7

Jini Distributed Events

Although these next three chapters may well be the toughest in the book, you should find them worthwhile – we will be covering some core cornerstone concepts within Jini: remote events, distributed leasing, and distributed transactions. A thorough understanding of these concepts will enable you to draw from them effectively when developing Jini applications. There is a lot of code spread between the chapters, but we've tried to keep it simple and tightly focused on each of the new concepts being explored. The following is a summary of the coverage in the next three chapters.

Concept	Coverage	Description
Remote Events	This Chapter	Extends the Java event model across the network. Provides a flexible asynchronous communications system. Jini distributed events mechanism is built on top of RMI, making it very simple to use. Designed to be 'chainable' or 'composable', allowing the possibility of using third-party services to process or handle events in the form of a pipeline.
Distributed Leasing	Chapter 8	Associates resource allocation on a remote server with a finite time lease that eventually expires. Distributed leasing allows resources to be automatically reclaimed in case of system or network failure. It forces the client to show 'proof of interest' by renewing the lease on regular interfaces. It contributes to the long-term health of a Jini system by letting the network self-heal, purging itself of resource leaks and zombie allocations, over a long period of operating time. We will examine distributed leasing from both the lease holder (resource user) and lease grantor (resource allocator) point of view.

Table continued on following page

Concept	Coverage	Description
Distributed Transactions	Chapter 9	Provides a 'standard' means of synchronizing the operation or state changes of multiple distributed services. Jini provides the specification for a TransactionManager that can co-ordinate a 'two phase commit' protocol amongst multiple distributed participants. The exact semantics of a transaction in Jini is completely application dependent, the ACID properties of classic transaction theory are not enforced (although they can be for certain applications). We will examine distributed transactions from both the view of an external transaction user, or an internal transaction participant.

In this chapter, our focus will be on distributed events – the main mechanism for asynchronous communications in the Jini world. We will see how Jini extends the model of Java events, one that AWT and Swing programmers will be familiar with, to one that works between remote Java VMs in a distributed network. We will use several code examples to illustrate the concepts. We will see:

- How remote events are implemented
- The importance of selecting event IDs and sequence numbers for distributed events
- The ability – by design – to 'chain' multiple event receivers/senders together, and the advantage of this 'composable' design
- How to use third-party event handling service

Remote Events in Jini

It's unlikely, though not impossible, that anyone who's been developing in Java for any length of time has managed to bypass events. Certainly if you've been programming with either AWT, Swing or JavaBeans you'll have come across various forms of listeners and events.

When using AWT events, the event producer (that sends the event) manages a set of objects registered by the event consumer (that receives the event). Each of these objects, supplied by consumers, managed by the producer, implements a 'listener' interface. They are often called listener objects. Once the consumer has registered a listener with the event producer (usually via an `addXXXListener()` method call of the source object), it can go about its own flow of logic. When the event producer is ready to fire (send) an event, it will go through its list of listeners and call the `notify()` method (or equivalent) on the listener interface. An `XXXEvent` object is passed as an argument to the `notify()` method, the consumer can retrieve event specific information from this `XXXEvent` object. This procedure is essentially a callback into the sink object. It is up to the sink object, the one that supplies the listener object in the first place, to implement the logic within the `notify()`method.

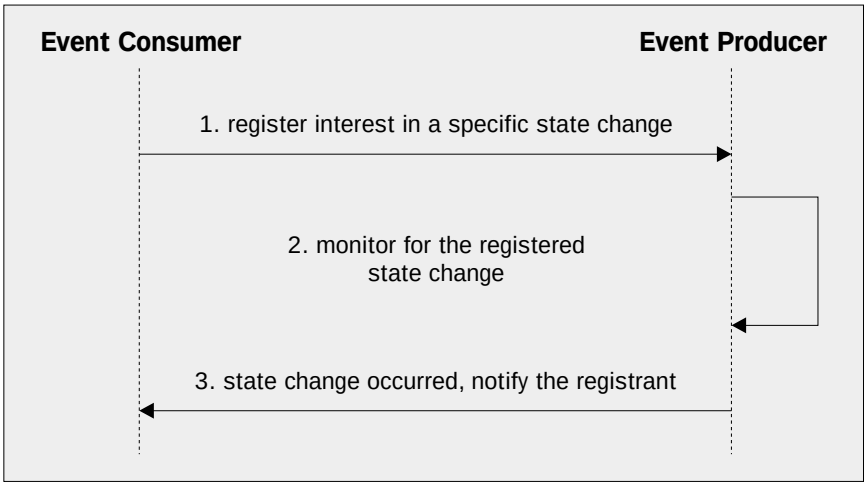
Within AWT, Swing, and JavaBeans, all this happens locally within a single VM. Jini extends this across the network, and across multiple VMs. All events thrown are subclasses of `net.jini.core.event.RemoteEvent`, and all listener objects must implement the `net.jini.core.event.RemoteEventListener` interface. This is an RMI based remote interface.

Event Concepts in Detail

What we're calling an event is, in real terms, a notification involving an event consumer and an event producer:

- The event consumer registers an interest with the producer, usually associated with a state change of the generator (for example, whenever a GUI button is clicked in AWT)
- The producer tracks registrations, and monitors for related state changes within its own realm (for example, waiting for a mouse event corresponding to the click of the button)
- Should the appropriate change take place, the producer notifies the consumer (for example, the button handling notifies the application code of the 'click')

So it looks something like this:



Notification, here, is asynchronous with respect to the normal flow of programming logic of the listener, meaning notifications of state changes in the generator are sent as and when the state changes, not when the interested application checks back for state changes, as is the case with polling. So how does the producer go about notifying the consumer?

Asynchronous Notification

When we take a closer look at how the producer goes about notifying the consumer, we will see that it actually calls the `notify()` method of the listener interface. This listener interface is implemented by an object that is supplied by the consumer.

In most circumstances, the event generator's method invocation on the event consumer is a synchronous call. Which means that the event generator will be typically blocked while the method is invoked, and not continue with its own logic until the invocation returns. Event generators should protect themselves from listeners with poorly written event handlers, typically accomplished by placing notification into a separate thread to the main event generating program thread. This will ensure that misbehaving consumers (that is, one that takes an exceedingly long time to process an event) cannot stop the main logic of the producer from executing.

Java Event Notification Mechanism

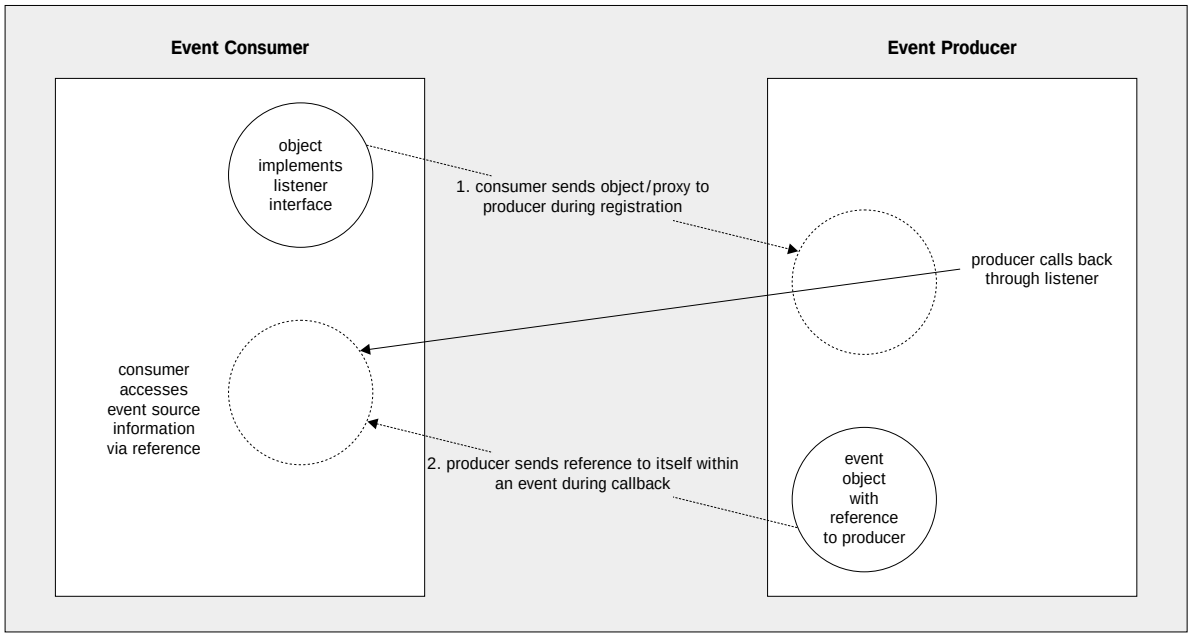
Most Java event notifications are implemented through the familiar listener code pattern.

The interaction is between a consumer and a producer of notifications. A listener is a Java object that implements a listener interface. The consumer passes a listener object to the event producer as a form of registration (indeed, sometimes the event consumer and the listener are the same Java object). The onus is on the event producer to keep track of this registration.

A typical producer can take registrations from many consumers on many state changes of interest, and must manage registration and un-registering appropriately.

When making event notifications, the producer provides an object to the consumer that implements the event interface – this is nearly always a data object. The actual type (subclass) of the event object will give the event handler information about the type of notification, while member variables of the object will provide additional information. One member variable of the event object is the source of the event; this is a reference to the producer itself.

This diagram illustrates the listener coding pattern.



Extending the Model Across the Network

When the event listener code pattern is extended across a network of multiple machines, something interesting occurs. Although the logical concept remains the same, the unreliable nature of the underlying delivery mechanism and infrastructure of a multiple machined network brings heavy-duty complexity to the simple programming paradigm. Jini's remote event API abstracts most of this complexity, and allows the programmer to respond to the new uncertainties.

First, let's look at the way remote events in Jini extend the single machine Java model. The consumer in this case still creates a Java object that implements a listener interface. However, since the event notification will be made remotely, there are really only two choices:

- Make the listener interface a remote interface via RMI
- Keep the listener interface local, but demand that the consumer supplies a proxy object containing the interface (allowing the consumer to use means other than RMI to communicate with the proxy)

For the provision of event notification, the first choice is by far the simplest, and it is also the chosen path of Jini. Jini defines a specific remote (RMI) interface as the remote event listener:

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

So the programmer of the consumer application is responsible for generating RMI stubs for the listener using the rmic tool, and providing a means of downloading the stubs, even though the consumer is really a client on a higher conceptual level. This makes good sense; the client is temporarily an RMI server when event notification is taking place through the remote listener interface.

The event generator supplies an event object as a parameter on the event notification that contains state information: an event ID, a sequence number, a hand-back serialized object (`MarshalledObject`), and a remote reference back to the producer that fired the event. The hand-back `MarshalledObject` is one supplied by the consumer itself as part of the listener registration; this can be a reference to state information within the consumer. The event object is an instance of a `RemoteEvent` subclass. `RemoteEvent` is defined as:

```
public class RemoteEvent extends java.util.EventObject
{
    public RemoteEvent(Object source,long eventID, long seqNum,
        MarshalledObject handback)
    public Object getSource () {…}
    public long getID() {…}
    public long getSequenceNumber() {…}
    public MarshalledObject getRegistrationObject() {…}
}
```

This means that another RMI stub is required from the listener in order to communicate with the event producer using the source reference contained in the event. However, as you'll usually use listener RMI stubs to make the initial remote event registration possible (assuming that the service/producer chose RMI as the transport mechanism for its service proxy), this simply means that additional class that needs to be in the RMI download server (or JAR file).

The state information from the producer is usually carried in a `RemoteEvent` subclass that may have additional fields and methods. The consumer uses the hand-back object to attach information to a specific event registration; this `MarshalledObject` is handed back to the consumer during notification.

An Example of Remote Events

We saw Jini events in action in an earlier example. Recall the `DiscoveryListener` interface that we implemented in Chapter 6. The event handler provides the `discovered()` method call that the `LookupDiscovery` helper class used to inform us when it had discovered lookup services.

```
public interface DiscoveryListener extends EventListener
{
    public void discovered(DiscoveryEvent e);
    public void discarded(DiscoveryEvent e);
}
```

Here, the `discovered()` method invocation is made asynchronously with respect to the normal execution flow of our application. The main program blocks the end of its flow so that the process itself can stay alive to handle the event notification whenever it happens (that is, when the discovery occurs).

The Problems Inherent to Remote Events

If you've worked with RMI, you'll have soon realized that any remote method might throw a `RemoteException` at any time. Which is another way of saying that any message sent between networked machines (and Java VMs of course) is liable to fail.

As either the machines in the network, or the network itself may fail at any time, and for any length of time, event registration should be treated as a type of scarce resource. In other words, event registrations should be leased to ensure the long-lived stability of the Jini system as a whole. This will prevent the producer from keeping too many stale registrations around, or wasting time firing events to consumers who may have either crashed or ceased their interest in the event without de-registering.

When remote event notifications are sent over a network, the following limitations are evident:

- Event delivery order cannot be generally assured, especially when there may be intermediary event processors involved
- Success of event delivery itself cannot be generally assured
- Latency (or delay) in getting an event delivered can be substantial when compared to the actual processing required for the event

These restrictions, combined with the possibility of occasional consumer or producer failure, make the software designer's life significantly more difficult. One must take these constraints into account when designing applications that work with remote events. For example, one could consider batching events together should there be a likelihood of many events traveling between consumer/producer pairs. This can substantially cut down on the average latency time and reduce overall network traffic. One may also provide a reliable event delivery mechanism on top of the basic Jini event infrastructure. Building any of this event processing functionality is beyond the scope of this chapter; however, we will have enough information to attack such a project by the end of this chapter.

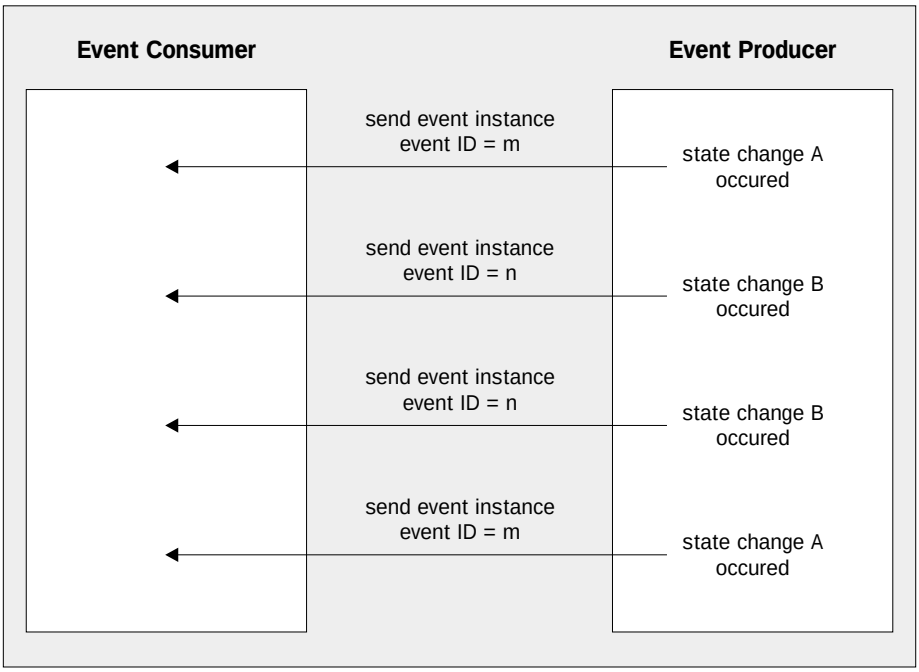
Selection of Event ID and Sequence Number

To assist in implementation of intermediary event processing services, Jini has provided the following:

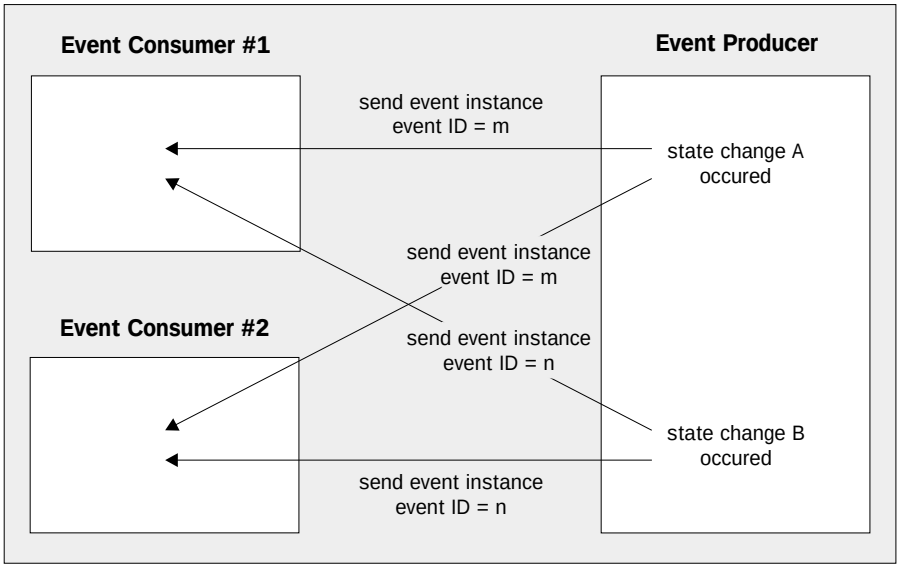
- Event IDs
- Sequence numbers with each event
- Simple event interfaces that are completely composable

The event ID uniquely identifies the type of event, and/or the registration instance, from a producer. Since a single consumer can be registered for many different types of events with a single producer, this will enable the consumer to quickly determine what a specific notification is all about.

Here we see this use of the event ID:

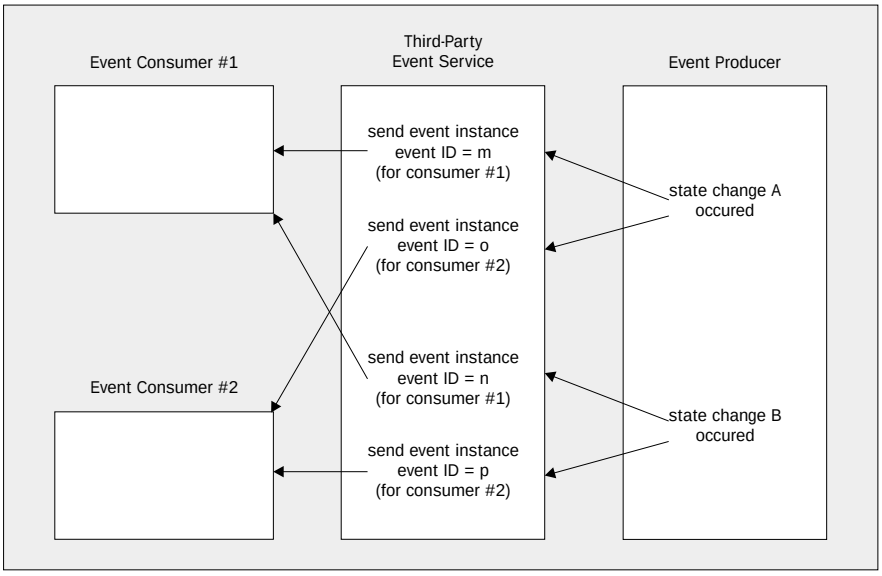


Typically, within one single producer, the same event ID can (but may not necessarily) be used for the same event type across multiple consumers, which allows one-to-one mapping of the event ID and type of event. The following diagram illustrates this use of the event ID by the producer.



As we will see in the next section when we talk about third-party events handling, we may need more than one event ID per event type. Third-party events handlers are useful, for example, to provide an in-order delivery of events even though the events may actually be delivered out of order (that is, the third-party stores and sorts the events before delivering to the consumer). Such a middleman service may register for event notification on behalf of multiple consumers. These registrations may be with the same consumer for a particular type of event. In order for the middleman service to distinguish between the different clients when the event is actually received, the middleman service will require the event ID to represent a unique 'registration number' instead of the simple event type – which is not unique from the view of the middleman service. In fact, reggie provides an event ID that is based on the registration, and is thus compatible with the deployment of middleman services.

This figure illustrates the implementation for an event ID based on the registration number.



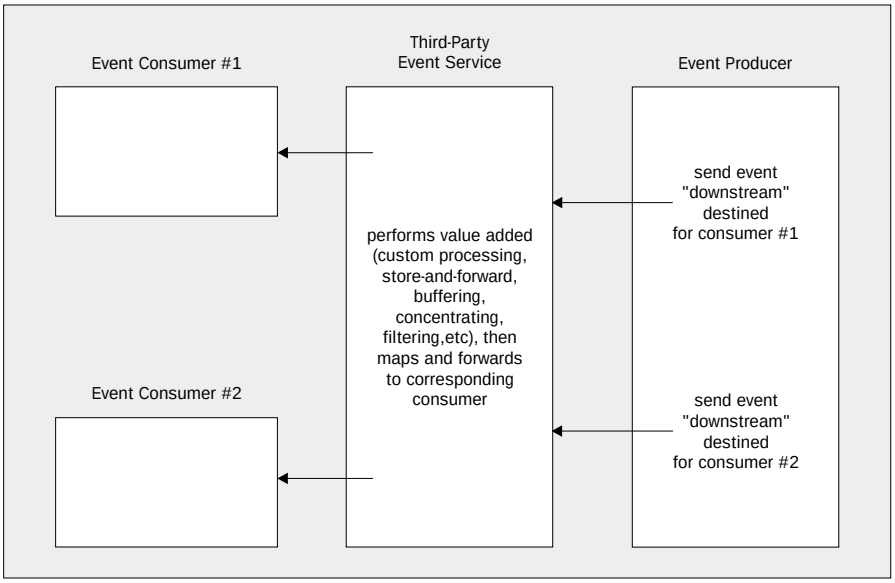
Sequence numbers specify the order in which events were generated by the event generator. It is not guaranteed that the events will arrive at the listener in the order they were generated.

The Jini specification requires that the sequence numbers of events, associated with a specific event ID and from a specific event producer, increase, but it doesn't specify what that increment should be. For example, if you receive two events with the same event ID from the same producer and one contains the sequence number 5 while the other one contains the sequence number 18, you can safely conclude that the event with the sequence number 18 was sent later by the event generator, even if you do receive it before you receive the event with sequence number 5. You can also specify, although the Jini specifications do not require it, a full ordered sequence of numbers. In this case the generator should increment the sequence number by 1 whenever events with the same event ID are fired. Other than providing a means of determining order, this can also keep count of the number of events that may be missing for the consumer (if we know in advance that the increment is 1).

This is vital for applications that cannot tolerate missing events. Our earlier example would have indicated, for example, that there were 18-5 = 13 events that had been generated for the same event ID between the two events that we have on hand.

Third-Party Event Handling

There are plenty of legitimate design scenarios where one might want to make use of third-party events handling, where someone else handles event registrations and/or event handling on your behalf. The following diagram illustrates this:



The clients and services that use third-party event handling generally fall into one or more of the following three categories. The event consumers or event producers:

- Do not want to handle events themselves
- Cannot handle the events themselves
- Want the added value that the third-party brings

The service may not want to handle the service because of the complex logic such handling requires, or because it may be positioned badly within the network for event handling – on a slow part of a network, or behind a firewall, for example.

There are actually quite a few examples of services that could not manage events, even if they wanted to: that is a Jini service that runs only on demand (that is, activatable services), services that run on limited Java platforms without RMI support, or even services that run on devices that don't have a Java VM at all. In these cases, a third-party that represents the service and handles events for it allows it to participate fully in a Jini network.

Services might elect to use a third-party for value added reasons: services like event ordering services, for example, event concentrating services, reliable event delivery services, event consolidation services, and so on.

Since version 1.1, the Jini distribution has included a third-party event mailbox service, code-named Mercury. Mercury receives events on behalf of event consumers, buffers or stores them, and allows consumers to retrieve these events at their convenience. We'll examine Mercury's functionality and put it to action in Chapter 11.

Event Routing Through Pipelines

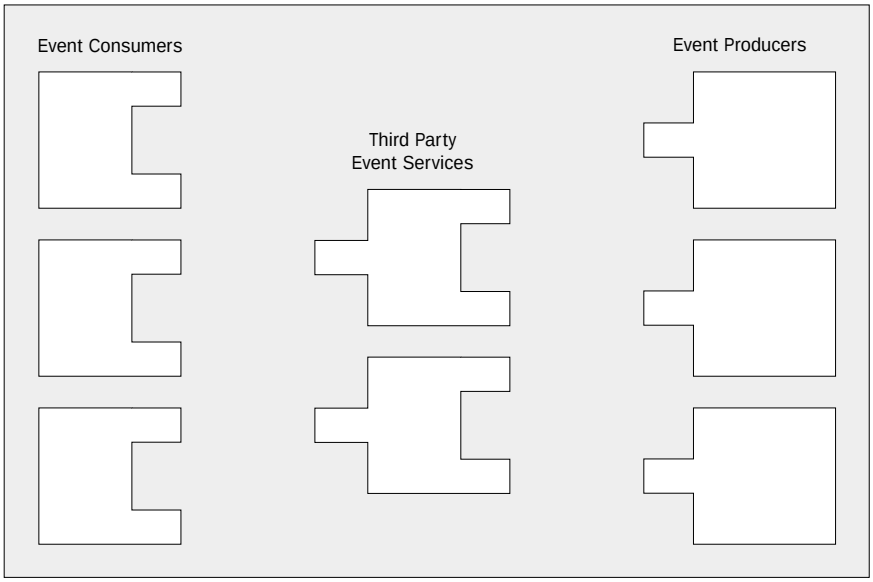
Hooking up third-party event delegates allows you to create a pipeline value-adding model. In effect, a series of specialized third-party handlers can be chained together in both the routing and processing of events.

A value adding, third-party handler can add generic functionality to event handling (store-and-forward functionality, for example, with an event mailbox like Mercury), or it can process associated event data across the pipeline (active processing of the attached data with the event).

This is useful if you're in the business of re-using software as well, allowing **composability** of sorts. You can compose functional blocks dynamically at deployment time to create newly combined functionality. Unix veterans will recognize this as a primary advantage provided by the operating system through the pipe '|' operator and utilities like 'tee'.

It is the simplicity of Jini's remote event mechanism that makes composability possible. Rather than using a specific listener interface and event type for each and every event type, as Swing and AWT do, Jini has one remote listener interface to fit all consumers and likewise one single remote event definition. This means that all event sources and event sinks are directly plug compatible – they can be plugged into one another without worrying about incompatibility.

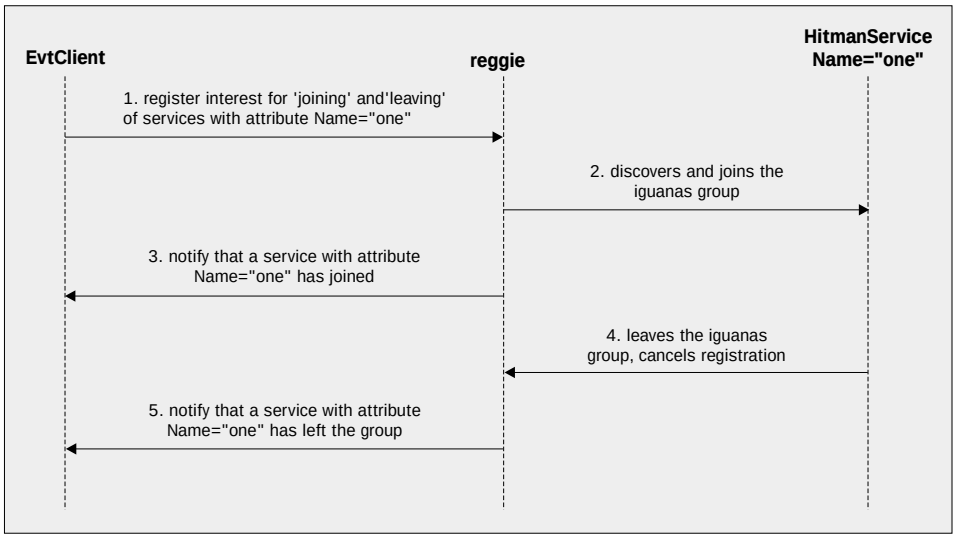
The following diagram illustrates the plug compatible action of Jini remote event sources and sinks.



Any third-party event service has to use the event ID to distinguish the information contained in the 'consumer, event type, generator' trio. The service must map the event ID to the trio, and all other services in the pipeline must understand the specific conventions that have been used in order for them to work together.

Implementing an Event Consumer

So let's implement a remote event consumer. You can find the code for this consumer in the `ch7\code\EvtClient` directory of the source code distribution. We'll be using the same client framework for our later explorations, but for now, this client will be interacting with a lookup service for registration and receipt of remote events. The lookup service will be the reference implementation. Reggie will generate remote events to the client based on the internal state changes in the proxies store. The following illustration shows how `EvtClient` works.



The `EvtClient`:

- Discovers an instance of a lookup service in the group iguanas
- Registers with the lookup service in order to be notified when a specific service (with attribute Name="one") joins and leaves the federation
- Catches the join and leave remote notifications from the lookup service, and prints the event ID and sequence number information to the standard output

The Event Producer Role of a Lookup Service

A Jini lookup service supports remote event registration directly through the registrar proxy. A client of the service can specify an interest in specific state changes within its service/proxy data store. The registration is performed via the `ServiceRegistrar` interface that we're already familiar with. There is one specific method in the `ServiceRegistrar` interface that is responsible for event registration:

```
public interface ServiceRegistrar
{
    ServiceRegistration register(ServiceItem item, long leaseDuration)throws
        RemoteException;
    Object lookup(ServiceTemplate tmpl) throws RemoteException;
    ServiceMatches lookup(ServiceTemplate tmpl, int maxMatches) throws
        RemoteException;

    EventRegistration notify(ServiceTemplate tmpl, int transitions,
        RemoteEventListener listener,MarshalledObject handback, long
        leaseDuration) throws RemoteException;
...
}
```

A template can specify the services state changes that can be monitored by the lookup service. The template may contain interface/class types, service IDs, and attributes as in service lookup. The state changes, specified in terms of transitions with respect to the template, are:

```
int TRANSITION_MATCH_NOMATCH = 1 << 0;
int TRANSITION_NOMATCH_MATCH = 1 << 1;
int TRANSITION_MATCH_MATCH = 1 << 2;
```

In other words, every time a change occurs in the proxy registration database, the lookup service will attempt to match the supplied template. It remembers the result of the previous match and will send notification only if the desired transition between match and no match is detected (for `TRANSITION_MATCH_NOMATCH`), or between no match and match (for `TRANSITION_NOMATCH_MATCH`), or if some attribute of a service is changed (for `TRANSITION_MATCH_MATCH`).

This method returns an `EventRegistration` instance. `EventRegistration` is a class specified as:

```
public class EventRegistration implements java.io.Serializable
{
    public EventRegistration(long eventID,
    Object eventSource,
    Lease eventLease,
```

```
        long seqNum) {…}
        public long getID() {…}
        public Object getSource() {…}
        public Lease getLease() {…}
        public long getSequenceNumber() {…}
    }
```

Event Consumer Jini Client Coding

Here is the code for the `EvtClient.java` implementation:

```
    import net.jini.discovery.*;
    import net.jini.core.lookup.*;
    import java.io.IOException;
    import java.rmi.RemoteException;
    import java.rmi.RMISecurityManager;
```

First, we need to import support for RMI calls, leasing, attribute entry, and template matching.

```
    import java.rmi.server.UnicastRemoteObject;
    import java.rmi.MarshalledObject;
    import net.jini.core.event.*;
    import net.jini.core.lease.*;
    import net.jini.lease.LeaseRenewalManager;

    import net.jini.lookup.entry.Name;
    import net.jini.core.entry.Entry;
```

Note that `EvtClient` class itself extends `java.rmi.server.UnicastRemoteObject`. This means that any references passed out of `EvtClient` will be automatically 'made' remote by the RMI runtime. We do not have to export the references explicitly.

The implementation of `DiscoveryListener` is created for handling non-remote notifications whenever an instance of a lookup service is discovered within the Jini federation. On the other hand, the implementation of `RemoteEventListener` is for remote notification from the lookup service whenever an instance of a specific service of interest registers with, or leaves, the lookup service/federation, or when its registration with the lookup service changes.

```
    public class EvtClient extends UnicastRemoteObject implements
        DiscoveryListener, RemoteEventListener
    {
        protected  ServiceRegistrar[] registrars;
        static final int MAX_MATCHES = 5;
```

The `main()` method creates an instance of `EvtClient` and sits dormant while the client waits for remote notifications from the lookup service. Optionally, if a command line argument exists, it passes it into the constructor of `EvtClient`. This command line argument can be use to specify the groups that this `EvtClient` instance should attempt to join (that is, discover lookup services for).

```
        static public void main(String argv[])
          {
            EvtClient myApp = null;
```

```
          try
            {
               if (argv.length > 0)
               myApp = new EvtClient(argv);
            else
               myApp = new EvtClient(null);
               synchronized (myApp)
                 {
                    myApp.wait(0);
                 }
            }
            catch(Exception e)
              {
                 System.exit(0);
              }
        }
```

The `doLookupWork()` method, like the variations that we have seen in the previous chapter, examines the service item of the registrar (proxy of reggie) and prints out some interesting information such as its service ID and the group that it services.

```
   private void doLookupWork()
      {
         ServiceMatches matches = null;
         String [] groups;
         String msg = null;
         if(registrars.length > 0)
           {
              msg = "";
              System.out.println("-----------------------------------------");
              System.out.println("Registrar: " + registrars[0].getServiceID());
              try
                {
                  groups = registrars[0].getGroups();
                  if (groups.length > 0)

                     for (int o=0; o<groups.length; o++)
                       {
                          msg += groups[o] + " ";
                       }
                  System.out.println("Groups Supported: " + msg);
                }
              catch (Exception e)
                {
                   e.printStackTrace();
                   System.exit(1);
                }
           }
      }
```

The constructor creates an `RMISecurityManager` as the client will be downloading proxies from the lookup service. It also starts a `LookupDiscovery` instance to handle discovery of lookup services within the specified groups from the input argument (or null which is the same as `LookupDiscovery.ALL_GROUPS`).

```
       public EvtClient() throws RemoteException
          {
            ClientCore(null);
          }

       public EvtClient(String [] ingroups)  throws RemoteException
          {
             ClientCore(ingroups);
          }

       private void ClientCore(String [] ingroups) throws RemoteException
          {
           if (System.getSecurityManager() == null)
             {
                System.setSecurityManager(new RMISecurityManager());
             }
           LookupDiscovery discover = null;

           try
              {
                 discover = new LookupDiscovery(LookupDiscovery.NO_GROUPS);
                 discover.addDiscoveryListener(this);
                 discover.setGroups(ingroups);
              }
           catch(IOException e)
              {
                 System.err.println(e.toString());
                 e.printStackTrace();
                 System.exit(1);
              }
          }
```

The `discovered()` method is an event notification method on the `DiscoveryListener` interface. It is called when the discovery protocol has produced a set of lookup service proxies. Here we save the reference to the proxies and simply call the `doLookupWork()` and the `doEventReg()`methods.

```
      public synchronized void discovered(DiscoveryEvent evt) {
             registrars = evt.getRegistrars();
             doLookupWork();
             doEventReg();
       }

       public void discarded(DiscoveryEvent evt) {

       }
```

The `doEventReg()` creates a template to specify matches based on attribute `Name="one"`. We submit the template to the `notify()` method of the reggie proxy, and request for notification on both `TRANSITION_NOMATCH_MATCH` and `TRANSITION_MATCH_NOMATCH` transitions by logically `OR`ing them. The consumer in this case is also the listener, therefore we pass the `this` reference for the `RemoteEventListener` argument. RMI runtime will automatically export our server and supply our stub object (`EvtClient_Stub`) to the client via the codebase. For the handback object, we have used a marshalled version of the reggie proxy itself. It is done here just to show how to create a `MarshalledObject`, and is not actually used. We request a permanent lease on the registration, which we know reggie will likely not grant. Whatever duration reggie decides to grant, we will pass the lease returned within the `EventRegistration` instance to a `LeaseRenewalManager` that will handle the 'forever' renewal of lease for us automatically.

```
public  EventRegistration myReg = null;

    protected void doEventReg()
       {
          if (registrars.length > 0)
            {
               Entry myAttrib[] = new Entry[1];
               myAttrib[0] = new Name("one");

            try
               {
                   myReg = registrars[0].notify(new
                       ServiceTemplate(null,null,myAttrib) /* tmpl */,
                       ServiceRegistrar.TRANSITION_NOMATCH_MATCH |
                       ServiceRegistrar.TRANSITION_MATCH_NOMATCH,
                       this,
                       new MarshalledObject(registrars[0])   /* handback */,
                       Lease.FOREVER  );
               }
             catch (RemoteException ex)
                {
                  ex.printStackTrace();
                  System.exit(1);
                }
             catch (IOException ex)
                {
                  ex.printStackTrace();
                  System.exit(1);
                }

                new LeaseRenewalManager(myReg.getLease(), Lease.FOREVER, null);
             }
        }
```

Finally, the `notify()` method implements the only method on the `RemoteEventListener` interface. This is essentially the event notification method. Here, we simply dissect and print some interesting information on the `RemoteEvent` that is supplied. Since we know that the event is actually a subclass of `RemoteEvent` called `ServiceEvent`, we can obtain the additional information provided by this event.

```
      public synchronized void notify (RemoteEvent inEvt) throws
         UnknownEventException, RemoteException
         {
            ServiceEvent srvEvt = (ServiceEvent) inEvt;
            System.out.println("got a notification from:");
            ServiceRegistrar mySrc = (ServiceRegistrar) srvEvt.getSource();
            System.out.println("   Source service ID: " + mySrc.getServiceID());
            System.out.println("   Event ID: " + srvEvt.getID());
            System.out.println("   Sequence Number: " +
                srvEvt.getSequenceNumber());
            System.out.println("   Due to Proxy: " + srvEvt.getServiceID());

            if (srvEvt.getServiceItem() == null)
               System.out.println("   Proxy Deleted");
             else
                System.out.println("   Attributes: " +
                    srvEvt.getServiceItem().attributeSets[0]);
               System.out.println("   Transition: " + srvEvt.getTransition());
         }
}
```

Run the batch file to compile the code:

```
..\bats\buildit EvtClient.java
```

Next, we need to create the RMI stubs.

Creating RMI Stubs

As we saw earlier, the `RemoteEventListener` interface is an RMI interface. We create RMI stubs using the rmic utility provided by the JDK. A batch file called `makejar.bat` is supplied for this purpose; you will find it in the `ch7\code\EvtClient` directory. The batch file contains:

```
call ..\bats\setpaths.bat
rmic -classpath %JINIJARS%;. -v1.2 EvtClient
jar cvf EvtClient-dl.jar EvtClient_Stub.class
copy EvtClient-dl.jar %WROXHOME%
```

rmic is used to create the RMI stubs required by `EvtClient`. A JAR file is then created with the stub and copied to the root of the stubs class server.

> **Case-Sensitivity Caveat**
> **Be very careful of case-sensitivity if you're creating JAR files on a Win32 system. Spelling `EvtClient_Stub.class` as `EvtClient_stub.class` will still work on a Win32 command line, but create a horrible mess when you get around to debugging: it will lead to service event notifications that cannot locate the classes it needs.**

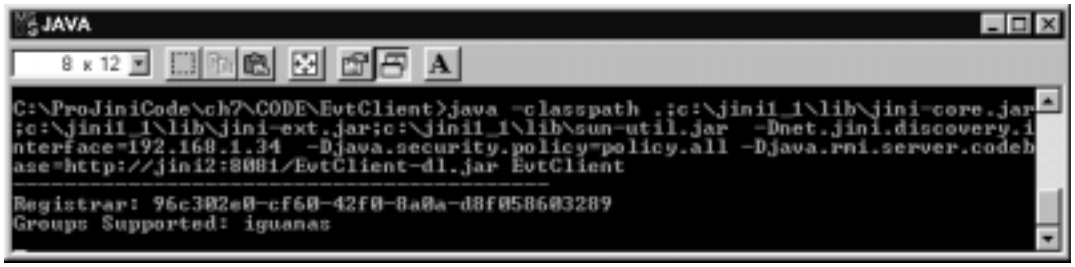
Testing the Event Client

To test the event client, we will need to:

- Change directory to the ch7\code\bats directory where the startup batch files are located
- Delete the log directories for reggie and rmid to clear up previously stored states from other experiments; you can use the runclean.bat file under Windows 95/98 (or manually remove the directories in Win2000 or NT)
- Start the HTTP class server by starting runhttpd.bat
- Start the stubs serving class server by starting runhttpdstubs.bat
- Start the RMI Activation daemon, RMID, by starting runrmid.bat
- Start one copy of reggie on the iguanas group by starting the runlookup1.bat file
- Wait for the setup VM to complete, now reggie is running properly
- Change directory to Ch7\code\EvtClient directory
- Start our event client by executing the runevt.bat file in this directory, it contains:

```
call ..\bats\setpaths.bat
java -classpath .;%JINIJARS%  -Dnet.jini.discovery.interface=%ADAPTERIP%  -
Djava.security.policy=policy.all -
Djava.rmi.server.codebase=http://%STUBHOST%/EvtClient-dl.jar EvtClient %1 %2
```

At this point, you should see our client discover the lookup service, and an indication of the registration with the service for event notifications. The output from the client should be similar to this:



Since there aren't any services joining or leaving the federation, there is no transition from the match/no-match states within the lookup service, so we ought to start such a service.

Go to the `Ch9\code\hitman` directory, we will borrow a service for this purpose. Don't worry about what it does for now, because we will be taking a detailed look at this service in the transaction coverage of this chapter. For now, you only need to know that the proxy registered by this service will have the attribute `Name="one"` that is necessary to trigger the desired transition. This will cause reggie to send an event to our waiting client. Compile the `Hitman` service by using:

```
..\bats\buildit HitmanService.java
```

Create the stubs by using:

```
makejar
```

Create a directory for the log files:

```
md hitlogs
```

Finally, start the `HitmanService` by running the `runhit1.bat` file.

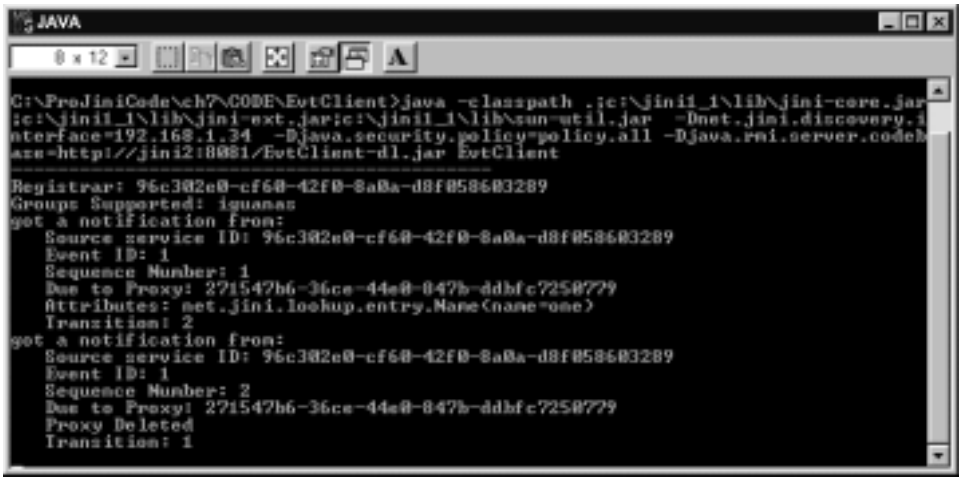
The following image shows what the `HitmanService` should look like after it has started. It actually has a GUI. In fact, if you do click on the button, it will do an orderly un-registration with the lookup service by canceling the lease on the proxy registration – but don't click it yet:

Shortly after starting the service, you should see the remote event being caught and decoded by our `EvtClient`. This shows something close to what you should see:

What's happened is that the search template that specifies the attribute `Name="one"` has switched from a no_match to match state when the `HitmanService` proxy is registered. This is exactly what the client registered for.

To observe the event sent when transitioning from match to no_match (numeric value is 2) click the button on the `HitmanService` GUI to perform an orderly de-registration of the service's proxy.

You should see the `EvtClient` printing out the details of the new event received. Here we see this new decoded output.



You may want to repeat this and observe the event ID and sequence number being assigned by reggie in this scenario.

Implementing an Event Producer

Now to turn the tables and see how the services implement a remote event producer. As you might expect, this is a bit more complicated than implementing a consumer.

Specifically, the event generator needs to:

- Provide event registration (Jini does not specify how this should be implemented) and keep track of the registrations
- Monitor and scan for the state changes
- Co-ordinate event firing to all the interested consumers
- Assign `eventIDs` and sequence numbers to satisfy the application's needs

We'll provide an event registration mechanism for the event generator through a new interface called `IndexKeeperRemote`. This interface inherits from `IndexKeeper`, but adds a remote event registration method to it:

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

public interface IndexKeeperRemote extends IndexKeeper,java.rmi.Remote
  {
```

```
      public EventRegistration addRemoteListener(RemoteEventListener listener)
         throws java.rmi.RemoteException;

   }
```

Note that the interface is also made remote, unlike the original `IndexKeeper` implementation. This means that the proxy object that is stored within the lookup service will be an RMI stub to an implementation of `IndexKeeperRemote` – and not a custom `IndexKeeper` object as we have seen previously.

The state-change we're looking for is simply a specified elapse of time. We'll use a thread that goes to sleep, waking at intermittent intervals to fire events to interested clients. It co-ordinates the firing of events by sequentially scanning the list of registered listeners, and creates a thread that will fire the event. This means that the event producing application will not be blocked by a notification call taking a long time to process (out of order event delivery will also be a distinct possibility).

The event ID is assigned according to the position of the registration in the list of listener registrations. This will be the same for all events from the same registration. The sequence number is implemented globally across all event types, and will increment irregularly should there be more than one type of event registration. In other words, the sequence numbers aren't ordered.

The code for the service can be found in the `ch7\code\EvtService` directory. Here is the `EvtService.java` source code:

```
import java.rmi.Remote;
import java.rmi.server.RemoteObject;
```

We need to import RMI support, lookup entry for attribute management, discovery protocol support and the `JoinManager`.

```
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 net.jini.lookup.ServiceIDListener;
import net.jini.lookup.entry.Name;
import net.jini.core.entry.Entry;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import net.jini.lookup.JoinManager;
import net.jini.discovery.*;
import net.jini.core.lookup.ServiceID;
```

The `IdPrinter` defined implements the `ServiceIDListener` interface. This is the event notification interface (non-remote) for the service ID assigned by the lookup service. `JoinManager` will make an event notification on the `serviceIDNotify()`method upon a service ID assignment. We print it to standard output as we did before.

```
class IdPrinter implements ServiceIDListener
   {
       int myIndex;
       public IdPrinter(int initIndex)
          {
             myIndex = initIndex;
          }

       // A real service will save the serviceID in persistent store
       public void serviceIDNotify(ServiceID serviceID)
          {
             System.out.println("instance " + myIndex + " has been assigned
                 service ID: " + serviceID.toString());
          }
   }
```

Our `EvtService` class, which implements the remotely callable `IndexKeeperRemote` interface, inherits from `java.rmi.server.UnicastRemoteObject` to make RMI reference handling simple.

```
public class EvtService extends  UnicastRemoteObject implements
    IndexKeeperRemote
       {
          static final int MAX_INSTANCES = 3;
          static final String [] GROUPS1 =  { "iguanas"};
```

`myIndex`, as you might recall from the last chapter, is an instance number given to a specific instance of the service proxy. Since our proxy is actually an RMI stub in this case, the index is assigned to the server-side instance. We maintain a worker thread that will wake up periodically to perform event firing.

The variable `evtSeqNum` holds the global event sequence number, and is initialized to 1. It increments across all events. The worker thread will be incrementing this sequence number by 1 each time it fires an event – regardless of event ID or event type. `meExported` is a temporary holder for the exported instance of the `evtService` itself, and it will be used later by the worker thread to send a remote reference of the source object (within the `RemoteEvent` object) to a listener.

```
          int myIndex = 0;
          Thread worker = null;
          RemoteObject meExported = null;
          long evtSeqNum = 1L;
          protected JoinManager myJM = null;
```

The `main()` method creates an instance of `EvtService`, passing it an integer index that may be supplied as a command line argument. It then sits idle while `JoinManager` and the worker event firing thread goes to work.

```
public static void main(String argv[])
   {
      EvtService myApp = null;
      try
        {
           if (argv.length > 0)
              myApp = new EvtService(Integer.parseInt(argv[0]));
           else
```

```
                myApp = new EvtService(300);
          }
        catch (Exception e)
          {
             e.printStackTrace();
             System.exit(1);
          }

        synchronized(myApp)
          {
            try
              {
                 myApp.wait(0);
              }
            catch(InterruptedException e)
              {
                 System.exit(0);
              }
          }
      }  // of main()
```

The `EvtService` itself creates an RMI security manager for the `RemoteEventListener` callback.

```
      LookupDiscoveryManager ldm = null;
      JoinManager jm = null;

      public EvtService() throws RemoteException
        {
           StartService(-1);
        }

      public EvtService(int inIndex) throws RemoteException
        {
          StartService(inIndex);
        }
      private void StartService(int inIndex) throws RemoteException
        {
          myIndex = inIndex;
          meExported = this;
          if (System.getSecurityManager() == null)
            {
              System.setSecurityManager(new RMISecurityManager());
            }
```

It creates the attribute `Name="one"` so that the client will be able to find the service proxy. It also creates the worker thread to fire events occasionally. We will see the sleeper class that implements this thread later. Note that the thread is created in the suspended state at this time. It then creates a `LookupDiscoveryManager` instance to handle discovery protocol for the iguanas group. This `LookupDiscoveryManager` instance, along with the attributes, the RMI proxy, and an instance of the `IdPrinter` class, are supplied as parameters to start the `JoinManager` operation. The `JoinManager` will perform the Join protocol and assure that our service will be available within the iguanas group as long as it is kept alive.

```
          String [] groupsToDiscover = GROUPS1;
          Entry [] attributes = new Entry[1];
          attributes[0] = new Name("one");
          worker = new Thread(new Sleeper(this));
           try
             {
               ldm = new LookupDiscoveryManager(groupsToDiscover,
                         null /* unicast locators */,
                         null /* DiscoveryListener */);
               jm = new JoinManager(this,  /* service */
                        attributes,
                        new IdPrinter(myIndex) /* ServiceIDListener*/,
                        ldm /* DiscoveryManagement */,
                        null /* LeaseManager */);

             }
           catch(Exception e)
             {
                e.printStackTrace();
                System.exit(1);
             } // catch
           worker.start();

        }   //EvtService
            attributes[0] = new Name("one");
         worker = new Thread(new Sleeper(this));
           try
             {
                ldm = new LookupDiscoveryManager(groupsToDiscover,
                          null /* unicast locators */,
                          null /* DiscoveryListener */);
                jm = new JoinManager(this,  /* service */
                         attributes,
                         new IdPrinter(myIndex) /* ServiceIDListener*/,
                         ldm /* DiscoveryManagement */,
                         null /* LeaseManager */);
             }
           catch(Exception e)
             {
               e.printStackTrace();
               System.exit(1);
             } // catch
            worker.start();

        }   //EvtService
```

The `getIndex()` method did not change implementation. However, it now retrieves the index for this service instance. This means that every `getIndex()` call will result in an RMI call from the client to the service.

```
      public int getIndex() throws RemoteException
        {
          return myIndex;
        }
```

Tracking Event Registrations

For convenience, we'll use the general and lightweight `javax.swing.event.EventListenerList` to implement the list of listeners. Although this class can manage heterogeneous lists consisting of listeners of different types, we won't be using this feature of the list in this example.

```
      protected EventListenerList myList = new EventListenerList();
```

Here is the event registration handling routine we specified in the new `IndexKeeperRemote` interface. As the Jini specifications don't make recommendations on the correct implementation of registration, we're going to borrow the JavaBeans conventions.

```
public synchronized EventRegistration addRemoteListener(RemoteEventListener
    listener) throws RemoteException
{
    System.out.println("got a registration!");
    myList.add(RemoteEventListener.class, listener);
```

After we have added the listener to the list, we'll find out which position it is actually stored in and note this for the event ID – taking advantage of this specific list implementation and knowing that the position will not change unless the listener element is removed (for this simple example, we will ignore the need to remove listeners).

```
       //find out where it is added to determine event ID
         long tpEventID = 0;
         Object[] listenerList = myList.getListenerList();
         for (int i = listenerList.length - 2; i >= 0; i -= 2)
           {
             if (listenerList[i+1].equals(listener))
               {
                  tpEventID = i+1;
                  break;
               }
           }
        return new EventRegistration(tpEventID, this, null, evtSeqNum);
}
```

Once we have the event ID, we can return the source producer, event ID, current sequence number, and a lease back to the client within an `EventRegistration` instance. We're not creating a lease at this point, but we will later on.

Firing Events

The next method, `fireRemoteEvent()` iterates through the listener list and fires an event for each listener within the list. It will be executed by the worker thread.

```
public void fireRemoteEvent()
  {
     RemoteEvent anEvent = null;
     System.out.println("Checking for listeners...");
     Object[] listeners = myList.getListenerList();

     for (int i = listeners.length - 2; i >= 0; i -= 2)
       {
```

```
          if (listeners[i] == RemoteEventListener.class)
            { // redundant check in      our case
              RemoteEventListener aList = (RemoteEventListener) listeners[i+1];
          try
            {
              if (anEvent == null)
                {
                  anEvent = new RemoteEvent(meExported, i+1,
                              evtSeqNum++, null);
                }
```

Note that the worker thread itself makes use of short-lived threads to fire the events. This will make sure that the worker thread will not be blocked by a `notify()` call on a listener that does not return for a long time. Remember that `notify()` calls are synchronous RMI calls.

```
                System.out.println("Fired one event...");
                // use short temporary threads to avoid blocking
                new Thread(new Notifier(aList, anEvent)).start();
            }
          catch(Exception ex)
            {
              ex.printStackTrace();
            }
           }// of if listener==RemoteEventLister.class
         } // of for i
    }    // of fireRemoteEvent
}
```

The sleeper class actually implements the logic of the worker thread. It will sleep for 30 seconds, wake up and fire remote events until the service terminates.

```
class Sleeper implements Runnable
  {
    EvtService myService;

    public Sleeper(EvtService inServ)
      {
        myService = inServ;
      }
    public void run()
      {
        int loopCounter = 1;
        while(true)
          {
             System.out.println("in loop... " + loopCounter++);
             System.out.flush();
           try
             {
                Thread.sleep(30000L);
             }
           catch (Exception e) {}
           myService.fireRemoteEvent();
          }
      }
  }
```

The class `Notifier` implements the actual notification for any specific listener. It is a worker thread that is created by the main worker thread. The lifetime of this thread is exactly one single event notification. This is not good production code for most VMs – a more complex thread-pool based implementation would be significantly more efficient. However, we've kept things simple here to help emphasize the event handling and firing logic.

```
class Notifier implements Runnable
  {
    RemoteEventListener myListener;
    RemoteEvent myEvent;
    public Notifier( RemoteEventListener inLis, RemoteEvent inEvt)
      {
        myListener = inLis;
        myEvent = inEvt;
      }
    public void run()
      {
        try
          {
             myListener.notify(myEvent);
          }
        catch (Exception e)
          {
            e.printStackTrace();
          }
      }
  }
```

That is all the code to the service. From the `ch7\code\EvtService` directory, compile the code using:

```
..\bats\buildit EvtService.java
```

Making RMI Stubs

We need to create RMI stubs for the `IndexKeeperRemote` remote interface that we are implementing. These stubs will also be the proxy objects that will be shipped to the lookup service. We can make the RMI stub using the batch file in the `ch7\code\EvtService` directory: `makejar.bat. it contains:`

```
call ..\bats\setpaths.bat
rmic -classpath %JINIJARS%;. -v1.2 EvtService
jar cvf EvtService-dl.jar EvtService_Stub.class IndexKeeperRemote.class
IndexKeeper.class
copy EvtService-dl.jar %WROXHOME%
```

The batch file above will create the JAR file containing the stub and related class. It will also copy the resulting JAR file into the root directory of the HTTP class server for stubs as well.

We are now ready to test the event producer service. However, we do not yet have a client that understands the `IndexKeeperRemote` interface used for this service.

Modifying the EvtClient to use EvtService

Since the `EvtClient` application was first coded to use reggie for event registration, we can quickly adapt it for use with our `EvtService`.

You will find the source code in the ch7\code\EvtClient2 directory of the distribution. We have actually inherited from EvtClient and then made it compatible with EvtService. Because of this, you must copy over EvtClient.class from the ch7\code\EvtClient directory before compiling the source.

In addition, we will also need to copy over the IndexKeeperRemote.java interface file and the IndexKeeper.java file that it is based on.

The doEventReg() method now uses the class type IndexKeeperRemote to find the proxy instead of attributeName="one". It also calls the addRemoteListener() method to add itself as a listener, instead of the notify() method of the ServiceRegistrar as before.

```
protected void doEventReg()
   {
     if (registrars.length > 0)
       {
          Class [] myClassType = { IndexKeeperRemote.class };

      try
        {
        IndexKeeperRemote myES = (IndexKeeperRemote) registrars[0].lookup(new
                                    ServiceTemplate(null,myClassType,null));
         if (myES != null)
           {
             myES.addRemoteListener(this);
             System.out.println("registered our interest in the event...");
           }
         else
             System.out.println("cannot find any proxy for event service...");
        }
          catch (RemoteException ex)
             {
                ex.printStackTrace();
                System.exit(1);
             }
          catch (IOException ex)
             {
                ex.printStackTrace();
                System.exit(1);
             }
        }
   }
```

The other minor modification required is to change the reference from the ServiceRegistrar to the IndexKeeperRemote interface within the notify()event notification method.

```
public synchronized void notify (RemoteEvent inEvt) throws
   UnknownEventException, RemoteException
    {
      RemoteEvent srvEvt =  inEvt;
      System.out.println("got a notification from:");
      IndexKeeperRemote mySrc = (IndexKeeperRemote) srvEvt.getSource();
      System.out.println("   Source instance: " + mySrc.getIndex());
      System.out.println("   Event ID: " + srvEvt.getID());
      System.out.println("   Sequence Number: " + srvEvt.getSequenceNumber());
    }
```

Those are all the changes necessary. Follow the steps listed earlier to compile, make stub, and make the necessary JAR file for this new EvtClient.

Testing Our Own Remote Event Producer and Consumer

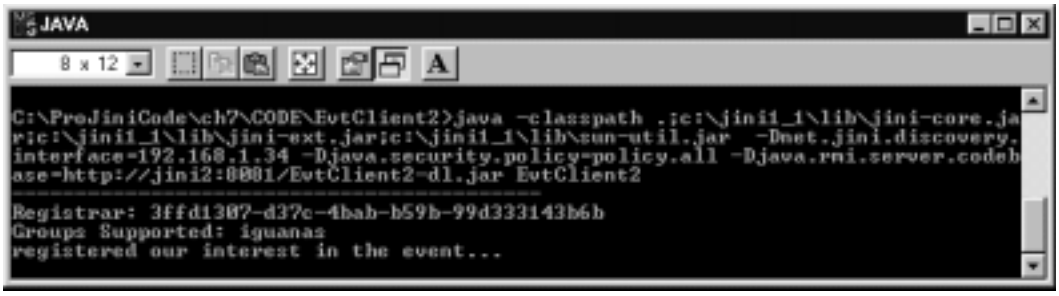
To test out the new `EvtClient2` and the `EvtService`, we perform the following steps.

- Change directory to the location of the startup files
- Clean up the log files by running runclean.bat or remove the log directories manually.
- Start the class server by running the batch file runhttpd.bat
- Start the stub class server by running the batch file runhttpdstubs.bat
- Start the rmid activation daemon by running the batch file runrmid.bat; this should also start the reggie instance from before
- Change directory to the location of the EvtService by running the runevt.bat file. This file contains:

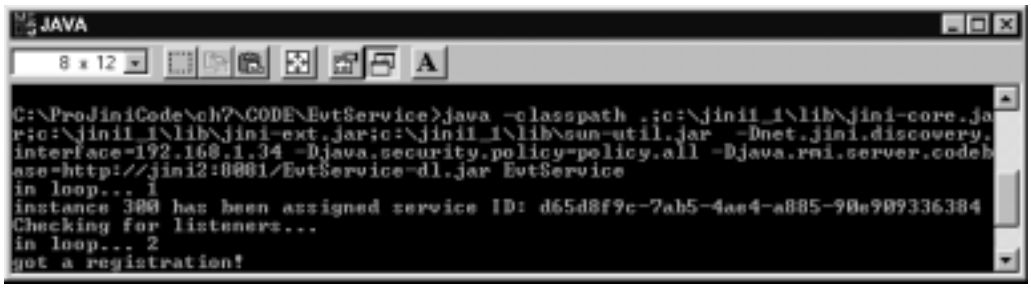
```
call ..\bats\setpaths.bat
java -classpath .;%JINIJARS%  -Dnet.jini.discovery.interface=%ADAPTERIP% -
Djava.security.policy=policy.all -
Djava.rmi.server.codebase=http://%STUBHOST%/EvtService-dl.jar EvtService %1 %2 %3
```

At this point, you should see the service coming up and a service ID being assigned by reggie

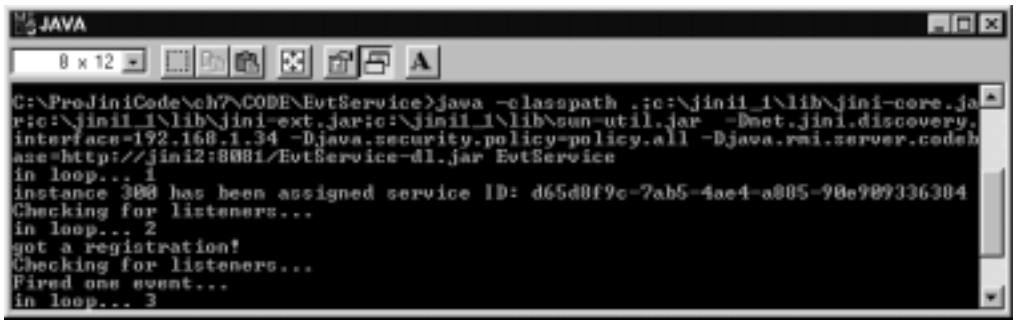
- Open another command shell with the JDK/Jini environment set, and change directory to Ch7\code\EvtClient2
- Start an instance of the client using the runevt.bat file in this directory. You should see the client starting up, locate the lookup service, and then make a registration with the service



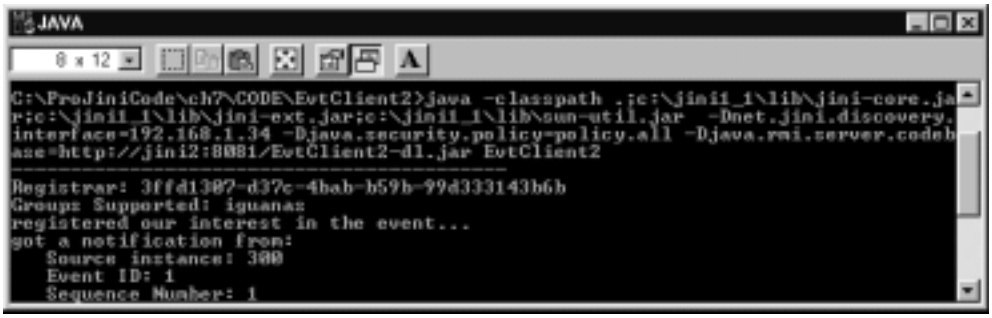
On the service side, you'll see a message indicating the worker thread has checked the listener list, and you should see the event registration on the service output.



If you wait until the next worker thread sweep, you should see the worker thread announce that it has fired an event.



You can also see in the client output that the remote event notification has been received and decoded. The remote instance number printed (`300`) was obtained by a call to the `getIndex()` method which executes on the service side.



So we've successfully created a remote event consumer that works well with lookup services like reggie, by creating an event producer service and modifying the original consumer to work with our service.

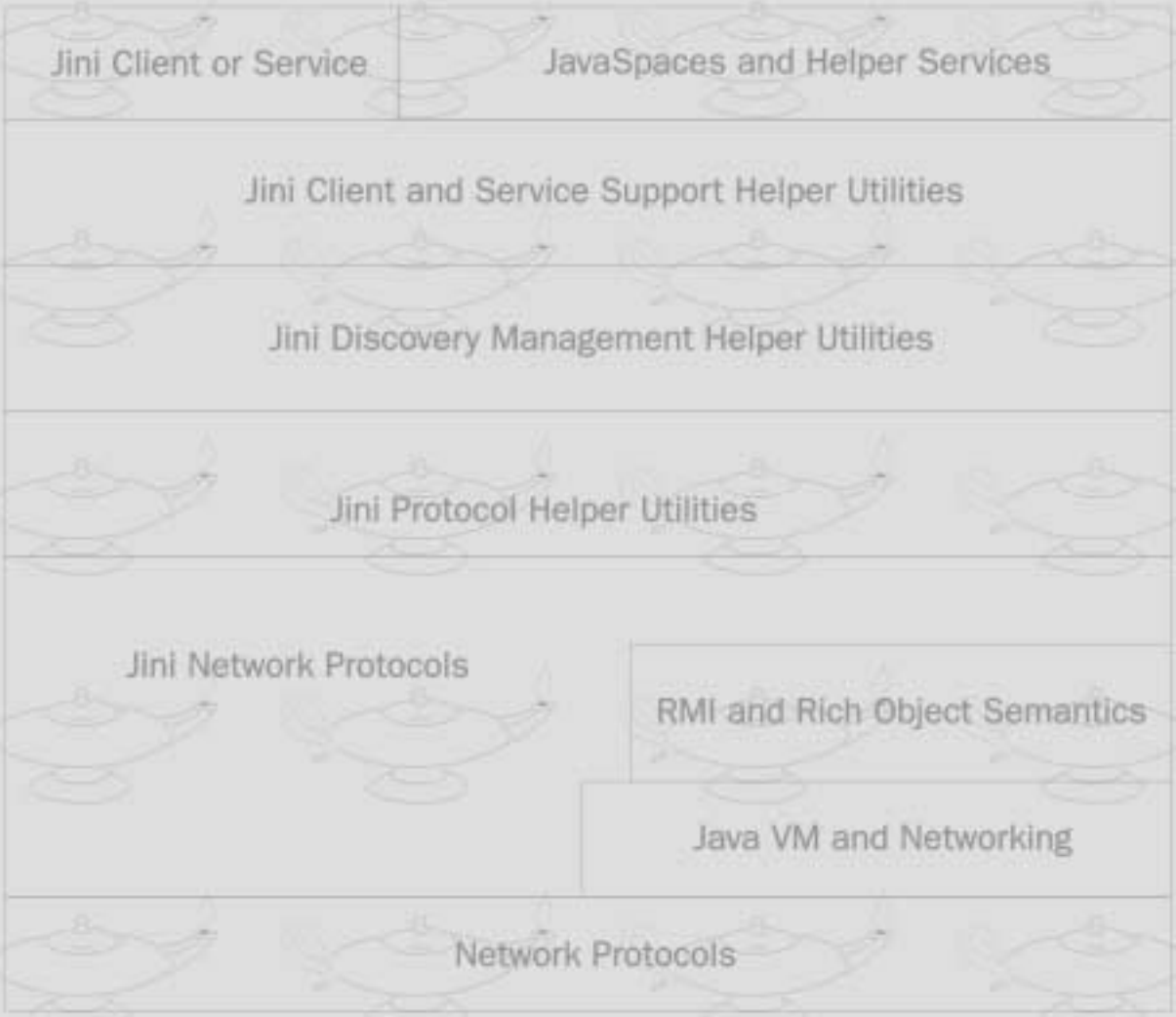
Summary

In this chapter, we have thoroughly explored remote events in the Jini context. First, we reviewed and re-examined the Java event model, and saw how remote events extend the same listener code pattern from intra-Java VM to across the network. The unreliable transport across a network has made event implementation across a network more difficult to implement than the local version.

We saw the single interface `RemoteEventListener`, and single class `RemoteEvent` that Jini's remote events are based on. We talked about the ability to compose multiple Jini services together to form an event processing pipeline. The concept of third-party event services was introduced, and the importance of selecting the right algorithm for generating event ID and sequence numbers was discussed.

Through hands-on coding, we worked with the Jini lookup service (reggie) by registering to receive remote events whenever a registered service changes states. Finally, we created our own event notification service that will accept event registrations from clients. We have also modified the original event client to work with our own event notification service.

In the next section, we will cover the one issue that we have purposely avoided in our implementation – distributed leases in Jini.


Jini Client or Service
JavaSpaces and Helper Services
Jini Client and Service Support Helper Utilities
Jini Discovery Management Helper Utilities
Jini Protocol Helper Utilities
Jini Network Protocols
RMI and Rich Object Semantics
Java VM and Networking
Network Protocols