to receive replies
    try {
        Thread.currentThread().sleep(100000L);
    } catch(java.lang.InterruptedException e) {
        // do nothing
    }
}

public ReggieMonitor() {
    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();

    for (int n = 0; n < registrars.length; n++) {
        System.out.println("Service lookup found");
        ServiceRegistrar registrar = registrars[n];
        if (registrar == null) {
            System.out.println("registrar null");
            continue;
        }
        try {
            System.out.println("Lookup service at " +
                          registrar.getLocator().getHost());
        } catch(RemoteException e) {
                System.out.println("Lookup service info unavailable");
            }

            try {
                observers.add(new RegistrarObserver(registrar));
            } catch(RemoteException e) {
                System.out.println("adding observer failed");
            }

            ServiceTemplate templ = new ServiceTemplate(null, new Class[]
{Object.class}, null);
            ServiceMatches matches = null;
            try {
                matches = registrar.lookup(templ, 10);
            } catch(RemoteException e) {
                System.out.println("lookup failed");
            }

            for (int m = 0; m < matches.items.length; m++) {
                if (matches.items[m] != null && matches.items[m].service != null) {
                    System.out.println("Reg knows about " + matches.items[m].ser-
vice.toString() +
                                       " with id " + matches.items[m].serviceID);
                }
            }

        }
    }

    public void discarded(DiscoveryEvent evt) {
        // remove observer
    }
} // ReggieMonitor
```

Summary

This chapter has looked at how the remote event differs from the other event models in Java and at how to create and use them. Jini events allow distributed components to inform other components when they change state and to supply enough support information for listeners to determine the nature of the change. This adds an asynchronous state-change mechanism to Jini, which can allow more flexible systems to be built.

CHAPTER 15

ServiceDiscoveryManager

Both clients and services need to find lookup services. Both can do this using low-level core classes, or discovery utilities such as LookupDiscoveryManager. Once a lookup service is found, a service just needs to register with it and try to keep the lease alive for as long as it wants to. A service can make use of the JoinManager class for this.

The ServiceDiscoveryManager class performs client-side functions similar to that of JoinManager for services, and simplifies the task of finding services. The ServiceDiscoveryManager class is only available in Jini 1.1.

ServiceDiscoveryManager Interface

The ServiceDiscoveryManager class is a utility class designed to help in the various client-side lookup cases that can occur:

- A client may wish to use a service immediately or later.
- A client may want to use multiple services.
- A client will want to find services by their interfaces, but may also want to apply additional criteria, such as being a "fast enough" printer.
- A client may just wish to use a service if it is available at the time of the request, but alternatively may want to be informed of new services becoming available and to respond to this new availability (for example, a service browser).

Due to the variety of possible cases, the ServiceDiscoveryManager class is more complex than JoinManager. Its interface includes the following:

```
package net.jini.lookup;

public class ServiceDiscoveryManager {
    public ServiceDiscoveryManager(DiscoveryManagement discoveryMgr,
                                   LeaseRenewalManager leaseMgr)
          throws IOException;
    LookupCache createLookupCache(ServiceTemplate tmpl,
                                  ServiceItemFilter filter,
                                  ServiceDiscoveryListener listener);

    ServiceItem[] lookup(ServiceTemplate tmpl,
                         int maxMatches, ServiceItemFilter filter);

    ServiceItem lookup(ServiceTemplate tmpl,
                       ServiceItemFilter filter);

    ServiceItem lookup(ServiceTemplate tmpl,
                       ServiceItemFilter filter, long wait);

    ServiceItem[] lookup(ServiceTemplate tmpl,
                         int minMaxMatch, int maxMatches,
                         ServiceItemFilter filter, long wait);

    void terminate();
}
```

ServiceItemFilter Interface

Most methods of the client lookup manager require a ServiceItemFilter. This is a simple interface designed to be an additional filter on the client side to help in finding services. The primary way for a client to find a service is to ask for an instance of an interface, possibly with additional entry attributes. This matching is performed on the lookup service, and it only involves a form of exact pattern matching. It allows the client to ask for a toaster that will handle two slices of toast exactly, but not for one that will toast two or more.

Performing arbitrary Boolean matching on the lookup service raises a security issue as it would involve running some code from the client or service in the lookup service, and it also raises a possible performance issue for the lookup service. This means that enhancing the matching process in the lookup service is unlikely to ever occur, so any more sophisticated matching must be done by the client. The ServiceItemFilter allows additional Boolean filtering to be performed on the client side, by client code, so these issues are local to the client only.

The ServiceItemFilter interface is as follows:

```
package net.jini.lookup;

public interface ServiceItemFilter {
    boolean check(ServiceItem item);
}
```

A client filter will implement this interface to perform additional checking.

Client-side filtering will not solve all of the problems of locating the "best" service. Some situations will still require other services that know "local" information, such as distances in a building.

Finding a Service Immediately

The simplest scenario for a client is that it wants to find a service immediately, use it, and then (perhaps) terminate. The client will be prepared to wait a certain amount of time before giving up. All issues of discovery can be given to the `ServiceDiscoveryManager`, and the task of finding a service can be given to a method such as `lookup()` with a `wait` parameter. The `lookup()` method will block until a suitable service is found or the time limit is reached. If the time limit is reached, a `null` object will be returned; otherwise a non-`null` service object will be returned.

```
package client;

import common.FileClassifier;
import common.MIMEType;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;

/**
 * ImmediateClientLookup.java
 */

public class ImmediateClientLookup {

    private static final long WAITFOR = 100000L;

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

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(2*WAITFOR);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public ImmediateClientLookup() {
        ServiceDiscoveryManager clientMgr = null;

        System.setSecurityManager(new RMISecurityManager());

        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            clientMgr = new ServiceDiscoveryManager(mgr,
                                               new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        Class [] classes = new Class[] {FileClassifier.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        ServiceItem item = null;
        // Try to find the service, blocking till timeout if necessary
        try {
            item = clientMgr.lookup(template,
                                    null, /* no filter */
                                    WAITFOR /* timeout */);
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        if (item == null) {
            // couldn't find a service in time
            System.out.println("no service");
            System.exit(1);
        }

        // Get the service
        FileClassifier classifier = (FileClassifier) item.service;
        if (classifier == null) {
            System.out.println("Classifier null");
            System.exit(1);
        }

        // Now we have a suitable service, use it
        MIMEType type;
        try {
            String fileName;

            // Try several file types: .txt, .rtf, .abc
            fileName = "file1.txt";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);

            fileName = "file2.rtf";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);

            fileName = "file3.abc";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);
        } catch(java.rmi.RemoteException e) {
            System.err.println(e.toString());
        }
        System.exit(0);
    }

    private void printType(String fileName, MIMEType type) {
        System.out.print("Type of " + fileName + " is ");
        if (type == null) {
            System.out.println("null");
        } else {
            System.out.println(type.toString());
        }
    }
} // ImmediateClientLookup
```

Using a Filter

An example in Chapter 13 discussed how to select a printer with a speed greater than a certain value. This type of problem is well suited to the `ServiceDiscoveryManager` using a ServiceItemFilter. The ServiceItemFilter interface has a check() method, which is called on the client side to perform additional filtering of services. This method can accept or reject a service based on criteria supplied by the client.

The following program illustrates how this check() method can be used to select only printer services with a speed greater than 24 pages per minute:

```
package client;

import common.Printer;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.ServiceItemFilter;
/**
 * TestPrinterSpeedFilter.java
 */

public class TestPrinterSpeedFilter implements ServiceItemFilter {
    private static final long WAITFOR = 100000L;

    public TestPrinterSpeedFilter() {
        ServiceDiscoveryManager clientMgr = null;

        System.setSecurityManager(new RMISecurityManager());

        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            clientMgr = new ServiceDiscoveryManager(mgr,
                                               new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        Class[] classes = new Class[] {Printer.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
        ServiceItem item = null;
        try {
            item = clientMgr.lookup(template,
                                    this, /* filter */
                                    WAITFOR /* timeout */);
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        if (item == null) {
            // couldn't find a service in time
            System.exit(1);
        }

        Printer printer = (Printer) item.service;
        // Now use the printer
        // ...
    }

    public boolean check(ServiceItem item) {
        // This is the filter
        Printer printer = (Printer) item.service;
        if (printer.getSpeed() > 24) {
            return true;
        } else {
            return false;
        }
    }

    public static void main(String[] args) {

        TestPrinterSpeed f = new TestPrinterSpeed();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(2*WAITFOR);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

} // TestPrinterSpeed
```

Building a Cache of Services

A client may wish to make use of a service multiple times. If the client simply found a suitable reference to a service, then before each use it would have to check whether the reference was still valid, and if not, it would need to find another one. The client may also want to use minor variants of a service, such as a fast printer one time and a slow one the next. While this management can be done easily enough in each case, the ServiceDiscoveryManager can supply a cache of services that will do this work for you. This cache will monitor lookup services to keep the cache as up-to-date as possible.

The cache is defined as an interface:

```
package net.jini.lookup;

public interface LookupCache {
    public ServiceItem lookup(ServiceItemFilter filter);
    public ServiceItem[] lookup(ServiceItemFilter filter,
                                int maxMatches);
    public void addListener(ServiceDiscoveryListener l);
    public void removeListener(ServiceDiscoveryListener l);
    public void discard(Object serviceReference);
    void terminate();
}
```

A suitable implementation object can be created by the ServiceDiscoveryManager method:

```
LookupCache createLookupCache(ServiceTemplate tmpl,
                              ServiceItemFilter filter,
                              ServiceDiscoveryListener listener);
```

We will ignore the ServiceDiscoveryListener until the next section of this chapter. It can be set to null in createLookupCache().

The LookupCache created by createLookupCache() takes a template for matching against interface and entry attributes. In addition, it also takes a filter to perform additional client-side Boolean filtering of services. The cache will then maintain a set of references to services matching the template and passing the filter. These references are all local to the client and consist of the service proxies and their attributes downloaded to the client. Searching for a service can then be done by local methods: LookupCache.lookup(). These can take an additional filter that can be used to further refine the set of services returned to the client.

The search in the cache is done directly on the proxy services and attributes already found by the client, and does not involve querying lookup services.

Essentially, this involves a tradeoff of lookup service activity while the client is idle to produce fast local response when the client is active.

There are versions of `ServiceDiscoveryManager.lookup()` with a time parameter, which block until a service is found or the method times out. These methods do not use polling, but instead use event notification because they are trying to find services based on remote calls to lookup services. The `lookup()` methods of `LookupCache` do not implement such a blocking call because the methods run purely locally, and it is reasonable to poll the cache for a short time if need be.

Here is a version of the file classifier client that creates and examines the cache for a suitable service:

```
package client;

import common.FileClassifier;
import common.MIMEType;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.lookup.LookupCache;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;

/**
 * CachedClientLookup.java
 */

public class CachedClientLookup {

    private static final long WAITFOR = 100000L;

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

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

```
public CachedClientLookup() {
    ServiceDiscoveryManager clientMgr = null;
    LookupCache cache = null;

    System.setSecurityManager(new RMISecurityManager());

    try {
        LookupDiscoveryManager mgr =
            new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                       null /* unicast locators */,
                                       null /* DiscoveryListener */);
        clientMgr = new ServiceDiscoveryManager(mgr,
                                       new LeaseRenewalManager());
    } catch(Exception e) {
        e.printStackTrace();
        System.exit(1);
    }

    Class [] classes = new Class[] {FileClassifier.class};
    ServiceTemplate template = new ServiceTemplate(null, classes,
                                                   null);


    try {
        cache = clientMgr.createLookupCache(template,
                                            null, /* no filter */
                                            null /* no listener */);
    } catch(Exception e) {
        e.printStackTrace();
        System.exit(1);
    }

    // loop until we find a service
    ServiceItem item = null;
    while (item == null) {
        System.out.println("no service yet");
        try {
            Thread.currentThread().sleep(1000);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
        // see if a service is there now
        item = cache.lookup(null);
    }
    FileClassifier classifier = (FileClassifier) item.service;
```

```
        if (classifier == null) {
            System.out.println("Classifier null");
            System.exit(1);
        }

        // Now we have a suitable service, use it
        MIMEType type;
        try {
            String fileName;

            fileName = "file1.txt";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);

            fileName = "file2.rtf";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);

            fileName = "file3.abc";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);
        } catch(java.rmi.RemoteException e) {
            System.err.println(e.toString());
        }
        System.exit(0);
    }

    private void printType(String fileName, MIMEType type) {
        System.out.print("Type of " + fileName + " is ");
        if (type == null) {
            System.out.println("null");
        } else {
            System.out.println(type.toString());
        }
    }
} // CachedClientLookup
```

Running the CachedClientLookup

While it is okay to poll the local cache, the cache itself must get its contents from lookup services, and in general it is not okay to poll these because that involves possibly heavy network traffic. The cache itself gets its information by registering itself as a listener for service events from the lookup services (as explained in Chapter 14).

The lookup services will then call `notify()` on the cache listener. This call is a remote call from the remote lookup service to the local cache, done (probably) using an RMI stub. In fact, the Sun implementation of `ServiceDiscoveryManager` uses a nested class, `ServiceDiscoveryManager.LookupCacheImpl.LookupListener`, which has an RMI stub.

In order for the cache to actually work, it is necessary to set the RMI codebase property, `java.rmi.server.codebase`, to a suitable location for the class files (such as an HTTP server), and to make sure that the class `net/jini/lookup/ServiceDiscoveryManager$LookupCacheImpl$LookupListener_Stub.class` is accessible from this codebase. The stub file may be found in the `lib/jini-ext.jar` library in the Jini 1.1 distribution. It has to be extracted from there and placed in the codebase using a command such as this:

```
unzip jini-ext.jar 'net/jini/lookup/ServiceDiscoveryManager$LookupCache-
Impl$LookupListener_Stub.class' -d /home/WWW/htdocs/classes
```

Note that the specification just says that this type of thing has to be done but does not descend to details about the class name—that is left to the documentation of the `ServiceDiscoveryManager` as implemented by Sun. If another implementation is made of the Jini classes, then it would probably use a different remote class.

Monitoring Changes to the Cache

The cache uses remote events to monitor the state of lookup services. It includes a local mechanism to pass some of these changes to a client by means of the `ServiceDiscoveryListener` interface:

```
package net.jini.lookup;
interface ServiceDiscoveryListener {
    void serviceAdded(ServiceDiscoveryEvent event);
    void serviceChanged(ServiceDiscoveryEvent event);
    void serviceRemoved(ServiceDiscoveryEvent event);
}
```

The `ServiceDiscoveryListener` methods take a parameter of type `ServiceDiscoveryEvent`. This class has methods:

```
package net.jini.lookup;

class ServiceDiscoveryEvent extends EventObject {
    ServiceItem getPostEventServiceItem();
    ServiceItem getPreEventServiceItem();
}
```

Clients are not likely to be interested in all events generated by lookup services, even for the services in which they are interested. For example, if a new service registers itself with ten lookup services, they will all generate transition events from `NO_MATCH` to `MATCH`, but the client will usually only be interested in seeing the first of these—the other nine are just repeated information. Similarly, if a service's lease expires from one lookup service, then that doesn't matter much; but if it expires from all lookup services that the client knows of, then it does matter, because the service is no longer available to it. The cache consequently prunes events so that the client gets information about the real services rather than information about the lookup services.

In Chapter 14, an example was given on monitoring changes to services from a lookup service viewpoint, reporting each change to lookup services. A client-oriented view just monitors changes in services themselves, which can be done easily using `ServiceDiscoveryEvent` objects:

```
package client;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.lookup.ServiceDiscoveryListener;
import net.jini.lookup.ServiceDiscoveryEvent;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceItem;
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.LookupCache;

/**
 * ServiceMonitor.java
 */

public class ServiceMonitor implements ServiceDiscoveryListener {

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

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(100000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }
```

```
public ServiceMonitor() {
    ServiceDiscoveryManager clientMgr = null;
    LookupCache cache = null;

    System.setSecurityManager(new RMISecurityManager());

    try {
        LookupDiscoveryManager mgr =
            new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                       null /* unicast locators */,
                                       null /* DiscoveryListener */);
        clientMgr = new ServiceDiscoveryManager(mgr,
                                       new LeaseRenewalManager());
    } catch(Exception e) {
        e.printStackTrace();
        System.exit(1);
    }

    ServiceTemplate template = new ServiceTemplate(null, null,
                                                   null);
    try {
        cache = clientMgr.createLookupCache(template,
                                            null, /* no filter */
                                            this /* listener */);
    } catch(Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}

// methods for ServiceDiscoveryListener
public void serviceAdded(ServiceDiscoveryEvent evt) {
    // evt.getPreEventServiceItem() == null
    ServiceItem postItem = evt.getPostEventServiceItem();
    System.out.println("Service appeared: " +
                       postItem.service.getClass().toString());
}

public void serviceChanged(ServiceDiscoveryEvent evt) {
    ServiceItem preItem = evt.getPostEventServiceItem();
    ServiceItem postItem = evt.getPreEventServiceItem() ;
    System.out.println("Service changed: " +
                       postItem.service.getClass().toString());
    }
    public void serviceRemoved(ServiceDiscoveryEvent evt) {
        // evt.getPostEventServiceItem() == null
        ServiceItem preItem = evt.getPreEventServiceItem();
        System.out.println("Service disappeared: " +
                           preItem.service.getClass().toString());
    }

} // ServiceMonitor
```

Summary

The client lookup manager can handle a variety of common situations that arise as clients need to find services under different situations.

CHAPTER 16

Transactions

TRANSACTIONS ARE A NECESSARY PART of many distributed operations. Frequently two or more objects may need to synchronize changes of state so that they all occur, or none occur. This happens in situations such as control of ownership, where one party has to give up ownership at the same time as another asserts ownership. What has to be avoided is only one party performing the action, which could result in the property having either no owners or two owners.

The theory of transactions often includes mention of the "ACID" properties:

- **Atomicity**: All the operations of a transaction must take place, or none of them do.
- **Consistency**: The completion of a transaction must leave the participants in a "consistent" state, whatever that means. For example, the number of owners of a resource must remain at one.
- **Isolation**: The activities of one transaction must not affect any other transactions.
- **Durability**: The results of a transaction must be persistent.

The practice of transactions is that they use the two-phase commit protocol. This requires that participants in a transaction are asked to "vote" on a transaction. If all participants agree to go ahead, then the transaction "commits," which is binding on all the participants. If any "abort" during this voting stage, this forces abortion of the transaction for all participants.

Jini has adopted the syntax of the two-phase commit method. It is up to the clients and services within a transaction to observe the ACID properties if they choose to do so. Jini essentially supplies the mechanism of two-phase commit and leaves the policy to the participants in a transaction.

Transaction Identifiers

Restricting Jini transactions to a two-phase commit model without associating a particular semantics to it means that a transaction can be represented in a simple way, as a `long` identifier. This identifier is obtained from a transaction manager and will uniquely label the transaction to that manager. (It is not guaranteed to be unique between managers, though—unlike service IDs.) All participants in the transaction communicate with the transaction manager using this identifier to label which transaction they belong to.

The participants in a transaction may disappear, or the transaction manager may disappear. As a result, transactions are managed by a lease, which will expire unless it is renewed. When a transaction manager is asked for a new transaction, it returns a `TransactionManager.Created` object, which contains the transaction identifier and lease:

```
public interface TransactionManager {
    public static class Created {
        public final long id;
        public final Lease lease;
    }
    ...
}
```

A `Created` object may be passed around between participants in the lease, and one of them will need to look after lease renewals. All the participants will use the transaction identifier in communication with the transaction manager.

TransactionManager

A transaction manager looks after the two-phase commit protocol for all the participants in a transaction. It is responsible for creating a new transaction with its `create()` method. Any of the participants can force the transaction to abort by calling `abort()`, or they can force it to the two-phase commit stage by calling `commit()`.

```
public interface TransactionManager {

    Created create(long leaseFor) throws ...;
    void join(long id, TransactionParticipant part,
              long crashCount) throws ...;
    void commit(long id) throws ...;
    void abort(long id) throws ...;
    ...
}
```

When a participant joins a transaction, it registers a listener of type `TransactionParticipant`. If any participant calls `commit()`, the transaction manager starts the voting process using all of these listeners. If all of these are prepared to commit, then the manager moves all of these listeners to the commit stage. Alternatively, any of the participants can call `abort()`, which forces all of the listeners to abort.

TransactionParticipant

When an object becomes a participant listener in a transaction, it allows the transaction manager to call various methods:

```
public interface TransactionParticipant ... {

    int prepare(TransactionManager mgr, long id) throws ...;
    void commit(TransactionManager mgr, long id) throws ...;
    void abort(TransactionManager mgr, long id) throws ...;
    int prepareAndCommit(TransactionManager mgr, long id) throws ...;
}
```

These methods are triggered by calls made upon the transaction manager. For example, if one client calls the transaction manager to abort, then the transaction manager calls all the listeners to abort.

The "normal" mode of operation (that is, when nothing goes wrong with the transaction) is for a call to be made on the transaction manager to commit. It then enters the two-phase commit stage where it asks each participant listener to first `prepare()` and then to either `commit()` or `abort()`.

Mahalo

`Mahalo` is a transaction manager supplied by Sun as part of the Jini distribution. It can be used without any changes. It runs as a Jini service, like `reggie`, and like all Jini services it has two parts: the part that runs as a server, needing its own set of class files in `mahalo.jar`, and the set of class files that need to be available to clients in `mahalo-dl.jar`. It also needs a security policy, an HTTP server, and log files.

`Mahalo` can be started using a command line like this:

```
java  -Djava.security.policy=policy.all \
      -Dcom.sun.jini.mahalo.managerName=TransactionManager \
      -jar /home/jan/tmpdir/jini1_0/lib/mahalo.jar \
      http://`hostname`:8080/mahalo-dl.jar \
      /home/jan/projects/jini/doc/policy.all \
      /tmp/mahalo_log public &
```

A Transaction Example

The classic use of transactions is to handle money transfers between accounts. In this scenario there are two accounts, one of which is debited and the other credited.

This is not a very exciting example, so we shall try a more complex situation. Suppose a service decides to charge for its use. If a client decides this cost is reasonable, it will first credit the service and then request that the service be performed.

The actual accounts will be managed by an Accounts service, which will need to be informed of the credits and debits that occur. A simple Accounts model is one in which the service gets some sort of customer ID from the client, and passes its own ID and the customer ID to the Accounts service, which manages both accounts. This is simple, it is prone to all sorts of e-commerce issues that we will not go into, and it is similar to the way credit cards work!

Figure 16-1 shows the messages in a normal sequence diagram. The client makes a `getCost()` call to the service and receives the `cost` in return. It then makes a `credit()` call on the service, which makes a `creditDebit()` call on the Accounts service before returning. The client then makes a final `requestService()` call on the service and gets back a `result`.

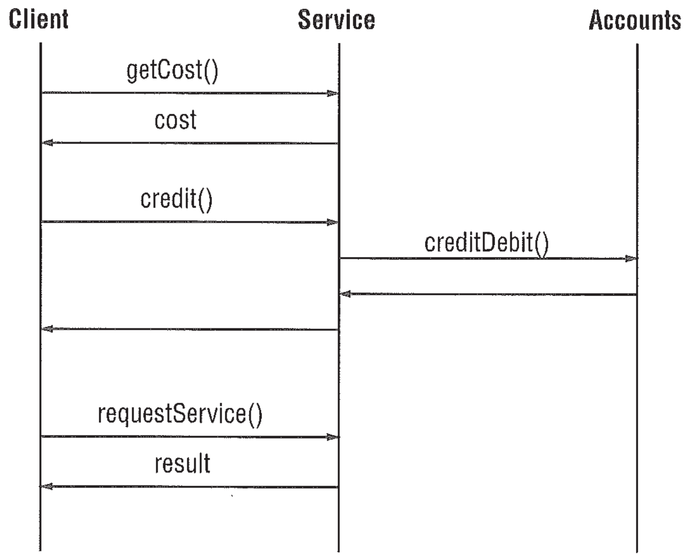


Figure 16-1. Sequence diagram for credit/debit example

There are a number of problems with the sequence of steps that can benefit by using a transaction model. The steps of `credit()` and `creditDebit()` should certainly be performed either both together or not at all. But in addition there is the issue of the quality of the service—suppose the client is not happy with the results from the service and would like to reclaim its money, or better yet, not spend it in the first case! If we include the delivery of the service in the transaction, then there is the opportunity for the client to abort the transaction before it is committed.

Figure 16-2 shows the larger set of messages in the sequence diagram for normal execution. As before, the client requests the cost from the service, and after getting this, it asks the transaction manager to create a transaction and receives back the transaction ID. It then joins the transaction itself. When it asks the service to credit an amount, the service also joins the transaction. The service then asks the account to `creditDebit()` the amount, and as part of this, the account also joins the transaction. The client then requests the service and gets the result. If all is fine, it then asks the transaction manager to `commit()`, which triggers the prepare-and-commit phase. The transaction manager asks each participant to `prepare()`, and if it gets satisfactory replies from each, it then asks each one to `commit()`.

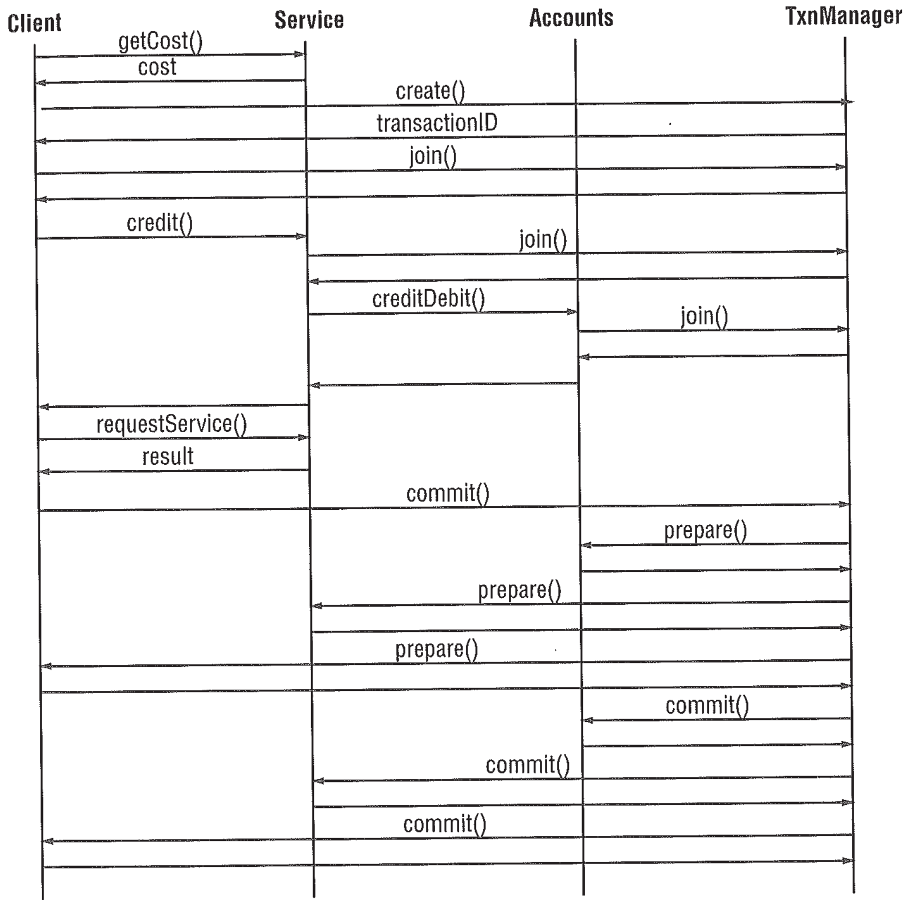


Figure 16-2. Sequence diagram for credit/debit example with transactions

There are several points of failure in this transaction:

- The cost may be too high for the client. However, at this stage the client has not created or joined a transaction, so this doesn't matter.
- The client may offer too little by way of payment to the service. The service can signal this by joining the transaction and then aborting it. This will ensure that the client has to roll back the transaction. (Of course, it could instead throw a `NotEnoughPayment` exception—joining and aborting is used for illustrating transaction possibilities.)
- There may be a time delay between finding the price and asking for the service. The price may have gone up in the meantime! The service would then abort the transaction, forcing the client and the accounts to roll back.
- After the service is performed, the client may decide that the result was not good enough, and refuse to pay. Aborting the transaction at this stage would cause the service and accounts to roll back.
- The Accounts service may abort the transaction if sufficient client funds are unavailable.

PayableFileClassifierImpl

The service we will use here is a version of the familiar file classifier that requires a payment before it will divulge the MIME type for a filename. A bit unrealistic, perhaps, but that doesn't matter for our purposes here.

There will be a `PayableFileClassifier` interface, which extends the `FileClassifier` interface. We will also make it extend the `Payable` interface, just in case we want to charge for other services. In line with other interfaces, we shall extend this to a `RemotePayableFileClassifier` and then implement this with a `PayableFileClassifierImpl`.

The `PayableFileClassifierImpl` can use the implementation of the `rmi.FileClassifierImpl`, so we shall make it extend this class. We also want it to be a participant in a transaction, so it must implement the `TransactionParticipant` interface. This leads to the inheritance diagram shown in Figure 16-3, which isn't really as complex as it looks.

The first new element in this hierarchy is the interface `Payable`:

```
package common;

import java.io.Serializable;
```

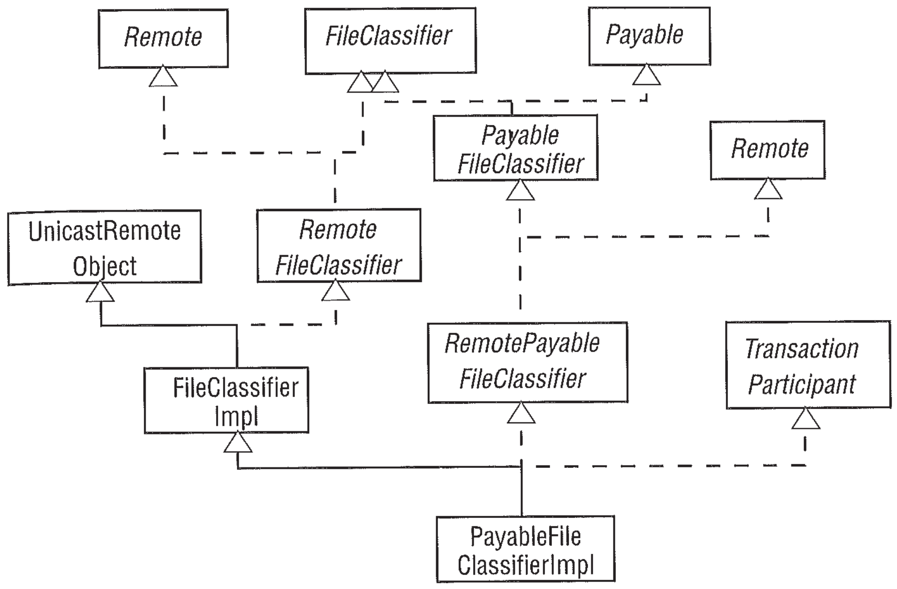


Figure 16-3. Class diagram for transaction participant

```
import net.jini.core.transaction.server.TransactionManager;

/**
 * Payable.java
 */

public interface Payable extends Serializable {

    void credit(long amount, long accountID,
                TransactionManager mgr,
                long transactionID)
        throws java.rmi.RemoteException;

    long getCost() throws java.rmi.RemoteException;
} // Payable
```

Extending `Payable` is the `PayableFileClassifier` interface:

```
package common;

/**
 * PayableFileClassifier.java
 */
```

```
public interface PayableFileClassifier extends FileClassifier, Payable {

} // PayableFileClassifier
```

PayableFileClassifier will be used by the client to search for the service. The service will use a RemotePayableFileClassifier, which is a simple extension to this:

```
package txn;

import common.PayableFileClassifier;
import java.rmi.Remote;

/**
 * RemotePayableFileClassifier.java
 */

public interface RemotePayableFileClassifier extends PayableFileClassifier, Remote
{

} // RemotePayableFileClasssifier
```

The implementation of this service joins the transaction, finds an Accounts service from a known location (using unicast lookup), registers the money transfer, and then performs the service. This implementation doesn't keep any state information that can be altered by the transaction. When asked to prepare() by the transaction manager it can just return NOTCHANGED. If there was state, the prepare() and commit() methods would have more content. The prepareAndCommit() method can be called by a transaction manager as an optimization, and the version given in this example follows the specification given in the "Jini Transaction" chapter of *The Jini Specification* by Ken Arnold et al. The following program gives this service implementation:

```
package txn;

import common.MIMEType;
import common.Accounts;
import rmi.FileClassifierImpl;
//import common.PayableFileClassifier;
//import common.Payable;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.transaction.server.TransactionParticipant;
import net.jini.core.transaction.server.TransactionConstants;
import net.jini.core.transaction.UnknownTransactionException;
import net.jini.core.transaction.CannotJoinException;
import net.jini.core.transaction.CannotAbortException;
import net.jini.core.transaction.server.CrashCountException;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;

/**
 * PayableFileClassifierImpl.java
 */

public class PayableFileClassifierImpl extends FileClassifierImpl
    implements RemotePayableFileClassifier, TransactionParticipant {

    protected TransactionManager mgr = null;
    protected Accounts accts = null;
    protected long crashCount = 0; // ???
    protected long cost = 10;
    protected final long myID = 54321;

    public PayableFileClassifierImpl() throws java.rmi.RemoteException {
        super();

        System.setSecurityManager(new RMISecurityManager());
    }

    public void credit(long amount, long accountID,
                       TransactionManager mgr,
                       long transactionID) {
        System.out.println("crediting");

        this.mgr = mgr;

        // before findAccounts
        System.out.println("Joining txn");
        try {
            mgr.join(transactionID, this, crashCount);
        } catch(UnknownTransactionException e) {
            e.printStackTrace();
        } catch(CannotJoinException e) {
            e.printStackTrace();
        } catch(CrashCountException e) {
                e.printStackTrace();
            } catch(RemoteException e) {
                e.printStackTrace();
            }
            System.out.println("Joined txn");


            findAccounts();

            if (accts == null) {
                try {
                    mgr.abort(transactionID);
                } catch(UnknownTransactionException e) {
                    e.printStackTrace();
                } catch(CannotAbortException e) {
                    e.printStackTrace();
                } catch(RemoteException e) {
                    e.printStackTrace();
                }
            }

            try {
                accts.creditDebit(amount, accountID, myID,
                                  transactionID, mgr);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
            }


        }

        public long getCost() {
            return cost;
        }

        protected void findAccounts() {
            // find a known account service
            LookupLocator lookup = null;
            ServiceRegistrar registrar = null;

            try {
                lookup = new LookupLocator("jini://localhost");
            } catch(java.net.MalformedURLException e) {
                System.err.println("Lookup failed: " + e.toString());
            System.exit(1);
        }

        try {
            registrar = lookup.getRegistrar();
        } catch (java.io.IOException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        } catch (java.lang.ClassNotFoundException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        }
        System.out.println("Registrar found");

        Class[] classes = new Class[] {Accounts.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
        try {
            accts = (Accounts) registrar.lookup(template);
        } catch(java.rmi.RemoteException e) {
            System.exit(2);
        }
    }

    public MIMEType getMIMEType(String fileName) throws RemoteException {

        if (mgr == null) {
            // don't process the request
            return null;
        }

        return super.getMIMEType(fileName);
    }


    public int prepare(TransactionManager mgr, long id) {
        System.out.println("Preparing...");
        return TransactionConstants.PREPARED;
    }

    public void commit(TransactionManager mgr, long id) {
        System.out.println("committing");

    }
    public void abort(TransactionManager mgr, long id) {
        System.out.println("aborting");
    }

    public int prepareAndCommit(TransactionManager mgr, long id) {
        int result = prepare(mgr, id);
        if (result == TransactionConstants.PREPARED) {
            commit(mgr, id);
            result = TransactionConstants.COMMITTED;
        }
        return result;
    }


} // PayableFileClassifierImpl
```

AccountsImpl

We shall assume that all accounts in this example are managed by a single Accounts service that knows about all accounts by using a long identifier. These should be stored in permanent form, and there should be proper crash-recovery mechanisms, etc. For simplicity, we shall just use a hash table of accounts, with uncommitted transactions kept in a "pending" list. When commitment occurs, the pending transaction takes place.

Figure 16-4 shows the Accounts class diagram.

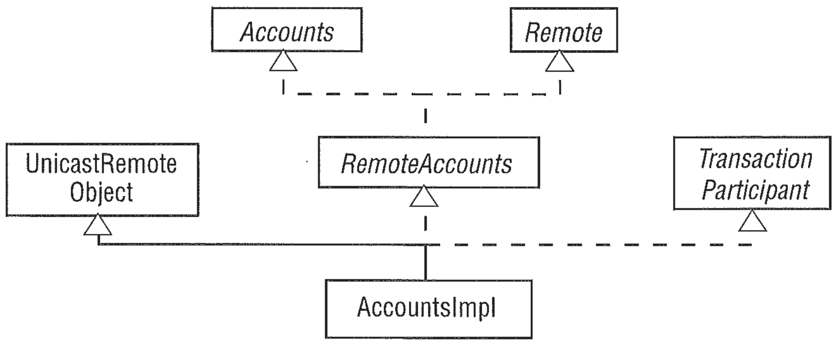


Figure 16-4. Class diagram for Accounts

The Accounts interface looks like this:

```
/**
 * Accounts.java
 */

package common;

import net.jini.core.transaction.server.TransactionManager;

public interface Accounts  {

    void creditDebit(long amount, long creditorID,
                     long debitorID, long transactionID,
                     TransactionManager tm)
        throws java.rmi.RemoteException;

} // Accounts
```

and this is the implementation:

```
/**
 * AccountsImpl.java
 */

package txn;

// import common.Accounts;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.transaction.server.TransactionParticipant;
import net.jini.core.transaction.server.TransactionConstants;
import java.rmi.server.UnicastRemoteObject;
import java.util.Hashtable;
// import java.rmi.RMISecurityManager;
// debug
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;
// end debug

public class AccountsImpl extends UnicastRemoteObject
    implements RemoteAccounts, TransactionParticipant, java.io.Serializable {

    protected long crashCount = 0; // value??
    protected Hashtable accountBalances = new Hashtable();
    protected Hashtable pendingCreditDebit = new Hashtable();

    public AccountsImpl() throws java.rmi.RemoteException {
        // System.setSecurityManager(new RMISecurityManager());
    }

    public void creditDebit(long amount, long creditorID,
                            long debitorID, long transactionID,
                            TransactionManager mgr) {

        // Ensure stub class is loaded by getting its class object.
        // It has to be loaded from the same place as this object
        java.rmi.Remote stub = null;
        try {
            stub = toStub(this);
        } catch(Exception e) {
            System.out.println("To stub failed");
            e.printStackTrace();
        }
        System.out.println("To stub found");
        String annote =
java.rmi.server.RMIClassLoader.getClassAnnotation(stub.getClass());
        System.out.println("from " + annote);
        try {
            Class cl = java.rmi.server.RMIClassLoader.loadClass(annote,
                                                   "txn.AccountsImpl_Stub");
        } catch(Exception e) {
            System.out.println("To stub class failed");
            e.printStackTrace();
        }
        System.out.println("To stub class ok");

        // mgr = findManager();
        try {
            System.out.println("Trying to join");
            mgr.join(transactionID, this, crashCount);
        } catch(net.jini.core.transaction.UnknownTransactionException e) {
            e.printStackTrace();
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
        } catch(net.jini.core.transaction.server.CrashCountException e) {
            e.printStackTrace();
        } catch(net.jini.core.transaction.CannotJoinException e) {
            e.printStackTrace();
        }
        System.out.println("joined");
        pendingCreditDebit.put(new TransactionPair(mgr,
                                                   transactionID),
                               new CreditDebit(amount, creditorID,
                                               debitorID));
    }

    // findmanager debug hack
    protected TransactionManager findManager() {
        // find a known account service
        LookupLocator lookup = null;
        ServiceRegistrar registrar = null;
        TransactionManager mgr = null;

        try {
            lookup = new LookupLocator("jini://localhost");
        } catch(java.net.MalformedURLException e) {
            System.err.println("Lookup failed: " + e.toString());
            System.exit(1);
        }

        try {
            registrar = lookup.getRegistrar();
        } catch (java.io.IOException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        } catch (java.lang.ClassNotFoundException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        }
        System.out.println("Registrar found");

        Class[] classes = new Class[] {TransactionManager.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
        try {
            mgr = (TransactionManager) registrar.lookup(template);
        } catch(java.rmi.RemoteException e) {
            System.exit(2);
        }
        return mgr;
    }
    public int prepare(TransactionManager mgr, long id) {
        System.out.println("Preparing...");
        return TransactionConstants.PREPARED;
    }

    public void commit(TransactionManager mgr, long id) {
        System.out.println("committing");

    }


    public void abort(TransactionManager mgr, long id) {
        System.out.println("aborting");
    }

    public int prepareAndCommit(TransactionManager mgr, long id) {
        int result = prepare(mgr, id);
        if (result == TransactionConstants.PREPARED) {
            commit(mgr, id);
            result = TransactionConstants.COMMITTED;
        }
        return result;
    }

    class CreditDebit {
        long amount;
        long creditorID;
        long debitorID;

        CreditDebit(long a, long c, long d) {
            amount = a;
            creditorID = c;
            debitorID = d;
        }
    }

    class TransactionPair {

        TransactionPair(TransactionManager mgr, long id) {

        }
    }
} // AccountsImpl
```

Client

The final component in this application is the client that starts the transaction. The simplest code for this would just use the blocking `lookup()` method of `ClientLookupManager` to find first the service and then the transaction manager. We will use the longer way to show various ways of doing things.

This implementation uses a nested class that extends `Thread`. Because of this, it cannot extend `UnicastRemoteObject` and so is not automatically exported. In order to export itself, it has to call the `UnicastRemoteObject.exportObject()` method. This must be done before the call to join the transaction, which expects a remote object.

```
package client;

import common.PayableFileClassifier;
import common.MIMEType;

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.transaction.server.TransactionManager;
import net.jini.core.transaction.server.TransactionConstants;
import net.jini.core.transaction.server.TransactionParticipant;
// import com.sun.jini.lease.LeaseRenewalManager;
import net.jini.lease.LeaseRenewalManager;
import net.jini.core.lease.Lease;
import net.jini.lookup.entry.Name;
import net.jini.core.entry.Entry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

/**
 * TestTxn.java
 */

public class TestTxn implements DiscoveryListener {

    PayableFileClassifier classifier = null;
    TransactionManager mgr = null;

    long myClientID; // my account id
```

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

    // stay around long enough to receive replies
    try {
        Thread.currentThread().sleep(100000L);
    } catch(java.lang.InterruptedException e) {
        // do nothing
    }
}

public TestTxn() {
    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();

    for (int n = 0; n < registrars.length; n++) {
        System.out.println("Service found");
        ServiceRegistrar registrar = registrars[n];

        new LookupThread(registrar).start();
    }

        // System.exit(0);
}

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

```
public class LookupThread extends Thread implements TransactionParticipant,
                                                    java.io.Serializable {

    ServiceRegistrar registrar;
    long crashCount = 0; // ???

    LookupThread(ServiceRegistrar registrar) {
        this.registrar = registrar;
    }


    public void run() {
        long cost = 0;

        // try to find a classifier if we haven't already got one
        if (classifier == null) {
            System.out.println("Searching for classifier");
            Class[] classes = new Class[] {PayableFileClassifier.class};
            ServiceTemplate template = new ServiceTemplate(null, classes,
                                                           null);

            try {
                Object obj = registrar.lookup(template);
                System.out.println(obj.getClass().toString());
                Class cls = obj.getClass();
                Class[] clss = cls.getInterfaces();
                for (int n = 0; n < clss.length; n++) {
                    System.out.println(clss[n].toString());
                }
                classifier = (PayableFileClassifier) registrar.lookup(template);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(2);
            }
            if (classifier == null) {
                System.out.println("Classifier null");
            } else {
                System.out.println("Getting cost");
                try {
                    cost = classifier.getCost();
                } catch(java.rmi.RemoteException e) {
                    e.printStackTrace();
                }
                if (cost > 20) {
```

```
                System.out.println("Costs too much: " + cost);
                classifier = null;
            }
        }


    }

    // try to find a transaction manager if we haven't already got one
    if (mgr == null) {
        System.out.println("Searching for txnmgr");

        Class[] classes = new Class[] {TransactionManager.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);



        /*
        Entry[] entries = {new Name("TransactionManager")};
        ServiceTemplate template = new ServiceTemplate(null, null,
                                                       entries);
        */

        try {
            mgr = (TransactionManager) registrar.lookup(template);
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
            System.exit(2);
        }
        if (mgr == null) {
            System.out.println("Manager null");
            return;
        }

    }

    if (classifier != null && mgr != null) {
        System.out.println("Found both");
        TransactionManager.Created tcs = null;


        System.out.println("Creating transaction");
        try {
            tcs = mgr.create(Lease.FOREVER);
```

```
} catch(java.rmi.RemoteException e) {
    mgr = null;
    return;
} catch(net.jini.core.lease.LeaseDeniedException e) {
    mgr = null;
    return;
}

long transactionID = tcs.id;

// join in ourselves
System.out.println("Joining transaction");

// but first, export ourselves since we
// don't extend UnicastRemoteObject
try {
    UnicastRemoteObject.exportObject(this);
} catch(RemoteException e) {
    e.printStackTrace();
}

try {
    mgr.join(transactionID, this, crashCount);
} catch(net.jini.core.transaction.UnknownTransactionException e) {
    e.printStackTrace();
} catch(java.rmi.RemoteException e) {
    e.printStackTrace();
} catch(net.jini.core.transaction.server.CrashCountException e) {
    e.printStackTrace();
} catch(net.jini.core.transaction.CannotJoinException e) {
    e.printStackTrace();
}

new LeaseRenewalManager().renewUntil(tcs.lease,
                                     Lease.FOREVER,
                                     null);
System.out.println("crediting...");
try {
    classifier.credit(cost, myClientID,
                      mgr, transactionID);
} catch(Exception e) {
    System.err.println(e.toString());
}
```

```
            System.out.println("classifying...");
            MIMEType type = null;
            try {
                type = classifier.getMIMEType("file1.txt");
            } catch(java.rmi.RemoteException e) {
                System.err.println(e.toString());
            }

            // if we get a good result, commit, else abort
            if (type != null) {
                System.out.println("Type is " + type.toString());
                System.out.println("Calling commit");
                // new CommitThread(mgr, transactionID).run();

                try {
                    System.out.println("mgr state " +
                                       mgr.getState(transactionID));
                    mgr.commit(transactionID);
                } catch(Exception e) {
                    e.printStackTrace();
                }

            } else {
                try {
                    mgr.abort(transactionID);
                } catch(java.rmi.RemoteException e) {
                } catch(net.jini.core.transaction.CannotAbortException e) {
                } catch( net.jini.core.transaction.UnknownTransactionException
                         e) {
                }
            }
        }
    }

    public int prepare(TransactionManager mgr, long id) {
        System.out.println("Preparing...");
        return TransactionConstants.PREPARED;
    }

    public void commit(TransactionManager mgr, long id) {
        System.out.println("committing");
    }
```

```
        public void abort(TransactionManager mgr, long id) {
            System.out.println("aborting");

        }

        public int prepareAndCommit(TransactionManager mgr, long id) {
            int result = prepare(mgr, id);
            if (result == TransactionConstants.PREPARED) {
                commit(mgr, id);
                result = TransactionConstants.COMMITTED;
            }
            return result;
        }

    } // LookupThread

    class CommitThread extends Thread {
        TransactionManager mgr;
        long transactionID;

        public CommitThread(TransactionManager m, long id) {
            mgr = m;
            transactionID = id;
            try {
                Thread.sleep(1000);
            } catch(Exception e) {
            }
        }

        public void run() {
            try {
                mgr.abort(transactionID);
            } catch(Exception e) {
                e.printStackTrace();
            }
        }
    } // CommitThread

} // TestTxn
```

Summary

Transactions are needed to coordinate changes of state across multiple clients and services. The Jini transaction model uses a simple model of transactions, with details of semantics being left to the clients and services. The Jini distribution supplies a transaction manager, called `Mahalo`, that can be used.

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 that consists of a number of LEGO parts, a microcomputer called the RCX, 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.

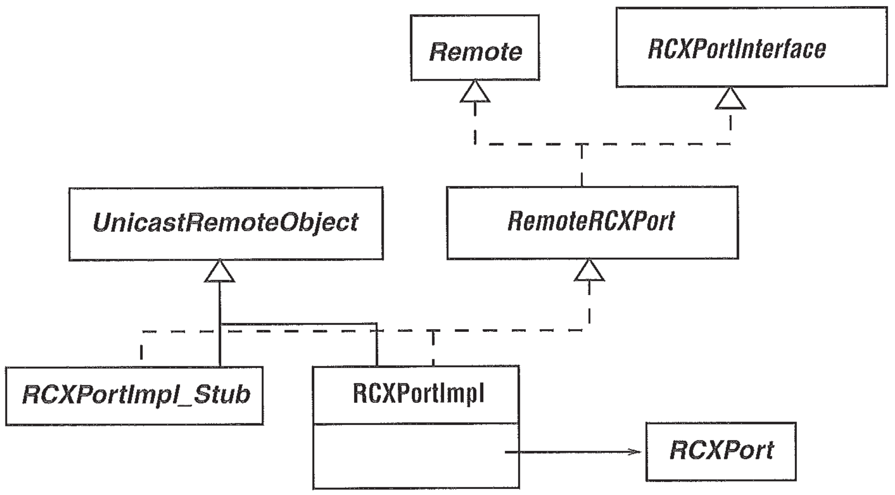


Figure 17-1. Class diagram for MINDSTORMS with RMI proxies

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), is pictured in Figure 17-2.

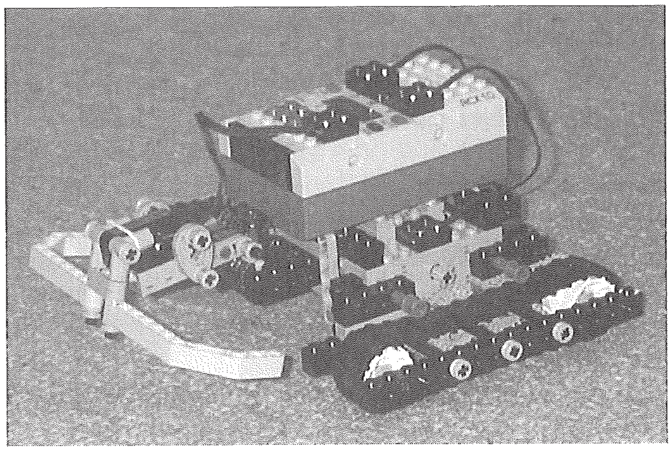


Figure 17-2. RoverBot MINDSTORMS robot

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.

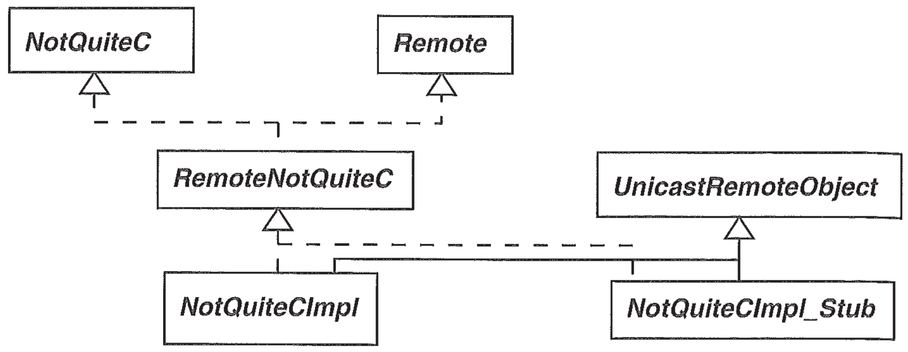


Figure 17-3. Class diagram for nqc *with RMI proxy*

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.

CHAPTER 18

CORBA and Jini

THERE ARE MANY DIFFERENT DISTRIBUTED SYSTEM ARCHITECTURES in addition to Jini. Many have only limited use, but some such as DCOM and CORBA are widely used, and there are many systems that have been built using these other distributed frameworks. This chapter looks at the similarities and differences between Jini and CORBA and shows how services built using one architecture can be used by another.

CORBA

Like Jini, CORBA is an infrastructure for distributed systems. CORBA was designed out of a different background than Jini, and there are some minor and major differences between the two.

- CORBA allows for specification of objects that can be distributed. The concentration is on distributed objects rather than on distributed services.
- CORBA is language-independent, using an Interface Definition Language (IDL) for specifying interfaces.
- CORBA objects can be implemented in a number of languages, including C, C++, SmallTalk, and Java
- Current versions of CORBA pass remote object references, rather than complete object instances. Each CORBA object lives within a server, and the object can only act within this server. This is more restricted than Jini, where an object can have instance data and class files sent to a remote location to execute there. This limitation in CORBA may change in future with pass-by-value parameters to methods.

IDL is a language that allows the programmer to specify the interfaces of a distributed object system. The syntax is similar to C++ but does not include any implementation-level constructs, so it allows definitions of data types (such as structures and unions), constants, enumerated types, exceptions, and interfaces. Within interfaces, it allows the declaration of attributes and operations (methods). The complete IDL specification can be found on the Object Management Group (OMG) Web site (`http://www.omg.org/`).

The book *Java Programming with CORBA* by Andreas Vogel and Keith Duddy (http://www.wiley.com/compbooks/vogel) contains an example of a room-booking service specified in CORBA IDL and implemented in Java. This defines interfaces for Meeting, a MeetingFactory factory to produce them, and a Room. A room may have a number of meetings in slots (hourly throughout the day), and there are support constants, enumerations, and typedefs to support this. In addition, exceptions may be thrown under various error conditions. The IDL that follows differs slightly from that given in the book, in that definitions of some data types that occur within interfaces have been "lifted" to a more global level, because the mapping from IDL to Java has changed slightly for elements nested within interfaces since that book was written. The following is the modified IDL for the room-booking service:

```
module corba {
module RoomBooking {

    interface Meeting {

        // A meeting has two read-only attributes that describe
        // the purpose and the participants of that meeting.

        readonly attribute string purpose;
        readonly attribute string participants;

        oneway void destroy();
    };

    interface MeetingFactory {

        // A meeting factory creates meeting objects.

        Meeting CreateMeeting( in string purpose, in string participants);
    };

    // Meetings can be held between the usual business hours.
    // For the sake of simplicity there are 8 slots at which meetings
    // can take place.

    enum Slot { am9, am10, am11, pm12, pm1, pm2, pm3, pm4 };

    // since IDL does not provide means to determine the cardinality
    // of an enum, a corresponding MaxSlots constant is defined.

    const short MaxSlots = 8;
    exception NoMeetingInThisSlot {};
    exception SlotAlreadyTaken {};

    interface Room {

        // A Room provides operations to view, make, and cancel bookings.
        // Making a booking means associating a meeting with a time slot
        // (for this particular room).

        // Meetings associates all meetings (of a day) with time slots
        // for a room.

        typedef Meeting Meetings[ MaxSlots ];

        // The attribute name names a room.

        readonly attribute string name;

        // View returns the bookings of a room.
        // For simplicity, the implementation handles only bookings
        // for one day.

        Meetings View();

        void Book( in Slot a_slot, in Meeting  a_meeting )
            raises(SlotAlreadyTaken);

        void Cancel( in Slot  a_slot )
            raises(NoMeetingInThisSlot);
    };
};
};
```

CORBA to Java Mapping

CORBA has bindings to a number of languages. That is, there is a translation from IDL to each language, and there is a runtime environment that supports objects written in these languages. A recent addition is Java, and this binding is still under active development (that is, the core is basically settled, but some parts are still changing). This binding must cover all elements of IDL. Here is a horribly brief summary of the CORBA translations:

- **Module**—A module is translated to a Java package. All elements within the module becomes classes or interfaces within the package.
- **Basic types**—Most of the basic types map in a straightforward manner—a CORBA `int` becomes a Java `int`, a CORBA `string` becomes a Java `java.lang.String`, and so on. Some are a little tricky, such as the unsigned types, which have no Java equivalent.
- **Constant**—Constants within a CORBA IDL interface are mapped to constants within the corresponding Java interface. Constants that are "global" have no direct equivalent in Java, and so are mapped to Java interfaces with a single field that is the value.
- **Enum**—Enumerated types have no direct Java equivalent, and so are mapped into a Java interface with the enumeration as a set of integer constants.
- **Struct**—A CORBA IDL structure is implemented as a Java class with instance variables for all fields.
- **Interface**—A CORBA IDL interface translates into a Java interface.
- **Exception**—A CORBA IDL exception maps to a final Java class.

This mapping does not conform to naming conventions, such as those established for Java Beans. For example, the IDL declaration `readonly string purpose` becomes the Java accessor method `String purpose()` rather than `String getPurpose()`. Where Java code is generated, the generated names will be used, but in methods that I write, I will use the more accepted naming forms.

Jini Proxies

A Jini service exports a proxy object that acts within the client on behalf of the service. On the service provider side, there may be service backend objects, completing the service implementation. The proxy may be fat or thin, depending on circumstances.

In Chapter 17 the proxy had to be thin: all it does is pass on requests to the service backend, which is linked to the hardware device, and the service cannot move, because it has to talk to a particular serial port. (The proxy may have an extensive user interface, but the Jini community seems to feel that any user interface should be in `Entry` objects rather than in the proxy itself.) Proxy objects created as RMI proxies are similarly thin, just passing on method calls to the service backend which is implemented as remote objects.

CORBA services can be delivered to any accessible client. Each service is limited to the server on which it is running, so they are essentially immobile, but they can be found by a variety of methods, such as a CORBA naming or trading service. These search methods can be run by any client, anywhere. A search will return a reference to a remote object, which is essentially a thin proxy to the CORBA service. Similarly, if a CORBA method call creates and returns an object, then it will return a remote reference to that object, and the object will continue to exist on the server where it was created. (The new CORBA standards will allow objects to be returned by value. This is not yet commonplace and will probably be restricted to a few languages, such as C++ and Java.)

The simplest way to make a CORBA object available to a Jini federation is to build a Jini service that is at the same time a CORBA client. The service acts as a bridge between the two protocols. Really, this is just the same as MINDSTORMS—anything that talks a different protocol (hardware or software) will require a bridge between itself and Jini clients.

Most CORBA implementations use a protocol called IIOP (Internet Inter-ORB Protocol), which is based on TCP. The current Jini implementation is also TCP-based, so there is a confluence of transport methods, which normally would not occur. A bridge would usually be fixed to a particular piece of hardware, but here it is not necessary due to this confluence.

A Jini service has a lot of flexibility in implementation and can choose to place logic in the proxy, in the backend, or anywhere else for that matter. The combination of Jini flexibility and IIOP allows a larger variety of implementation possibilities than is possible with fixed pieces of hardware such as MINDSTORMS. Here are a couple of examples:

- The Jini proxy could invoke the CORBA naming service lookup to locate the CORBA service, and then make calls directly on the CORBA service from the client. This is a fat proxy model in which the proxy contains all of the service implementation. There is no need for a service backend, and the service provider just exports the service object as proxy and then keeps the leases for the lookup services alive.

- The Jini proxy could be an RMI stub, passing on all method calls to a backend service running as an RMI remote object in the service provider. This is a thin proxy with fat backend, where all service implementation is done on the backend. The backend uses the CORBA naming service lookup to find the CORBA service and then makes calls on this CORBA service from the backend.

A Simple CORBA Example

The standard introductory example to any new system is "hello world." and it seems to get more complex with every advance that is made in computing technology! A CORBA version can be defined by the following IDL:

```
module corba {
    module HelloModule {
        interface Hello {
            string getHello();
        };
    };
};
```

This code can be compiled into Java using a compiler such as Sun's `idltojava` (or another CORBA 2.2 compliant compiler). This results in a `corba.HelloModule` package containing a number of classes and interfaces. `Hello` is an interface that is used by a CORBA client (in Java).

```
package corba.HelloModule;
public interface Hello
    extends org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity {
    String getHello();
}
```

CORBA Server in Java

A server for the hello IDL can be written in any language with a CORBA binding, such as C++. Rather than get diverted into other languages, though, we will stick to a Java implementation. However, this language choice is not forced on us by CORBA.

The server must create an object that implements the `Hello` interface. This is done by creating a servant that inherits from the `HelloImplBase` and then registering it with the CORBA ORB (Object Request Broker—this is the CORBA *backplane*, which acts as the runtime link between different objects in a CORBA system). The *servant* is the CORBA term for what we have been calling the "backend service" in Jini, and this object is created and run by the server. The server must also find a name server and register the name and the servant implementation. The servant implements the `Hello` interface. The server can just sleep to continue existence after registering the servant.

```
/**
 * CorbaHelloServer.java
 */
package corba;

import corba.HelloModule.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class CorbaHelloServer {

    public CorbaHelloServer() {

    }

    public static void main(String[] args) {
        try {
            // create a Hello implementation object
            ORB orb = ORB.init(args, null);
            HelloImpl hello = new HelloImpl();
            orb.connect(hello);

            // get the name server
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext namingContext = NamingContextHelper.narrow(objRef);

            // bind the Hello service to the name server
            NameComponent nameComponent = new NameComponent("Hello", "");
            NameComponent path[] = {nameComponent};
            namingContext.rebind(path, hello);

            // sleep
            java.lang.Object sleep = new java.lang.Object();
            synchronized(sleep) {
                sleep.wait();
            }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

} // CorbaHelloServer

class HelloImpl extends _HelloImplBase {
    public String getHello() {
        return("hello world");
    }
}
```

CORBA Client in Java

A standalone client finds a proxy implementing the `Hello` interface with methods such as one that looks up a CORBA name server. The name server returns a `org.omg.CORBA.Object`, which is cast to the interface type by the `HelloHelper` method `narrow()` (the Java cast method is not used). This proxy object can then be used to call methods back in the CORBA server.

```
/**
 * CorbaHelloClient.java
 */
package corba;

import corba.HelloModule.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class CorbaHelloClient {

    public CorbaHelloClient() {

    }

    public static void main(String[] args) {
        try {
            ORB orb = ORB.init(args, null);

            // get the name server
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext namingContext = NamingContextHelper.narrow(objRef);

             // get the Hello proxy
            NameComponent nameComponent = new NameComponent("Hello", "");
            NameComponent path[] = {nameComponent};
            org.omg.CORBA.Object obj = namingContext.resolve(path);
            Hello hello = HelloHelper.narrow(obj);
            // now invoke methods on the CORBA proxy
            String hello = hello.getHello();
            System.out.println(hello);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

} // CorbaHelloClient
```

Jini Service

In order to make the CORBA object accessible to the Jini world, it must be turned into a Jini service. At the same time it must remain in a CORBA server, so that it can be used by ordinary CORBA clients. So we can do nothing to the CORBA server. Instead, we need to build a Jini service that will act as a CORBA client. This service will then be able to deliver the CORBA service to Jini clients.

The Jini service can be implemented as a fat proxy delivered to a Jini client. The Jini service implementation is moved from the Jini server to a Jini client as the service object. Once in the client, the service implementation is responsible for locating the CORBA service by using the CORBA naming service, and it then translates client calls on the Jini service directly into calls on the CORBA service. The processes that run in this, with their associated Jini and CORBA objects, are shown in Figure 18-1.

The Java interface for this service is quite simple and basically just copies the interface for the CORBA service:

```
/**
 * JiniHello.java
 */
package corba;

import java.io.Serializable;

public interface JiniHello extends Serializable {

    public String getHello();
} // JiniHello
```

The `getHello()` method for the CORBA IDL returns a `string`. In the Java binding this becomes an ordinary Java `String`, and the Jini service can just use this type. The next example (in the "Room-Booking Example" section) will show a more

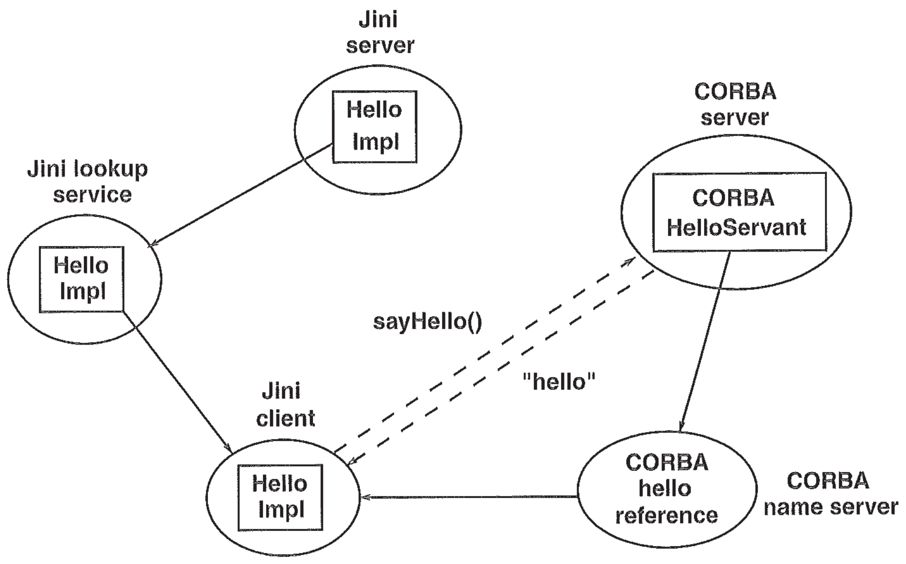


Figure 18-1. CORBA and Jini services

complex case where CORBA objects may be returned. Note that because this is a fat service, any implementation will get moved across to a Jini client and will run there, so the service only needs to implement `Serializable`, and its methods do not need to throw `Remote` exceptions, since they will run locally in the client.

The implementation of this Jini interface will basically act as a CORBA client. Its `getHello()` method will contact the CORBA naming service, find a reference to the CORBA `Hello` object, and call its `getHello()` method. The Jini service can just return the string it gets from the CORBA service.

```
/**
 * JiniHelloImpl.java
 */
package corba;

import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import corba.HelloModule.*;

public class JiniHelloImpl implements JiniHello {

    protected Hello hello = null;
    protected String[] argv;
    public JiniHelloImpl(String[] argv) {
        this.argv = argv;
    }

    public String getHello() {

        if (hello == null) {
            hello = getHello();
        }
        // now invoke methods on the CORBA proxy
        String hello = hello.getHello();
        return hello;
    }

    protected Hello getHello() {
        ORB orb = null;
        // Act like a CORBA client
        try {
            orb = ORB.init(argv, null);

            // get the CORBA name server
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext namingContext = NamingContextHelper.narrow(objRef);

             // get the CORBA Hello proxy
            NameComponent nameComponent = new NameComponent("Hello", "");
            NameComponent path[] = {nameComponent};
            org.omg.CORBA.Object obj = namingContext.resolve(path);
            Hello hello = HelloHelper.narrow(obj);
            return hello;
        } catch(Exception e) {
            e.printStackTrace();
            return null;
        }
    }
} // JiniHelloImpl
```

Jini Server and Client

The Jini server that exports the service doesn't contain anything new compared to the other service providers we have discussed. It creates a new `JiniHelloImpl` object and exports it using a `JoinManager`:

```
joinMgr = new JoinManager(new JiniHelloImpl(argv), ...)
```

Similarly, the Jini client doesn't contain anything new, except that it catches CORBA exceptions. After lookup discovery, the code is as follows:

```
try {
    hello = (JiniHello) registrar.lookup(template);
} catch(java.rmi.RemoteException e) {
    e.printStackTrace();
    System.exit(2);
}
if (hello == null) {
   System.out.println("hello null");
    return;
}
String msg;
try {
   msg = hello.getHello();
    System.out.println(msg);
} catch(Exception e) {
   // a CORBA runtime error may occur
    System.err.println(e.toString());
}
```

Building the Simple CORBA Example

Compared to the Jini-only examples that have been looked at so far, the major additional step in this CORBA example is to build the Java classes from the IDL specification. There are a number of CORBA IDL-to-Java compilers. One of these is the Sun compiler `idltojava`, which is available from `java.sun.com`. This (or another compiler) needs to be run on the IDL file to produce the Java files in the `corba.HelloModule` package. The files that are produced are standard Java files, and they can be compiled using your normal Java compiler. They may need some CORBA files in the `CLASSPATH` if required by your vendor's implementation of CORBA. Files produced by `idltojava` do not need any extra classes.

The Jini server, service, and client are also normal Java files, and they can be compiled like earlier Jini files, with the `CLASSPATH` set to include the Jini libraries.

Running the Simple CORBA Example

There are a large number of elements and processes that must be set running to get this example working satisfactorily:

1. A CORBA name server must be set running. In the JDK 1.2 distribution is a server, `tnameserv`. By default, this runs on TCP port 900. Under Unix, access to this port is restricted to system supervisors. It can be set running on this port by a supervisor, or it can be started during boot time. An ordinary user will need to use the option `-ORBInitialPort` port to run it on a port above 1024:

   ```
   tnameserv -ORBInitialPort 1055
   ```

   All CORBA services and clients should also use this port number.

2. The Java version of the CORBA service can then be started with this command:

   ```
   java corba.CorbaHelloServer -ORBInitialPort 1055
   ```

3. Typical Jini support services will need to be running, such as a Jini lookup service, the RMI daemon `rmid`, and HTTP servers to move class definitions around.

4. The Jini service can be started with this command:

   ```
   java corba.JiniHelloServer -ORBInitialPort 1055
   ```

5. Finally, the Jini client can be run with this command:

   ```
   java client.TestCorbaHello -ORBInitialPort 1055
   ```

CORBA Implementations

There are interesting considerations about what is needed in Java to support CORBA. The example discussed previously uses the CORBA APIs that are part of the standard OMG binding of CORBA to Java. The packages rooted in `org.omg` are in major distributions of JDK 1.2, such as the Sun SDK. This example should compile properly with most Java 1.2 compilers using these OMG classes.

Sun's JDK 1.2 runtime includes a CORBA ORB, and the example will run as is, using this ORB. However, there are many implementations of CORBA ORBs, and they do not always behave in quite the same way. This can affect compilation and runtime results. Which CORBA ORB is used is determined at runtime, based on properties. If a particular ORB is not specified, then it defaults to the Sun-supplied ORB (using Sun's SDK). To use another ORB, such as the Orbacus ORB, the following code needs to be inserted before the call to `ORB.init()`:

```
java.util.Properties props = System.getProperties();
props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
props.put("org.omg.CORBA.ORBSingletonClass",
          "com.ooc.CORBA.ORBSingleton");
System.setProperties(props);
```

Similar code is required for the ORBS from IONA and other vendors.

Variations in CORBA implementations could affect the runtime behavior of the client: if the proxy expects to use a particular ORB other than the default, then the class files for that ORB must be available to the client or be downloadable across the network. Alternatively, the proxy could be written to use the default Sun ORB, and then may need to make inter-ORB calls between the Sun ORB and the actual ORB used by the CORBA service. Such issues take us beyond the scope of this chapter, though. Vendor documentation for each CORBA implementation should give more information on any additional requirements.

Room-Booking Example

The IDL for a room-booking problem was briefly discussed in the introductory "CORBA" section in this chapter. This room-booking example has a few more complexities than the previous example. The problem here is to have a set of rooms, and for each room have a set of bookings that can be made for that room. The bookings may be made on the hour, from 9 a.m. until 4 p.m. (this only covers the bookings for one day). Bookings may be cancelled after they are made. A room can be queried for the set of bookings it has: it returns an array of meetings, which are `null` if no booking has been made, or non-null including the details of the participants and the purpose of the meeting.

There are other things to consider in this example:

- Each room is implemented as a separate CORBA object. There is also a "meeting factory" that produces more objects. This is a system with multiple CORBA objects residing on many CORBA servers. There are several possibilities for implementing a system with multiple objects.

- Some of the methods return CORBA objects, and these may need to be exposed to clients. This is not a problem if the client is a CORBA client, but here we will have Jini clients.

- Some of the methods throw user-defined exceptions, in addition to CORBA-defined exceptions. Both of these need to be handled appropriately.

CORBA Objects

CORBA defines a set of "primitive" types in the IDL, such as integers of various sizes, chars, etc. The language bindings specify the primitive types in each language that they are converted into. For example, the CORBA wide character (`wchar`) becomes a Java Unicode `char`. Things are different for non-primitive objects, which depend on the target language. For example, an IDL *object* turns into a Java *interface.*

The room-booking IDL defines CORBA interfaces for `Meeting`, `MeetingFactory`, and `Room`. These can be implemented in any suitable language and need not be in Java—the Java binding will convert these into Java interfaces. A CORBA client written in Java will get objects that implement these interfaces, but these objects will essentially be references to remote CORBA objects. Two things are certain about these references:

- CORBA interfaces generate Java interfaces, such as `Hello`. These inherit from `org.omg.CORBA.portable.IDLEntity`, which implements `Serializable`. As a result, the references can be moved around like Jini objects, but they lose their link to the CORBA ORB that created them and may end up in a different namespace, where the reference makes no sense. Therefore, CORBA references cannot be usefully moved around. At present, the best way to move them around is to convert them to "stringified" form and move that around, though this may change when CORBA pass-by-value objects become common. Note that the serialization method that gives a string representation of a CORBA object is not the same as the Java one: the CORBA method serializes the remote reference, whereas the Java method serializes the object's instance data.

- The references do not subclass from `UnicastRemoteObject` or `Activatable`. The Java runtime will not use an RMI stub for them.

If a Jini client gets local references to these objects and keeps them local, then it can use them via their Java interfaces. If they need to be moved around the network, then appropriate "mobile" classes will need to be defined and the information copied across to them from the local objects. For example, the CORBA `Meeting` interface generates the following Java interface:

```
/*
 * File: ./corba/RoomBooking/Meeting.java
 * From: RoomBooking.idl
 * Date: Wed Aug 25 11:30:25 1999
 *   By: idltojava Java IDL 1.2 Aug 11 1998 02:00:18
 */

package corba.RoomBooking;
public interface Meeting
    extends org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity {
    String purpose();
    String participants();
    void destroy()
;
}
```

To make the information from a CORBA `Meeting` available as a mobile Jini object, we would need an interface like this:

```
/**
 * JavaMeeting.java
 */
package corba.common;

import java.io.Serializable;
import org.omg.CORBA.*;
import corba.RoomBooking.*;
import java.rmi.RemoteException;

public interface JavaMeeting extends Serializable {
    String getPurpose();
    String getParticipants();
    Meeting getMeeting(ORB orb);
} // JavaMeeting
```

The first two methods in the preceding interface allow information about a meeting to be accessible to applications that do not want to contact the CORBA service. The third allows a CORBA object reference to be reconstructed within a new ORB. A suitable implementation is as follows:

```
/**
 * JavaMeetingImpl.java
 */
package corba.RoomBookingImpl;

import corba.RoomBooking.*;
import org.omg.CORBA.*;
import corba.common.*;

/**
 * A portable Java object representing a CORBA object.
 */
public class JavaMeetingImpl implements JavaMeeting {
    protected String purpose;
    protected String participants;
    protected String corbaObj;

    /**
     * get the purpose of a meeting for a Java client
     * unaware of CORBA
     */
    public String getPurpose() {
        return purpose;
    }

    /**
     * get the participants of a meeting for a Java client
     * unaware of CORBA
     */
    public String getParticipants() {
        return participants;
    }

    /**
     * reconstruct a meeting using a CORBA orb in the target JVM
     */
    public Meeting getMeeting(ORB orb) {
        org.omg.CORBA.Object obj = orb.string_to_object(corbaObj);
        Meeting m = MeetingHelper.narrow(obj);
        return m;
    }

    /**
     * construct a portable Java representation of the CORBA
     * Meeting using the CORBA orb on the source JVM
     */
    public JavaMeetingImpl(Meeting m, ORB orb) {
        purpose = m.purpose();
        participants = m.participants();
        corbaObj = orb.object_to_string(m);
    }
} // JavaMeetingImpl
```

Multiple Objects

The implementation of the room-booking problem in the Vogel and Duddy book (*Java Programming with CORBA,* `http://www.wiley.com/compbooks/vogel`) runs each room as a separate CORBA object, each with its own server. A meeting factory creates meeting objects that are kept within the factory server and passed around by reference. So, for a distributed application with ten rooms, there will be eleven CORBA servers running.

There are several possible ways of bringing this set of objects into the Jini world so that they are accessible to a Jini client:

1. A Jini server may exist for each CORBA server.
    - Each Jini server may export fat proxies, which build CORBA references in the same Jini client.
    - Each Jini server may export a thin proxy, with a CORBA reference held in each of these servers.
2. A single Jini server may be built for the federation of all the CORBA objects.
    - The single Jini server exports a fat proxy, which builds CORBA references in the Jini client.
    - The single Jini server exports a thin proxy, with all CORBA references within this single server.

The first of these pairs of options essentially isolates each CORBA service into its own Jini service. This may be appropriate in an open-ended system where there may be a large set of CORBA services, only some of which are needed by any application.

The second pair of options deals with the case where services come logically grouped together, such that one cannot exist without the other, even though they may be distributed geographically.

Intermediate schemes exist, where some CORBA services have their own Jini service, while others are grouped into a single Jini service. For example, rooms may be grouped into buildings and cannot exist without these buildings, whereas a client may only want to know about a subset of buildings, say those in New York.

Many Fat Proxies

We can have one Jini server for each of the CORBA servers. The Jini servers can be running on the same machines as the CORBA ones, but there is no necessity from either Jini or CORBA for this to be so. On the other hand, if a client is running as an applet, then applet security restrictions may force all the Jini servers to run on a single machine, the same one as an applet's HTTP server.

The Jini proxy objects exported by each Jini server may be fat ones, which connect directly to the CORBA server. Thus, each proxy becomes a CORBA client, as was the case in the "hello world" example. Within the Jini client, we do not just have one proxy, but many proxies. Because they are all running within the same address space, they can share CORBA references—there is no need to package a CORBA reference as a portable Jini object. In addition, the Jini client can just use all of these CORBA references directly, as instance objects of interfaces. This situation is shown in Figure 18-2.

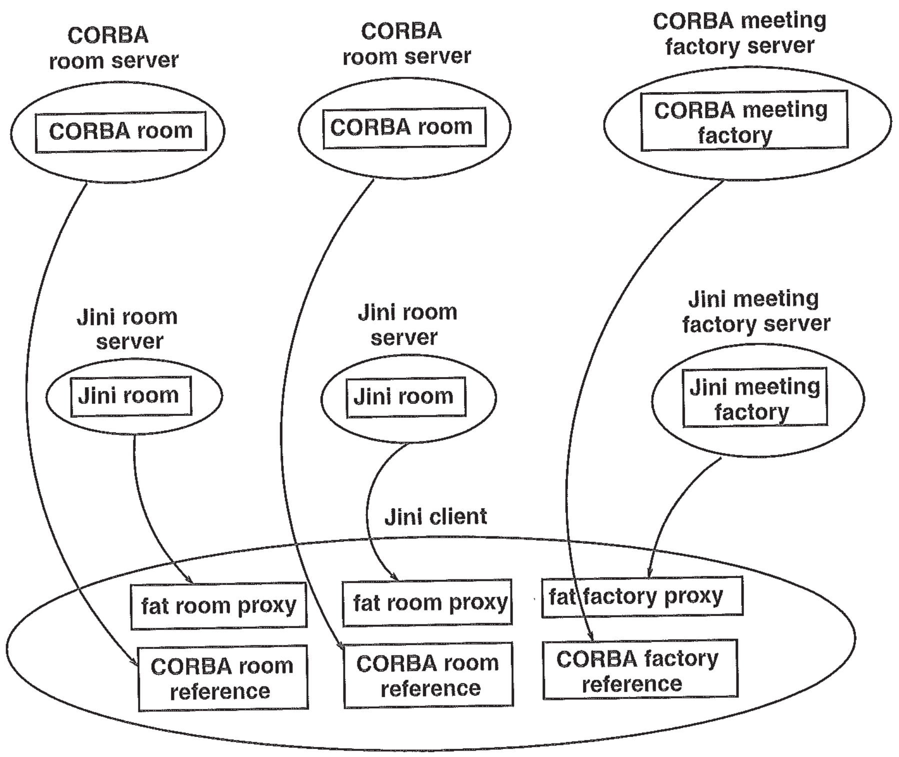


Figure 18-2. CORBA and Jini services for fat proxies

The CORBA servers are all accessed from within the Jini client. This arrangement may be ruled out if the client is an applet and the servers are on different machines.

Many Thin Proxies

The proxies exported can be thin, such as RMI stubs. In this case, each Jini server is acting as a CORBA client. This situation is shown in Figure 18-3.

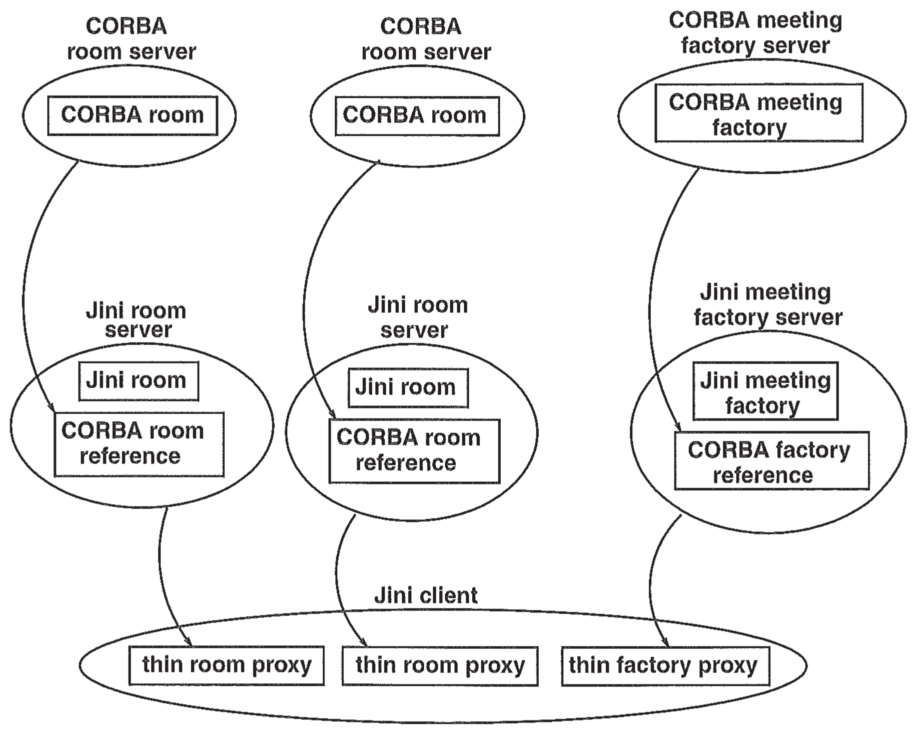


Figure 18-3. CORBA and Jini services for thin proxies

If all the Jini servers are collocated on the same machine, then this becomes a possible architecture suitable for applets. The downside of this approach is that all the CORBA references are within different JVMs. In order to move the reference for a meeting from the Jini meeting factory to one of the Jini rooms, it may be necessary to wrap it in a portable Jini object, as discussed previously. The Jini client will also need to get information about the CORBA objects, which can be gained from these portable Jini objects.

Single Fat Proxy

An alternative to Jini servers for each CORBA server is to have a single Jini bridge server into the CORBA federation. This can be a feasible alternative when the set of CORBA objects form a complete system or module, and it makes sense to treat them as a unit. Then you have the choices again of where to locate the CORBA references—either in the Jini server or in a proxy. Placing them in a fat proxy is shown in Figure 18-4.

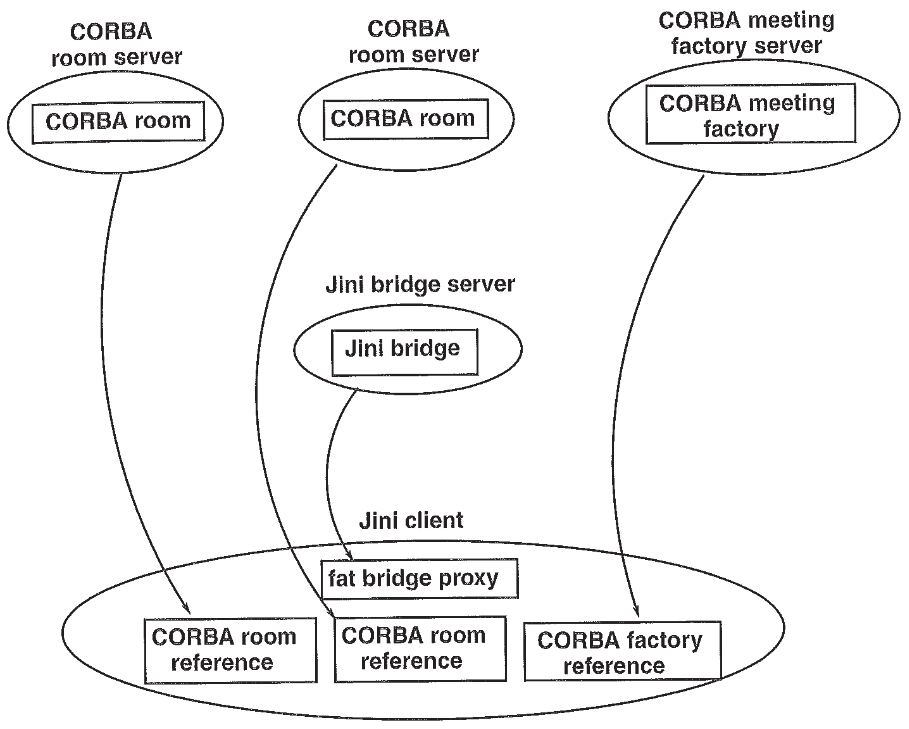


Figure 18-4. CORBA and Jini services for single fat proxy

Single Thin Proxy

Placing all the CORBA references on the server side of a Jini service means that a Jini client only needs to make one network connection to the service. This scenario is shown in Figure 18-5. This is probably the best option from a security viewpoint of a Jini client.

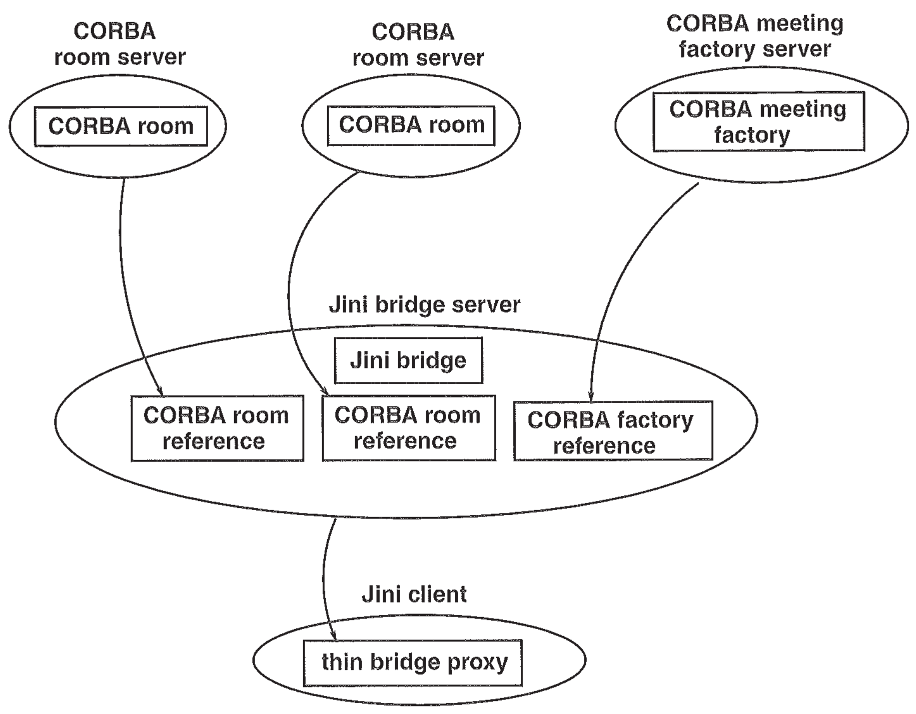


Figure 18-5. CORBA and Jini services for single thin proxy

Exceptions

CORBA methods can throw exceptions of two types: system exceptions and user exceptions. System exceptions subclass from `RuntimeException` and so are unchecked. They do not need to have explicit `try...catch` clauses around them. If an exception is thrown, it will be caught by the Java runtime and will generally halt the process with an error message. This would result in a CORBA client dying, which would generally be undesirable. Many of these system exceptions will be caused by the distributed nature of CORBA objects, and probably should be caught explicitly. If they cannot be handled directly, then to bring them into line with the Jini world, they can be wrapped as "nested exceptions" within a `Remote` exception and thrown again.

User exceptions are declared in the IDL for the CORBA interfaces and methods. These exceptions are checked, and need to be explicitly caught (or re-thrown) by Java methods. If a user exception is thrown, this will be because of some semantic error within one of the objects and will be unrelated to any networking or remote issues. User exceptions should be treated as they are, without wrapping them in `Remote` exceptions.

Interfaces for Single Thin Proxy

This and the following sections build a single thin proxy for a federation of CORBA objects. The Vogel and Duddy book gives a CORBA client to interact with the CORBA federation, and this is used as the basis for the Jini services and clients.

Using a thin proxy means that all CORBA-related calls will be placed in the service object, and will be made available to Jini clients only by means of portable Jini versions of the CORBA objects. These portable objects are defined by two interfaces, the `JavaRoom` interface

```
/**
 * JavaRoom.java
 */
package corba.common;

import corba.RoomBooking.*;
import java.io.Serializable;
import org.omg.CORBA.*;
import java.rmi.RemoteException;

public interface JavaRoom extends Serializable {
    String getName();
    Room getRoom(ORB orb);
} // JavaRoom
```

and the `JavaMeeting` interface

```
/**
 * JavaMeeting.java
 */
package corba.common;

import java.io.Serializable;
import org.omg.CORBA.*;
import corba.RoomBooking.*;
import java.rmi.RemoteException;

public interface JavaMeeting extends Serializable {
    String getPurpose();
    String getParticipants();
    Meeting getMeeting(ORB orb);
} // JavaMeeting
```

The bridge interface between the CORBA federation and the Jini clients has to provide methods for making changes to objects within the CORBA federation and for obtaining information from them. For the room-booking system, this requires the ability to book and cancel meetings within rooms, and also the ability to view the current state of the system. Viewing is accomplished by three methods: updating the current state, getting a list of rooms, and getting a list of bookings for a room.

```
/**
 * RoomBookingBridge.java
 */

package corba.common;

import java.rmi.RemoteException;
import corba.RoomBooking.*;
import org.omg.CORBA.*;

public interface RoomBookingBridge extends java.io.Serializable {

    public void cancel(int selected_room, int selected_slot)
        throws RemoteException, NoMeetingInThisSlot;
    public void book(String purpose, String participants,
                     int selected_room, int selected_slot)
        throws RemoteException, SlotAlreadyTaken;
    public void update()
        throws RemoteException, UserException;
    public JavaRoom[] getRooms()
        throws RemoteException;
    public JavaMeeting[] getMeetings(int room_index)
        throws RemoteException;
} // RoomBookingBridge
```

There is a slight legacy in this interface that comes from the original "monoblock" CORBA client by Vogel and Duddy. In that client, because the GUI interface elements and the CORBA references were all in the one client, simple shareable structures, such as arrays of rooms and arrays of meetings, were used. Meetings and rooms could be identified simply by their index in the appropriate array. In splitting the client apart into multiple (and remote) classes, this is not really a good idea anymore because it assumes a commonality of implementation across objects, which may not occur. It doesn't seem worthwhile being too fussy about that here, though.

RoomBookingBridge Implementation

The room-booking Jini bridge has to perform all CORBA activities and to wrap these up as portable Jini objects. A major part of this is locating the CORBA services, which here are the meeting factory and the rooms. We do not want to get too involved in these issues here. The meeting factory can be found in essentially the same way as the hello server was earlier, by looking up its name. Finding the rooms is harder, as these are not known in advance. Essentially, the equivalent of a directory has to be set up on the name server, which is known as a "naming context." Rooms are registered within this naming context by their servers, and the client gets this context and then does a search for its contents.

The Jini component of this object is that it subclasses from `UnicastRemoteObject` and implements a `RemoteRoomBookingBridge`, which is a remote version of `RoomBookingBridge`. It is also worthwhile noting how CORBA exceptions are caught and wrapped in `Remote` exceptions.

```
/**
 * RoomBookingBridgeImpl.java
 */
package corba.RoomBookingImpl;

import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import corba.RoomBooking.*;
import corba.common.*;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class RoomBookingBridgeImpl extends UnicastRemoteObject implements Remote-
RoomBookingBridge {

    private MeetingFactory meeting_factory;
    private Room[] rooms;
    private Meeting[] meetings;
    private ORB orb;
    private NamingContext room_context;

    public RoomBookingBridgeImpl(String[] args)
        throws RemoteException, UserException {
        try {
            // initialize the ORB
            orb = ORB.init(args, null);
    }
    catch(SystemException system_exception ) {
        throw new RemoteException("constructor RoomBookingBridge: ",
                                  system_exception);
    }
    init_from_ns();
    update();
}

public void init_from_ns()
    throws RemoteException, UserException {

    // initialize from Naming Service
    try {
        // get room context
        String str_name = "/BuildingApplications/Rooms/";
        org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
        NamingContext namingContext = NamingContextHelper.narrow(objRef);
        NameComponent nc = new NameComponent(str_name,  " ");
        NameComponent path[] = {nc};

        org.omg.CORBA.Object roomRef = namingContext.resolve(path);
        room_context = NamingContextHelper.narrow(roomRef);
        if( room_context == null ) {
            System.err.println( "Room context is null," );
            System.err.println( "exiting ..." );
            System.exit( 1 );
        }

        // get MeetingFactory from Naming Service
        str_name = "/BuildingApplications/MeetingFactories/MeetingFactory";
        nc = new NameComponent(str_name, " ");
        path[0] = nc;
        meeting_factory =
            MeetingFactoryHelper.narrow(namingContext.resolve(path));
        if( meeting_factory == null ) {
            System.err.println(
                "No Meeting Factory registered at Naming Service" );
            System.err.println( "exiting ..." );
            System.exit( 1 );
        }
    }
    catch(SystemException system_exception ) {
            throw new RemoteException("Initialize ORB", system_exception);
        }
    }

    public void update()
        throws RemoteException, UserException {

        try {
            // list rooms
            // initialize binding list and binding iterator
            // Holder objects for out parameter
            BindingListHolder blHolder = new BindingListHolder();
            BindingIteratorHolder biHolder = new BindingIteratorHolder();
            BindingHolder bHolder = new BindingHolder();
            Vector roomVector = new Vector();
            Room aRoom;

            // we are 2 rooms via the room list
            // more rooms are available from the binding iterator
            room_context.list( 2, blHolder, biHolder );

            // get rooms from Room context of the Naming Service
            // and put them into the roomVector
            for(int i = 0; i < blHolder.value.length; i++ ) {
                aRoom = RoomHelper.narrow(
                    room_context.resolve( blHolder.value[i].binding_name ));
                roomVector.addElement( aRoom );
            }

            // get remaining rooms from the iterator
            if( biHolder.value != null ) {
                while( biHolder.value.next_one( bHolder ) ) {
                    aRoom = RoomHelper.narrow(
                        room_context.resolve( bHolder.value.binding_name ) );
                    if( aRoom != null ) {
                        roomVector.addElement( aRoom );
                    }
                }
            }

            // convert the roomVector into a room array
            rooms = new Room[ roomVector.size() ];
            roomVector.copyInto( rooms );
            // be friendly with system resources
            if(  biHolder.value != null )
                biHolder.value.destroy();
        }

        catch(SystemException system_exception) {
            throw new RemoteException("View", system_exception);
            // System.err.println("View: " + system_exception);
        }
    }

    public void cancel(int selected_room, int selected_slot)
        throws RemoteException, NoMeetingInThisSlot {
        try {
            rooms[selected_room].Cancel(
                Slot.from_int(selected_slot) );
            System.out.println("Cancel called" );
        }
        catch(SystemException system_exception) {
            throw new RemoteException("Cancel", system_exception);
        }
    }

    public void book(String purpose, String participants,
                     int selected_room, int selected_slot)
        throws RemoteException, SlotAlreadyTaken {
        try {
            Meeting meeting =
                meeting_factory.CreateMeeting(purpose, participants);
            System.out.println( "meeting created" );
            String p = meeting.purpose();
            System.out.println("Purpose: "+p);
            rooms[selected_room].Book(
                Slot.from_int(selected_slot), meeting );
            System.out.println( "room is booked" );
        }
        catch(SystemException system_exception ) {
           throw new RemoteException("Booking system exception", system_exception);
        }
    }

    /**
     * return a list of the rooms as portable JavaRooms
     */
    public JavaRoom[] getRooms() {
        int len = rooms.length;
        JavaRoom[] jrooms = new JavaRoom[len];
        for (int n = 0; n < len; n++) {
            jrooms[n] = new JavaRoomImpl(rooms[n]);
        }
        return jrooms;
    }

    public JavaMeeting[] getMeetings(int room_index) {
        Meeting[] meetings = rooms[room_index].View();
        int len = meetings.length;
        JavaMeeting[] jmeetings = new JavaMeeting[len];
        for (int n = 0; n < len; n++) {
            if (meetings[n] == null) {
                jmeetings[n] = null;
            } else {
                jmeetings[n] = new JavaMeetingImpl(meetings[n], orb);
            }
        }
        return jmeetings;
    }
} // RoomBookingBridgeImpl
```

Other Classes

The Java classes and servers implementing the CORBA objects are mainly unchanged from the implementations given in the Vogel and Duddy book. They can continue to act as CORBA servers to the original clients. I replaced the "easy naming" naming service in their book with a later one with the slightly more complex standard mechanism for creating contexts and placing names within this context. This mechanism can use the `tnameserv` CORBA naming server, for example.

I have modified the Vogel and Duddy room-booking client a little bit, but its essential structure remains unchanged. The GUI elements, for example, were not altered. All CORBA-related code was removed from the client and placed into the bridge classes.

The Vogel and Duddy code samples can all be downloaded from a public Web site (`http://www.wiley.com/compbooks/vogel`) and come with no author attribution or copyright claim. The client is also quite lengthy since it has plenty of GUI inside, so I won't complete the code listing here. The code for all my classes, and the modified code of the Vogel and Duddy classes, is given in the subdirectory `corba` of the `programs.zip` file that can be found at `http://www.apress.com`.

Building the Room-Booking Example

The `RoomBooking.idl` IDL interface needs to be compiled to Java by a suitable IDL-to-Java compiler, such as Sun's `idltojava`. This produces classes in the `corba.RoomBooking` package. These can then all be compiled using the standard Java classes and any CORBA classes needed.

The Jini server, service, and client are also normal Java files and can be compiled like earlier Jini files, with the `CLASSPATH` set to include the Jini libraries.

Running the Room-Booking Example

There are a large number of elements and processes that must be set running to get this example working satisfactorily:

1. A CORBA name server must be set running, as in the earlier example. For example, you could use the following command:

   ```
   tnameserv -ORBInitialPort 1055
   ```

   All CORBA services and clients should also use this port number.

2. A CORBA server should be started for each room, with the first parameter being the "name" of the room:

   ```
   java corba.RoomBookingImpl.RoomServer "freds room" -ORBInitialPort 1055
   ```

3. A CORBA server should be started for the meeting factory:

   ```
   java corba.RoomBookingImpl.MeetingFactoryServer -ORBInitialPort 1055
   ```

4. Typical Jini support services will need to be running, such as a lookup service, the RMI daemon `rmid`, and HTTP servers to move class definitions around.

5. The Jini service can be started with this command:

   ```
   java corba.RoomBookingImpl.RoomBookingBridgeServer -ORBInitialPort 1055
   ```

6. Finally, the Jini client can be run with this command:

   ```
   java corba.RoomBookingImpl.RoomBookingClientApplication -ORBInitialPort
   1055
   ```

Migrating a CORBA Client to Jini

Both of the examples in this chapter started life as pure CORBA systems written by other authors, with CORBA objects delivered by servers to a CORBA client. The clients were both migrated in a series of steps to Jini clients of a Jini service acting as a front-end to CORBA objects. For those in a similar situation, it may be worthwhile to spell out the steps I went through in doing this for the room-booking problem:

1. The original client was a single client, mixing GUI elements, CORBA calls, and glue to hold it all together. This had a number of objects playing different roles all together, without a clear distinction about roles in some cases. The first step was to decide on the architectural constraint: one Jini service, or many.

2. A single Jini service was chosen (for no other reason than it looked to offer more complexities). This implied that all CORBA-related calls had to be collected into a single object, the `RoomBookingBridgeImpl`. At this stage, the `RoomBookingBridge` interface was not defined—that came after the implementation was completed (okay, I hang my head in shame, but I was trying to adapt existing code rather than starting from scratch). At this time, the client was still running as a pure CORBA client—no Jini mechanisms had been introduced.

3. Once all the CORBA related code was isolated into one class, another architectural decision had to be made: whether this was to function as a fat or thin proxy. The decision to make it thin in this case was again based on interest rather than functional reasons.

4. The GUI elements left behind in the client needed to access information from the CORBA objects. In the thin proxy model, this meant that portable Jini objects had to be built to carry information out of the CORBA world. This led to interfaces such as `JavaRoom` and implementations such as `JavaRoomImpl`. The GUI code in the client had no need to directly modify fields in these objects, so they ended up as read-only versions of their CORBA sources. (If a fat proxy had been used, this step of creating portable Jini objects would not have been necessary.)

5. The client was modified to use these portable Jini objects, and the `RoomBookingBridgeImpl` was changed to return these objects from its methods. Again, this was all still done within the CORBA world, and no Jini services were yet involved. This looked like a good time to define the `RoomBookingBridge` interface, when everything had settled down.

6. Finally, the `RoomBookingBridgeImpl` was turned into a `UnicastRemoteObject` and placed into a Jini server. The client was changed to look up a `RoomBookingBridge` service rather than create a `RoomBookingBridgeImpl` object.

At the end of this, I had an implementation of a Jini service with a thin RMI proxy. The CORBA objects and servers had not been changed at all. The original CORBA client had been split into two, with the Jini service implementing all of the CORBA lookups. These were exposed to the client through a set of facades that gave it the information it needed.

The client was still responsible for all of the GUI aspects, and so was acting as a "knowledgeable" client. If needed, these GUI elements could be placed into `Entry` objects, and also could be exported as part of the service.

Jini Service as a CORBA Service

We have looked at making CORBA objects into Jini services. Is it possible to go the other way, and make a Jini service appear as a CORBA object in a CORBA federation? Well, it should be. Just as there is a mapping from CORBA IDL to Java, there is also a mapping of a suitable subset of Java into CORBA IDL. Therefore, a Jini service interface can be written as a CORBA interface. A Jini client could then be written as the implementation of a CORBA server to this IDL.

At present, with a paucity of Jini services, it does not seem worthwhile to explore this in detail. This may change in the future, though.

Summary

CORBA is a separate distributed system from Jini. However, it is quite straightforward to build bridges between the two systems, and there are a number of different possible architectures. This makes it possible for CORBA services to be used by Jini clients.

CHAPTER 19

User Interfaces for Jini Services

SOME EARLIER CHAPTERS HAVE USED CLIENTS with graphical user interfaces to services. Clients may not always know which is the most appropriate user interface, and sometimes may not even know of any suitable user interface. Services should be able to define their own user interfaces, and the question of how they should best do this is explored in this chapter. We'll also look at how clients can discover, download, and use these user interfaces.

User Interfaces as Entries

Interaction with a service is specified by its interface, and the interaction will be the same across all implementations of the interface. This consistency doesn't allow any flexibility in using the service, since a client will only know about the methods defined in the interface. The interface is the defining level for using this type of service.

However, services can be implemented in many different ways, and service implementations do in fact differ. For example, one service may be offered on a "take it or leave it" basis, while another might have a warranty attached. This does not affect how the client calls a service, but it may affect whether or not the client wants to use one service implementation or another. There is a need to allow for this, and the mechanism used in Jini is to put these differences in `Entry` objects. Typical objects supplied by vendors may include `Name` and `ServiceInfo`.

Clients can make use of the type interface and these additional entry items primarily in the selection of a service. But once clients have the service, are they just constrained to use it via the type interface? The type interface is designed to allow a client application to use the service in a programmatic way by calling methods. However, many services could probably benefit from some sort of user interface (UI). For example, a printer may supply a method to print a file, but it may have the capability to print multiple copies of the same file. Rather than relying on the client to be smart enough to figure this out, the printer vendor may want to call attention to this option by supplying a user-interface object with a special component for the number of copies.

NOTE *In this chapter I talk about interfaces we have been using throughout the book, and also about user interfaces. To avoid possible confusion, in this chapter I will use the term "type interface" to refer to a Java interface as used in the rest of this book, and "user interface" for any sort of interaction with the user.*

A client can only be expected to know about the type interface of a service. If it uses this to build a user interface, then at best it could only manage a fairly generic one that will work for all service implementations. A vendor will know much more detail about any particular implementation of a service, and so the vendor is best placed to supply the user interface. In some cases, the service vendor may be unwilling or incapable of supplying user interfaces for a service, and a third party may supply it.

When your video player becomes Jini-enabled, it would be a godsend for someone to supply a decent user interface for it, since the video-player vendors seem generally incapable of doing so! The `Entry` objects are not just restricted to providing static data; as Java objects, they are perfectly capable of running as user-interface objects.

User interfaces are not yet part of the Jini standard, but the Jini community (with a semi-formal organization as the "Jini Community") is moving toward a standard way of specifying many things, including user-interface standards and guidelines. Guideline number one from the serviceUI group is this: user interfaces for a service should be given in `Entry` objects.

User Interfaces from Factory Objects

In Chapter 13, some discussion was given to the location of code, using user-interface components as examples. That chapter suggested that user interfaces should not be created on the server side but on the client side—the user interface should be exported as a factory object that can create the user-interface on the client side.

More arguments can be given to support this approach:

- A service exported from a low-resource computer, such as an embedded Java engine, may not have the classes on the service side needed to create the user-interface (it may not have the Swing or even the AWT libraries).

- There may be many potential user interfaces for any particular service: Palm handhelds (many with small grayscale screens) require a different interface than a high-end workstation with a huge screen and enormous numbers of colors. It is not reasonable to expect the service to create every one of these user interfaces, but it could export factories capable of doing so.

- Localization of internationalized services cannot be done on the service side, only on the client side.

The service should export zero or more user-interface factories, with methods to create the interface, such as `getJFrame()`. The service and its user-interface factory will both be retrieved by the client. The client will then create the user interface. Note that the factory will not know the service object beforehand; if the factory was given one during its construction (on the service side), the factory would end up with a service-side copy of the service instead of a client-side copy. Therefore, when the factory is asked for a user-interface (on the client side), it should be passed the service. In fact, the factory should probably be passed all of the information about the service, as retrieved in the `ServiceItem` from a lookup service.

A typical factory is the one that returns a `JFrame`. This is defined by the type interface as follows:

```
package net.jini.lookup.ui.factory;

import javax.swing.JFrame;

public interface JFrameFactory {
    String TOOLKIT = "javax.swing";
    String TYPE_NAME = "net.jini.lookup.ui.factory.JFrameFactory";

    JFrame getJFrame(Object roleObject);
}
```

The factory imports the minimum number of classes needed to compile the type interface. The `JFrameFactory` above needs to import `javax.swing.JFrame` because the `getJFrame()` method returns a `JFrame`. An implementation of this type interface will probably use many more classes. The `roleObject` passes any necessary information to the UI constructor. This is usually the `ServiceItem`, as it contains all the information (including the service) that was retrieved from a lookup service. The factory can then create an object that acts as a user interface to the service, and can use any additional information in the `ServiceItem`, such as entries for `ServiceInfo` or `ServiceType`, which could be shown, say, in an "About" box.

A factory that returns a visual component, such as a `JFrame`, should not make the component visible. This will allow the client to set the `JFrame`'s size and placement before showing it. Similarly, a "playable" user interface, such as an audio file, should not be in a "playing" state.

Current Factories

A service may supply lots of these user interface factories, each capable of creating a different user interface object. This allows for the differing capabilities of viewing devices, or even for different user preferences. One user may always like a Web-style interface, another may be content with an AWT interface, a third may want the accessibility mechanisms possible with a Swing interface, and so on.

The set of proposed factories currently includes the following:

- `DialogFactory`, which returns an instance of `java.awt.Dialog` (or one of its subclasses)
- `FrameFactory`, which returns an instance of `java.awt.Frame` (or one of its subclasses)
- `JComponentFactory`, which returns an instance of `javax.swing.JComponent` (or one of its subclasses, such as a `JList`)
- `JDialogFactory`, which returns an instance of `javax.swing.JDialog` (or one of its subclasses)
- `JFrameFactory`, which returns an instance of `javax.swing.JFrame` (or one of its subclasses)
- `PanelFactory`, which returns an instance of `java.awt.Panel` (or one of its subclasses)

These factories are all defined as interfaces. An implementation will define a `getXXX()` method that will return a user interface object. The current set of factories returns objects that belong to the Swing or AWT classes. Factories added in later iterations of the specification may return objects belonging to other user interface styles, such as speech objects. Although an interface may specify that a method, such as `getJFrame()`, will return a `JFrame`, an implementation will in fact return a subclass of this, which also implements a role interface.

Marshalling Factories

There may be many factories for a service, and each of them will generate a different user interface. These factories and their user interfaces will be different for each service. The standard factory interfaces will probably be known to both clients and services, but the actual implementations of these will only be known to services (or maybe to third-party vendors who add a user interface to a service).

If a client receives a ServiceItem containing entries with many factory implementation objects, it will need to download the class files for all of these as it instantiates the Entry objects. There is a strong chance that each factory will be bundled into a jar file that also contains the user interface objects themselves, so if the entries directly contain the factories, then the client will need to download a set of class files before it even goes about the business of deciding which of the possible user interfaces it wants to select.

This downloading may take time on a slow connection, such as a wireless or home network link. It may also cost memory, which may be scarce in small devices such as PDAs. Therefore, it is advantageous to hide the actual factory classes until the client has decided that it does in fact want a particular class. Then, of course, it will have to download all of the class files needed by that factory.

In order to hide the factories, they are wrapped in a MarshalledObject. This keeps a representation of the factory and also a reference to its codebase, so that when it is unwrapped, the necessary classes can be located and downloaded. Clients should have the class files for MarshalledObject, because this class is part of the Java core. By putting a factory object into entries in this form, no attempt is made to download the actual classes required by the factory until it is unmarshalled.

The decision as to whether or not to unmarshall a class can be made on a separate piece of information, such as a set of Strings that hold the names of the factory class (and all of its superclasses and interfaces). This level of indirection is a bit of a nuisance, but not too bad:

```
if (typeNames.contains("net.jini.lookup.ui.factory.JFrameFactory") {
    factory = (JFrameFactory) marshalledObject.get();
    ....
}
```

A client that does not want to use a JFrameFactory will just not perform the preceding Boolean test. It will not call the unmarshalling get() method and will not attempt the coercion to JFrameFactory. This will avoid downloading classes that are not wanted. This indirection does place a responsibility on service-side programmers to ensure that the coercion will be correct. In effect, this is a maneuver to circumvent the type-safe model of Java purely for optimization purposes.

There is one final wrinkle when loading the class files for a factory: a running JVM may have many class loaders. When loading the files for a factory, you want to make sure that the class loader is one that will actually download the class files across the network as required. The class loader associated with the service itself will be the most appropriate loader for this.

UIDescriptor

An entry for a factory must contain the factory, itself, hidden in a MarshalledObject and some string representation of the factory's class(es). It may also need other descriptive information about the factory. The UIDescriptor captures all this:

```
package net.jini.lookup.entry;

public class UIDescriptor extends AbstractEntry {

    public String role;
    public String toolkit;
    public Set attributes;
    public MarshalledObject factory;

    public UIDescriptor();
    public UIDescriptor(String role, String toolkit,
                        Set attributes, MarshalledObject factory);

    public final Object getUIFactory(ClassLoader parentLoader)
          throws IOException, ClassNotFoundException;
}
```

There are several features in the UIDescriptor that we haven't mentioned yet, and the factory type appears to be missing (it is one of the attributes).

Toolkit

A user interface will typically require a particular package to be present or it will not function. For example, a factory that creates a JFrame will require the javax.swing package. These requirements can provide a quick filter for whether or not to accept a factory—if it is based on a package the client doesn't have, then it can just reject this factory.

This isn't a bulletproof means of selection. For example, the Java Media Framework is a fixed-size package designed to handle lots of different media types, so if your user interface is a QuickTime movie, you might specify the JMF package. However, the media types handled by the JMF package are not fixed, and they can depend on native code libraries. For example, the current Solaris version of the JMF package has a native code library to handle MPEG movies, which is not present in the Linux version. Having the package specified by the toolkit does not guarantee that the class files for this user interface will be present. It is primarily intended to narrow lookups based on the UIs offered.

Role

There are many possible roles for a user interface. For example, a typical user may be using the service, in which case the UI plays the "main" role. Alternatively, a system administrator may be managing the service, and he or she might require a different user interface, in which case the UI then plays the "admin" role.

The `role` field in a `UIDescriptor` is intended to describe these possible variations in the use of a user interface. The value of this field is a string, and to reduce the possibility of spelling errors that are not discovered until runtime, the value should be one of several constant string values. These string constants are defined in a set of type interfaces known as *role* interfaces. There are currently three role interfaces:

- The `net.jini.lookup.ui.MainUI` role is for the standard user interface used by ordinary clients of the service:

```
package net.jini.lookup.ui;
public interface MainUI {
    String ROLE = "net.jini.lookup.ui.MainUI";
}
```

- The `net.jini.lookup.ui.AdminUI` role is for use by the service's administrator:

```
package net.jini.lookup.ui;
public interface AdminUI {
    String ROLE = "net.jini.lookup.ui.AdminUI";
}
```

- The `net.jini.lookup.ui.AboutUI` role is for information about the service, which can be presented by a user interface object:

```
package net.jini.lookup.ui;
public interface AboutUI {
    String ROLE = "net.jini.lookup.ui.AboutUI";
}
```

A service will specify a role for each of the user interfaces it supplies. This role is given in a number of ways for different objects:

- The `role` field in the `UIDescriptor` must be set to the `String ROLE` of one of these three role interfaces.

- The user interface indicates that it acts a role by implementing the particular role specified.

- The factory does not explicitly know about the role, but the factory contained in a `UIDescriptor` must produce a user interface implementing the role.

The service must ensure that the `UIDescriptors` it produces follows these rules. How it actually does so is not specified. There are several possibilities, including these:

- When a factory is created, the role is passed in through a constructor. It can then use this role to cast the `roleObject` in the `getXXX()` method to the expected class (currently this is always a `ServiceItem`).

- There could be different factories for different roles, and the `UIDescriptor` should have the right factory for that role.

The factory could perform some sanity checking if desired; since all `roleObjects` are (presently) the service items, it could search through these items for the `UIDescriptor`, and then check that its `role` matches what the factory expects.

There has been much discussion about "flavors" of roles, such as an "expert" role or a "learner" role. This has been deferred because it is too complicated, at least for the first version of the specification.

Attributes

The `attributes` section of a `UIDescriptor` can carry any other information about the user interface object that the service thinks might be useful to clients trying to decide which user interface to choose. Currently this includes the following:

- A `UIFactoryTypes` object, which contains a set of `Strings` for the fully qualified class names of the factory that this entry contains. The current factory hierarchy is very shallow, so this may be just a singleton set, like this:

  ```
  Set attribs = new HashSet();
  Set typeNames = new HashSet();
  typeNames.add(JFrameFactory.TYPE_NAME);
  attribs.add(new UIFactoryTypes(typeNames));
  ```

  Note that a client is not usually interested in the actual type of the factory, but rather in the interface it implements. This is just like Jini services themselves, where we only need to know the methods that can be called and are not concerned with the implementation details.

- An AccessibleUI object. Inclusion of this object indicates that the user interface implements javax.accessibility.Accessible and that the user interface would work well with assistive technologies.

- A Locales object, which specifies the locales supported by the user interface.

- A RequiredPackages object, which contains information about all of the packages that the user interface needs to run. This is not a guarantee that the user interface will actually run, nor a guarantee that it will be a usable interface, but it may help a client decide whether or not to use a particular user interface.

File Classifier UI Example

The file classifier has been used throughout this book as a simple example of a service to illustrate various features of Jini. We can use it here too, by supplying simple user interfaces to the service. Such a user interface would consist of a text field for entering a filename, and a display to show the MIME type of the filename. There is only a "main" role for this service, as no administration needs to be performed.

Figure 19-1 shows what a user interface for a file classifier could look like.

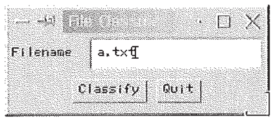
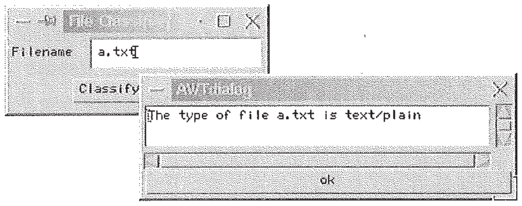


Figure 19-1. FileClassifier user interface

After the service has been invoked, it could pop up a dialog box, as shown in Figure 19-2.



The type of file a.txt is text/plain

Figure 19-2. FileClassifier return dialog box

A factory for the "main" role that will produce an AWT `Frame` is shown next:

```
/**
 * FileClassifierFrameFactory.java
 */

package ui;

import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.lookup.entry.UIDescriptor;
import java.awt.Frame;
import net.jini.core.entry.Entry;
import net.jini.core.lookup.ServiceItem;

public class FileClassifierFrameFactory implements FrameFactory {

    /**
     * Return a new FileClassifierFrame that implements the
     * MainUI role
     */
    public Frame getFrame(Object roleObject) {
        // we should check to see what role we have to return
        if (! (roleObject instanceof ServiceItem)) {
            // unknown role type object
            // can we return null?
            return null;
        }
        ServiceItem item = (ServiceItem) roleObject;

        // Do sanity checking that the UIDescriptor has a MainUI role
        Entry[] entries = item.attributeSets;
        for (int n = 0; n < entries.length; n++) {
            if (entries[n] instanceof UIDescriptor) {
                UIDescriptor desc = (UIDescriptor) entries[n];
                if (desc.role.equals(net.jini.lookup.ui.MainUI.ROLE)) {
                    // Ok, we are in the MainUI role, so return a UI for that
                   Frame frame = new FileClassifierFrame(item, "File Classifier");
                    return frame;
                }
            }
        }
        // couldn't find a role the factory can create
        return null;
    }
} // FileClassifierFrameFactory
```

The user interface object that performs this role is as follows:

```
/**
 * FileClassifierFrame.java
 */

package ui;

import java.awt.*;
import java.awt.event.*;
import net.jini.lookup.ui.MainUI;
import net.jini.core.lookup.ServiceItem;
import common.MIMEType;
import common.FileClassifier;
import java.rmi.RemoteException;

/**
 * Object implementing MainUI for FileClassifier.
 */
public class FileClassifierFrame extends Frame implements MainUI {

    ServiceItem item;
    TextField text;

    public FileClassifierFrame(ServiceItem item, String name) {
        super(name);

        Panel top = new Panel();
        Panel bottom = new Panel();
        add(top, BorderLayout.CENTER);
        add(bottom, BorderLayout.SOUTH);

        top.setLayout(new BorderLayout());
        top.add(new Label("Filename"), BorderLayout.WEST);
        text = new TextField(20);
        top.add(text, BorderLayout.CENTER);

        bottom.setLayout(new FlowLayout());
        Button classify = new Button("Classify");
        Button quit = new Button("Quit");
        bottom.add(classify);
        bottom.add(quit);
        // listeners
        quit.addActionListener(new QuitListener());
        classify.addActionListener(new ClassifyListener());

        // We pack, but don't make it visible
        pack();
    }

    class QuitListener implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            System.exit(0);
        }
    }

    class ClassifyListener implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            String fileName = text.getText();
            final Dialog dlg = new Dialog((Frame) text.getParent().getParent());
            dlg.setLayout(new BorderLayout());
            TextArea response = new TextArea(3, 20);

            // invoke service
            FileClassifier classifier = (FileClassifier) item.service;
            MIMEType type = null;
            try {
                type = classifier.getMIMEType(fileName);
                if (type == null) {
                    response.setText("The type of file " + fileName +
                                     " is unknown");
                } else {
                    response.setText("The type of file " + fileName +
                                     " is " + type.toString());
                }
            } catch(RemoteException e) {
                response.setText(e.toString());
            }

            Button ok = new Button("ok");
            ok.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    dlg.setVisible(false);
                }
            });
            dlg.add(response, BorderLayout.CENTER);
            dlg.add(ok, BorderLayout.SOUTH);
            dlg.setSize(300, 100);
            dlg.setVisible(true);
        }
    }

} // FileClassifierFrame
```

The server that delivers both the service and the user interface has to prepare a UIDescriptor. In this case, it only creates one such object for a single user interface, but if the server exported more interfaces, it would simply create more descriptors. Here is the server code:

```
/**
 * FileClassifierServer.java
 */

package ui;

import complete.FileClassifierImpl;

import net.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceRegistrar;
import java.rmi.RemoteException;
import net.jini.lookup.ServiceIDListener;
import net.jini.lease.LeaseRenewalManager;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.DiscoveryListener;
import net.jini.core.entry.Entry;

import net.jini.lookup.ui.MainUI;
import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.lookup.entry.UIDescriptor;
import net.jini.lookup.ui.attribute.UIFactoryTypes;

import java.rmi.MarshalledObject;
import java.io.IOException;
import java.util.Set;
import java.util.HashSet;
```

```
public class FileClassifierServer
    implements ServiceIDListener {

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

        // stay around forever
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(InterruptedException e) {
                // do nothing
            }
        }
    }

    public FileClassifierServer() {

        JoinManager joinMgr = null;

        // The typenames for the factory
        Set typeNames = new HashSet();
        typeNames.add(FrameFactory.TYPE_NAME);

        // The attributes set
        Set attribs = new HashSet();
        attribs.add(new UIFactoryTypes(typeNames));

        // The factory
        MarshalledObject factory = null;
        try {
            factory = new MarshalledObject(new FileClassifierFrameFactory());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(2);
        }
        UIDescriptor desc = new UIDescriptor(MainUI.ROLE,
                                             FileClassifierFrameFactory.TOOLKIT,
                                             attribs,
                                             factory);

        Entry[] entries = {desc};
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            joinMgr = new JoinManager(new FileClassifierImpl(), /* service */
                                      entries /* attr sets */,
                                      this /* ServiceIDListener*/,
                                      mgr /* DiscoveryManagement */,
                                      new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void serviceIDNotify(ServiceID serviceID) {
        // called as a ServiceIDListener
        // Should save the ID to permanent storage
        System.out.println("got service ID " + serviceID.toString());
    }

} // FileClassifierServer
```

Finally, a client needs to look for and use this user interface. The client finds a service as usual and then does a search through the `Entry` objects, looking for a `UIDescriptor`. Once it has a descriptor, it can check whether the descriptor meets the requirements of the client. Here we shall check whether it plays a `MainUI` role and can generate an AWT `Frame`:

```
package client;

import common.FileClassifier;
import common.MIMEType;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ClientLookupManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;
import net.jini.core.entry.Entry;
import net.jini.lookup.ui.MainUI;
import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.lookup.entry.UIDescriptor;
import net.jini.lookup.ui.attribute.UIFactoryTypes;

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

import java.util.Set;
import java.util.Iterator;
import java.net.URL;

/**
 * TestFrameUI.java
 */

public class TestFrameUI {

    private static final long WAITFOR = 100000L;

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

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(2*WAITFOR);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestFrameUI() {
        ClientLookupManager clientMgr = null;

        System.setSecurityManager(new RMISecurityManager());

        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            clientMgr = new ClientLookupManager(mgr,
                                                new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        Class [] classes = new Class[] {FileClassifier.class};
        UIDescriptor desc = new UIDescriptor(MainUI.ROLE,
                                             FrameFactory.TOOLKIT,
                                             null, null);
        Entry [] entries = {desc};

        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       entries);

        ServiceItem item = null;
        try {
            item = clientMgr.lookup(template,
                                    null, /* no filter */
                                    WAITFOR /* timeout */);
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        if (item == null) {
            // couldn't find a service in time
            System.out.println("no service");
            System.exit(1);
        }

        // We now have a service that plays the MainUI role and
        // uses the FrameFactory toolkit of "java.awt".
        // We now have to find if there is a UIDescriptor
        // with a Factory generating an AWT Frame
        checkUI(item);
    }

    private void checkUI(ServiceItem item) {
        // Find and check the UIDescriptors
        Entry[] attributes = item.attributeSets;
        for (int m = 0; m < attributes.length; m++) {
            Entry attr = attributes[m];
            if (attr instanceof UIDescriptor) {
                // does it deliver an AWT Frame?
                checkForAWTFrame(item, (UIDescriptor) attr);
            }
        }
    }

    private void checkForAWTFrame(ServiceItem item, UIDescriptor desc) {
        Set attributes = desc.attributes;
        Iterator iter = attributes.iterator();
        while (iter.hasNext()) {
            // search through the attributes, to find a UIFactoryTypes
            Object obj = iter.next();
            if (obj instanceof UIFactoryTypes) {
                UIFactoryTypes types = (UIFactoryTypes) obj;
                // see if it produces an AWT Frame Factory
                if (types.isAssignableTo(FrameFactory.class)) {
                    FrameFactory factory = null;
                    try {
                        factory = (FrameFactory) desc.getUIFactory(this.getClass().
                                                                getClassLoader());
                    } catch(Exception e) {
                        e.printStackTrace();
                        continue;
                    }

                    Frame frame = factory.getFrame(item);
                    frame.setVisible(true);
                }
            }
        }
    }


} // TestFrameUI
```

Images

User interfaces often contain images. They may be used as icons in toolbars, as general images on the screen, or as the icon image when the application is iconified. When a user interface is created on the client, these images will also need to be created and installed in the relevant part of the application. Images are not serializable, so they cannot be created on the server and exported as live objects in some manner. They need to be created from scratch on the client.

The Swing package contains a convenience class called `ImageIcon`. This class can be instantiated from a byte array, a filename, or most interestingly here, from a URL. So, if an image is stored where an HTTP server can find it, the `ImageIcon` constructor can use this version directly. There may be failures in this approach: the URL may be incorrect or malformed or the image may not exist on the HTTP server. Suitable code to create an image from a URL is as follows:

```
ImageIcon icon = null;
        try {
            icon = new ImageIcon(new URL("http://localhost/images/MINDSTORMS.ps"));
            switch (icon.getImageLoadStatus()) {
            case MediaTracker.ABORTED:
            case MediaTracker.ERRORED:
                System.out.println("Error");
                icon = null;
                break;
            case MediaTracker.COMPLETE:
                System.out.println("Complete");
                break;
            case MediaTracker.LOADING:
                System.out.println("Loading");
                break;
            }
        } catch(java.net.MalformedURLException e) {
            e.printStackTrace();
        }
        // icon is null or is a valid image
```

ServiceType

A user interface may use code like that in the previous section directly to include images. The service may also supply useful images and other human-oriented information in a `ServiceType` entry object. The `ServiceType` class is defined as follows:

```
package net.jini.lookup.entry;

public class ServiceType {
    public String getDisplayName();      // Return the localized display
                                         // name of this service.
    public Image getIcon(int iconKind)   // Get an icon for this service.
    public String getShortDescription()  // Return a localized short
                                         // description of this service.
}
```

The class is supplied with empty implementations, returning `null` for each method. A service will need to supply a subclass with useful implementations of the methods. This is a useful class that could be used to supply images and information that may be common to a number of different user interfaces for a service, such as a minimized image.

MINDSTORMS UI Example

In Chapter 17, an example was given, in the "Getting It Running" section, of a client supplying a user interface to a MINDSTORMS service. This client not only knew that the service was a MINDSTORMS robot, but that it was a particular robot for which it could use a customized UI. In this section, we'll give two user interfaces for the MINDSTORMS "RoverBot," one of which is fairly general and could be used for any robot, and another that is customized to the RoverBot. The service is responsible for creating and exporting both of these user interfaces to a client.

RCXLoaderFrame

A MINDSTORMS robot is primarily defined by the `RCXPort` interface. The Jini version is defined by the `RCXPortImplementation` interface:

```
/**
 * 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
```

This type interface allows programs to be downloaded and run and instructions to be sent for direct execution. As it stands, the client needs to call these interface methods directly. To make it more useable for the human trying to drive a robot, some sort of user interface would be useful.

There can be several general purpose user interfaces for the RCX robot, including these:

- Enter machine code (somehow) and download that.
- Enter RCX assembler code in the form of strings, and then assemble and download them.
- Enter NQC (Not Quite C) code, and then compile and download it.

The set of RCX classes by Laverde at `http://www.escape.com/~dario/java/rcx` includes a standalone application called `RCXLoader`, which does the second of these options. We can steal code from `RCXLoader` and some of his other classes to define an `RCXLoaderFrame` class:

```
package rcx.jini;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
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;
import rcx.*;

/*
 * RCXLoaderFrame
 * @author Dario Laverde
 * @author Jan Newmarch
 * @version 1.1
 * Copyright 1999 Dario Laverde, under terms of GNU LGPL
 */

public class RCXLoaderFrame extends Frame
    implements ActionListener, WindowListener, RemoteEventListener
{
    private String      portName;
    private RCXPortInterface     rcxPort;
    private Panel       textPanel;
    private Panel       topPanel;
    private TextArea    textArea;
    private TextField   textField;
    private Button      tableButton;
    private Properties  parameters;
    private int inByte;
    private int charPerLine = 48;
    private int lenCount;
    private StringBuffer sbuffer;
    private byte[] byteArray;
    private Frame opcodeFrame;
    private TextArea opcodeTextArea;

    public static Hashtable Opcodes=new Hashtable(55);
```

```
static {
    Opcodes.put(new Byte((byte)0x10),"PING           ,void, void,P");
    Opcodes.put(new Byte((byte)0x12),"GETVAL         ,
    byte src byte arg, short val,P");
    Opcodes.put(new Byte((byte)0x13),"SETMOTORPOWER  ,
    byte motors byte src byte arg, void,CP");
    Opcodes.put(new Byte((byte)0x14),"SETVAL         ,
    byte index byte src byte arg, void,CP");
    // Opcodes truncated to save space in listing
}

// added port interface parameter to Dario's code
public RCXLoaderFrame(RCXPortInterface port) {
        super("RCX Loader");

    // changed from Dario's code
    rcxPort = port;

    addWindowListener(this);

    topPanel  = new Panel();
    topPanel.setLayout(new BorderLayout());

    tableButton = new Button("table");
    tableButton.addActionListener(this);

    textField = new TextField();
    // textField.setEditable(false);
    // textField.setEnabled(false);
    // tableButton.setEnabled(false);
    textField.addActionListener(this);

    textPanel = new Panel();
    textPanel.setLayout(new BorderLayout(5,5));

    topPanel.add(textField,"Center");
    topPanel.add(tableButton,"East");
    textPanel.add(topPanel,"North");

    textArea = new TextArea();
    // textArea.setEditable(false);
    textArea.setFont(new Font("Courier",Font.PLAIN,12));
    textPanel.add(textArea,"Center");
```

```
        add(textPanel, "Center");

        textArea.setText("initializing...\n");

        Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
        setBounds(screen.width/2-370/2,screen.height/2-370/2,370,370);
        // setVisible(true);

        // changed listener type from Dario's code
        try {
            // We are remote to the object we are listening to
            // (the RCXPort), so the RCXPort must get a stub object
            // for us. We have subclassed from Frame, not from
            // UnicastRemoteObject. So we must export ourselves
            // for the remote references to work
            UnicastRemoteObject.exportObject(this);
            rcxPort.addListener(this);
        } catch(Exception e) {
            textArea.append(e.toString());
        }
        tableButton.setEnabled(true);
    }

    public void actionPerformed(ActionEvent evt) {
        Object obj = evt.getSource();
        if(obj==textField) {
            String strInput = textField.getText();
            textField.setText("");
            textArea.append("> "+strInput+"\n");
            try {
                byteArray = rcxPort.parseString(strInput);
            } catch(RemoteException e) {
                textArea.append(e.toString());
            }
            // byteArray = RCXOpcode.parseString(strInput);
            if(byteArray==null) {
                textArea.append("Error: illegal hex character or length\n");
                return;
            }
            if(rcxPort!=null) {
                try {
                    if(!rcxPort.write(byteArray)) {
                        textArea.append("Error: writing data to port
                    "+portName+"\n");
                }
            } catch(Exception e) {
                textArea.append(e.toString());
            }
        }
    }
    else if(obj==tableButton) {
        // make this all in the ui side
        showTable();
        setLocation(0,getLocation().y);
    }
}

public void windowActivated(WindowEvent e) { }
public void windowClosed(WindowEvent e) { }
public void windowDeactivated(WindowEvent e) { }
public void windowDeiconified(WindowEvent e) { }
public void windowIconified(WindowEvent e) { }
public void windowOpened(WindowEvent e) { }
public void windowClosing(WindowEvent e) {
    /*
    if(rcxPort!=null)
            rcxPort.close();
    */
    System.exit(0);
}

public void notify(RemoteEvent evt) throws UnknownEventException,
                                            java.rmi.RemoteException {

    long id = evt.getID();
    long seqNo = evt.getSequenceNumber();
    if (id == RCXPortInterface.MESSAGE_EVENT) {
        byte[] message = rcxPort.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) + " ");
        }
        textArea.append(sbuffer.toString());
        System.out.println("MESSAGE: " + sbuffer.toString());
        } else if (id == RCXPortInterface.ERROR_EVENT) {
            textArea.append(rcxPort.getError(seqNo));
        } else {
            throw new UnknownEventException("Unknown message " + evt.getID());
        }
    }

    public void showTable()
    {
        if(opcodeFrame!=null)
        {
            opcodeFrame.dispose();
            opcodeFrame=null;
            return;
        }
        opcodeFrame = new Frame("RCX Opcodes Table");
        Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
        opcodeFrame.setBounds(screen.width/2-70,0,
                              screen.width/2+70,screen.height-25);
        opcodeTextArea = new TextArea("   Opcode          ,parameters, response,
                                    C=program command P=remote command\n",60,100);
        opcodeTextArea.setFont(new Font("Courier",Font.PLAIN,10));
        opcodeFrame.add(opcodeTextArea);
        Enumeration k = Opcodes.keys();
        for (Enumeration e = Opcodes.elements(); e.hasMoreElements();) {
            String tmp = Integer.toHexString(((Byte)k.nextElement()).intValue());
            tmp = tmp.substring(tmp.length()-2)+" "+(String)e.nextElement()+"\n";
            opcodeTextArea.append(tmp);
        }
        opcodeTextArea.setEditable(false);
        opcodeFrame.setVisible(true);
    }
}
```

RCXLoaderFrameFactory

The factory object for the RCX is now easy to define—it just returns a `RCXLoaderFrame` in the `getUI()` method:

```
/**
 * RCXLoaderFrameFactory.java
 */
```

```
package rcx.jini;

import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.core.lookup.ServiceItem;
import java.awt.Frame;

public class RCXLoaderFrameFactory implements FrameFactory {

    public Frame getFrame(Object roleObj) {
        ServiceItem item= (ServiceItem) roleObj;
        RCXPortInterface port = (RCXPortInterface) item.service;
        return new RCXLoaderFrame(port);
    }

} // RCXLoaderFrameFactory
```

Exporting the FrameFactory

The factory object is exported by making it a part of a `UIDescriptor` entry object with a role, toolkit, and attributes:

```
Set typeNames = new HashSet();
typeNames.add(FrameFactory.TYPE_NAME);

Set attribs = new HashSet();
attribs.add(new UIFactoryTypes(typeNames));
// add other attributes as desired

MarshalledObject factory = null;
try {
    factory = new MarshalledObject(new
                                   RCXLoaderFrameFactory());
} catch(Exception e) {
    e.printStackTrace();
    System.exit(2);
}

UIDescriptor desc = new UIDescriptor(MainUI.ROLE,
                                     FrameFactory.TOOLKIT,
                                     attribs,
                                     factory);
Entry[] entries = {desc};
```

```
JoinManager joinMgr = new JoinManager(impl,
                                       entries,
                                       this,
                                       new LeaseRenewalManager());
```

Customized User Interfaces

The RCXLoaderFrame is a general interface to any RCX robot. Of course, there could be many other such interfaces, differing in the classes used, the amount of international support, the appearance, etc. All the variations, however, will just use the standard RCXPortInterface, because that is all they know about.

The LEGO pieces can be combined in a huge variety of ways, and the RCX itself is programmable, so you can build an RCX car, an RCX crane, an RCX maze-runner, and so on. Each different robot can be driven by the general interface, but most could benefit from a custom-built interface for that type of robot. This is typical: for example, every blender could be driven from a general blender user interface (using the possibly forthcoming standard blender interface :-). But the blenders from individual vendors would have their own customized user interface for their brand of blender.

I have been using an RCX car. While it can do lots of things, it has been convenient to use five commands for demonstrations: forward, stop, back, left, and right, with a user interface as shown in Figure 19-3.

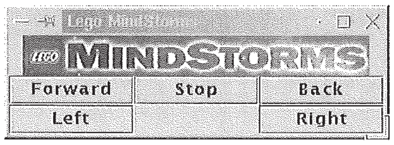


Figure 19-3. This is a control panel taken from the LEGO® MINDSTORMS™ Robotics Invention System RCX programming system.

In Chapter 17, this appearance was hard-coded into the client. Since the client was just searching for *any* MINDSTORMS robot, it really shouldn't know about this sort of detail and should get this user interface from the robot service.

CarJFrame

The CarJFrame class produces the user interface as a Swing JFrame, with the buttons generating specific RCX code for this model.

```
package rcx.jini;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.UnknownEventException;
import java.net.URL;
import java.rmi.RemoteException;

class CarJFrame extends JFrame
    implements RemoteEventListener, ActionListener {

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

    protected int state = STOPPED;

    protected RCXPortInterface port = null;

    JFrame frame;
    JTextArea text;

    public CarJFrame(RCXPortInterface port) {
        super() ;
        this.port = port;

        frame = new JFrame("LEGO MINDSTORMS");
        Container content = frame.getContentPane();
        JLabel label = null;
        ImageIcon icon = null;
        try {
            icon = new ImageIcon(new
                       URL("http://www.LEGOMINDSTORMS.com/images/home_logo.ps"));
            switch (icon.getImageLoadStatus()) {
            case MediaTracker.ABORTED:
            case MediaTracker.ERRORED:
                System.out.println("Error");
                icon = null;
                break;
            case MediaTracker.COMPLETE:
                System.out.println("Complete");
                break;
            case MediaTracker.LOADING:
                System.out.println("Loading");
                break;
            }
        } catch(java.net.MalformedURLException e) {
            e.printStackTrace();
        }
        if (icon != null) {
            label = new JLabel(icon);
        } else {
            label = new JLabel("MINDSTORMS");
        }

        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();
    byte[] command;

    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());
        }
    }
}
```

CarJFrameFactory

The factory generates a `CarJFrame` object, like this:

```
/**
 * CarJFrameFactory.java
 */

package rcx.jini;

import net.jini.lookup.ui.factory.JFrameFactory;
import net.jini.core.lookup.ServiceItem;
import javax.swing.JFrame;

public class CarJFrameFactory implements JFrameFactory {

    public JFrame getJFrame(Object roleObj) {
        ServiceItem item  = (ServiceItem) roleObj;
        RCXPortInterface port = (RCXPortInterface) item.service;
        return new CarJFrame(port);
    }

} // CarJFrameFactory
```

Exporting the FrameFactory

Both of the user interfaces discussed—the RCXLoaderFrame and the CarJFrame—can be exported by expanding the set of Entry objects.

```
// generic UI
Set genericAttribs = new HashSet();
Set typeNames = new HashSet();
typeNames.add(FrameFactory.TYPE_NAME);
genericAttribs.add(new UIFactoryTypes(typeNames));
MarshalledObject genericFactory = null;
try {
    genericFactory = new MarshalledObject(new
                                  RCXLoaderFrameFactory());
} catch(Exception e) {
    e.printStackTrace();
            System.exit(2);
}
UIDescriptor genericDesc = new UIDescriptor(MainUI.ROLE,
                                            FrameFactory.TOOLKIT,
                                               genericAttribs,
                                            genericFactory);

// car UI
Set carAttribs = new HashSet();
typeNames = new HashSet();
typeNames.add(JFrameFactory.TYPE_NAME);
carAttribs.add(new UIFactoryTypes(typeNames));
MarshalledObject carFactory = null;
try {
    carFactory = new MarshalledObject(new CarJFrameFactory());
} catch(Exception e) {
    e.printStackTrace();
    System.exit(2);
}
UIDescriptor carDesc = new UIDescriptor(MainUI.ROLE,
                                        JFrameFactory.TOOLKIT,
                                        carAttribs,
                                        carFactory);

Entry[] entries = {genericDesc, carDesc};

JoinManager joinMgr = new JoinManager(impl,
                                      entries,
                                        this,
                                        new LeaseRenewalManager());
```

The RCX Client

The following client will start up all user interfaces that implement the main UI role and that use a `Frame` or `JFrame`:

```
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;
import net.jini.core.entry.Entry;
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.lookup.ServiceItem;

import net.jini.lookup.entry.UIDescriptor;
import net.jini.lookup.ui.MainUI;
import net.jini.lookup.ui.attribute.UIFactoryTypes;
import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.lookup.ui.factory.JFrameFactory;

import java.util.Set;
import java.util.Iterator;

/**
 * TestRCX2.java
 */
```

```
public class TestRCX2 implements DiscoveryListener {

    public static void main(String argv[]) {
        new TestRCX2();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(1000000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestRCX2() {
        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;

        UIDescriptor desc = new UIDescriptor(MainUI.ROLE, null, null, null);
        Entry[] entries = {desc};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       entries);

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Service found");
            ServiceRegistrar registrar = registrars[n];
            ServiceMatches matches = null;
            try {
                matches = registrar.lookup(template, 10);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(2);
            }
            for (int nn = 0; nn < matches.items.length; nn++) {
                ServiceItem item = matches.items[nn];
                port = (RCXPortInterface) item.service;
                if (port == null) {
                    System.out.println("port null");
                    continue;
                }

                Entry[] attributes = item.attributeSets;
                for (int m = 0; m < attributes.length; m++) {
                    Entry attr = attributes[m];
                    if (attr instanceof UIDescriptor) {
                        showUI(port, item, (UIDescriptor) attr);
                    }
                }
            }



        }
    }

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

    private void showUI(RCXPortInterface port,
                        ServiceItem item,
                        UIDescriptor desc) {
        Set attribs = desc.attributes;
        Iterator iter = attribs.iterator();
        while (iter.hasNext()) {
            Object obj = iter.next();
            if (obj instanceof UIFactoryTypes) {
                UIFactoryTypes types = (UIFactoryTypes) obj;
                Set typeNames = types.getTypeNames();
                if (typeNames.contains(FrameFactory.TYPE_NAME)) {
                    FrameFactory factory = null;
                    try {
                       factory = (FrameFactory) desc.getUIFactory(this.getClass().
                                                                getClassLoader());
                    } catch(Exception e) {
                        e.printStackTrace();
                        continue;
                    }
                    Frame frame = factory.getFrame(item);
                    frame.setVisible(true);
                } else if (typeNames.contains(JFrameFactory.TYPE_NAME)) {
                    JFrameFactory factory = null;
                    try {
                       factory = (JFrameFactory) desc.getUIFactory(this.getClass().
                                                                getClassLoader());
                    } catch(Exception e) {
                        e.printStackTrace();
                        continue;
                    }
                    JFrame frame = factory.getJFrame(item);
                    frame.setVisible(true);
                }
            } else {
                System.out.println("non-gui entry");
            }
        }
    }
} // TestRCX
```

Summary

The serviceUI group is evolving a standard mechanism for services to distribute user interfaces for Jini services. The preference is to do this by `Entry` objects that contain factories for producing user interfaces.

CHAPTER 20

Activation

MANY OF THE EXAMPLES IN EARLIER CHAPTERS use RMI proxies for services. These services subclass `UnicastRemoteObject` and live within a server whose principal task is to keep the service alive and registered with lookup services. If the server fails to renew leases, then lookup services will eventually discard it; if it fails to keep itself and its service alive, then the service will not be available when a client wants to use it.

This results in a server and a service that will be idle most of the time, probably swapped out to disk, but still using virtual memory. In JDK 1.2, the memory requirements on the server side can be enormous (hopefully this will be fixed, but at the moment this is a severe embarrassment to Java and a potential threat to the success of Jini). In JDK 1.2, there is an extension to RMI called Activation, which allows an idle object to die and be recalled to life when needed. In this way, it does not occupy virtual memory while idle. Of course, a process needs to be alive to restore such objects, and RMI supplies a daemon `rmid` to manage this. In effect, `rmid` acts as another virtual memory manager because it stores information about dormant Java objects in its own files and restores them from there as needed.

There is a serious limitation to `rmid`: it is a Java program itself, and when running also uses enormous amounts of memory. So it only makes sense to use this technique when you expect to be running a number of largely idle services on the same machine. When a service is recalled to life, or activated, a new JVM may be started to run the object. This again increases memory use.

If memory use were the only concern, there are a variety of other systems, such as `echidna`, that run multiple applications within a single JVM. These may be adequate to solve the memory issues. However, RMI Activation is also designed to work with distributed objects and allows JVMs to hold remote references to objects that are no longer active. Instead of throwing a remote exception when trying to access these objects, the Activation system tries to resurrect the object using `rmid` to give a valid (and new) reference. Of course, if it fails to do this, it will throw an exception anyway.

The principal place that this is used in the standard Jini distribution is with the `reggie` lookup service. `reggie` is an activatable service that starts, registers itself with `rmid`, and then exits. Whenever lookup services are required, `rmid` restarts `reggie` in a new JVM. Clients of the lookup service are unaware of this mechanism; they simply make calls on their proxy `ServiceRegistration` object and the Activation system looks after the rest. The main problem is for the system administrator—getting `reggie` to work in the first place!

A Service Using Activation

The major concepts in Activation are the activatable object itself (which extends `java.rmi.activation.Activatable`) and the environment in which it runs, an `ActivationGroup`.

A JVM may have an activation group associated with it. If an object needs to be activated and there is already a JVM running its group, then it is restarted within that JVM. Otherwise, a new JVM is started. An activation group may hold a number of cooperating objects.

The next sections show how to create a service as an activatable object that starts life in a server that sets up the activation group. Issues related to activation, such as security and state maintenance, will also be discussed.

The Service

An activable object subclasses from `Activatable` and uses a special two-argument constructor that will be called when the object needs to be reconstructed. There is a standard implementation of this constructor that just calls the superclass constructor:

```
public ActivatableImpl(ActivationID id, MarshalledObject data)
    throws RemoteException {
    super(id, 0);
}
```

(The use of the marshalled object parameter is discussed later in the "Maintaining State" section). Adding this constructor is all that is normally needed to change a remote service (that implements `UnicastRemoteObject`) into an activatable service. For example, an activatable version of the remote file classifier described in Chapter 9 in the "RMI Proxy for FileClassifier" section is as follows:

```
package activation;

import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;

import common.MIMEType;
import common.FileClassifier;
import rmi.RemoteFileClassifier;

/**
```

```
 * FileClassifierImpl.java
 */

public class FileClassifierImpl extends Activatable
                               implements RemoteFileClassifier {

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        if (fileName.endsWith(".gif")) {
            return new MIMEType("image", "gif");
        } else if (fileName.endsWith(".jpeg")) {
            return new MIMEType("image", "jpeg");
        } else if (fileName.endsWith(".mpg")) {
            return new MIMEType("video", "mpeg");
        } else if (fileName.endsWith(".txt")) {
            return new MIMEType("text", "plain");
        } else if (fileName.endsWith(".html")) {
            return new MIMEType("text", "html");
        } else
            // fill in lots of other types,
            // but eventually give up and
            return new MIMEType(null, null);
    }


    public FileClassifierImpl(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException {
        super(id, 0);
    }

} // FileClassifierImpl
```

Note that an activatable object cannot have a default no-args constructor to initialize itself, since this new constructor is required for the object to be constructed by the activation system.

The Server

The server needs to create an activation group for the objects to run in. The main issue involved here is to set a security policy file. There are two security policies in activatable objects: the policy used to create the server and export the service, and the policy used to run the service. The activation group sets a policy file for running methods of the service object. The policy file for the server is set using the normal -Djava.security.policy=... argument to start the server. After setting various parameters, the activation group is set for the JVM by ActivationGroup.createGroup().

Remote objects that subclass UnicastRemoteObject are created in the normal way using a constructor on the server. Activatable objects are not constructed in the server but are instead registered with rmid, which will look after construction on an as-needed basis.

In order to create activatable objects, rmid needs to know the class name and the location of the class files. The server wraps these up in an ActivationDesc, and registers this with rmid by using Activatable.register(). This returns an RMI stub object that can be registered with lookup services using the ServiceRegistrar.register() methods. This is also a little different from subclasses of UnicastRemoteObject, which pass an object that is converted to a stub by the RMI runtime. The required actions, in point form, are as follows:

- A service creates a subclass of UnicastRemoteObject using its constructor.
- A subclass of Activatable is created by rmid using a special constructor.
- For a UnicastRemoteObject object, the server needs to know the class files for the class in its CLASSPATH and the client needs to know the class files for the stub from an HTTP server.
- For an Activatable object, rmid needs to know the class files from an HTTP server, the server must be able to find the stub files from its CLASSPATH, and the client must be able to get the stub files from an HTTP server.
- A server hands a UnicastRemoteObject object to the ServiceRegistrar.register(). This is converted to the stub by the RMI runtime.
- A server gets a stub for an Activatable object from Activatable.register(). This stub is given directly to ServiceRegistrar.register().

Changes need to be made to servers that export activatable objects instead of unicast remote objects. The server in Chapter 9, in the "RMI Proxy for FileClassifier" section, creates a unicast remote object and exports its RMI proxy to lookup services by passing the remote object to the ServiceRegistrar.register() method. The changes for such servers to export activatable objects are as follows:

- An activation group has to be created with a security policy file.
- The service is not created explicitly but is instead registered with rmid.

- The return object from the registration is a stub that can be registered with lookup services.
- Leasing vanishes—the server just exits. The service will just expire after a while. See the "LeaseRenewalService" section later in the chapter for more details on how to keep the service alive.

The file classifier server using an activatable service would look like this:

```
package activation;

import rmi.RemoteFileClassifier;

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.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
import java.rmi.RMISecurityManager;
import java.rmi.MarshalledObject;
import java.rmi.activation.ActivationDesc;
import java.rmi.activation.ActivationGroupDesc;
import java.rmi.activation.ActivationGroupDesc.CommandEnvironment;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationGroup;
import java.rmi.activation.ActivationGroupID;

import java.util.Properties;

import java.rmi.activation.UnknownGroupException;
import java.rmi.activation.ActivationException;
import java.rmi.RemoteException;

/**
 * FileClassifierServer.java
 */

public class FileClassifierServer implements DiscoveryListener {

    static final protected String SECURITY_POLICY_FILE =
        "/home/jan/projects/jini/doc/policy.all";
    // Don't forget the trailing '/'!
```

```
static final protected String CODEBASE = "http://localhost/classes/";

// protected FileClassifierImpl impl;
protected RemoteFileClassifier stub;

public static void main(String argv[]) {
    new FileClassifierServer(argv);
    // stick around while lookup services are found
    try {
        Thread.sleep(10000L);
    } catch(InterruptedException e) {
        // do nothing
    }
    // the server doesn't need to exist anymore
    System.exit(0);
}

public FileClassifierServer(String[] argv) {
    // install suitable security manager
    System.setSecurityManager(new RMISecurityManager());

    // Install an activation group
    Properties props = new Properties();
    props.put("java.security.policy",
            SECURITY_POLICY_FILE);
    ActivationGroupDesc.CommandEnvironment ace = null;
    ActivationGroupDesc group = new ActivationGroupDesc(props, ace);
    ActivationGroupID groupID = null;
    try {
        groupID = ActivationGroup.getSystem().registerGroup(group);
    } catch(RemoteException e) {
        e.printStackTrace();
        System.exit(1);
    } catch(ActivationException e) {
        e.printStackTrace();
        System.exit(1);
    }

    try {
        ActivationGroup.createGroup(groupID, group, 0);
    } catch(ActivationException e) {
        e.printStackTrace();
        System.exit(1);
    }
```

```
        String codebase = CODEBASE;
        MarshalledObject data = null;
        ActivationDesc desc = null;
        try {
            desc = new ActivationDesc("activation.FileClassifierImpl",
                                       codebase, data);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }

        try {
            stub = (RemoteFileClassifier) Activatable.register(desc);
        } catch(UnknownGroupException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        }

        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();
        RemoteFileClassifier service;

        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];
            // export the proxy service
            ServiceItem item = new ServiceItem(null,
                                               stub,
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.print("Register exception: ");
                e.printStackTrace();
                // System.exit(2);
                continue;
            }
            try {
                System.out.println("service registered at " +
                                   registrar.getLocator().getHost());
            } catch(Exception e) {
            }
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }
} // FileClassifierServer
```

Running the Service

The service backend and the server must be compiled as usual, and in addition, an RMI stub object must be created for the service backend using the `rmic` compiler (in JDK 1.2, at least). The class files for the stub must be copied to somewhere where an HTTP server can deliver them to clients. This is the same as for any other RMI stubs.

There is an extra step that must be performed for Activatable objects: the activation server `rmid` must be able to reconstruct a copy of the service backend (the client must be able to reconstruct a copy of the service's stub). This means that `rmid` must have access to the class files of the service backend, either from an HTTP server or from the file system. In the previous server, the `codebase` property in the `ActivationDesc` is set to an HTTP URL, so the class files for the service backend must be accessible to an HTTP server. Note that `rmid` does not get the class files for a service backend from the `CLASSPATH`, but from the codebase of the service. The HTTP server need not be on the same machine as the service backend.

Before starting the service provider, an rmid process must be set running on the same machine as the service provider. An HTTP server must be running on a machine specified by the codebase property on the service. The service provider can then be started. This will register the service with rmid and will copy a stub object to any lookup services that are found. The server can then terminate. (As mentioned earlier, this will cause the service's lease to expire, but techniques to handle this are described later).

In summary, there are typically three processes involved in getting an activatable service running:

- The service provider, which specifies the location of class files in its codebase.
- rmid, which must be running on the same machine as the service provider and must be started before the service provider. It gets class files using the codebase of the service.
- An HTTP server, which can be on a different machine and is pointed to by the codebase.

While the service remains registered with lookup services, clients can download its RMI stub. The service will be created on demand by rmid. You only need to run the server once, since rmid keeps information about the service in its own log files.

Security

The JVM for the service will be created by rmid and will be running in the same environment as rmid. Such things as the current directory for the service will be the same as for rmid, not from where the server ran. Similarly, the user ID for the service will be the user ID of rmid. This is a potential security problem in multi-user systems. For example, any user on a Unix system could write a service that attempts to read the shadow password file on the system, as an activatable service. Once registered with rmid, this same user could write a client that calls the appropriate methods on the service. If rmid is running in privileged mode, owned by the superuser of the system, then the service will run in that same mode and will happily read any file in the entire file system! For safety, rmid should probably be run using the user ID nobody, much like the recommendations for HTTP servers.

Some of the security issues with rmid have been addressed in JDK 1.3. These were discussed in Chapter 12, and they allow a security policy to be associated with each activatable service.

Non-Lazy Services

The types of services discussed in this chapter so far are "lazy" services, activated on demand when their methods are called. This reduces memory use at the expense of starting up a new JVM when required. Some services need to be continuously alive but can still benefit from the logging mechanism of `rmid`. If `rmid` crashes and is restarted, or the machine is rebooted and `rmid` restarts, then the server is able to use its log files to restart any "active" services registered with it, as well as to restore "lazy" services on demand. By making services non-lazy and ensuring that `rmid` is started on reboot, you can avoid messing around with boot configuration files.

Maintaining State

An activatable object is created afresh each time a method is called on it, using its two-argument constructor. The default action, calling `super(id, 0)` will result in the object being created in the same state on each activation. However, method calls on objects (apart from `get...()` methods) usually result in a change of state of the object. Activatable objects will need some way of reflecting this change on each activation, and saving and restoring state using a disk file typically does this.

When an object is activated, one of the parameters passed to it is a `MarshalledObject` instance. This is the same object that was passed to the activation system in the `ActivationDesc` parameter to `Activation.register()`. This object does not change between different activations, so it cannot hold changing state, but only data, which is fixed for all activations. A simple use for it is to hold the name of a file that can be used for state. Then, on each activation the object can restore state by reading stored information. On each subsequent method call that changes state, the information in the file can be overwritten.

The mutable file classifier example was discussed in Chapter 14—it could be sent `addType()` and `removeType()` messages. It begins with a given set of MIME type/ file extension mappings. State here is very simple; it is just a matter of storing all the file extensions and their corresponding MIME types in a `Map`. If we turn this into an activatable object, we store the state by just storing the map. This map can be saved to disk using `ObjectOutputStream.writeObject()`, and it can be retrieved by `ObjectInputStream.readObject()`. More complex cases might need more complex storage methods.

The very first time a mutable file classifier starts on a particular host, it should build its initial state file. There are a variety of methods that could be used. For example, if the state file does not exist, then the first activation could detect this and construct the initial state at that time. Alternatively, a method such as `init()` could be defined, to be called once after the object has been registered with the activation system.

The "normal" way of instantiating an object—through a constructor—doesn't work very well with activatable objects. If a constructor for a class doesn't start by calling another constructor with `this(...)` or `super(...)`, then the no-argument superclass constructor `super()` is called. However, the class `Activatable` doesn't have a no-args constructor, so you can't subclass from `Activatable` and have a constructor such as `FileClassifierMutable(String stateFile)` that doesn't use the activation system.

You can avoid this problem by not inheriting from `Activatable` and registering explicitly with the activation system, like this:

```
public FileClassifierMutable(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException {
        Activatable.exportObject(this, id, 0);
        // continue with instantiation
```

Nevertheless, this is a bit clumsy: you create an object solely to build up initial state, and then discard it because the activation system will recreate it on demand.

The technique we'll use here is to create initial state if the attempt to restore state from the state file fails for any reason when the object is activated. This is done in the `restoreMap()` method called from the constructor `FileClassifierMutable(ActivationID id, MarshalledObject data)`. The name of the file is extracted from the marshalled object passed in as parameter.

```
package activation;

import java.io.*;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;

import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.EventRegistration;
import java.rmi.RemoteException;
import net.jini.core.event.UnknownEventException ;

import javax.swing.event.EventListenerList;

import common.MIMEType;
import common.MutableFileClassifier;
import mutable.RemoteFileClassifier;
import java.util.Map;
import java.util.HashMap;
```

```
/**
 * FileClassifierMutable.java
 */

public class FileClassifierMutable extends  Activatable
                                implements RemoteFileClassifier {

    /**
     * Map of String extensions to MIME types
     */
    protected Map map = new HashMap();

    /**
     * Permanent storage for the map while inactive
     */
    protected String mapFile;

    /**
     * Listeners for change events
     */
    protected EventListenerList listenerList = new EventListenerList();

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        System.out.println("Called with " + fileName);

        MIMEType type;
        String fileExtension;
        int dotIndex = fileName.lastIndexOf('.');

        if (dotIndex == -1 || dotIndex + 1 == fileName.length()) {
            // can't find suitable suffix
            return null;
        }

        fileExtension= fileName.substring(dotIndex + 1);
        type = (MIMEType) map.get(fileExtension);
        return type;

    }

    public void addType(String suffix, MIMEType type)
        throws java.rmi.RemoteException {
```

```
    map.put(suffix, type);
    fireNotify(MutableFileClassifier.ADD_TYPE);
    saveMap();
}

public void removeMIMEType(String suffix, MIMEType type)
    throws java.rmi.RemoteException {
    if (map.remove(suffix) != null) {
        fireNotify(MutableFileClassifier.REMOVE_TYPE);
        saveMap();
    }
}

public EventRegistration addRemoteListener(RemoteEventListener listener)
    throws java.rmi.RemoteException {
    listenerList.add(RemoteEventListener.class, listener);

    return new EventRegistration(0, this, null, 0);
}

// Notify all listeners that have registered interest for
// notification on this event type.  The event instance
// is lazily created using the parameters passed into
// the fire method.

protected void fireNotify(long eventID) {
    RemoteEvent remoteEvent = null;

    // Guaranteed to return a non-null array
    Object[] listeners = listenerList.getListenerList();

    // Process the listeners last to first, notifying
    // those that are interested in this event
    for (int i = listeners.length - 2; i >= 0; i -= 2) {
        if (listeners[i] == RemoteEventListener.class) {
           RemoteEventListener listener = (RemoteEventListener) listeners[i+1];
            if (remoteEvent == null) {
                remoteEvent = new RemoteEvent(this, eventID,
                                              0L, null);
            }
            try {
                listener.notify(remoteEvent);
            } catch(UnknownEventException e) {
                e.printStackTrace();
            } catch(RemoteException e) {
                e.printStackTrace();
            }
        }
    }
}

/**
 * Restore map from file.
 * Install default map if any errors occur
 */
public void restoreMap() {
    try {
        FileInputStream istream = new FileInputStream(mapFile);
        ObjectInputStream p = new ObjectInputStream(istream);
        map = (Map) p.readObject();

        istream.close();
    } catch(Exception e) {
        e.printStackTrace();
        // restoration of state failed, so
        // load a predefined set of MIME type mappings
        map.put("gif", new MIMEType("image", "gif"));
        map.put("jpeg", new MIMEType("image", "jpeg"));
        map.put("mpg", new MIMEType("video", "mpeg"));
        map.put("txt", new MIMEType("text", "plain"));
        map.put("html", new MIMEType("text", "html"));

        this.mapFile = mapFile;
        saveMap();
    }
}

/**
 * Save map to file.
 */
public void saveMap() {
    try {
        FileOutputStream ostream = new FileOutputStream(mapFile);
        ObjectOutputStream p = new ObjectOutputStream(ostream);
        p.writeObject(map);
        p.flush();
        ostream.close();
    } catch(Exception e) {
            e.printStackTrace();
        }
    }

    public FileClassifierMutable(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException {
        super(id, 0);
        try {
            mapFile = (String) data.get();
        } catch(Exception e) {
            e.printStackTrace();
        }
        restoreMap();
    }
} // FileClassifierMutable
```

The difference between the server for this service and the last one is that we now have to prepare a marshalled object for the state file and register it with the activation system. Here the filename is hard-coded, but it could be given as a command line argument (as services such as `reggie` do).

```
package activation;

import mutable.RemoteFileClassifier;

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.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
import java.rmi.RMISecurityManager;
import java.rmi.MarshalledObject;
import java.rmi.activation.ActivationDesc;
import java.rmi.activation.ActivationGroupDesc;
import java.rmi.activation.ActivationGroupDesc.CommandEnvironment;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationGroup;
import java.rmi.activation.ActivationGroupID;

import java.util.Properties;

import java.rmi.activation.UnknownGroupException;
import java.rmi.activation.ActivationException;
import java.rmi.RemoteException;

/**
 * FileClassifierServerMutable.java
 */

public class FileClassifierServerMutable implements DiscoveryListener {

    static final protected String SECURITY_POLICY_FILE =
        "/home/jan/projects/jini/doc/policy.all";
    // Don't forget the trailing '/'!
    static final protected String CODEBASE = "http://localhost/classes/";
    static final protected String LOG_FILE = "/tmp/file_classifier";

    // protected FileClassifierImpl impl;
    protected RemoteFileClassifier stub;

    public static void main(String argv[]) {
        new FileClassifierServerMutable(argv);
        // stick around while lookup services are found
        try {
            Thread.sleep(10000L);
        } catch(InterruptedException e) {
            // do nothing
        }
        // the server doesn't need to exist anymore
        System.exit(0);
    }

    public FileClassifierServerMutable(String[] argv) {
        // install suitable security manager
        System.setSecurityManager(new RMISecurityManager());

        // Install an activation group
        Properties props = new Properties();
        props.put("java.security.policy",
                SECURITY_POLICY_FILE);
        ActivationGroupDesc.CommandEnvironment ace = null;
        ActivationGroupDesc group = new ActivationGroupDesc(props, ace);
        ActivationGroupID groupID = null;
        try {
            groupID = ActivationGroup.getSystem().registerGroup(group);
        } catch(RemoteException e) {
```

```
        e.printStackTrace();
        System.exit(1);
    } catch(ActivationException e) {
        e.printStackTrace();
        System.exit(1);
    }

    try {
        ActivationGroup.createGroup(groupID, group, 0);
    } catch(ActivationException e) {
        e.printStackTrace();
        System.exit(1);
    }

    String codebase = CODEBASE;
    MarshalledObject data = null;
    try {
        data = new MarshalledObject(LOG_FILE);
    } catch(java.io.IOException e) {
        e.printStackTrace();
    }

    ActivationDesc desc = null;
    try {
        desc = new ActivationDesc("activation.FileClassifierMutable",
                                  codebase, data);
    } catch(ActivationException e) {
        e.printStackTrace();
        System.exit(1);
    }

    try {
        stub = (RemoteFileClassifier) Activatable.register(desc);
    } catch(UnknownGroupException e) {
        e.printStackTrace();
        System.exit(1);
    } catch(ActivationException e) {
        e.printStackTrace();
        System.exit(1);
    } catch(RemoteException e) {
        e.printStackTrace();
        System.exit(1);
    }
```

```
        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();
        RemoteFileClassifier service;

        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];

            // export the proxy service
            ServiceItem item = new ServiceItem(null,
                                               stub,
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.print("Register exception: ");
                e.printStackTrace();
                // System.exit(2);
                continue;
            }
            try {
                System.out.println("service registered at " +
                                   registrar.getLocator().getHost());
            } catch(Exception e) {
            }
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }
```

```
} // FileClassifierServerMutable
```

This example used a simple way of storing state. Sun uses a far more complex system in many of its services, such as `reggie`—a "reliable log" in the package `com.sun.jini.reliableLog`. However, this package is not a part of standard Jini, so it may change or even be removed in later versions of Jini. There is nothing to stop you from using it, though, if you need a robust storage mechanism.

LeaseRenewalService

Activatable objects are an example of services that are not continuously alive. Mobile services, such as those that will exist on mobile phones, are another. These services will be brought to life on demand (as activatable objects), or will join the network on occasion. These services raise a number of problems, and one was skirted around in the last section: How do you renew leases when the object is not alive?

Activatable objects are brought back to life when methods are invoked on them, and the expiration of a lease does not cause any methods to be invoked. There is no "lease-expiring event" generated that could cause a listener method to be invoked, either. It is true that a `ServiceRegistrar` such as `reggie` will generate an event when a lease changes status, but this is a "service removed" event rather than a "service about to be removed" event—at that point it is too late.

If a server is alive, then it can use a `LeaseRenewalManager` to keep leases alive, but there are two problems with this: first the renewal manager works by sleeping and waking up just in time to renew the leases, and second, if the server exits, then no `LeaseRenewalManager` will continue to run.

Jini 1.1 supplies a lease renewal service that partly solves these problems. Since it runs as a service, it has an independent existence that does not depend on the server for any other service. It can act like a `LeaseRenewalManager` in keeping track of leases registered with it, renewing them as needed. In general, it can keep leases alive without waking the service itself, which can slumber until it is activated by clients calling methods.

There is a small hiccup in this system, though: how long should the `LeaseRenewalService` keep renewing leases for a service? The `LeaseRenewalManager` utility has a simple solution: keep renewing while the server for that service is alive. If the server dies, taking down a service, then it will also take down the `LeaseRenewalManager` running in the same JVM, so leases will expire, as expected, after an interval.

But this mechanism won't work for `LeaseRenewalService` because the managed service can disappear without the `LeaseRenewalService` knowing about it. So the lease renewal must be done on a leased basis itself! The `LeaseRenewalService` will renew leases for a service only for a particular amount of time, as specified by a lease. The service will still have to renew its lease, but with a `LeaseRenewalService` instead of a bunch of lookup services. The lease granted by this service will need to be of much longer duration than those granted by the lookup services for this to be of value.

Activatable services can only be woken by calling one of their methods. The `LeaseRenewalService` accomplishes this by generating renewal events in advance and calling a `notify()` method on a listener. If the listener is the activatable object, this will wake it up so that it can perform the renewal. If the `rmid` process managing the service has died or is unavailable, then the event will not be delivered and the `LeaseRenewalService` can remove this service from its renewal list.

This is not quite satisfactory for other types of "dormant" services, such as might exist on mobile phones, since there is no equivalent of `rmid` to handle activation. Instead, the mobile phone service might say that it will connect once a day and renew the lease, as long as the `LeaseRenewalService` agrees to keep the lease for at least a day. This is still "negotiable," in that the service asks for a duration and the `LeaseRenewalService` replies with a value that might not be so long. Still, it should be better than dealing with the lookup services.

The Norm Service

Jini 1.1 supplies an implementation of `LeaseRenewalService` called `norm`. This is a non-lazy Activatable service that requires `rmid` to be running. This is run with the following command

```
java -jar [setup_jvm_options] executable_jar_file
          codebase_arg  norm_policy_file_arg
          log_directory_arg
          [groups] [server_jvm] [server_jvm_args]
```

as in the following

```
java -jar \
        -Djava.security.policy=/files/jini1_1/example/txn/policy.all \
        /files/jini1_1/lib/norm.jar \
        http://`hostname`:8080/norm-dl.jar \
        /files/jini1_1/example/books/policy.all /tmp/norm_log
```

The first security file defines the policy that will be used for the server startup. The `norm.jar` file contains the class files for the `norm` service. This exports RMI stubs, and the class definitions for these are in `norm-dl.jar`. The second security file defines the policy file that will be used in the execution of the `LeaseRenewalService` methods. Finally, the log file is used to keep state, so that it can keep track of the leases it is managing.

The norm service will maintain a set of leases for a period of up to two hours. The reggie lookup service only grants leases for five minutes, so using this service increases the amount of time between renewing leases by a factor of over 20.

Using the LeaseRenewalService

The norm service exports an object of type LeaseRenewalService, which is defined by the following interface:

```
package net.jini.lease;

public interface LeaseRenewalService {
    LeaseRenewalSet createLeaseRenewalSet(long leaseDuration)
        throws java.rmi.RemoteException;
}
```

A server that wants to use the lease renewal service will first find this service and then call the createLeaseRenewal() method. The server requests a leaseDuration value, measured in milliseconds, for the lease service to manage a set of leases. The lease service creates a lease for this request, but the lease time may be less than the requested time (for norm, it is a maximum of two hours). In order for the lease service to continue to manage the set beyond the lease's expiry, the lease must be renewed before expiration. Since the service may be inactive at the time of expiry, the LeaseRenewalSet can be asked to register a listener object that will receive an event containing the lease. This will activate a dormant listener so that the listener can renew the lease in time. If the lease for the LeaseRenewalSet is allowed to lapse, then eventually all the leases for the services it was managing will also expire, making the services unavailable.

The LeaseRenewalSet returned from createLeaseRenewalSet has an interface including the following:

```
package net.jini.lease;

public interface LeaseRenewalSet {
    public void renewFor(Lease leaseToRenew,
                         long membershipDuration)
              throws RemoteException;
    public EventRegistration setExpirationWarningListener(
                         RemoteEventListener listener,
                         long minWarning,
                         MarshalledObject handback)
              throws RemoteException;
    ....
}
```

The `renewFor()` method adds a new lease to the set being looked after. The `LeaseRenewalSet` will keep renewing the lease until either the requested `membershipDuration` expires or the lease for the whole `LeaseRenewalSet` expires (or until an exception happens, like a lease being refused).

Setting an expiration warning listener means that the `notify()` method of the listener will be called at least `minWarning` milliseconds before the lease for the set expires. The event argument will actually be an `ExpirationWarningEvent`:

```
package net.jini.lease;

public class ExpirationWarningEvent extends RemoteEvent {
    Lease getLease();
}
```

This allows the listener to get the lease for the `LeaseRenewalSet` and (probably) renew it. Here is a simple activatable class that can renew the lease:

```
package activation;

import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.lease.Lease;
import net.jini.lease.ExpirationWarningEvent;

public class RenewLease extends Activatable
    implements RemoteEventListener {

    public RenewLease(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException {
        super(id, 0);
    }

    public void notify(RemoteEvent evt) {
        System.out.println("expiring... " + evt.toString());
        ExpirationWarningEvent eevt = (ExpirationWarningEvent) evt;
        Lease lease = eevt.getLease();
        try {
            // This is short, for testing. Try 2+ hours
            lease.renew(20000L);
        } catch(Exception e) {
            e.printStackTrace();
        }
        System.out.println("Lease renewed for " +
                           (lease.getExpiration() -
                            System.currentTimeMillis()));
    }
}
```

The server will need to register the service and export it as an activatable object. This is done in exactly the same way as in the `FileClassifierServer` example of the first section of this chapter. In addition, it will need to do a few other things:

- It will need to register the lease listener with the activation system as an activatable object.
- It will need to find a `LeaseRenewalService` from a lookup service.
- It will need to register all leases from lookup services with the `LeaseRenewalService`. Since it may find lookup services before it finds the renewal service, it will need to keep a list of lookup services found before finding the service, in order to register them with it.

Adding these additional requirements to the `FileClassifierServer` of the first section results in this server:

```
package activation;

import rmi.RemoteFileClassifier;

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.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.lease.Lease;
import net.jini.lease.LeaseRenewalService;
import net.jini.lease.LeaseRenewalSet;
import java.rmi.RMISecurityManager;
import java.rmi.MarshalledObject;
import java.rmi.activation.ActivationDesc;
import java.rmi.activation.ActivationGroupDesc;
import java.rmi.activation.ActivationGroupDesc.CommandEnvironment;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationGroup;
import java.rmi.activation.ActivationGroupID;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;

import java.util.Properties;
import java.util.Vector;

import java.rmi.activation.UnknownGroupException;
import java.rmi.activation.ActivationException;
import java.rmi.RemoteException;

/**
 * FileClassifierServerLease.java
 */

public class FileClassifierServerLease
    implements DiscoveryListener {

    static final protected String SECURITY_POLICY_FILE =
        "/home/jan/projects/jini/doc/policy.all";
    // Don't forget the trailing '/'!
    static final protected String CODEBASE = "http://localhost/classes/";

    protected RemoteFileClassifier stub;

    protected RemoteEventListener leaseStub;

    // Lease renewal management
    protected LeaseRenewalSet leaseRenewalSet = null;

    // List of leases not yet managed by a LeaseRenewalService
    protected Vector leases = new Vector();

    public static void main(String argv[]) {
        new FileClassifierServerLease(argv);
        // stick around while lookup services are found
        try {
            Thread.sleep(10000L);
    } catch(InterruptedException e) {
        // do nothing
    }
    // the server doesn't need to exist anymore
    System.exit(0);
}

public FileClassifierServerLease(String[] argv) {
    // install suitable security manager
    System.setSecurityManager(new RMISecurityManager());

    // Install an activation group
    Properties props = new Properties();
    props.put("java.security.policy",
            SECURITY_POLICY_FILE);
    ActivationGroupDesc.CommandEnvironment ace = null;
    ActivationGroupDesc group = new ActivationGroupDesc(props, ace);
    ActivationGroupID groupID = null;
    try {
        groupID = ActivationGroup.getSystem().registerGroup(group);
    } catch(RemoteException e) {
        e.printStackTrace();
        System.exit(1);
    } catch(ActivationException e) {
        e.printStackTrace();
        System.exit(1);
    }

    try {
        ActivationGroup.createGroup(groupID, group, 0);
    } catch(ActivationException e) {
        e.printStackTrace();
        System.exit(1);
    }

    String codebase = CODEBASE;
    MarshalledObject data = null;
    ActivationDesc desc = null;
    ActivationDesc descLease = null;
    try {
        desc = new ActivationDesc("activation.FileClassifierImpl",
                                  codebase, data);
        descLease = new ActivationDesc("activation.RenewLease",
                                  codebase, data);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }

        try {
            stub = (RemoteFileClassifier) Activatable.register(desc);
            leaseStub = (RemoteEventListener) Activatable.register(descLease);
        } catch(UnknownGroupException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        }

        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();
        RemoteFileClassifier service;

        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];

            // export the proxy service
            ServiceItem item = new ServiceItem(null,
                                               stub,
                                               null);
            ServiceRegistration reg = null;
            try {
```

```
            reg = registrar.register(item, Lease.FOREVER);
        } catch(java.rmi.RemoteException e) {
            System.err.print("Register exception: ");
            e.printStackTrace();
            // System.exit(2);
            continue;
        }
        try {
            System.out.println("service registered at " +
                               registrar.getLocator().getHost());
        } catch(Exception e) {
        }

        Lease lease = reg.getLease();
        // if we have a lease renewal manager, use it
        if (leaseRenewalSet != null) {
            try {
                leaseRenewalSet.renewFor(lease, Lease.FOREVER);
            } catch(RemoteException e) {
                e.printStackTrace();
            }
        } else {
            // add to the list of unmanaged leases
            leases.add(lease);
            // see if this lookup service has a lease renewal manager
            findLeaseService(registrar);
        }
    }
}

public void findLeaseService(ServiceRegistrar registrar) {
    System.out.println("Trying to find a lease service");
    Class[] classes = {LeaseRenewalService.class};
    ServiceTemplate template = new ServiceTemplate(null, classes,
                                                   null);
    LeaseRenewalService leaseService = null;
    try {
        leaseService = (LeaseRenewalService) registrar.lookup(template);
    } catch(RemoteException e) {
        e.printStackTrace();
        return;
    }
    if (leaseService == null) {
        System.out.println("No lease service found");
```

```
                return;
            }
            try {
                // This time is unrealistically small - try 10000000L
                leaseRenewalSet = leaseService.createLeaseRenewalSet(20000);
                System.out.println("Found a lease service");
                // register a timeout listener
                leaseRenewalSet.setExpirationWarningListener(leaseStub, 5000,
                                                             null);
                // manage all the leases found so far
                for (int n = 0; n < leases.size(); n++) {
                    Lease ll = (Lease) leases.elementAt(n);
                    leaseRenewalSet.renewFor(ll, Lease.FOREVER);
                }
                leases = null;
            } catch(RemoteException e) {
                e.printStackTrace();
            }
            Lease renewalLease = leaseRenewalSet.getRenewalSetLease();
            System.out.println("Lease expires in " +
                               (renewalLease.getExpiration() -
                                System.currentTimeMillis()));
        }

        public void discarded(DiscoveryEvent evt) {

        }
    } // FileClassifierServerLease
```

LookupDiscoveryService

It is easy enough for a server to discover all of the lookup services within reach at the time it is started, by using LookupDiscovery. While the server continues to stay alive, any new lookup services that start will also be found by LookupDiscovery. But if the server terminates, which it will for activable services, then any new lookup services will probably never be found. This will result in the service not being registered with them, which could mean that clients may not find it. This is analogous to leases not being renewed if the server terminates.

Jini 1.1 supplies a service, the LookupDiscoveryService, that can be used to continuously monitor the state of lookup services. It will monitor them on behalf of a service that will most likely want to register with each new lookup service as it starts. If the service is an activatable one, the server that would have done registered the service will have terminated, as its role would have just been to register the service with rmid.

When there is a change to lookup services, the LookupDiscoveryService needs to notify an object about this by sending it a remote event (actually of type RemoteDiscoveryEvent). But again, we do not want to have a process sitting around waiting for such notification, so the listener object will probably also be an activatable object.

The LookupDiscoveryService interface has the following specification:

```
package net.jini.discovery;
public interface LookupDiscoveryService {
    LookupDiscoveryRegistration register(String[] groups,
                                         LookupLocator[] locators,
                                         RemoteEventListener listener,
                                         MarshalledObject handback,
                                         long leaseDuration);
}
```

Calling the register() method will begin a multicast search for the groups and unicast lookup for the locators. The registration is leased and will need to be renewed before expiring (a lease renewal service can be used for this). Note that the listener cannot be null–this is simple sanity checking, for if the listener was null, then the service could never do anything useful.

A lookup service in one of the groups can start or terminate, or it can change its group membership in such a way that it now does (or doesn't) meet the group criteria. A lookup service in the locators list can also start or stop. These will generate RemoteDiscoveryEvent events and call the notify() method of the listener. The event interface includes the following:

```
package net.jini.discovery;

public interface RemoteDiscoveryEvent {
    ServiceRegistrar[] getRegistrars();
    boolean isDiscarded();
    ...
}
```

The list of registrars is the set that triggered the event. The isDiscarded() method is used to check whether the lookup service is a "discovered" lookup service or a "discarded" lookup service. An initial event is not posted when the listener is registered: the set of lookup services that are initially found can be retrieved from the LookupDiscoveryRegistration object returned from the register() method by its getRegistrars().

The Fiddler Service

The Jini 1.1 release includes an implementation of the lookup discovery service called `fiddler`. It is a non-lazy activatable service and is started much like other services, such as `reggie`:

```
java -jar [setup_jvm_options] executable_jar_file
     codebase_arg  fiddler_policy_file_arg
     log_directory_arg [groups and locators]
     [server_jvm] [server_jvm_args]
```

For example,

```
java -jar \
        -Djava.security.policy=/files/jini1_1/example/txn/policy.all \
        /files/jini1_1/lib/fiddler.jar \
        http://`hostname`:8080/norm-dl.jar \
        /files/jini1_1/example/books/policy.all /tmp/fiddler_log
```

Using the LookupDiscoveryService

An activatable service can make use of a lease renewal service to look after the leases for discovered lookup services. It can find these lookup services by means of a lookup discovery service. The logic that manages these two services is a little tricky.

While lease management can be done by the lease renewal service, the lease renewal set will also be leased and will need to be renewed on occasion. The lease renewal service can call an activatable `RenewLease` object to do this, as shown in the preceding section of this chapter.

The lookup discovery service is also a leased service—it will only report changes to lookup services while its own lease is current. Therefore, the lease from this service will have to be managed by the lease renewal service, in addition to the leases for any lookup services discovered.

The primary purpose of the lookup discovery service is to call the `notify()` method of some object when information about lookup services changes. This object should also be an activatable object. We define a `DiscoveryChange` object with a `notify()` method to handle changes in lookup services. If a lookup service has disappeared, we don't worry about it. If a lookup service has been discovered, we want to register the service with it and then manage the resultant lease. This means that the `DiscoveryChange` object must know both the service to be registered and the lease renewal service. This is static data, so these two objects can be passed in an array of two objects as the `MarshalledObject` to the activation constructor.

The class itself can be implemented as shown here:

```
package activation;

import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.lease.Lease;
import net.jini.lease.ExpirationWarningEvent;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.lease.LeaseRenewalSet;
import net.jini.discovery.RemoteDiscoveryEvent;
import java.rmi.RemoteException;
import  net.jini.discovery.LookupUnmarshalException;

import rmi.RemoteFileClassifier;

public class DiscoveryChange extends Activatable
    implements RemoteEventListener {

    protected LeaseRenewalSet leaseRenewalSet;
    protected RemoteFileClassifier service;

    public DiscoveryChange(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException {
        super(id, 0);
        Object[] objs = null;
        try {
            objs = (Object []) data.get();
        } catch(ClassNotFoundException e) {
            e.printStackTrace();
        } catch(java.io.IOException e) {
            e.printStackTrace();
        }
        service = (RemoteFileClassifier) objs[0];
        leaseRenewalSet= (LeaseRenewalSet) objs[1];
    }

    public void notify(RemoteEvent evt) {
        System.out.println("lookups changing... " + evt.toString());
```

```
            RemoteDiscoveryEvent revt = (RemoteDiscoveryEvent) evt;

            if (! revt.isDiscarded()) {
                // The event is a discovery event
                ServiceItem item = new ServiceItem(null, service, null);
                ServiceRegistrar[] registrars = null;
                try {
                    registrars = revt.getRegistrars();
                } catch(LookupUnmarshalException e) {
                    e.printStackTrace();
                    return;
                }
                for (int n = 0; n < registrars.length; n++) {
                    ServiceRegistrar registrar = registrars[n];

                    ServiceRegistration reg = null;
                    try {
                        reg = registrar.register(item, Lease.FOREVER);
                        leaseRenewalSet.renewFor(reg.getLease(), Lease.FOREVER);
                    } catch(java.rmi.RemoteException e) {
                        System.err.println("Register exception: " + e.toString());
                    }
                }
            }
        }
}
```

The server must install an activation group and then find activation proxies for the service itself and also for the lease renewal object. After this, it can use a `ClientLookupManager` to find the lease service and register the lease renewal object with it. Now that it has a proxy for the service object, and also a lease renewal service, it can create the marshalled data for the lookup discovery service and register this with `rmid`. Then we can find the lookup discovery service and register our discovery change listener `DiscoveryChange` with it. At the same time, we have to register the service with all the lookup services the lookup discovery service finds on initialization.

This all leads to the following server:

```
package activation;

import rmi.RemoteFileClassifier;

import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.LookupDiscoveryService;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.discovery.LookupDiscoveryRegistration;
import net.jini.discovery.LookupUnmarshalException;

import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.lease.Lease;

import net.jini.lease.LeaseRenewalService;
import net.jini.lease.LeaseRenewalSet;
import net.jini.lease.LeaseRenewalManager;

import net.jini.lookup.ClientLookupManager;

import java.rmi.RMISecurityManager;
import java.rmi.activation.ActivationDesc;
import java.rmi.activation.ActivationGroupDesc;
import java.rmi.activation.ActivationGroupDesc.CommandEnvironment;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationGroup;
import java.rmi.activation.ActivationGroupID;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;
import java.rmi.activation.UnknownGroupException;
import java.rmi.activation.ActivationException;
import java.rmi.RemoteException;

import java.util.Properties;
import java.util.Vector;



/**
 * FileClassifierServerDiscovery.java
 */

public class FileClassifierServerDiscovery
    /* implements DiscoveryListener */ {
```

```
private static final long WAITFOR = 10000L;

static final protected String SECURITY_POLICY_FILE =
    "/home/jan/projects/jini/doc/policy.all";
// Don't forget the trailing '/'!
static final protected String CODEBASE = "http://localhost/classes/";

protected RemoteFileClassifier serviceStub;

protected RemoteEventListener leaseStub,
                              discoveryStub;

// Services
protected LookupDiscoveryService discoveryService = null;
protected LeaseRenewalService leaseService = null;

// Lease renewal management
protected LeaseRenewalSet leaseRenewalSet = null;

// List of leases not yet managed by a LeaseRenewalService
protected Vector leases = new Vector();

protected ClientLookupManager clientMgr = null;

public static void main(String argv[]) {
    new FileClassifierServerDiscovery();
    // stick around while lookup services are found
    try {
        Thread.sleep(20000L);
    } catch(InterruptedException e) {
        // do nothing
    }
    // the server doesn't need to exist anymore
    System.exit(0);
}

public FileClassifierServerDiscovery() {
    // install suitable security manager
    System.setSecurityManager(new RMISecurityManager());

    installActivationGroup();

    serviceStub = (RemoteFileClassifier)
            registerWithActivation("activation.FileClassifierImpl", null);
    leaseStub = (RemoteEventListener)
                registerWithActivation("activation.RenewLease", null);

    initClientLookupManager();

    findLeaseService();

    // the discovery change listener needs to know
    // the service and the lease service
    Object[] discoveryInfo = {serviceStub, leaseRenewalSet};
    MarshalledObject discoveryData = null;
    try {
        discoveryData = new MarshalledObject(discoveryInfo);
    } catch(java.io.IOException e) {
        e.printStackTrace();
    }
    discoveryStub = (RemoteEventListener)
                registerWithActivation("activation.DiscoveryChange",
                                       discoveryData);


    findDiscoveryService();

}

public void installActivationGroup() {

    Properties props = new Properties();
    props.put("java.security.policy",
              SECURITY_POLICY_FILE);
    ActivationGroupDesc.CommandEnvironment ace = null;
    ActivationGroupDesc group = new ActivationGroupDesc(props, ace);
    ActivationGroupID groupID = null;
    try {
        groupID = ActivationGroup.getSystem().registerGroup(group);
    } catch(RemoteException e) {
        e.printStackTrace();
        System.exit(1);
    } catch(ActivationException e) {
        e.printStackTrace();
        System.exit(1);
    }

    try {
            ActivationGroup.createGroup(groupID, group, 0);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public Object registerWithActivation(String className, MarshalledObject data) {
        String codebase = CODEBASE;
        ActivationDesc desc = null;
        Object stub = null;

        try {
            desc = new ActivationDesc(className,
                                      codebase, data);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }

        try {
            stub = Activatable.register(desc);
        } catch(UnknownGroupException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        }
        return stub;
    }

    public void initClientLookupManager() {
        LookupDiscoveryManager lookupDiscoveryMgr = null;
        try {
            lookupDiscoveryMgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            clientMgr = new ClientLookupManager(lookupDiscoveryMgr,
                                                new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void findLeaseService() {
        leaseService = (LeaseRenewalService)
findService(LeaseRenewalService.class);
        if (leaseService == null) {
            System.out.println("Lease service null");
        }
        try {
            leaseRenewalSet = leaseService.createLeaseRenewalSet(20000);
            leaseRenewalSet.setExpirationWarningListener(leaseStub, 5000,
                                                          null);
        } catch(RemoteException e) {
            e.printStackTrace();
        }
    }

    public void findDiscoveryService() {
        discoveryService = (LookupDiscoveryService)
                           findService(LookupDiscoveryService.class);
        if (discoveryService == null) {
            System.out.println("Discovery service null");
        }
        LookupDiscoveryRegistration registration = null;
        try {
            registration =
                discoveryService.register(LookupDiscovery.ALL_GROUPS,
                                          null,
                                          discoveryStub,
                                          null,
                                          Lease.FOREVER);
        } catch(RemoteException e) {
            e.printStackTrace();
        }
        // manage the lease for the lookup discovery service
        try {
            leaseRenewalSet.renewFor(registration.getLease(), Lease.FOREVER);
        } catch(RemoteException e) {
            e.printStackTrace();
        }
        // register with the lookup services already found
        ServiceItem item = new ServiceItem(null, serviceStub, null);
        ServiceRegistrar[] registrars = null;
        try {
            registrars = registration.getRegistrars();
        } catch(RemoteException e) {
            e.printStackTrace();
            return;
        } catch(LookupUnmarshalException e) {
            e.printStackTrace();
            return;
        }

        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
                leaseRenewalSet.renewFor(reg.getLease(), Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.println("Register exception: " + e.toString());
            }
        }
    }

    public Object findService(Class cls) {
        Class [] classes = new Class[] {cls};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        ServiceItem item = null;
        try {
            item = clientMgr.lookup(template,
                                    null, /* no filter */
                                    WAITFOR /* timeout */);
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        if (item == null) {
            // couldn't find a service in time
            System.out.println("No service found for " + cls.toString());
            return null;
        }
        return item.service;
    }
} // FileClassifierServerDiscovery
```

Summary

Some objects may not always be available, either because of mobility issues or because they are activatable objects. This chapter has dealt with activatable objects, and also with some of the special services that are needed to properly support these transient objects.

Index

The Story Behind Apress

APRESS IS AN INNOVATIVE PUBLISHING COMPANY devoted to meeting the needs of existing and potential programming professionals. Simply put, the "A" in Apress stands for the "author's press™." Our unique author-centric approach to publishing grew from conversations between Dan Appleman and Gary Cornell, authors of best-selling, highly regarded computer books. They wanted to create a publishing company that emphasized quality above all—a company whose books would be considered the best in their market.

To accomplish this goal, they knew it was necessary to attract the very best authors—established authors whose work is already highly regarded, and new authors who have real-world practical experience that professional software developers want in the books they buy. Dan and Gary's vision of an author-centric press has already attracted many leading software professionals—just look at the list of Apress titles on the following pages.

Would You Like to Write for Apress?

APRESS IS RAPIDLY EXPANDING its publishing program. If you can write and refuse to compromise on the quality of your work, if you believe in doing more then rehashing existing documentation, and if you are looking for opportunities and rewards that go far beyond those offered by traditional publishing houses, we want to hear from you!

Consider these innovations that we offer every one of our authors:

- Top royalties with *no* hidden switch statements. For example, authors typically only receive half of their normal royalty rate on foreign sales. In contrast, Apress' royalty rate remains the same for both foreign and domestic sales.

- A mechanism for authors to obtain equity in Apress. Unlike the software industry, where stock options are essential to motivate and retain software professionals, the publishing industry has stuck to an outdated compensation model based on royalties alone. In the spirit of most software companies, Apress reserves a significant portion of its equity for authors.

- Serious treatment of the technical review process. Each Apress book has a technical reviewing team whose remuneration depends in part on the success of the book since they, too, receive a royalty.

Moreover, through a partnership with Springer-Verlag, one of the world's major publishing houses, Apress has significant venture capital behind it. Thus, Apress has the resources both to produce the highest quality books *and* to market them aggressively.

If you fit the model of the Apress author who can write a book that gives the "professional what he or she needs to know™," then please contact any one of our editorial directors, Gary Cornell (gary_cornell@apress.com), Dan Appleman (dan_appleman@apress.com), or Karen Watterson (karen_watterson@apress.com), for more information on how to become an Apress author.

Apress Titles

ISBN	LIST PRICE	AVAILABLE	AUTHOR	TITLE
1-893115-01-1	$39.95	Now	Appleman	Dan Appleman's Win32 API Puzzle Book and Tutorial for Visual Basic Programmers
1-893115-23-2	$29.95	Now	Appleman	How Computer Programming Works
1-893115-09-7	$24.95	Now	Baum	Dave Baum's Definitive Guide to LEGO MINDSTORMS
1-893115-84-4	$29.95	Now	Baum, Gasperi, Hempel, Villa	Extreme MINDSTORMS
1-893115-82-8	$59.95	Now	Ben-Gan/Moreau	Advanced Transact-SQL for SQL Server 2000
1-893115-14-3	$39.95	Winter 2000	Cornell/Jezak	Visual Basic Add-Ins and Wizards: Increasing Software Productivity
1-893115-85-2	$34.95	Winter 2000	Gilmore	A Programmer's Introduction to PHP 4.0
1-893115-17-8	$59.95	Now	Gross	A Programmer's Introduction to Windows DNA
1-893115-86-0	$34.95	Now	Gunnerson	A Programmer's Introduction to C#
1-893115-10-0	$34.95	Now	Holub	Taming Java Threads
1-893115-04-6	$34.95	Now	Hyman/Vaddadi	Mike and Phani's Essential C++ Techniques
1-893115-79-8	$49.95	Now	Kofler	Definitive Guide to Excel VBA
1-893115-75-5	$44.95	Now	Kurniawan	Internet Programming with VB
1-893115-19-4	$49.95	Now	Macdonald	Serious ADO: Universal Data Access with Visual Basic
1-893115-06-2	$39.95	Now	Marquis/Smith	A Visual Basic 6.0 Programmer's Toolkit
1-893115-22-4	$27.95	Now	McCarter	David McCarter's VB Tips and Techniques
1-893115-76-3	$49.95	Now	Morrison	C++ For VB Programmers
1-893115-80-1	$39.95	Now	Newmarch	A Programmer's Guide to Jini Technology
1-893115-81-X	$39.95	Now	Pike	SQL Server: Common Problems, Tested Solutions
1-893115-20-8	$34.95	Now	Rischpater	Wireless Web Development
1-893115-24-0	$49.95	Now	Sinclair	From Access to SQL Server
1-893115-16-X	$49.95	Now	Vaughn	ADO Examples and Best Practices

ISBN	LIST PRICE	AVAILABLE	AUTHOR	TITLE
1-893115-83-6	$44.95	Winter 2000	Wells	Code Centric: T-SQL Programming with Stored Procedures and Triggers
1-893115-05-4	$39.95	Winter 2000	Williamson	Writing Cross-Browser Dynamic HTML
1-893115-02-X	$49.95	Now	Zukowski	John Zukowski's Definitive Guide to Swing for Java 2
1-893115-78-X	$49.95	Now	Zukowski	Definitive Guide to Swing for Java 2, Second Edition

To order, call (800) 777-4643 or email `sales@apress.com`.

COMPATIBLE WITH JINI™ 1.1

A Programmer's Guide to Jini™ Technology

- Covers latest utilities and services of the newly released Jini™ 1.1
- Uses JDK 1.2 and 1.3
- Deals with user interfaces with Jini™ services
- Shows how to transform hardware devices into Jini™ services
- Expanded and revised version of the most popular online Jini™ tutorial

JINI™ IS SUN'S ATTEMPT TO MAKE transparent distributed computing a reality. So what does this mean? Imagine living in a world where you could move to a new office in another country or check into a hotel and plug your notebook or PDA directly into the network at that location. Your notebook would be immediately recognized and have access to the services at that location—without having you go through complex setup procedures in an unknown environment. Jini™ is Sun's Java™-based technology that has the potential to make this possible.

But this kind of technology, which will be so simple to use, is not easy for the programmer to implement! Newmarch's book is the best place to learn exactly what you will need to know in order to enter this brave new world. Newmarch's comprehensive treatment of Jini™ technology starts with the basics of how Jini™ clients, services, and devices join a Jini™ network and how clients use the Jini™ lookup service to see and use services in the network. After covering the basics, Newmarch moves on to explain how events and security are handled in the Jini™ framework. The last half of the book gives detailed coverage of many advanced topics that most books skim over, including how to use Jini™ with CORBA, transactions, user interfaces for services, and remote events. There's even coverage of how to use Jini™ for robotics applications, using the popular LEGO® MINDSTORMS™ Robotics Invention Kit as the vehicle!

All the sample code in this book can be downloaded from `http://www.apress.com`.

ABOUT THE AUTHOR

Jan Newmarch teaches in the Faculty of Information Science and Engineering at the University of Canberra, Australia. He is a well-known figure in the Java™ and Jini™ communities. He wrote the online tutorial that gave thousands of people their first introduction to Jini™, and it has been expanded and revised in this book. His areas of expertise include Java™, electronic commerce, user interfaces, and distributed computing.

US $39.95

Shelve in
Computer Programming,
Software Engineering,
Distributed Computing

Apress™
www.apress.com

6 89253 15801 2

ISBN 1-893115-80-1
53995>
9 781893 115804

APPENDIX C

University Libraries

Search this catalog and more

inPrimo

click to try

Database | Search | Headings | Titles | Patron | Login | Request | History | Help

Database Name: George Mason University

Search Request: Simple Search = 45821831

Search Results: Displaying 1 of 1 entries

Bibliographic | Holdings | MARC format

A programmer's guide to Jini technology / Jan Newmarch.

000 00767cam a2200229Ia 45e
001 925274
005 20010524111632.0
008 010131s2000 caua 001 0 eng d
035 __ **|a** (OCoLC)ocm45821831
040 __ **|a** VYF **|c** VYF **|d** TXA **|d** VGM
020 __ **|a** 1893115801
090 __ **|a** QA76.9.D5 **|b** N48 2000
049 __ **|a** VGMM
100 1_ **|a** Newmarch, Jan.
245 12 **|a** A programmer's guide to Jini technology / **|c** Jan Newmarch.
260 __ **|a** Berkeley, CA : **|b** Apress ; **|a** New York : **|b** distributed to the book trade in the United States by Springer-Verlag, **|c** c2000.
300 __ **|a** xxi, 448 p. : **|b** ill. ; **|c** 24 cm.
440 _0 **|a** Books for professionals by professionals
500 __ **|a** Includes index.
630 00 **|a** Jini.
650 _0 **|a** Electronic data processing **|x** Distributed processing.
994 __ **|a** E0 **|b** VGM

Record Options

Select Download Format	Full Record	Format for Print/Save
Enter your email address:		Email

Databases Search *Headings* Titles Patron Login Request

University Libraries