CHAPTER 16

Remote Events

Components of a system can change state and may need to inform other components that this change has happened. JavaBeans 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 examines 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 covers how leases are managed by event sources. Finally, we'll consider 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 called 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. (Addison-Wesley, 1995), but this is modified by other pressures, such as JavaBeans.

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 dispatch to listeners, and 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`, and so on. 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 JavaBeans.

JavaBeans is also the influence behind `PropertyChange` events, which get delivered whenever a bean changes a "bound" or "constrained" property value. These are delivered to the

PropertyChangeListener's propertyChange() method and to the VetoableChangeListener's vetoableChange() method. These events 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 would 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.
- JavaBeans can require method names and event types that vary. This requires the availability of classes across the network, which is more complex than a single method on a single event type (the original Observer pattern used a single method, for simplicity).

Remote Events

Unlike the large number of event classes in AWT and Swing (for example), Jini typically uses events of one type, the RemoteEvent, or a small number of subclasses of RemoteEvent. The class has these public 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 JavaBeans 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 can be 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 a value such as MOUSE_PRESSED or MOUSE_RELEASED. These fields are not seen by the AWT programmer because the AWT event dispatch system uses the id field to choose an appropriate method, 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 sorts 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 timestamps on mouse events. For example, a message with an id of ADD_TYPE and a sequence number of 10 could correspond to the state change "added MIME type text/xml for files with suffix .xml." Another event with an id of REMOVE_TYPE and a sequence number of 11 would be taken as a later event, even if it arrived earlier. 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 idea 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 can register a different object for events, so for example your stock broker 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 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 be for only 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. Note that in Jini 1, the source object was typically this, and the programmer would rely on Java substituting a proxy. In Jini 2.0, the proxy will have to be explicitly given, for example:

```
    new EventRegistration(0L, proxy, null, 0L)
```

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

```
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, 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 can be implemented by using a single-valued variable, as shown in Figure 16-1.

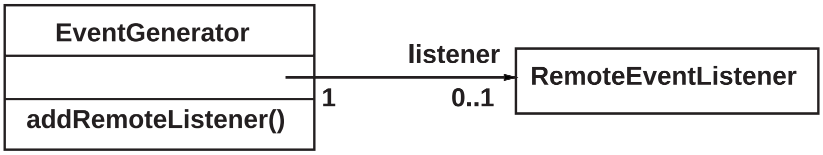

Figure 16-1. *A single listener*

The simplest case of event registration is as follows:

```
protected RemoteEventListener listener = null;
public EventRegistration addRemoteListener(RemoteEventListener listener)
       throws java.util.TooManyListenersException {
    if (this.listener == null {
        this.listener = listener;
    } else {
        throw new java.util.TooManyListenersException();
    }
    return new EventRegistration(0L, proxy, 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(). In Jini 2.0, the source object will be a proxy:

```
protected void fireNotify(long eventID,
                          long seqNum) {
    if (listener == null) {
        return;
    }
    RemoteEvent remoteEvent = new RemoteEvent(proxy, 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 in the registration and notify methods. Far more complex is the addition of a non-null lease. First, the event source has to decide on a *lease policy*—that is, for what periods of time is it going to 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 all list handling to the convenience class, as shown in Figure 16-2.

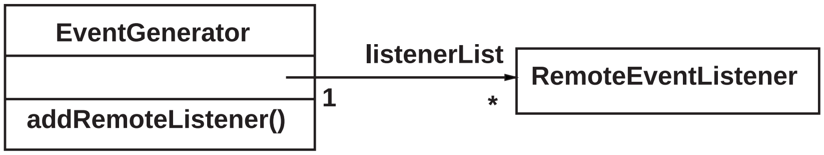

Figure 16-2. *Multiple listeners*

A version 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(OL, proxy, null, OL);
}
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(proxy, 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 notification of these listeners 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 you are dealing with proxy objects, not the real objects.

This is the same with remote events, except that in this case you 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 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 file name extensions built in. However, it may 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, it 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 file name 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 file name extension. When the classifier installs a new 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; 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, knowing 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 previously discussed. 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 16-3.

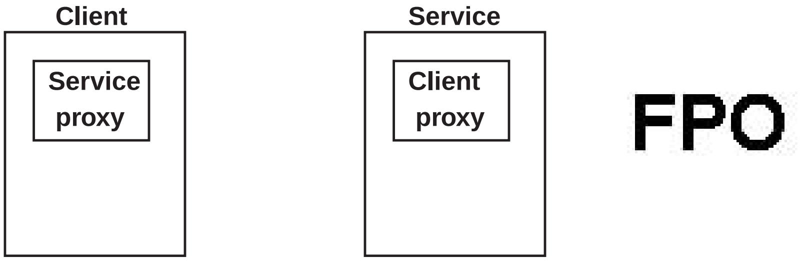

Figure 16-3. *Proxies for services and listeners*

File Classifier with Events

Let's make this discussion more concrete by looking at a new file classifier that can have its set of mappings dynamically updated. In the last chapter, we also considered such a situation, but from the point of view of leasing such additions. In this chapter, we ignore leasing issues and concentrate on generating events as the mappings change.

The first interface required is `MutableFileClassifier`, which is known to all objects. This interface adds methods to add and remove types, and also to register listeners for events. The event types are labeled with two constants. The listener model is simple, and it 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 removeType(String suffix)
        throws java.rmi.RemoteException;
    public EventRegistration addRemoteListener(RemoteEventListener listener)
        throws java.rmi.RemoteException;
} // MutableFileClasssifier
```

The RemoteFileClassifier interface 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
```

The implementation changes from a static list of if...then statements to a dynamic map keyed on file suffixes. It manages the event listener list for multiple listeners in the simple way discussed earlier. It generates events whenever a new suffix/type is added or successfully removed.

There are, however, several subtleties related to proxies. When a listener registers by addListener(), an EventRegistration is returned. This EventRegistration contains the service object (or rather, its proxy). Similarly, when notify() is called on the listener, it is passed a RemoteEvent, and this also contains the service (or rather, its proxy). With the “old” version of RMI, all of the work to do with proxies was looked after by the Java runtime, But with the Jeri model, handling of proxies must be made explicit. This means that the implementation object must know its proxy in order to prepare EventRegistration and RemoteEvent objects.

In all of the servers we have seen so far, the server creates the service and then goes on to create its proxy. This means that the service normally does not know its proxy. One way to overcome this is for the service to implement a method such as setProxy(); another way is for the service to create its own proxy and make it available to the server with a method such as getProxy(). Jini from version 2.0 has an interface, ProxyAccessor, that supports the second method.

```
interface ProxyAccessor {
    public Object getProxy();
}
```

The implementation needs to be passed enough information (e.g., a configuration) in its constructor to create a proxy. The methods addType() and removeType() manipulate the map of MIME types and also call firNotify() to generate events:

```
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 java.rmi.Remote;
import net.jini.core.event.UnknownEventException ;
import javax.swing.event.EventListenerList;
import net.jini.export.*;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.BasicILFactory;
import net.jini.export.ProxyAccessor;
import net.jini.config.*;
import common.MIMEType;
import common.MutableFileClassifier;
import java.util.Map;
import java.util.HashMap;
/**
 * FileClassifierImpl.java
 */
public class FileClassifierImpl implements RemoteFileClassifier, ProxyAccessor {
    /**
     * 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;
    protected Remote proxy;
    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 {
        System.out.println("type added");
        map.put(suffix, type);
        fireNotify(ADD_TYPE);
    }
    public void removeType(String suffix)
        throws java.rmi.RemoteException {
        System.out.println("Type removed");
        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,
                                     proxy,
                                     null, /* Lease is null for simplicity only.
                                              It should be e.g. a LandlordLease
                                           */
                                     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(proxy, eventID,
                                                  seqNum++, null);
                }
```

```
                try {
                    listener.notify(remoteEvent);
                } catch(UnknownEventException e) {
                    e.printStackTrace();
                } catch(RemoteException e) {
                    // Remove this listener from the list due to failure
                    listenerList.remove(RemoteEventListener.class, listener);
                    System.out.println("notification failed, listener removed");
                }
            }
        }
    }
    // Implementation for ProxyAccessor
    public Object getProxy() {
        return proxy;
    }
    public FileClassifierImpl()  throws java.rmi.RemoteException {
        // empty constructor for proxy generation
    }
    public FileClassifierImpl(String[] configArgs)  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"));
        try {
            // get the configuration (by default a FileConfiguration)
            Configuration config = ConfigurationProvider.getInstance(configArgs);

            // and use this to construct an exporter
            Exporter exporter = (Exporter) config.getEntry( "FileClassifierServer",
                                                            "exporter",
                                                            Exporter.class);
            // export an object of this class
            proxy = exporter.export(this);
        } catch(Exception e) {
            System.err.println(e.toString());
            e.printStackTrace();
            System.exit(1);
        }
    }
} // FileClassifierImpl
```

The server changes by passing in configuration information to the implementation's constructor and then getting the proxy from it in order to register the service.

```
package mutable;
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 java.rmi.RMISecurityManager;
import java.rmi.Remote;
import net.jini.config.*;
import net.jini.export.*;
/**
 * FileClassifierServer.java
 */
public class FileClassifierServer
    implements ServiceIDListener {
    // explicit proxy for Jini 2.0
    protected Remote proxy;
    protected FileClassifierImpl impl;
    private static String CONFIG_FILE = "jeri/file_classifier_server.config";

    public static void main(String argv[]) {
        FileClassifierServer server = new FileClassifierServer();
        // stay around forever
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(InterruptedException e) {
                // do nothing
            }
        }
    }
    public FileClassifierServer() {
        String[] configArgs = new String[] {CONFIG_FILE};
        try {
            impl = new FileClassifierImpl(configArgs);
        } catch(Exception e) {
            System.err.println("New impl: " + e.toString());
            System.exit(1);
        }
        proxy = (Remote) impl.getProxy();
        // install suitable security manager
        System.setSecurityManager(new RMISecurityManager());
        JoinManager joinMgr = null;
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null,  // unicast locators
                                           null); // DiscoveryListener
            joinMgr = new JoinManager(proxy, // service proxy
                                      null,  // attr sets
                                      this,  // ServiceIDListener
                                      mgr,   // DiscoveryManager
                                      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
```

The client must have an object that implements `RemoteEventListener`:

```
package client;
import common.MutableFileClassifier;
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.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.*;
import java.rmi.server.ExportException;
import net.jini.export.Exporter;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.tcp.TcpServerEndpoint;
/**
 * TestFileClassifierEvent.java
 */
```

```
public class TestFileClassifierEvent implements DiscoveryListener,
                                                RemoteEventListener {
    public static void main(String argv[]) {
        TestFileClassifierEvent client = new TestFileClassifierEvent();
        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(100000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }
    public TestFileClassifierEvent() {
        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[] {MutableFileClassifier.class};
        MutableFileClassifier classifier = null;
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Lookup service found");
            ServiceRegistrar registrar = registrars[n];
            try {
                classifier = (MutableFileClassifier) registrar.lookup(template);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                continue;
            }
            if (classifier == null) {
                System.out.println("Classifier null");
                continue;
            }
            // Add ourselves as an event listener
            Exporter exporter = new BasicJeriExporter(TcpServerEndpoint.getIn-
stance(0),
                                                  new BasicILFactory());
            // export an object of this class
            RemoteEventListener proxy = null;
            try {
                proxy =  (RemoteEventListener) exporter.export(this);
            } catch (ExportException e) {
                e.printStackTrace();
                continue;
            }
            try {
                classifier.addRemoteListener(proxy);
            } catch (RemoteException e) {
                e.printStackTrace();
                continue;
            }

            // Add some types to the service to generate events
            try {
                classifier.addType("ps", new MIMEType("text", "postscript"));
                classifier.removeType("ps");
            } catch(java.rmi.RemoteException e) {
                System.err.println(e.toString());
                continue;
            }
        }
    }
    public void discarded(DiscoveryEvent evt) {
        // empty
    }
    public void notify(RemoteEvent evt) {
        System.out.println("Event of type " + evt.getID());
    }
} // TestFileClassifier
```

Leasing Event Listeners

The implementation presented in the previous section creates a `null` object for a lease. This is not correct; it should be a non-null object. However, conceptually there is nothing here that we have not already covered in earlier chapters. See, for example, the section in Chapter 15 titled "Leased Changes to a Service" for how to add a landlord lease.

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, many times 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; and so on.

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 detect 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 service 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 could set up a transition mask as a bitwise OR of the three service transitions and call notify() on the ServiceRegistrar object. Note that this method expects to receive a proxy object (this was implicit in Jini 1 but needs to be made explicit in Jini 2.0). 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 net.jini.lease.LeaseRenewalManager;
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;
import net.jini.config.*;
import net.jini.export.*;
import java.rmi.Remote;
public class RegistrarObserver 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(Configuration config,
                         ServiceRegistrar registrar) throws RemoteException {
        RemoteEventListener proxy;
        this.registrar = registrar;
        Exporter exporter = null;
        try {
            exporter = (Exporter) config.getEntry( "JeriExportDemo",
                                    "exporter",
                                    Exporter.class);
        } catch(ConfigurationException e) {
            e.printStackTrace();
            return;
        }
        // export an object of this class
        proxy = (RemoteEventListener) exporter.export(this);
        ServiceTemplate templ = new ServiceTemplate(null, null, null);
        EventRegistration reg = null;
        try {
            reg = registrar.notify(templ,
                         transitions,
                         proxy,
                         null,
                         Lease.ANY);
            System.out.println("notifed 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
```

A suitable driver for this is as follows:

```
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 net.jini.config.*;
import java.util.Vector;
import observer.RegistrarObserver;
/**
 * ReggieMonitor.java
 */
```

```
public class ReggieMonitor implements DiscoveryListener {
    private Vector observers = new Vector();
    private Configuration config;
    public static void main(String argv[]) {
        new ReggieMonitor(argv);
        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(100000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }
    public ReggieMonitor(String[] argv) {
        String[] configArgs = new String[] {argv[0]};
        try {
            // get the configuration (by default a FileConfiguration)
            config = ConfigurationProvider.getInstance(configArgs);
        } catch(Exception e) {
            System.err.println(e.toString());
            e.printStackTrace();
            System.exit(1);
        }
        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 infor unavailable");
            }
            try {
                observers.add(new RegistrarObserver(config, registrar));
            } catch(RemoteException e) {
                System.out.println("adding observer failed");
            }
            ServiceTemplate templ = new ServiceTemplate(null, new Class[] {Ob-
ject.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].service.toString() +
                                       " with id " +
                                       matches.items[m].serviceID);
                }
            }
        }
    }
    public void discarded(DiscoveryEvent evt) {
        // remove observer
    }
} // ReggieMonitor
```

Summary

This chapter looked at how the remote event differs from the other event models in Java and at how to create and use remote events. 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.