CHAPTER 8

Leasing

In distributed applications, there may be partial failures of the network or of components on the network. *Leasing* is a way for components to register that they are alive, but to ensure that they are "timed out" if they fail or are unreachable. Leasing is the mechanism used between applications to give access to resources over a period of time in an agreed-upon manner.

Leases are requested for periods of time, and these requests may be granted, modified, or denied. The most common example of a lease is when a service is registered with lookup services. A lookup service will not want to keep a service forever, because it may disappear. Keeping information about nonexistent services is a waste of resources on the lookup service and also may lead to clients wasting time trying to access services that aren't there. As a result, a lookup service will grant a lease saying that it will only keep information for a certain period of time, and the service can renew the lease later if desired.

Requesting and Receiving Leases

Leases are requested for a period of time. In Jini, a common use of leasing is for a service to request that a copy of the service be kept on a lookup service for a certain length of time, for delivery to clients on request. The service requests a time in the `ServiceRegistrar`'s `register()` method. Two special values of the time are as follows:

- `Lease.ANY`: The service lets the lookup service decide on the time.
- `Lease.FOREVER`: The request is for a lease that never expires.

The lookup service acts as the granter of the lease and decides how long it will actually create the lease for. (The lookup service from Sun typically sets the lease time as only five minutes.) Once it has done that, it will attempt to ensure that the request is honored for that period of time. The lease is returned to the service and is accessible through the `getLease()` method of the `ServiceRegistration` object. These objects are shown in Figure 8-1.

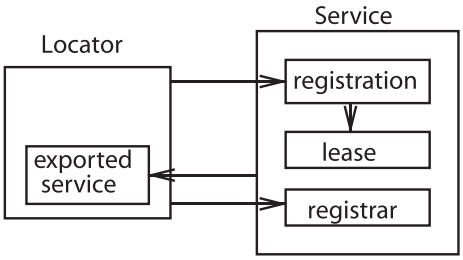

Figure 8-1. *Objects in a leased system*

```
ServiceRegistration reg = registrar.register();
Lease lease = reg.getLease();
```

The principal methods of the `Lease` object are as follows:

```
package net.jini.core;
public interface Lease {
    void cancel() throws
                  UnknownLeaseException,
                  java.rmi.RemoteException;
    long getExpiration();
    void renew(long duration) throws
                  LeaseDeniedException,
                  UnknownLeaseException,
                  java.rmi.RemoteException;
}
```

The expiration value from `getExpiration()` is the time in milliseconds since the beginning of the epoch (the same as in `System.currentTimeMillis()`). To find the amount of time still remaining from the present, the current time can be subtracted from this, as follows:

```
long duration = lease.getExpiration() - System.currentTimeMillis();
```

Cancellation

A service can cancel its lease by using `cancel()`. The lease communicates back to the lease management system on the lookup service, which cancels storage of the service.

Expiration

When a lease expires, it does so silently. That is, the lease granter (the lookup service) will not inform the lease holder (the service) that it has expired. While it might seem nice to get warning of a lease expiring so that it can be renewed, this would have to be in advance of the expiration (e.g., "I'm just about to expire; please renew me quickly!"), but this would complicate the leasing system and not be completely reliable anyway (e.g., how far in advance is soon enough?).

Instead, it is up to the service to call `renew()` before the lease expires if it wishes the lease to continue.

Renewing Leases

Jini supplies a `LeaseRenewalManager` class that looks after the process of calling `renew()` at suitable times.

```
package net.jini.lease;
public Class LeaseRenewalManager {
    public LeaseRenewalManager();
    public LeaseRenewalManager(Lease lease,
                               long expiration,
                               LeaseListener listener);
    public void renewFor(Lease lease, long duration,
                         LeaseListener listener);
    public void renewUntil(Lease lease,
                           long expiration,
                           LeaseListener listener);
    // etc
}
```

The `LeaseRenewalManager` manages a set of leases, which may be set by the constructor or added later by `renewFor()` or `renewUntil()`. The time requested in these methods is in milliseconds. The `expiration` time is measured since the epoch, whereas the `duration` time is measured from now.

Generally, leases will be renewed and the manager will function quietly. However, the lookup service may decide not to renew a lease and will cause an exception to be thrown. This exception will be caught by the renewal manager and will cause the listener's `notify()` method to be called with a `LeaseRenewalEvent` as a parameter, which will allow the application to take corrective action if its lease is denied. If the listener is `null`, then no notification will take place.

Granting and Handling Leases

The preceding discussion looked at leases from the side of the client that receives a lease and has to manage it. The converse of this is the agent that grants leases and has to manage things from its side. This section contains more advanced material that you can feel free to skip for now; it is not needed until Chapter 16. An example of creating a lease is also presented in Chapter 15.

A lease can be granted for almost any remote service—any one where one object wants to maintain information about another one that is not within the same virtual machine. As with other remote services, there are the added partial failure modes, such as network crash, remote service crash, timeouts, and so on. An object that keeps information on a remote service will hand out a lease to the service and will want the remote service to keep "pinging" it periodically to say that it is alive and that it wants the information kept. Without this periodic assurance, the object might conclude that the remote service has vanished or is somehow unreachable, and that it should discard the information about it.

Leases are a very general mechanism for allowing one service to have confidence in the existence of the other for a limited period. Because they are general, they allow for a great deal of flexibility in use. Because of the possible variety of services, some parts of the Jini lease mechanism cannot be completely defined and must be left as interfaces for applications to fill in. This generality means that all of the details are not filled in for you, as your own requirements cannot be completely predicted in advance.

A lease is given as an interface, and any agent that wishes to grant leases must implement this interface.

```
package net.jini.core.lease;
import java.rmi.RemoteException;
public interface Lease {
    long FOREVER;
    long ANY;
    long getExpiration();
    void cancel() throws UnknownLeaseException, RemoteException;
    void renew(long duration)
        throws LeaseDeniedException, UnknownLeaseException,
               RemoteException;
    void setSerialFormat(int format);
    int getSerialFormat();
    LeaseMap createLeaseMap(long duration);
    boolean canBatch(Lease lease);
}
```

Jini provides three implementations: an `AbstractLease`, and a subclass of this, a `LandlordLease`, which in turn has a subclass `ConstrainableLandlordLease`.

The main issues in implementing a particular lease class lie in setting a policy for handling the initial request for a lease period and in deciding what to do when a renewal request comes in. Some simple possibilities are as follows:

- Always grant the requested time.
- Ignore the requested time and always grant a fixed time.

Of course, there are many more possibilities based on the lessor's expected TTL, system load, and so forth.

There are other issues, though. Any particular lease will need a timeout mechanism. Also, a group of leases can be managed together, and this can reduce the amount of overhead of managing individual leases.

Abstract Lease

An *abstract lease* gives a basic implementation of a lease that can almost be used for simple leases.

```
package com.sun.jini.lease;
public abstract class AbstractLease implements Lease, java.io.Serializable {
    protected AbstractLease(long expiration);
    public long getExpiration();
    public int getSerialFormat();
    public void setSerialFormat(int format);
    public void renew(long duration);
    protected abstract long doRenew(long duration);
}
```

This class supplies straightforward implementations of much of the `Lease` interface, with three provisos:

- The constructor is `protected`, so that constructing a lease with a specified duration is devolved to a subclass. This means that lease duration policy must be set by this subclass.
- The `renew()` method calls into the abstract `doRenew()` method, again to force a subclass to implement a renewal policy.
- The `Lease` interface does not implement the `cancel()` and `createLeaseMap()` methods, so these must also be left to a subclass.

Thus, this class implements the easy things and leaves all matters of policy to concrete subclasses.

Landlord Package

The *landlord* is a package that allows more complex leasing systems to be built. It is not part of the Jini specification, but is supplied as a set of classes and interfaces. The set is not complete in itself—some parts are left as interfaces and need to have class implementations. These will be supplied by a particular application.

A landlord looks after a set of leases. Leases are identified to the landlord by a *cookie*, which is a unique identifier (`Uuid`) for each lease. A landlord does not need to create leases itself; it can use a *landlord lease factory* to do this. (But of course the landlord *can* create leases, depending on how an implementation is done.) When a client wishes to cancel or renew a lease, it asks the lease to perform the cancellation or renewal, and in turn the lease asks its landlord to perform the action. A client is unlikely to ask the landlord directly, as it will only have been given a lease, not a landlord.

The principal classes and interfaces in the `landlord` package are shown in Figure 8-2, where the interfaces are shown in italic font and the classes in normal font.

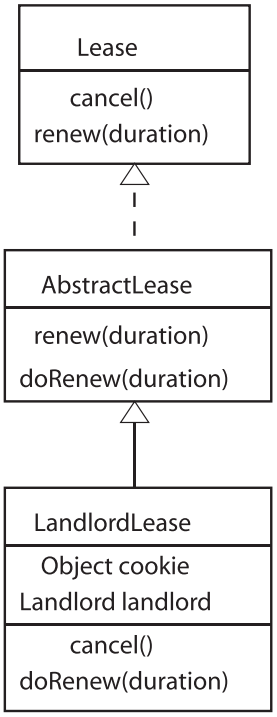

Figure 8-2. *Class diagram in the landlord package*

The interfaces assume that they will be implemented in certain ways, in that each implementation class will contain a reference to certain other interfaces. This doesn't show in the interface specifications, but can be inferred from the method calls.

For example, suppose we wish to develop a lease mechanism for a `Foo` resource. We would create a `FooLandlord` to create and manage leases for `Foo` objects. A minimal structure for this could be as shown in Figure 8-3.

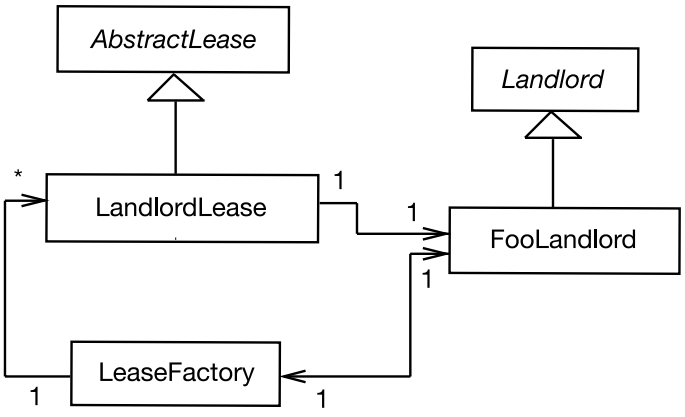

Figure 8-3. *Class diagram of a minimal landlord implementation*

The landlord creates a lease factory and asks it to create leases. Each lease contains a reference to its landlord. When requests are made of the lease, such as `renew()`, these are passed to the landlord to make a decision. However, the `renew()` request to the landlord does *not* pass in the lease, but just its UUID.

Information missing from Figure 8-3 includes how the resource itself is dealt with, where leases end up, and how leasing granting and renewal decisions are made:

- Leases are given to the client that is requesting a lease. Calls such as `renew()` are remote calls to the landlord. The landlord doesn't need a copy of the lease, but does need some representation of it, and that is the purpose of the cookie: it acts as a lessor-side representation of the lease.
- The resource being leased has a representation on the lessor. For example, a lookup service would have the marshalled form of the service proxy to manage. The lessor needs to have this representation plus the lease handle (the cookie) and information such as the lease duration and expiration. This information is given in an implementation of the `LeasedResource` interface.
- Decisions about granting or renewing leases would need to be made using the `LeasedResource`. While these decisions could be made by the landlord, it is cleaner to hand such a task to a separate object concerned with such policy decisions, which is the function of `LeasePeriodPolicy` objects. For example, the `FixedLeasePeriodPolicy` has a simple policy that grants lease times based on a fixed default and maximum lease.

These considerations lead to a more complex class diagram involving the resource and the policy classes, shown in Figure 8-4.

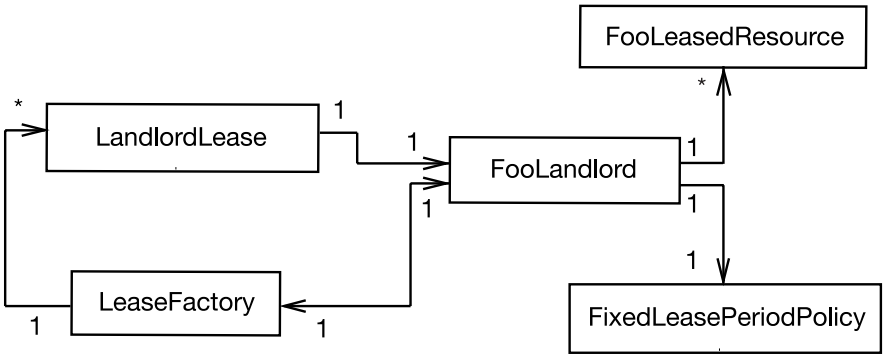

Figure 8-4. *Class diagram in a landlord implementation*

In this context, let's now consider some of these classes in more detail.

LandlordLease Class

The `LandlordLease` class extends `AbstractLease`. This class has the private fields `cookie` and `landlord`, as shown in Figure 8-5.

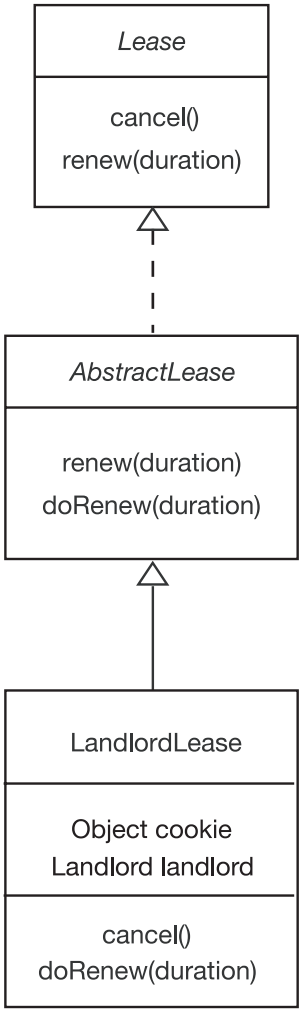

Figure 8-5. *The LandlordLease class diagram*

Implementation of the `cancel()` and `doRenew()` methods in `LandlordLease` is deferred to its landlord.

```
public void cancel() {
    landlord.cancel(cookie);
}
protected long doRenew(long renewDuration) {
    return landlord.renew(cookie, renewDuration);
}
```

The `LandlordLease` class can be used as is, with no subclassing needed. Note that the landlord system produces these leases, but does not actually keep them anywhere; they are passed

on to clients, which then use the lease to call the landlord and hence interact with the landlord lease system. Within the landlord system, the cookie is used as an identifier for the lease.

LeasedResource Interface

A `LeasedResource` is a convenience wrapper around a resource that includes extra information about a lease and methods for use by landlords. It defines an interface as follows:

```
public interface LeasedResource {
    public void setExpiration(long newExpiration);
    public long getExpiration();
    public Uuid getCookie();
}
```

This interface includes the cookie, a unique identifier for a lease within a landlord system, as well as expiration information for the lease. This is all the information maintained about the lease that has been given out to a client.

An implementation of `LeasedResource` will typically include the resource that is leased, plus a method of setting the cookie. The following code shows an example:

```
/**
 * FooLeasedResource.java
 */
package foolandlord;
import com.sun.jini.landlord.LeasedResource;
import net.jini.id.Uuid;
import net.jini.id.UuidFactory;
public class FooLeasedResource implements LeasedResource  {

    protected Uuid cookie;
    protected Foo foo;
    protected long expiration = 0;
    public FooLeasedResource(Foo foo) {
        this.foo = foo;
        cookie = UuidFactory.generate();
    }
    public void setExpiration(long newExpiration) {
        this.expiration = newExpiration;
    }
    public long getExpiration() {
        return expiration;
    }
    public Uuid getCookie() {
        return cookie;
    }
    public Foo getFoo() {
        return foo;
    }
} // FooLeasedResource
```

LeasePeriodPolicy Interface

A lease policy is used when a lease is first granted, and when it tries to renew itself. The time requested may be granted, modified, or denied. A lease policy is specified by the `LeasePeriodPolicy` interface.

```
package com.sun.jini.landlord;
public interface LeasePeriodPolicy {
    LeasePeriodPolicy.Result grant(LeasedResource resource, long requestedDuration);
    LeasePeriodPolicy.Result renew(LeasedResource resource, long requestedDuration);
}
```

An implementation of this policy is given by the `FixedLeasePeriodPolicy`. The constructor takes maximum and default lease values. It uses these to grant and renew leases.

Landlord Interface

The `Landlord` is the final interface in the package that we need for a basic landlord system. Other classes and interfaces, such as `LeaseMap`, are for handling of leases in batches, and will not be dealt with here. The `Landlord` interface is as follows:

```
package com.sun.jini.lease.landlord;
public interface Landlord extends Remote {
    public long renew(Uuid cookie, long extension)
        throws LeaseDeniedException, UnknownLeaseException, RemoteException;
    public void cancel(Uuid cookie)
        throws UnknownLeaseException, RemoteException;
    public RenewResults renewAll(Object[] cookies, long[] durations)
        throws RemoteException;

    public Map cancelAll(Uuid[] cookies)
        throws RemoteException;
}
```

The `renew()` and `cancel()` methods are usually called from the `renew()` and `cancel()` methods of a particular lease. An implementation of `Landlord`, such as `FooLandlord`, will probably have a table of `LeasedResource` objects indexed by the `Uuid`, so that it can work out which resource the request is about.

For any implementation of the `Landlord` interface, the methods `renewAll()` and `cancelAll()` will clearly just loop through the cookies and call `renew()` and `cancel()`, respectively, on each cookie. A convenience class, `LandlordUtil`, has the methods `renewAll()` and `cancelAll()`, which just do that, saving the programmer from having to write the same code for each implementation. This utility class needs to have an object that implements just `renew()` and `cancel()`, and the `LocalLandlord` interface has these two methods. So by making our `FooLandlord` also implement this interface, we can use the utility class to reduce the code we need to write.

The landlord won't make decisions itself about renewals. The `renew()` method needs to use a policy object to ask for renewal. In the `FooManager` implementation, it uses a `FixedLeasePeriodPolicy`.

There must be a method to ask for a new lease for a resource, and this is not specified by the landlord package. This request will probably be made on the lease-granting side, and this should have access to the landlord object, which forms a central point for lease management. So the `FooLandlord` will quite likely have a method such as the following:

```
public Lease newFooLease(Foo foo, long duration);
```

which will give a lease for a resource.

The lease used in the landlord package is a `LandlordLease`. This contains a private field, which is a reference to the landlord itself. The lease is given to a client as a result of `newFooLease()`, and this client will usually be a remote object. Giving the lease to the client will involve serializing the lease and sending it to this remote client. While serializing the lease, the landlord field will also be serialized and sent to the client.

When the client methods such as `renew()` are called, the implementation of the `LandlordLease` will make a call to the landlord, which by then will be remote from its origin. So the landlord object invoked by the lease will need to be a remote object making a remote call. In Jini 1.2, this would have been done by making `FooLandlord` a subclass of `UnicastRemoteObject`. In Jini 2.0, this is preferably done by explicitly exporting the landlord to get a proxy object. The code that follows uses a `BasicJeriExporter` (for simplicity), but it would be better to use a configuration.

Putting all this together for the `FooLandlord` class gives us this:

```
/**
 * FooLandlord.java
 */
package foolandlord;
import net.jini.core.lease.UnknownLeaseException;
import net.jini.core.lease.LeaseDeniedException;
import net.jini.core.lease.Lease;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.tcp.TcpServerEndpoint;
import net.jini.export.*;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Map;
import java.util.HashMap;
import net.jini.id.Uuid;
import com.sun.jini.landlord.Landlord;
import com.sun.jini.landlord.LeaseFactory;
import com.sun.jini.landlord.LeasedResource;
import com.sun.jini.landlord.FixedLeasePeriodPolicy;
import com.sun.jini.landlord.LeasePeriodPolicy;
import com.sun.jini.landlord.LeasePeriodPolicy.Result;
import com.sun.jini.landlord.Landlord.RenewResults;
import com.sun.jini.landlord.LandlordUtil;
import com.sun.jini.landlord.LocalLandlord;
import net.jini.id.UuidFactory;
```

```
public class FooLandlord implements Landlord, LocalLandlord {
    private static final long MAX_LEASE = Lease.FOREVER;
    private static final long DEFAULT_LEASE = 1000*60*5; // 5 minutes
    private Map leasedResourceMap = new HashMap();
    private LeasePeriodPolicy policy = new
        FixedLeasePeriodPolicy(MAX_LEASE, DEFAULT_LEASE);
    private Uuid myUuid = UuidFactory.generate();
    private LeaseFactory factory;
    public FooLandlord() throws java.rmi.RemoteException {
        Exporter exporter = new
            BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                              new BasicILFactory());
        Landlord proxy = (Landlord) exporter.export(this);
        factory = new LeaseFactory(proxy, myUuid);
    }

    public void cancel(Uuid cookie) throws UnknownLeaseException {
        if (leasedResourceMap.remove(cookie) == null) {
            throw new UnknownLeaseException();
        }
    }
    public Map cancelAll(Uuid[] cookies) {
        return LandlordUtil.cancelAll(this, cookies);
    }
    public long renew(Uuid cookie,
                      long extension) throws LeaseDeniedException,
                                             UnknownLeaseException {
        LeasedResource resource = (LeasedResource)
            leasedResourceMap.get(cookie);
        LeasePeriodPolicy.Result result = null;
        if (resource != null) {
            result = policy.renew(resource, extension);
        } else {
            throw new UnknownLeaseException();
        }
        return result.duration;
    }
    public Landlord.RenewResults renewAll(Uuid[] cookies, long[] durations) {
        return LandlordUtil.renewAll(this, cookies, durations);
    }
    public LeasePeriodPolicy.Result grant(LeasedResource resource,
                                          long requestedDuration)
        throws LeaseDeniedException {
        Uuid cookie = resource.getCookie();
        try {
            leasedResourceMap.put(cookie, resource);
        } catch(Exception e) {
```

```
            throw new LeaseDeniedException(e.toString());
        }
        return policy.grant(resource, requestedDuration);
    }
    public Lease newFooLease(Foo foo, long duration)
        throws LeaseDeniedException {
        FooLeasedResource resource = new FooLeasedResource(foo);
        Uuid cookie = resource.getCookie();
        // find out how long we should grant the lease for
        LeasePeriodPolicy.Result result = grant(resource, duration);
        long expiration = result.expiration;
        resource.setExpiration(expiration);
        Lease lease = factory.newLease(cookie, expiration);
        return lease;
    }
    public static void main(String[] args) throws RemoteException,
                                                   LeaseDeniedException,
                                                   UnknownLeaseException {
        // simple test harness

        long DURATION = 2000; // 2 secs;

        FooLandlord landlord = new FooLandlord();
        Lease lease = landlord.newFooLease(new Foo(), DURATION);
        long duration = lease.getExpiration() - System.currentTimeMillis();
        System.out.println("Lease granted for " + duration + " msecs");
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {
            // ignore
        }
        lease.renew(5000);
        duration = lease.getExpiration() - System.currentTimeMillis();
        System.out.println("Lease renewed for " + duration + " msecs");
        lease.cancel();
        System.out.println("Lease cancelled");
    }
} // FooLandlord
```

The Ant file for this is similar to those given before. I only give the parts that are different this time:

```
    <!-- Source files for the server -->
    <property name="src.files"
              value="
                     foolandlord/Foo.java
                     foolandlord/FooLeasedResource.java
```

```
                foolandlord/FooLandlord.java
               "/>
    <!-- Class files to run the server -->
    <property name="class.files"
              value="
                     foolandlord/Foo.class
                     foolandlord/FooLeasedResource.class
                     foolandlord/FooLandlord.class
                    "/>
    <!-- Class files for the client to download -->
    <property name="class.files.dl"
              value="
                    "/>
```

Summary

Leasing allows resources to be managed without complex garbage collection mechanisms. Leases received from services can be dealt with easily using `LeaseRenewalManager`. Entities that need to hand out leases can use a system, such as the landlord system, to handle these leases.