Jini ™ Distributed Leasing Specification

The Distributed Leasing specification defines a set of interfaces and associated conventions and protocols that allow objects in different Java™ virtual machines, perhaps located on different physical machines, to negotiate and establish leases for resources of various kinds. Along with the interfaces and conventions are a set of classes that allow programmers to use the interfaces to construct distributed programs using the leasing model. Leasing is not a stand-alone service or interface, but rather a component in a way of designing interfaces and interactions that are reliable and fault-tolerant in a distributed system.

THE NETWORK IS THE COMPUTER™

901 San Antonio Road
Palo Alto, CA 94303 USA
415 960-1300
fax 415 969-9131

Revision 1.0
January 25, 1999

Contents

Introduction 1

The purpose of the leasing interfaces defined in this document is to simplify and unify a particular style of programming for distributed systems and applications. This style, in which a resource offered by one object in a distributed system and used by a second object in that system, is based on a notion of granting a use to the resource for a certain period of time, negotiated by the two objects when access to the resource is first requested and given. Such a grant is known as a *lease*, and is meant to be similar to the notion of a lease used in everyday life. As in everyday life, the negotiation of a lease entails responsibilities and duties for both the grantor of the lease and the holder of the lease. Part of this specification is a detailing of these responsibilities and duties, as well as a discussion of when it is appropriate to use a lease in offering a distributed service.

There is no requirement that the leasing notions defined in this document be the only time-based mechanism used in software. Leases are a part of the programmer's arsenal, and other time-based techniques such as time-to-live, ping intervals, and keep-alives can be useful in particular situations. Leasing is not meant to replace these other techniques, but rather to enhance the set of tools available to the programmer of distributed systems.

1.1 Leasing and Distributed Systems

Distributed systems differ fundamentally from non-distributed systems in that there are situations where different parts of a cooperating group are unable to communicate, either because one of the members of the group has crashed or because the connection between the members in the group has failed. This partial failure can happen at any time, and can be intermittent or long-lasting.

The possibility of partial failure greatly complicates the construction of distributed systems in which components of the system that are not co-located provide resources or other services to each other. The programming model used most often in non-distributed computing, in which resources and services are granted until explicitly freed or given up, is open to failures caused by the inability to successfully make the explicit calls that cancel the use of the resource or system. Failure of this sort of system can result in resources never being freed, in services being delivered long after the recipient of the service has forgotten that the service was requested, and resource consumption that can grow without bounds.

To avoid these problems, we introduce the notion of a lease. Rather than granting services or resources until that grant has been explicitly cancelled by the party to whom the grant was made, a leased resource or service grant is time based. When the time for the lease has expired, the service ends or the resource is freed. The time period for the lease is determined when the lease is first granted, using a request/response form of negotiation between the party wanting the lease and the lease grantor. Leases may be renewed or cancelled before they expire by the holder of the lease, but in the case of no action (or in the case of a network or participant failure) the lease simply expires. When a lease expires, both the holder of the lease and the grantor of the lease know that the service or resource has been reclaimed.

While the notion of a lease was originally brought into the system as a way of dealing with partial failure, the technique is also useful for dealing with another problem faced by distributed systems. Distributed systems tend to be long-lived. In addition, since distributed systems are often providing resources that are shared by numerous clients in an uncoordinated fashion, such systems are much more difficult to “shut down” for maintenance purposes than systems which reside on a single machine.

As a consequence of this, distributed systems, especially those with persistent state, are prone to accumulations of outdated and unwanted information. While the accumulation of such information, which can include objects stored

for future use and subsequently forgotten, may be slow, the trend is always upward. Over the (comparatively) long life of a distributed system, such unwanted information can grow without upper bound, taking up resources and compromising the performance of the overall system.

A standard way of dealing with these problems is to consider the cleanup of unused resources to be system administration task. When such resources begin to get scarce, a human administrator is given the task of finding those resources that are no longer needed and deleting them. This solution, however, is error prone (since the administrator is often required to judge the use of a resource with no actual evidence concerning whether or not the resource is being used) and tends to happen only when resource consumption has gotten out of hand.

When such resources are leased, however, this accumulation of out-of-date information does not occur and resorting to manual cleanup methods is not needed. Information or resources that are leased remain in the system only as long as the lease for that information or resource is renewed. Thus information that is forgotten (either through program error, inadvertence, or system crash) will be deleted after some finite time. Note that this is not the same as garbage collection (although it is related in that it has to do with freeing up resources) since the information that is leased is not of the sort that would generally have any active reference to it. Rather, this is information that is stored for (possible) later retrieval which is no longer of any interest to the party that originally stored the information.

This model of persistence is one that requires renewed proof of interest to maintain the persistence. Information is kept (and resources used) only as long as that information is claimed to be of interest by someone (a claim that is shown by the act of renewing the lease). The interval for which the resource may be consumed without a proof of interest can vary, and is subject to negotiation by the party storing the information (which has expectations for how long it will be interested in the information) and the party in which the information is stored (which has requirements on how long it is willing to store something without proof that some party is interested).

The notion of persistence of information is not one of storing the information on stable storage (although it encompasses that notion). Persistent information, in this case, includes any information that has a lifetime longer than the lifetime of the process in which the request for storage originates.

Leasing also allows a form of programming in which the entity that reserves the information or resource is not the same as the entity that makes use of the information or resource. On such a model, a resource can be reserved (leased) by an entity on the expectation that some other entity will use the resource over some period of time. Rather than having to check back to see if the resource is used (or freed), a leased version of such a reservation allows the entity granted the lease to forget about the resource. Whether used or not, the resource will be freed when the lease has expired.

Leasing such information storage introduces a programming paradigm that is an extension of the model used by most programmers today. The current model is essentially one of infinite leasing, with information only being removed from persistent stores by the active deletion of such information. Databases and filesystems are perhaps the best known exemplars of such stores, with both holding any information placed in them until the information is explicitly deleted by some user or program.

1.2 Goals and Requirements

The requirements of this set of interfaces are

- to provide a simple way of indicating time-based resource allocation or reservation;
- to provide a uniform way of renewing and cancelling leases; and
- to show common patterns of use for interfaces using this set of interfaces.

The goals of this document are

- to describe the notion of a lease, and show some of the applications of that notion in distributed computing;
- to show the way in which this notion is used in a distributed system; and
- to indicate appropriate uses of the notion in applications built to run in a distributed environment.

1.3 Dependencies

This document relies on the following specifications:

- Java Remote Method Invocation Specification

1.4 Comments

Please direct comments to `jini-comments@java.sun.com`.

Basic Leasing Interfaces 2

The basic concept of leasing is that access to a resource or the request for some action is not open-ended with respect to time, but only granted for some particular interval. In general (although not always) this interval is determined by some negotiation between the object asking for the leased resource (which we will call the lease holder) and the object granting access for some period (which we will call the lease grantor).

In its most general form, a lease is used to associate a mutually agreed upon time interval with an agreement reached by two objects. The kinds of agreements that can be leased are varied, and can include such things as agreements on access to an object (references), agreements for taking future action (event notifications), agreements to supplying persistent storage (file systems, JavaSpaces systems), or agreements to advertise availability (naming or directory services).

While it is possible that a lease can be given which provides exclusive access to some resource, this is not required with the notion of leasing being offered here. Agreements that provide access to resources that are intrinsically sharable can have multiple concurrent leaseholders. Other resources may decide to only grant exclusive leases, combining the notion of leasing with a concurrency control mechanism.

2.1 Characteristics of a Lease

There are a number of characteristics of a lease that are important for understanding what a lease is and when it is appropriate to use a lease. Among these characteristics are

- A lease is a time period during which the grantor of the lease insures (to the best of the grantor's abilities) that the holder of the lease will have access to some resource. The time period of the lease can be determined solely by the lease grantor, or can be a period of time that is negotiated between the holder of the lease and the grantor of the lease. Duration negotiation need not be multi-round; it often suffices for the requestor to indicate the time desired and the grantor to return the actual time of grant.
- During the period of a lease, a lease can be cancelled by the entity holding the lease. Such a cancellation allows the grantor of the lease to clean up any resources associated with the lease, and obliges the grantor of the lease to not take any action involving the leaseholder that was part of the agreement that was the subject of the lease.
- A leaseholder can request that a lease be renewed. The renewal period can be for a different time than the original lease, and is also subject to negotiation with the grantor of the lease. The grantor may renew the lease for the requested period or a shorter period, or refuse to renew the lease at all. A renewed lease is just like any other lease, and is itself subject to renewal.
- A lease can expire. If a lease period has elapsed with no renewals, the lease expires and any resources associated with the lease may be freed by the lease grantor. Both the grantor and the holder are obliged to act as though the leased agreement is no longer in force. The expiration of a lease is similar to the cancellation of a lease, except that no communication is necessary between the leaseholder and the lease grantor.

Leasing is part of a programming model for building reliable distributed applications. In particular, leasing is a way of insuring that a uniform response to failure, forgetting, or disinterest is guaranteed; allowing agreements to be made that can then be forgotten without the possibility of unbounded resource consumption; and providing a flexible mechanism for duration based agreement.

2.2 Basic Operations

The Lease interface defines a type of object that is returned to the lease holder and issued by the lease grantor. The basic interface may be extended in ways that offer more functionality, but the basic interface is

```
package net.jini.core.lease;
import java.rmi.RemoteException;


public interface Lease {
    long FOREVER = Long.MAX_VALUE;
    long ANY = -1;

    int DURATION = 1;
    int ABSOLUTE = 2;


    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);
}
```

Particular instances of the Lease type will be created by the grantors of a lease, and returned to the holder of the lease as part of the return value from a call that allocates a leased resource. The actual implementation of the object, including the way (if any) in which the Lease object communicates with the grantor of the lease, is determined by the lease grantor and is hidden from the lease holder.

The interface defines two constants that can be used when requesting a lease. The first, FOREVER, can be used to request a lease that never expires. When granted such a lease, the leaseholder is responsible for insuring that the resource leased is freed when no longer needed. The second constant, ANY, is

used by the requestor to indicate that there is no particular lease time desired, and that the grantor of the lease should supply a time that is most convenient for the grantor.

If the request is for a particular duration, the lease grantor is required to grant a lease of no more than the requested period of time. A lease may be granted for a period of time shorter than that requested.

A second pair of constants is used to determine the format used in the serialized form for a `Lease` object; in particular, the serialized form used to represent the time at which the lease expires. If the serialized format is set to the value `DURATION`, the serialized form will convert the time of lease expiration into a duration (in milliseconds) from the time of serialization. This form is best used when transmitting a `Lease` object from one address space to another (via an RMI call) where it cannot be assumed that the address spaces have synchronized clocks. If the serialized format is set to `ABSOLUTE`, the time of expiration will be stored as an absolute time, calculated in terms of milliseconds since the beginning of the epoch.

The first method in the `Lease` interface, `getExpiration()`, returns a `long` that indicates the time, relative to the current clock, that the lease will expire. Following the usual convention in the Java™ programming language, this time is represented as milliseconds from the beginning of the epoch, and can be used to compare the expiration time of the lease with the result of a call to obtain the current time, `java.lang.System.currentTimeMillis()`.

The second method, `cancel()`, can be used by the leaseholder to indicate that it is no longer interested in the resource or information held by the lease. If the leased information or resource could cause a callback to the lease holder (or some other object on behalf of the lease holder), the lease grantor should not issue such a callback after the lease has been cancelled. The overall effect of a `cancel()` call is the same as lease expiration, but instead of happening at the end of a pre-agreed duration it happens immediately. If the lease being cancelled is unknown to the lease grantor, an `UnknownLeaseException` is thrown. The method can also throw a `RemoteException` if the implementation of the method requires calling a remote object which is the lease holder.

The third method, `renew()`, is used to renew a lease for an additional period of time. The length of the desired renewal is given, in milliseconds, in the parameter to the call. This duration is not added to the original lease, but is used to determine a new expiration time for the existing lease. This method

has no return value; if the renewal is granted, this is reflected in the lease object on which the call was made. If the lease grantor is unable or unwilling to renew the lease, a `RenewFailedException` is thrown. If a renewal fails, the lease is left intact for the same duration that was in force prior to the call to `renew()`. If the lease being renewed is unknown to the lease grantor, an `UnknownLeaseException` is thrown. The method can also throw a `RemoteException` if the implementation of the method requires calling a remote object which is the lease holder.

Two methods are concerned with the serialized format of a Lease object. The first, `setSerialFormat()`, takes an integer that indicates the appropriate format to use when serializing the format. The current supported formats are a duration format that stores the length of time (from the time of serialization) before the lease expires; and an absolute format, which stores the time (relative to the current clock) that the lease will expire. The durational format should be used when serializing a `Lease` object for transmission from one machine to another; the durational format should be used when storing a `Lease` object on stable store that will be read back later by the same process or machine. The default serialization format is durational. The second method, `getSerialForm()`, returns an integer indicating the format that will be used to serialize the `Lease` object.

The last two methods are used to aid in the batch renewal or cancellation of a group of `Lease` objects. The first of these, `createLeaseMap(long duration)` creates a `Map` object that can contain leases whose renewal or cancellation can be batched, and adds the current lease to that map. The current lease will be renewed for the duration indicated by the argument to the method when all of the leases in the `LeaseMap` are renewed. The second method, `batchWith(Lease lease)`, returns a boolean value indicating whether or not the lease given as an argument to the method can be batched (in `renew()` and `cancel()` calls) with the current lease. Whether or not two `Lease` objects can be batched is an implementation detail determined by the objects.

There are three types of `Exception` objects associated with the basic lease interface. All of these are used in the `Lease` interface itself, and two can be used by methods that grant access to a leased resource.

The `RemoteException` is imported from the package `java.rmi`. This exception is used to indicate a problem with any communication that might occur between the lease holder and the lease grantor if those objects are in separate virtual machines. The full specification of this exception can be found in the *Java™ Remote Method Invocation (RMI)* specification.

The `UnknownLeaseException` is used to indicate that the Lease object called is not known to the grantor of the lease. This can occur when a lease expires, or when a copy of a lease has been cancelled by some other leaseholder. This exception is defined as

```
package net.jini.core.lease;

public class UnknownLeaseException extends LeaseException
{
    public UnknownLeaseException(){
        super();
    }
    public UnknownLeaseException(String reason){
        super(reason);
    }
}
```

The final exception defined is the `LeaseDeniedException`, which can be thrown either by a call to `renew()` or a call to an interface that grants access to a leased resource. This exception indicates that the requested lease has been denied by the resource holder. The exception is defined as

```
package net.jini.core.lease;

public class LeaseDeniedException extends LeaseException
{
    public LeaseDeniedException(){
        super();
    }
    public LeaseDeniedException(String reason){
        super(reason);
    }
}
```

The `LeaseException` superclass is defined as

```
package net.jini.core.lease;

public class LeaseException extends Exception
{
    public LeaseException(){
```

```
        super();
    }
    public LeaseException(String reason){
        super(reason);
    }
}
```

The final basic interface defined for leasing is that of a `LeaseMap`, which allows groups of `Lease` objects to be renewed or cancelled using a single operation. The `LeaseMap` interface is

```
package net.jini.core.lease;
import java.rmi.RemoteException;
public interface LeaseMap extends java.util.Map {
    boolean canContainKey(Object key);
    void renewAll() throws LeaseMapException, RemoteException;
    void cancelAll() throws LeaseMapException, RemoteException;
}
```

A `LeaseMap` is an extension of the `java.util.Map` class that associates a `Lease` object with a `Long`. The `Long` is the duration for which the lease should be renewed whenever it is renewed. `Lease` objects and associated renewal durations can be entered and removed from a `LeaseMap` using the usual `Map` methods. An attempt to add a `Lease` object to a map containing other `Lease` objects for which `Lease.canBatch()` would return false will cause an `IllegalArgumentException` to be thrown, as will attempts to add a key that is not a `Lease` object or a value which is not a `Long`.

The first method defined in the `LeaseMap` interface, `canContainKey()`, takes a `Lease` object as an argument and returns `true` if that `Lease` object can be added to the `Map` and `false` otherwise. A `Lease` object can be added to a `Map` if that `Lease` object can be renewed in a batch with the other objects in the `LeaseMap`. The requirements for this are dependent on the implementation of the `Lease` object.

The second method, `renewAll()`, will attempt to renew all of the `Lease` objects in the `LeaseMap` for the duration associated with the `Lease` object. If all of the `Lease` objects are successfully renewed, the method will return

nothing. If some `Lease` objects fail to renew, those objects will be removed from the `LeaseMap` and will be contained in the thrown `LeaseMapException`.

The second method, `cancelAll()`, cancels all the `Lease` objects in the `LeaseMap`. If all cancels are successful, the method returns normally and leaves all leases in the map. If any of the `Lease` objects cannot be cancelled, they are removed from the `LeaseMap` and the operation throws a `LeaseMapException`.

The `LeaseMapException` class is defined as

```
package net.jini.core.lease;

import java.util.Map;

public class LeaseMapException extends LeaseException {

    public Map exceptionMap;

    public LeaseMapException(String s, Map exceptionMap) {
        super(s);
        this.exceptionMap = exceptionMap;
    }
}
```

Objects of type `LeaseMapException` contain a `Map` object which maps `Lease` objects (the keys) to `Exception` objects (the values). The `Lease` objects are the ones which could not be renewed or cancelled, and the `Exception` objects reflect the individual failures. For example, if a `LeaseMap.renew()` call fails because one of the leases has already expired, that lease would be taken out of the original `LeaseMap` and placed in the `Map` returned as part of the `LeaseMapException` object with an `UnknownLeaseException` object as the corresponding value.

2.3 Leasing and Time

The duration of a lease is determined when the lease is granted (or renewed). A lease is granted for a duration, rather than until some particular moment of time, since such a grant does not require that the clocks used by the client and the server be synchronized.

The difficulty of synchronizing clocks in a distributed system is well known. The problem is somewhat more tractable in the case of leases, which are expected to be for periods of minutes to months, as the accuracy of

synchronization required is expected to be in terms of minutes rather than nanoseconds. Over a particular local group of machines, a time service could be used that would allow this level of synchronization.

However, leasing is expected to be used by clients and servers who are widely distributed and might not share a particular time service. In such a case, clock drift of many minutes is a common occurrence. Because of this, the leasing specification has chosen to use durations rather than absolute time.

The reasoning behind such a choice is based on the observation that the accuracy of the clocks used in the machines that make up a distributed system is matched much more closely than the clocks on those systems. While there may be minutes of difference in the notion of the absolute time had by widely separated systems, there is much less likelihood of a significant difference over the rate of change of time in those systems. While there is clearly some difference in the notion of duration between systems (if there were not, synchronization for absolute time would be much easier), that difference is not cumulative in the way errors in absolute time are.

This decision does mean that holders of leases and grantors of leases need to be aware of some of the consequences of the use of durations. In particular, the amount of time needed to communicate between the lease holder and the lease grantor, which may vary from call to call, needs to be taken into account when renewing a lease. If a lease holder is calculating the absolute time (relative to the lease holder's clock) at which to ask for a renewal, that time should be based on the sum of the duration of the lease and the time at which the lease holder requested the lease, not on the duration and the time that the lease holder received the lease.

2.4 Serialized Forms

The `serialVersionUID` of `LeaseException` is -7902272546257490469. There are no serialized fields.

The `serialVersionUID` of `UnknownLeaseException` is -2921099330511429288. There are no serialized fields.

The `serialVersionUID` of `LeaseDeniedException` is 5704943735577343495. There are no serialized fields.

The `serialVersionUID` of `LeaseMapException` is -4854893779678486122. The single serialized field is the declared public field.

Example Supporting Classes 3

The basic `Lease` interface defined in the previous chapter allows leases to be granted by one object and handed to another as the result of a call that creates or provides access to some leased resource. The goal of the interface is to allow as much freedom as possible in implementation, both on the part of the party that is granting the lease (and thus is giving out the implementation that supports the Lease interface) and the party that receives the lease.

However, there are a number of classes that can be supplied that can simplify the handling of leases in some common cases. In this section, we will describe examples of these supporting classes, and show how these classes can be used with leased resources.

3.1 A Renewal Class

One of the common patterns with leasing is for the lease holder to request a lease with the intention of renewing the lease until finished with the resource. The period of time during which the resource is needed is unknown at the time of requesting the lease, so the requestor desires that the lease be renewed until an undetermined time in the future. Alternatively, the lease requestor may know how long the lease needs to be held, but the lease holder is unwilling to grant a lease for the full period of time. Again, the pattern will be to renew the lease for some period of time.

If the lease continues to be renewed, the lease holder doesn't want to be bothered with knowing about it, but if the lease is not renewed for some reason the leaseholder wants to be notified. Such a notification can be done using the usual inter-address space mechanisms for event notifications, by registering a listener of the appropriate type.

This functionality can be supplied by a class with an interface like the following

```
class LeaseRenew
{
    LeaseRenew(Lease toRenew,
               long renewTil,
               LeaseExpireListener listener);
    void addRenew(Lease toRenew,
                  long renewTil,
                  LeaseExpireListener listener);
    long getExpiration(Lease forLease)
        throws UnknownLeaseException;
    void setExpiration(Lease forLease,long toExpire)
        throws UnknownLeaseException;
    void cancel(Lease toCancel)
        throws UnknownLeaseException;
    void setLeaseExpireListener(Lease forLease,
                                LeaseExpireListener listener)
        throws UnknownLeaseException;
    void removeLeaseExpireListener(Lease forLease)
        throws UnknownLeaseException;
}
```

The constructor of this class takes a `Lease` object, presumably returned from some call that reserved a leased resource; an initial time indicating when the lease should be renewed until, and an object that is to be notified if a renewal fails before the time indicated in `renewTil`. This returns a `LeaseRenew` object, which will have it's own thread of control that will do the lease renewals.

Once a `LeaseRenew` object has been created, other leases can be added to the set that are renewed by that object using the `addRenew()` call. This call takes a Lease object, an expiration time or overall duration, and a listener to be informed if the lease cannot be renewed prior to the time requested. Internally to the `LeaseRenew` object, leases that can be batched can be placed into a `LeaseMap`.

The duration of a particular lease can be queried by a call to the method `getExpiration()`. This method takes a `Lease` object and returns the time at which that lease will be allowed to expire by the `LeaseRenew` object. Note that this is different from the `Lease.getExpiration()` method, which tells the time at which the lease will expire if not renewed. If there is no `Lease` object corresponding to the argument for this call being handled by the `LeaseRenew` object, a `UnknownLeaseException` will be thrown. This can happen either when no such `Lease` has ever been given to the `LeaseRenew` object, or when a `Lease` object that has been held has already expired or been cancelled. Notice that since this object is assumed to be in the same address space as the object that acquired the lease, we can assume that it shares the same clock with that object, and hence can use absolute time rather than a duration based system.

The `setExpiration()` method allows the caller to adjust the expiration time of any `Lease` object held by the `LeaseRenew` object. This method takes as arguments the `Lease` whose time of expiration is to be adjusted and the new expiration time. If no lease is held by the `LeaseRenew` object corresponding to the first argument, an `UnknownLeaseException` will be thrown.

A call to `cancel()` will result in the cancellation of the indicated `Lease` held by the `LeaseRenew` object. Again, if the lease has already expired on that object, an `UnknownLeaseException` will be thrown. It is expected that a call to this method will be made if the leased resource is no longer needed, rather than just dropping all references to the `LeaseRenew` object.

The final two methods, `setLeaseExpireListener()` and `removeLeaseExpireListener()`, allow setting and unsetting the destination of an event handler associated with a particular `Lease` object held by the `LeaseRenew` object. The handler will be called if the `Lease` object expires before the desired duration period has completed. Note that one of the properties of the example given here is that only one `LeaseExpireListener` can be associated with each `Lease`.

3.2 A Renewal Service

Objects that hold a lease that needs to be renewed may themselves be activatable, and thus unable to insure that they will be capable of renewing a lease at some particular time in the future (since they may not be active at that

time). For such objects, it might make sense to hand the lease renewal duty off to a service that could take care of lease renewal for the object, allowing that object to be deactivated without fear of losing its lease on some other resource.

The most straightforward way of accomplishing this is to hand the `Lease` object off to some object whose job it is to renew leases on behalf of others. This object will be remote to the objects to which it offers its service (otherwise it would be inactive when the others become inactive) but might be local to the machine; there could even be such services that are located on other machines.

The interface to such an object might look something like

```
interface LeaseRenewService extends Remote
{
    EventRegistration renew(Lease toRenew,
                            long renewTil,
                            RemoteEventListenter notifyBeforeDrop,
                            MarshalledObject returnOnNotify)
            throws RemoteException;
    void onRenewFailure(Lease toRenew,
                          RemoteEventListenter toNotify,
                          MarshalledObject returnOnNotify)
            throws RemoteException, UnknownLeaseException;
}
```

The first method, `renew()`, is the request to the object to renew a particular lease on behalf of the caller. The `Lease` object to be renewed is passed to the `LeaseRenewService` object, along with the length of time for which the lease is to be renewed. Since we are assuming that this service may not be on the same machine as the object that acquired the original lease, we return to a duration-based time system, since we cannot assume that the two systems have synchronized clocks.

Requests to renew a `Lease` are themselves leased. The duration of the lease is requested in the duration argument to the `renew()` method, and the actual time of the lease is returned as part of the `EventRegistration` return value. While it may seem odd to lease the service of renewing other leases, this does not cause an infinite regress. It is assumed that the `LeaseRenewService` will grant leases that are longer (perhaps significantly longer) than those in the leases that it is renewing. In this fashion, the `LeaseRenewService` can act as a concentrator for lease renewal messages.

The `renew()` method also takes a Remote`EventListener` and a `MarshalledObject` to be passed to that `RemoteEventListener`. This is because part of the semantics of the `renew()` call is to register interest in an event that can occur within the `LeaseRenewService` object. The registration is actually for a notification before the lease granted by the renewal service is dropped. This event notification can be directed back to the object that is the client of the renewal service, and will (if so directed) cause the object to be activated (if it is not already active). This gives the object a chance to renew the lease with the `LeaseRenewService` object before that lease is dropped.

The second method, `onRenewFailure()`, allows the client to register interest in the `LeaseRenewService` being unable to renew the `Lease` supplied as an argument to the call. This call also takes an `RemoteEventListener` object which is the target of the notification, and a `MarshalledObject` that will be passed as part of the notification. This allows the client to be informed if the `LeaseRenewService` is denied a lease renewal during the lease period offered to the client for such renewal. This call does not take a time period for the event registration, but instead will have the same duration as the leased renewal associated with the `Lease` object passed into the call, which should be the same as the Lease object supplied in a previous call to `renew()`. If the `Lease` is not one that is known to the `LeaseRenewService` object, an `UnknownLeaseException` will be thrown.

There is no need for a method allowing the cancellation of a lease renewal request. Since these requests are themselves leased, cancelling the lease with the `LeaseRenewService` will cancel both the renewing of the lease and any event registrations associated with that lease.