After acquiring references to the targeted lookup services, the lookup discovery service would pass those references to the entity, providing the entity with access to the services registered with each lookup service. In this way, the entity participates in the multicast discovery protocols through a proxy relationship with the lookup discovery service, gaining access not only to lookup services outside of its own range, but also to all of the services registered with those lookup services.

Note that the scenario just described does not come without restrictions. For the lookup discovery service to be able to "link" an entity with lookup services in the way just described, the lookup discovery service must be registered with a lookup service having a location that either is known to the entity or is within the multicast radius of the entity. Furthermore, the lookup discovery service must be running on a host that is located within the multicast radius of the lookup services with which the entity wishes to be linked. That is, the entity must be able to find the lookup discovery service, and the lookup discovery service must be able to find the other desired lookup services.

To address these scenarios, the lookup discovery service participates in both the multicast discovery protocols and the unicast discovery protocol on behalf of a registered discovering entity or *client*. This service will listen for and process multicast announcement packets from Jini lookup services and will, until successful, repeatedly attempt to discover specific lookup services that the client is interested in finding.

Upon discovery of a previously undiscovered lookup service of interest, the lookup discovery service notifies all entities that have requested the discovery of that lookup service that such an event has occurred. The event mechanism employed by the lookup discovery service satisfies the requirements defined in *The Jini Technology Core Platform Specification, "Distributed Events"*. Note that the entity that receives such an event notification does not have to be the client of the lookup discovery service; it may be a third-party event-handling service such as an event mailbox service. Once a client is notified of the discovery of a lookup service, it is left to the client to define the semantics of how it interacts with that lookup service. For example, the client may wish to join the lookup service, simply query it for other useful services, or both.

The lookup discovery service must be implemented as a well-behaved Jini service and must comply with all of the policies embodied in the Jini technology programming model. Thus, the resources granted by this service are leased, and implementations of this service must adhere to the distributed leasing model for Jini technology as defined in *The Jini Technology Core Platform Specification, "Distributed Leasing"*. That is, the lookup discovery service will grant its services for only a limited period of time without an active expression of continuing interest on the part of the client.

LD.1.1 Goals and Requirements

The requirements of the interfaces and classes specified in this document are:

- To define a service that not only employs the Jini discovery protocols to discover, by way of either group association or `LookupLocator` association, lookup services in which clients have registered interest, but that also notifies its clients of the discovery of those lookup services
- To provide this service in such a way that it can be used by entities that deactivate
- To comply with the policies of the Jini technology programming model

The goals of this document are as follows:

- To describe the lookup discovery service
- To provide guidance in the use and deployment of services that implement the `LookupDiscoveryService` interface and related classes and interfaces

LD.1.2 Other Types

The types defined in the specification of the `LookupDiscoveryService` interface are in the `net.jini.discovery` package. The following object types may be referenced in this chapter. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.discovery.LookupLocator
net.jini.core.event.EventRegistration
net.jini.core.event.RemoteEventListener
net.jini.core.lease.Lease
net.jini.core.lookup.ServiceID
net.jini.core.lookup.ServiceRegistrar
net.jini.discovery.DiscoveryEvent
net.jini.discovery.DiscoveryGroupManagement
net.jini.discovery.DiscoveryListener
java.io.IOException
java.rmi.MarshalledObject
java.rmi.NoSuchObjectException
java.rmi.RemoteException
java.util.Map
```

LD.2 The Interface

THE LookupDiscoveryService interface defines the service—referred to as the *lookup discovery service*—previously introduced in this specification. Through this interface, other Jini services and clients may request that discovery processing be performed on their behalf. This interface belongs to the net.jini.discovery package, and any service implementing this interface must comply with the definition of a Jini service. This interface is not a remote interface; each implementation of this service exports a front-end proxy object that implements this interface local to the client, using an implementation-specific protocol to communicate with the actual remote server (the back end). All of the proxy methods must obey normal Java Remote Method Invocation (RMI) remote interface semantics except where explicitly noted. Two proxy objects are equal (using the equals method) if they are proxies for the same lookup discovery service.

The one method defined in this interface throws a RemoteException, and requires only the default serialization semantics so that this interface can be implemented directly using Java RMI.

```
package net.jini.discovery;

public interface LookupDiscoveryService {
    public LookupDiscoveryRegistration register(
                                String[] groups,
                                LookupLocator[] locators,
                                RemoteEventListener listener,
                                MarshalledObject handback,
                                long leaseDuration)
                                        throws RemoteException;
}
```

When requesting a registration with the lookup discovery service, the client indicates the lookup services it is interested in discovering by submitting two sets of objects. Each set may contain zero or more elements. One set consists of the names of the groups whose members are lookup services the client wishes to be

discovered. The other set consists of `LookupLocator` objects, each corresponding to a specific lookup service the client wishes to be discovered.

For each successful registration the lookup discovery service will manage both the set of group names and the set of locators submitted. These sets will be referred to as the *managed set of groups* and the *managed set of locators*, respectively. The managed set of groups associated with a particular registration contains the names of the groups whose members consist of lookup services that the client wishes to be discovered through *multicast discovery*. Similarly, the managed set of locators contains instances of `LookupLocator`, each corresponding to a specific lookup service that the client wishes to be discovered through *unicast discovery*. The references to the lookup services that have been discovered will be maintained in a set referred to as the *managed set of lookup services* (or managed set of *registrars*).

Note that when the general term *managed set* is used, it should be clear from the context whether groups, locators, or registrars are being discussed. Furthermore, when the term *group discovery* or *locator discovery* is used, it should be taken to mean, respectively, the employment of either the multicast discovery protocols or the unicast discovery protocol to discover lookup services that correspond to members of the appropriate managed set.

LD.3 The Semantics

To employ the lookup discovery service to perform discovery on its behalf, a client must first register with the lookup discovery service by invoking the `register` method defined in the `LookupDiscoveryService` interface. The `register` method is the only method specified by this interface.

LD.3.1 Registration Semantics

An invocation of the `register` method produces an object—referred to as a *registration object* (or simply a *registration*)—that is mutable. That is, the registration object contains methods through which it may be changed. Because registrations are mutable, each invocation of the `register` method produces a new registration object. Thus, the `register` method is not idempotent.

The `register` method may throw a `RemoteException`. Typically, this exception occurs when there is a communication failure between the client and the lookup discovery service. When this exception does occur, the registration may or may not have been successful.

Each registration with the lookup discovery service is persistent across restarts (or crashes) of the lookup discovery service until the lease on the registration expires or is cancelled.

The `register` method takes the following as arguments:

- A `String` array, none of whose elements may be `null`, consisting of zero or more elements in which each element is the name of a group whose members are lookup services that the client requesting the registration wishes to be discovered via group discovery
- An array of `LookupLocator` objects, none of whose elements may be `null`, consisting of zero or more elements in which each element corresponds to a specific lookup service that the client requesting the registration wishes to be discovered via locator discovery

- A non-null RemoteEventListener object which specifies the entity that will receive events notifying the registration when a lookup service of interest is discovered or discarded
- Either null or an instance of MarshalledObject specifying an object that will be included in the notification event that the lookup discovery service sends to the registered listener
- A long value representing the amount of time (in milliseconds) for which the resources of the lookup discovery service are being requested

The register method returns an object that implements the LookupDiscoveryRegistration interface. It is through this returned object that the client interacts with the lookup discovery service. This interaction includes activities such as group and locator management, state retrieval, and discarding discovered but unavailable lookup services so that they are eligible for rediscovery (see Section LD.4.1, "The LookupDiscoveryRegistration Interface" for definition of the semantics of the methods of the LookupDiscoveryRegistration interface).

The groups argument takes a String array, none of whose elements may be null. Although it is acceptable to specify null (which is equivalent to DiscoveryGroupManagement.ALL_GROUPS) for the groups argument itself, if the argument contains one or more null elements, a NullPointerException is thrown. If the value is null, the lookup discovery service will attempt to discover all lookup services located within the multicast radius of the host on which the lookup discovery service is running. If an empty array (equivalent to DiscoveryGroupManagement.NO_GROUPS) is passed in, then no group discovery will be performed for the associated registration until the client, through the registration's setGroups or addGroups method, changes the contents of the managed set of groups to either a non-empty set of group names or null.

The locators argument takes an array of LookupLocator objects, none of whose elements may be null. If either the empty array or null is passed in as the locators argument, then no locator discovery will be performed for the associated registration until the client, through the registration's addLocators or setLocators method, changes the managed set of locators to a non-empty set of locators. Although it is acceptable to input null for the locators argument itself, if the argument contains one or more null elements, a NullPointerException is thrown.

If the register method is invoked with a set of group names and a set of locators in which either or both sets contain duplicate elements (where duplicate locators are determined by LookupLocator.equals), the invocation is equivalent to constructing this class with no duplicates in either set.

Upon discovery of a lookup service, through either group discovery or locator discovery, the lookup discovery service will send an event, referred to as a *discovered event*, to the listener associated with the registration produced by the call to `register`.

After initial discovery of a lookup service, the lookup discovery service will continue to monitor the group membership state reflected in the multicast announcements from that lookup service. Depending on the lookup service's current group membership, the lookup discovery service may send either a discovered event or an event referred to as a *discarded event*. The conditions under which either a discovered event or a discarded event will be sent are as follows:

- If the multicast announcements from an already discovered lookup service indicate that the lookup service is a member of a new group, a discovered event will be sent to the listener of each registration that has yet to receive a discovered event for that lookup service, but that has previously registered interest in the new group.
- If the multicast announcements from an already discovered lookup service indicate that the lookup service has changed its group membership in such a way that the lookup service is no longer of interest to one or more of the registrations that previously registered interest in the groups of that lookup service, a discarded event will be sent to the listener of each such registration. This type of discarded event is sometimes referred to as a *passive no-interest discarded event* ("passive" because the lookup discovery service, rather than the client, initiated the discard process).
- If the multicast announcements from an already discovered lookup service are no longer being received, a discarded event will be sent to the listener of each registration that previously registered interest in one or more of that lookup service's member groups. This type of discarded event is sometimes referred to as a *passive communication discarded event*.

It is important to note that when the lookup discovery service (passively) discards a lookup service, due to group membership changes (lost interest) or unavailability (communication failure), the discarded event will be sent to only the listeners of those registrations that have previously requested that the affected lookup service be discovered through at least group discovery. That is, the listener of any registration that is interested in the affected lookup service through *only* locator discovery will not be sent either type of passive discarded event. This is because the semantics of the lookup discovery service assume that since the client, through the registration request, expressed no interest in discovering the

lookup service through its group membership, the client must also have no interest in any group-related changes in that lookup service's state.

A more detailed discussion of the event semantics of the lookup discovery service is presented in Section LD.3.2, "Event Semantics".

A valid parameter must be passed as the `listener` argument of the `register` method. If a `null` value is input to this argument, then a `NullPointerException` will be thrown and the registration fails.

Note that if an indefinite exception occurs while attempting to send a discovered or discarded event to a registration's listener, the lookup discovery service will continue to attempt to send the event until either the event is successfully delivered or the client's lease on that registration expires. If an `UnknownEventException`, a bad object exception, or a bad invocation exception occurs while attempting to send a discovered or discarded event to a registration's listener, the lookup discovery service assumes that the client is in an unknown, possibly corrupt state, and will cancel the lease on the registration and clear the registration from its managed set.

The state information maintained by the lookup discovery service includes the set of group names, locators, and listeners submitted by each client through each invocation of the `register` method, with duplicates eliminated. This state information contains no knowledge of the clients that register with the lookup discovery service. Thus, there is no requirement that a client identify itself during the registration process.

LD.3.2 Event Semantics

For each registration created by the lookup discovery service, an event identifier will be generated that uniquely maps the registration to the listener as well as to the registration's managed set of groups and managed set of locators. This event identifier is returned as a part of the returned registration object and is unique across all other active registrations with the lookup discovery service.

Whenever the lookup discovery service finds a lookup service matching the discovery criteria of one or more of its registrations, it sends an instance of `RemoteDiscoveryEvent` (a subclass of `RemoteEvent`) to the listener corresponding to each such registration. The event sent to each listener will contain the appropriate event identifier.

Once an event signaling the discovery (by group or locator) of a desired lookup service has been sent, no other discovered events for that lookup service will be sent to a registration's listener until the lookup service is discarded (either actively, by the client through the registration, or passively by the lookup discovery service) and then rediscovered. Note that more information about what it

means for a lookup service to be discarded is presented in Section LD.3.1, "Registration Semantics" and the section of this specification titled "Discarding Lookup Services".

If, between the time a lookup service is discarded and the time it is rediscovered, a new registration is requested having parameters indicating interest in that lookup service, upon rediscovery of the lookup service an event will also be sent to that new registration's listener.

The sequence numbers for a given event identifier are strictly increasing (as defined in *The Jini Technology Core Platform Specification, "Distributed Events"*), which means that when any two such successive events have sequence numbers that differ by only a value of 1, then no events have been missed. On the other hand, when the set of received events is viewed in order, if the difference between the sequence numbers of two successive events is greater than 1, then one or more events may or may not have been missed. For example, a difference greater than 1 could occur if the lookup discovery service crashes, even if no events are lost because of the crash. When two such successive events have sequence numbers whose difference is greater than 1, there is said to be a *gap* between the events.

When a gap occurs between events, the local state (on the client) related to the discovered lookup services may or may not fall out of sync with the corresponding remote state maintained by the lookup discovery service. For example, if the gap corresponds to a missed event representing the (initial) discovery of a targeted lookup service, the remote state will reflect this discovery, whereas the client's local state will not. To allow clients to identify and correct such a situation, each registration object provides a method that returns a set consisting of the proxies to the lookup services that have been discovered for that registration. With this information the client can update its local state.

When requesting a registration with the lookup discovery service, a client may also supply (as a parameter to the `register` method) a reference to an object, wrapped in a `MarshalledObject`, referred to as a *handback*. When the lookup discovery service sends an event to a registration's listener, the event will also contain a reference to this handback object. The lookup discovery service will not change the handback object. That is, the handback object contained in the event sent by the lookup discovery service will be identical to the handback object registered by the client with the event mechanism.

The semantics of the object input to the `handback` argument are left to each client to define, although `null` may be input to this argument. The role of the `handback` object in the remote event mechanism is detailed in *The Jini Technology Core Platform Specification, "Distributed Events"*.

LD.3.3 Leasing Semantics

When a client registers with the lookup discovery service, it is effectively requesting a lease on the resources provided by that service. The initial duration of the lease granted to a client by the lookup discovery service will be less than or equal to the requested duration reflected in the value input to the `leaseDuration` argument. That value must be positive, `Lease.FOREVER`, or `Lease.ANY`. If any other value is input to this argument, an `IllegalArgumentException` will be thrown. The client may obtain a reference to the `Lease` object granted by the lookup discovery service through the associated registration returned by the service (see Section LD.4.1, “The LookupDiscoveryRegistration Interface”).

LD.4 Supporting Interfaces and Classes

THE lookup discovery service depends on the `LookupDiscoveryRegistration` interface, as well as on the concrete classes `RemoteDiscoveryEvent` and `LookupUnmarshalException`.

LD.4.1 The `LookupDiscoveryRegistration` Interface

When a client requests a registration with the lookup discovery service, an object that implements the `LookupDiscoveryRegistration` interface is returned. It is through this interface that the client manages the state of its registration with the lookup discovery service.

```
package net.jini.discovery;

public interface LookupDiscoveryRegistration {
    public EventRegistration getEventRegistration();
    public Lease getLease();
    public ServiceRegistrar[] getRegistrars()
                            throws LookupUnmarshalException,
                                   RemoteException;
    public String[] getGroups() throws RemoteException;
    public LookupLocator[] getLocators()
                                     throws RemoteException;
    public void addGroups(String[] groups)
                                     throws RemoteException;
    public void setGroups(String[] groups)
                                     throws RemoteException;
    public void removeGroups(String[] groups)
                                     throws RemoteException;
    public void addLocators(LookupLocator[] locators)
                                     throws RemoteException;
    public void setLocators(LookupLocator[] locators)
```

```
                                                throws RemoteException;
        public void removeLocators(LookupLocator[] locators)
                                                throws RemoteException;
        public void discard(ServiceRegistrar registrar)
                                                throws RemoteException;
    }
```

As with the `LookupDiscoveryService` interface, the `LookupDiscoveryRegistration` interface is not a remote interface. Each implementation of the lookup discovery service exports proxy objects that implement this interface local to the client, using an implementation-specific protocol to communicate with the actual remote server. All of the proxy methods must obey normal Java RMI remote interface semantics except where explicitly noted. Two proxy objects are equal (using the `equals` method) if they are proxies for the same registration created by the same lookup discovery service.

The discovery facility of the lookup discovery service, together with its event mechanism, make up the set of resources clients register to use. Because the resources of the lookup discovery service are leased, access is granted for only a limited period of time unless there is an active expression of continuing interest on the part of the client.

When a client uses the registration process to request that a lookup discovery service perform discovery of a set of desired lookup services, the client is also registered with the service's event mechanism. Because of this implicit registration with the event mechanism, the lookup discovery service "bundles" both resources under a single lease. When that lease expires, both discovery processing and event notifications will cease with respect to the registration that resulted from the client's request.

To facilitate lease management and event handling, the `LookupDiscoveryRegistration` interface defines methods that allow the client to retrieve its event registration information. Additional methods defined by this interface allow the client to retrieve references to the registration's currently discovered lookup services, as well as to modify the managed sets of groups and locators.

If the client's registration with the lookup discovery service has expired or been cancelled, then any invocation of a remote method defined in this interface will result in a `NoSuchObjectException`. That is, any method that communicates with the back end server of the lookup discovery service will throw a `NoSuchObjectException` if the registration on which the method is invoked no longer exists. Note that if a client receives a `NoSuchObjectException` as a result of an invocation of such a method, although the client can assume that the regis-

tration no longer exists, the client cannot assume that the lookup discovery service itself no longer exists.

Each remote method of this interface may throw a `RemoteException`. Typically, this exception occurs when there is a communication failure between the client and the lookup discovery service. Whenever this exception occurs as a result of the invocation of one of these methods, the method may or may not have completed its processing successfully.

LD.4.1.1 The Semantics

The methods defined by this interface are organized into a set of accessor methods, a set of group mutator methods, a set of locator mutator methods, and the `discard` method. Through the accessor methods, various elements of a registration's state can be retrieved. The mutator methods provide a mechanism for changing the set of groups and locators to be discovered for the registration. Through the `discard` method, a particular lookup service may be made eligible for rediscovery.

The Accessor Methods

The `getEventRegistration` method returns an `EventRegistration` object that encapsulates the information the client needs to identify a notification sent by the lookup discovery service to the registration's listener. This method is not remote and takes no arguments.

The `getLease` method returns the `Lease` object that controls a client's registration with the lookup discovery service. It is through the `Lease` object returned by this method that the client requests the renewal or cancellation of the registration with the lookup discovery service. This method is not remote and takes no arguments.

Note that the object returned by the `getEventRegistration` method also provides a `getLease` method. That method and the `getLease` method defined by the `LookupDiscoveryRegistration` interface both return the same `Lease` object. The `getLease` method defined here is provided as a convenience to avoid the indirection associated with the `getLease` method on the `EventRegistration` object, as well as to avoid the overhead of making two method calls.

The `getRegistrars` method returns a set of instances of the `ServiceRegistrar` interface. Each element in the set is a proxy to one of the lookup services that have already been discovered for the registration. Additionally, each element in the set will be unique with respect to all other elements in the set, as determined by the `equals` method provided by each element. The contents

of the set make up the current remote state of the set of lookup services discovered for the registration. This method returns a new array on each invocation.

This method can be used to maintain synchronization between the set of discovered lookup services making up a registration's local state on the client and the registration's corresponding remote state maintained by the lookup discovery service. The local state can become unsynchronized with the remote state when a gap occurs in the events received by the registration's listener.

According to the event semantics of the lookup discovery service, if there is no gap between two sequence numbers, no events have been missed and the states remain synchronized with each other; if there is a gap, events may or may not have been missed. Therefore, upon finding gaps in the sequence of events, the client can invoke this method and use the returned information to synchronize the local state with the remote state.

To construct its return set, the `getRegistrars` method retrieves from the lookup discovery service the set of lookup service proxies making up the registration's current remote state. When the lookup discovery service sends the requested set of proxies, the set is sent as a set of marshalled instances of the `ServiceRegistrar` interface. The lookup discovery service individually marshals each proxy in the set that it sends because if it were not to do so, *any* deserialization failure on the set would result in an `IOException`, and failure would be declared for the whole deserialization process, not just an individual element. This would mean that all elements of the set sent by the lookup discovery service—even those that were successfully deserialized—would be unavailable to the client. Individually marshalling each element in the set minimizes the "all or nothing" aspect of the deserialization process, allowing the client to recover those proxies that can be successfully unmarshalled and to proceed with processing that might not be possible otherwise.

When constructing the return set, this method attempts to unmarshal each element of the set of marshalled proxy objects sent by the lookup discovery service. When failure occurs while attempting to unmarshal any of those elements, this method throws an exception of type `LookupUnmarshalException` (described later). It is through the contents of that exception that the client can recover any available proxies and perform error handling related to the unavailable proxies. The contents of the `LookupUnmarshalException` provide the client with the following useful information:

- The knowledge that a problem has occurred while unmarshalling at least one of the elements making up the remote state of the registration's discovered lookup services

- The set of proxy objects that were successfully unmarshalled by the `getRegistrars` method
- The set of marshalled proxy objects that could not be unmarshalled by the `getRegistrars` method
- The set of exceptions corresponding to each failed attempt at unmarshalling

The type of exception that occurs when attempting to unmarshal an element of the set sent by the lookup discovery service is typically an `IOException` or a `ClassNotFoundException` (usually the more common of the two). A `ClassNotFoundException` occurs whenever a remote object on which the marshalled proxy depends cannot be retrieved and loaded, usually because the codebase of one of the object's classes or interfaces is currently "down." To address this situation, the client may wish to proceed with its processing using the successfully unmarshalled proxies, and attempt to unmarshal the unavailable proxies (or re-invoke this method) at some later time.

If the `getRegistrars` method returns successfully without throwing a `LookupUnmarshalException`, the client is guaranteed that all marshalled proxies belonging to the set sent by the lookup discovery service have each been successfully unmarshalled; the client then has a snapshot—relative to the point in time when this method is invoked—of the remote state of the lookup services discovered for the associated registration.

The `getGroups` method returns an array consisting of the group names from the registration's managed set; that is, the names of the groups the lookup discovery service is currently configured to discover for the associated registration. If the managed set of groups is empty, this method returns the empty array. If there is no managed set of groups associated with the registration (that is, the lookup discovery service is configured to discover `DiscoveryGroupManagement.ALL_GROUPS` for the registration), then `null` is returned.

The `getLocators` method returns an array consisting of the `LookupLocator` objects from the registration's managed set; that is, the locators of the specific lookup services the lookup discovery service is currently configured to discover for the associated registration. If the managed set of locators is empty, this method returns the empty array.

The Group Mutator Methods

With respect to a particular registration, the groups to be discovered may be modified using the methods described in this section. In each case, a set of groups is represented as a `String` array, none of whose elements may be `null`. If any set of groups input to one of these methods contains one or more `null` elements, a

`NullPointerException` is thrown. The empty set is denoted by the empty array (`DiscoveryGroupManagement.NO_GROUPS`), and "no set" is indicated by `null` (`DiscoveryGroupManagement.ALL_GROUPS`). No set indicates that all lookup services within the multicast radius should be discovered, regardless of group membership. Invoking any of these methods with an input set of groups that contains duplicate names is equivalent to performing the invocation with the duplicate group names removed from the input set.

The `addGroups` method adds a set of group names to the registration's managed set. This method takes one argument: a `String` array consisting of the set of group names with which to augment the registration's managed set.

If the registration has no current managed set of groups to augment, this method throws an `UnsupportedOperationException`. If the parameter value is `null`, this method throws a `NullPointerException`. If the parameter value is the empty array, then the registration's managed set of groups will not change.

The `setGroups` method replaces all of the group names in the registration's managed set with names from a new set. This method takes one argument: a `String` array consisting of the set of group names with which to replace the current names in the registration's managed set.

If `null` is passed to `setGroups`, the lookup discovery service will attempt to discover any undiscovered lookup services located within range of the lookup discovery service, regardless of group membership.

If the empty set is passed to `setGroups`, then group discovery will be halted until the registration's managed set of groups is changed—through a subsequent call to this method or to `addGroups`—to a set that is either a non-empty set of group names or `null`.

The `removeGroups` method deletes a set of group names from the registration's managed set. This method takes one argument: a `String` array containing the set of group names to remove from the registration's managed set.

If the registration has no current managed set of groups from which to remove elements, this method throws an `UnsupportedOperationException`. If `null` is input, this method throws a `NullPointerException`. If the registration does have a managed set of groups from which to remove elements, but either the input set is empty or none of the elements in the input set match any element in the managed set, then the registration's managed set of groups will not change.

Once a new group name has been placed in the registration's managed set as a result of an invocation of either `addGroups` or `setGroups`, if there are lookup services belonging to that group that have already been discovered for that registration, no event will be sent to the registration's listener for those particular lookup services. However, attempts to discover any undiscovered lookup services belonging to that group will continue to be made on behalf of the registration.

Any already discovered lookup service that is a member of one or more of the groups removed from the registration's managed set as a result of an invocation of either `setGroups` or `removeGroups` will be discarded and will no longer be eligible for discovery (for that registration), but only if that lookup service satisfies both of the following conditions:

- The lookup service is not a member of any group in the registration's new managed set resulting from the invocation of `setGroups` or `removeGroups`
- With respect to the registration, the lookup service is not currently eligible for discovery through locator discovery; that is, the lookup service does not correspond to any element in the registration's managed set of locators.

The Locator Mutator Methods

With respect to a particular registration, the set of locators to discover may be modified using the methods described in this section. In each case, a set of locators is represented as an array of `LookupLocator` objects, none of whose elements may be `null`. If any set of locators input to one of these methods contains one of more `null` elements, a `NullPointerException` is thrown. Invoking any of these methods with a set of locators that contains duplicate locators (as determined by `LookupLocator.equals`) is equivalent to performing the invocation with the duplicates removed from the input set.

The `addLocators` method adds a set of `LookupLocator` objects to the registration's managed set. This method takes one argument: an array consisting of the set of locators with which to augment the registration's managed set.

If `null` is passed to `addLocators`, a `NullPointerException` will be thrown. If the parameter value is the empty array, the registration's managed set of locators will not change.

The `setLocators` method replaces all of the locators in the registration's managed set with `LookupLocator` objects from a new set. This method takes one argument: an array consisting of the set of locators with which to replace the current locators in the registration's managed set.

If `null` is passed to `setLocators`, a `NullPointerException` will be thrown.

If the empty set is passed to `setLocators`, then locator discovery will be halted until the registration's managed set of locators is changed—through a subsequent call to this method or to `addLocators`—to a set that is non-`null` and non-empty.

The `removeLocators` method deletes a set of `LookupLocator` objects from the registration's managed set. This method takes one argument: an array contain-

ing the set of `LookupLocator` objects to remove from the registration's managed set.

If `null` is passed to `removeLocators`, a `NullPointerException` will be thrown. If any element of the set of locators to remove is not contained in the registration's managed set, `removeLocators` takes no action with respect to that element. If the parameter value is the empty array, the managed set of locators will not change.

Whenever a new locator is placed in the managed set as a result of an invocation of one of the locator mutator methods and that new locator equals none of the previously discovered locators (across all registrations), the lookup discovery service will attempt unicast discovery of the lookup service associated with the new locator.

If locator discovery is attempted for a registration, such discovery attempts will be repeated until one of the following events occurs:

- The lookup service is discovered
- The client's lease on the registration expires
- The client explicitly removes the locator from the registration's managed set

Upon discovery of the lookup service corresponding to the new locator, or upon finding a match between the new locator and a previously discovered lookup service, a discovered event will be sent to the registration's listener, unless that lookup service was previously discovered for that registration through group discovery.

Any already discovered lookup service corresponding to a locator that is removed from the registration's managed set as a result of an invocation of either `setLocators` or `removeLocators` will be discarded and will no longer be eligible for discovery, but only if it is not currently eligible for discovery through group discovery—that is, only if the lookup service is not also a member of one or more of the groups in the registration's managed set of groups.

Discarding Lookup Services

When the lookup discovery service removes an already discovered lookup service from a registration's managed set of lookup services, the lookup service is said to be *discarded*.

There are a number of situations in which the lookup discovery service will discard a lookup service:

- In response to a discard request resulting from an invocation of a registration's `discard` method
- In response to a declaration—via an invocation of one of the mutator methods on a registration—that there is no longer any interest in one or more of the registration's already discovered lookup services
- In response to the determination that the multicast announcements from an already discovered lookup service indicate that the lookup service has changed its group membership in such a way that the lookup service is no longer of interest to one or more of the registrations that previously registered interest in the groups of that lookup service
- In response to the determination that the multicast announcements from an already discovered lookup service are no longer being received

For each of these cases, whenever the lookup discovery service discards a lookup service, it will send an event to the registration's listener to notify it that the lookup service has been discarded.

The `discard` method provides a mechanism for registered clients to inform the lookup discovery service of the existence of an unavailable—or *unreachable*—lookup service, and to request that the lookup discovery service discard that lookup service and make it eligible for rediscovery.

The `discard` method takes a single argument: the proxy to the lookup service to discard. This method takes no action if the parameter to this method equals none of the proxies reflected in the managed set (using proxy equality as defined in *The Jini Technology Core Platform Specification,* "Lookup Service". If `null` is passed to `discard`, a `NullPointerException` is thrown.

Although the lookup discovery service monitors the multicast announcements from all discovered lookup services for indications of unavailability, it should be noted that there are conditions under which the lookup discovery service will not discard such a lookup service, even when the lookup service is found to be unreachable. Whether or not the lookup discovery service discards such an unreachable lookup service is dependent on how each registration is configured for discovery with respect to that lookup service. If every registration that is configured to discover the unreachable lookup service is configured to discover it through locator discovery only, the lookup discovery service will not discard the lookup service. In other words, in order for the lookup discovery service to discard a lookup service it has determined is unreachable, at least one registration must be configured for discovery of at least one group in which that lookup service is a member.

Thus, whenever a client determines that a previously discovered lookup service has become unreachable, it should not rely on the lookup discovery service to discard the lookup service. Instead, the client should inform the lookup discovery

service—through the invocation of the registration's `discard` method—that the previously discovered lookup service is no longer available and that attempts should be made to rediscover that lookup service for the registration. Typically, a client determines that a lookup service is unavailable when the client attempts to use the lookup service but receives an indefinite exception, a bad object exception, or a bad invocation exception as a result of the attempt.

Note that the lookup discovery service may be acting on behalf of numerous clients that have access to the same lookup service. If that lookup service becomes unavailable, many of those clients may invoke `discard` between the time the lookup service becomes unavailable and the time it is rediscovered. Upon the first invocation of `discard`, the lookup discovery service will re-initiate discovery of the relevant lookup service for the registration of the client that made the invocation. For all other invocations made prior to rediscovery, the registrations through which the invocation is made are sent a discarded event, and added to the list of registrations that will be notified when rediscovery of the lookup service does occur. That is, upon rediscovery of the lookup service, only those registrations through which the `discard` method was invoked will be notified.

Upon successful completion of the `discard` method, the proxy requested to be discarded is guaranteed to have been removed from the managed set of the registration through which the invocation was made. No such guarantee is made with respect to when the discarded event is sent to each such registration's listener. That is, the event notifying the listeners that the lookup service has been discarded may or may not be sent asynchronously.

LD.4.2 The `RemoteDiscoveryEvent` Class

When the lookup discovery service discovers or discards a lookup service matching the criteria established through one of its registrations, the lookup discovery service sends an instance of the `RemoteDiscoveryEvent` class to the `RemoteEventListener` implemented by the client and registered with the lookup discovery service.

```
package net.jini.discovery;

public class RemoteDiscoveryEvent extends RemoteEvent {
    public RemoteDiscoveryEvent(Object source,
                                long eventID,
                                long seqNum,
                                MarshalledObject handback,
                                boolean discarded,
```

```
                                    Map groups)
                                           throws IOException {…}

    public boolean isDiscarded() {…}
    public ServiceRegistrar[] getRegistrars()
                            throws LookupUnmarshalException {…}
    public Map getGroups() {…}
}
```

The `RemoteDiscoveryEvent` class provides an encapsulation of event information that the lookup discovery service uses to notify a registration of the occurrence of an event involving one or more `ServiceRegistrar` objects (lookup services) in which the registration has registered interest. The lookup discovery service passes an instance of this class to the registration's discovery listener when one of the following events occurs:

- Each lookup service referenced in the event has been discovered for the first time or rediscovered after having been discarded.
- Each lookup service referenced in the event has been either actively or passively discarded.

`RemoteDiscoveryEvent` is a subclass of `RemoteEvent`, adding the following additional items of abstract state:

- A `boolean` indicating whether the lookup services referenced by the event have been discovered or discarded
- A set of marshalled instances of the `ServiceRegistrar` interface having the characteristic that when each element is unmarshalled, the result is a proxy to one of the discovered or discarded lookup services referenced by the event
- A `Map` instance in which the elements of the map's key set are the instances of `ServiceID` that correspond to each lookup service reference returned in the event, and the map's value set contains the corresponding member groups of each lookup service reference

Methods are defined through which this additional state may be retrieved upon receipt of an instance of this class.

Clients need to know not only when a targeted lookup service has been discovered, but also when it has been discarded. The lookup discovery service uses an instance of `RemoteDiscoveryEvent` to notify a registration when either of these events occurs, as indicated by the value of the `boolean` state variable. When

the value of that variable is `true`, the event is referred to as a *discarded event*; when `false`, it is referred to as a *discovered event*.

LD.4.2.1 The Semantics

The constructor of the `RemoteDiscoveryEvent` class takes the following parameters as input:

- A reference to the lookup discovery service that generated the event
- The event identifier that maps a particular registration to both its listener and its targeted groups and locators
- The sequence number of the event being constructed
- The client-defined handback (which may be `null`)
- A flag indicating whether the event being constructed is a discovered event or a discarded event
- A `Map` whose key set contains the proxies to newly discovered or discarded lookup service(s) the event is to reference, and whose value set contains the corresponding member groups of each lookup service

If the `groups` parameter is empty, the constructor will throw an `IllegalArgumentException`. If null is input to the `groups` parameter, the constructor will throw a `NullPointerException`. If none of the proxies referenced in the `groups` parameter can be successfully serialized, the constructor will throw an `IOException`.

The `isDiscarded` method returns a `boolean` that indicates whether the event is a discovered event or a discarded event. If the event is a discovered event, then this method returns `false`. If the event is a discarded event, `true` is retuned.

The `getRegistrars` method returns an array consisting of instances of the `ServiceRegistrar` interface. Each element in the returned set is a proxy to one of the newly discovered or discarded lookup services that caused a `RemoteDiscoveryEvent` to be sent. Additionally, each element in the returned set will be unique with respect to all other elements in the set, as determined by the `equals` method provided by each element. This method does not make a remote call. With respect to multiple invocations of this method, each invocation will return a new array.

When the lookup discovery service sends an instance of `RemoteDiscoveryEvent` to the listener of a client's registration, the set of lookup service proxies contained in the event consists of marshalled instances of the `ServiceRegistrar` interface. The lookup discovery service individually marshals

each proxy associated with the event because if it were not to do so, *any* deserialization failure on the set would result in an `IOException`, and failure would be declared for the whole deserialization process, not just an individual element. This would mean that all elements of the set sent in the event—even those that can be successfully deserialized—would be unavailable to the client through this method. Just as with the `getRegistrars` method defined by the `LookupDiscoveryRegistration` interface, individually marshalling each element in the set minimizes the "all or nothing" aspect of the deserialization process, allowing the client to recover those proxies that can be successfully unmarshalled and to proceed with processing that might not be possible otherwise.

When constructing the return set, this method attempts to unmarshal each element of the set of marshalled proxy objects contained in the event. When failure occurs while attempting to unmarshal any of the elements of that set, this method throws an exception of type `LookupUnmarshalException`. It is through the contents of this exception that the client can recover any available proxies and perform error handling with respect to the unavailable proxies.

If the `getRegistrars` method returns successfully without throwing a `LookupUnmarshalException`, the client is guaranteed that all marshalled proxies sent in the event have each been successfully unmarshalled during that particular invocation. Furthermore, after the first such successful invocation, no more unmarshalling attempts will be made (because such attempts are no longer necessary), and all future invocations of this method are guaranteed to return an array with contents identical to the contents of the array returned by the first successful invocation.

Note that an array, rather than a single proxy, is returned by the `getRegistrars` method so that implementations of the lookup discovery service can choose to "batch" the information sent to a registration. With respect to discoveries, batching the information may be particularly useful when a client first registers with the lookup discovery service.

Upon initial registration, multiple lookup services are typically found over a short period of time, providing the lookup discovery service with the opportunity to send all of the initially discovered lookup services in only one event. Afterward, as so-called "late joiner" lookup services are found sporadically, the lookup discovery service may send events referencing only one lookup service.

Note that the event sequence numbers, as defined earlier in Section LD.3.2, "Event Semantics", are strictly increasing, even when the information is batched.

The `getGroups` method returns a `Map` in which the elements of the map's key set are the instances of `ServiceID` that correspond to each lookup service for which the event was constructed and sent. Each element of the returned map's value set is a `String` array containing the names of the member groups of the

associated lookup service whose ServiceID equals to the corresponding key. This method does not make a remote call. On each invocation of this method, the same Map object is returned; that is, a copy is not made.

The Map returned by the getGroups method is keyed by the ServiceID of each lookup service in the event, rather than by the proxy of each lookup service to avoid the deserialization issues addressed by the getRegistrars method. Thus, client's wishing to retrieve the set of member groups corresponding to any element of the array returned by the getRegistrars method, must use the ServiceID of the desired element from that array as the key to the get method of the Map returned by this method and then cast to String[].

LD.4.2.2 Serialized Forms

Class	serialVersionUID	Serialized Fields
RemoteDiscoveryEvent	-9171289945014585248L	boolean discarded ArrayList marshalledRegs ServiceRegistrar[] regs Map groups

LD.4.3 The LookupUnmarshalException Class

Recall that when unmarshalling an instance of MarshalledObject, one of the following checked exceptions is possible:

- An IOException, which can occur while deserializing the object from its internal representation
- A ClassNotFoundException, which can occur if, while deserializing the object from its internal representation, either the class file of the object cannot be found, or the class file of an interface or class referenced by the object being deserialized cannot be found. Typically, a ClassNotFoundException occurs when the codebase from which to retrieve the needed class file is not currently available

The LookupUnmarshalException class provides a mechanism that clients of the lookup discovery service may use for efficient handling of the exceptions that may occur when unmarshalling elements of a set of marshalled instances of the ServiceRegistrar interface. When elements in such a set are unmarshalled, the

`LookupUnmarshalException` class may be used to collect and report pertinent information generated when failure occurs during the unmarshalling process.

```
package net.jini.discovery;

public class LookupUnmarshalException extends Exception {
    public LookupUnmarshalException
                    (ServiceRegistrar[] registrars,
                     MarshalledObject[] marshalledRegistrars,
                     Throwable[] exceptions) {…}
    public LookupUnmarshalException
                    (ServiceRegistrar[] registrars,
                     MarshalledObject[] marshalledRegistrars,
                     Throwable[] exceptions,
                     String message) {…}
    public ServiceRegistrar[] getRegistrars() {…}
    public MarshalledObject[] getMarshalledRegistrars() {…}
    public Throwable[] getExceptions() {…}
}
```

The `LookupUnmarshalException` class is a subclass of `Exception`, adding the following additional items of abstract state:

- A set of `ServiceRegistrar` instances in which each element is the result of a successful unmarshalling attempt
- A set of marshalled instances of `ServiceRegistrar` in which each element is the result of an unsuccessful unmarshalling attempt
- A set of exceptions (`IOException`, `ClassNotFoundException`, or some unchecked exception) in which each element corresponds to one of the unsuccessful unmarshalling attempts

When exceptional conditions occur while unmarshalling a set of marshalled instances of `ServiceRegistrar`, the `LookupUnmarshalException` class can be used not only to indicate that an exceptional condition has occurred, but also to provide information that can be used to perform error handling activities such as:

- Determining if it is feasible to continue with processing
- Reporting errors
- Attempting recovery
- Performing debug activities

LD.4.3.1 The Semantics

The constructor of the `LookupUnmarshalException` class has two forms. The first form of the constructor takes the following parameters as input:

- An array containing the set of instances of `ServiceRegistrar` that were successfully unmarshalled
- An array containing the set of marshalled `ServiceRegistrar` instances that could not be unmarshalled
- An array containing the set of exceptions that occurred during the unmarshalling process

The second form of the constructor takes the same arguments as the first and one additional argument: a `String` describing the nature of the exception.

Each element in the `exceptions` parameter should be an instance of `IOException`, `ClassNotFoundException`, or some unchecked exception. Furthermore, there is a one-to-one correspondence between each element in the `exceptions` parameter and each element in the `marshalledRegistrars` parameter. That is, the element of the `exceptions` parameter corresponding to index *i* should be an instance of the exception that occurred while attempting to unmarshal the element at index *i* of the `marshalledRegistrars` parameter.

If the number of elements in the `exceptions` parameter does not equal the number of elements in the `marshalledRegistrars` parameter, the constructor will throw an `IllegalArgumentException`.

The `getRegistrars` method is an accessor method that returns an array consisting of instances of `ServiceRegistrar`, where each element of the array corresponds to a successfully unmarshalled object. Note that the same array is returned on each invocation of this method; that is, a copy is not made.

The `getMarshalledRegistrars` method is an accessor method that returns an array consisting of instances of `MarshalledObject`, where each element of the array is a marshalled instance of the `ServiceRegistrar` interface and corresponds to an object that could not be successfully unmarshalled. Note that the same array is returned on each invocation of this method; that is, a copy is not made.

The `getExceptions` method is an accessor method that returns an array consisting of instances of `Throwable`, where each element of the array corresponds to one of the exceptions that occurred during the unmarshalling process. Each element in the return set is an instance of `IOException`, `ClassNotFoundException`, or some unchecked exception. Additionally, there should be a one-to-one correspondence between each element in the array returned by this method and the

array returned by the `getMarshalledRegistrars` method. Note that the same array is returned on each invocation of this method; that is, a copy is not made.

LD.4.3.2 Serialized Forms

Class	`serialVersionUID`	Serialized Fields

`LookupUnmarshalException`	2956893184719950537L	`ServiceRegistrar[] registars`
		`MarshalledObject[] marshalledRegistrars`
		`Throwable[] exceptions`

LR

Jini Lease Renewal Service Specification

LR.1 Introduction

LEASING is a key concept in the Jini architecture; in general, Jini technology-enabled services (*Jini services*) grant access to a resource only for as long as the clients of those Jini services actively express interest in the resource being maintained. This pattern is in contrast to many other systems, in which access to a resource is granted until the client explicitly releases the resource. Using a leasing model generally makes a distributed system more robust by allowing stale information and services to be cleaned up, but it also places additional requirements on clients and services.

A client of a leased service may run into difficulties if that client deactivates. Unless the client ensures that some other process renews the client's leases while it is inactive, or that the client is activated before its leases begin to expire, the client will lose access to the resources it has acquired. This loss can be particularly dramatic in the case of lookup service registrations. A service's registration with a lookup service is leased—if the service deactivates (maybe to conserve computational resources on its host) and it does not take appropriate steps, its registrations with lookup services will expire, and before long it will be inaccessible. If that service becomes active only when clients invoke its methods, it may never become active again, because at this point new clients may not be able to find it.

The need to renew leases creates a constant load on clients, servers, and the network. Although batching lease renewals can help (see *The Jini Technology Core Platform Specification, "Distributed Leasing")*, a given client is unlikely to have very many leases granted by any one service at any given time, thus reducing the opportunities for meaningful batching.

This additional load may be an especially great burden on clients that always have the ability to access the network but cannot be continuously connected. A cell phone always has the ability to connect; however, being connected all the time will drain its batteries and accumulate airtime charges. One or two leases may not pose a problem, but a large number of leases could force the phone to be on the network all the time.

A lease renewal service can help mitigate these problems. Clients that wish to become inactive can pass the responsibility for renewing the leases they have been granted to a renewal service. Those clients can then deactivate without risk of losing access to the resources that they have acquired. Clients that have continuous access to the network but cannot be continuously connected, such as the cell phone described previously, can also register with a renewal service that can be continuously connected. The renewal service will renew the client's leases, allowing the client to remain disconnected most of the time. Lastly, if multiple clients pass their leases to a given renewal service, more opportunities for batching renewals will be created.

Like other Jini services, the lease renewal service will grant its services for only a limited period of time without an active expression of continuing interest. To break the recursive cycle that would otherwise result, the renewal service provides an optional event that is triggered before the leases that it grants expire. This event gives activatable processes that have deactivated the opportunity to wake up and renew their lease with the renewal service. Although it may seem odd for the lease renewal service to lease its resources, it is very important that it does so. If it did not, then the lease renewal service could be used to subvert the leasing model.

Lease renewal services are likely to grant longer leases than other Jini services. In some cases the lease may be so long that the client will not need to worry about renewing the lease at all. In other cases the lease may be long enough that a client that deactivates will rarely need to reactivate for the sole purpose of renewing its lease with the renewal service. In any case, the leases that the renewal service grants are likely to be sufficiently long such that the actual renewal calls do not place a significant additional load on the client, the renewal service, or the network.

LR.1.1 Goals and Requirements

The requirements of the set of classes and interfaces in this specification are:

- To provide a service for renewing leases

- To provide this service in such a way that it can be used by activatable processes that deactivate
- To provide this service in a way that does not overly weaken the leasing model

The goals of this specification are:

- To describe the lease renewal service
- To provide guidance in the use, deployment, and implementation of the lease renewal service

LR.1.2 Other Types

The types defined in the specification of the `LeaseRenewalService` interface are in the `net.jini.lease` package. The following object types may be referenced in this chapter. Whenever referenced, these object types will be referenced in unqualified form:

```
java.io.IOException
java.rmi.MarshalledObject
java.rmi.RemoteException
java.rmi.NoSuchObjectException
net.jini.core.lease.Lease
net.jini.core.lease.UnknownLeaseException
net.jini.core.event.RemoteEvent
net.jini.core.event.RemoteEventListener
net.jini.core.event.EventRegistration
```

LR.2 The Interface

THE `LeaseRenewalService` (in the `net.jini.lease` package) defines the interface to the renewal service. The interface is not a remote interface; each implementation of the renewal service exports proxy objects that implement the `LeaseRenewalService` interface local to the client, using an implementation-specific protocol to communicate with the actual remote server. All of the proxy methods obey normal RMI remote interface semantics. Two proxy objects are equal (using the `equals` method) if they are proxies for the same renewal service. All the methods of `LeaseRenewalService` throw `RemoteException` and require only the default serialization semantics. Therefore, `LeaseRenewalService` can be implemented directly using RMI.

```
package net.jini.lease;

public interface LeaseRenewalService {
    public LeaseRenewalSet createLeaseRenewalSet(
            long leaseDuration)
        throws RemoteException;
}
```

Clients of the renewal service organize the leases they wish to have renewed into *lease renewal sets* (or *sets,* for short). A method is provided by the `LeaseRenewalService` interface to create these sets. These sets are then populated by methods on the sets themselves. Two leases in the same set need not be granted by the same service or have the same expiration time; in addition, they can be added or removed from the set independently.

Every method invocation on a renewal service (whether the invocation is directly on the service or indirectly on a `set` the service has created) is atomic with respect other invocations.

The term *client lease* is used to refer to a lease that has been placed into a renewal set. Client leases are distinct from the leases that the renewal service grants on renewal sets it has created.

In general, there will be times when an implementation of the renewal service needs to pass one client lease as an argument to a method call on a second client

lease. There is a security risk in doing so, because such actions can let the second client lease "capture" the first. Implementations may want to verify that their clients can be trusted not to place leases in the set that would take such actions. Another alterative is to pass one `Lease` object to another only if they trust each other. Depending on the environment, conservative tests for such trust could include: ensuring the codebases of both leases are constructed from the same set of URLs, or that all of the URLs come from a common set of hosts or host/port pairs.

Each client lease has two expiration related times associated with it: the *desired expiration* time for the lease and the *actual expiration* time granted when the lease is created or last renewed. The desired expiration represents when the client would like the lease to expire. The actual expiration represents when the lease is going to expire if it is not renewed. Both time values are absolute times, not relative time durations. When a client lease's desired expiration arrives, the lease will be removed from the set without further client intervention.

Each client lease also has two other associated attributes: a *renewal duration* and a *remaining desired duration*. The remaining desired duration is always the desired expiration less the current time. The renewal duration is usually a positive number and represents the duration that will be requested when the renewal service renews the client lease, unless the renewal duration is greater than the remaining desired duration. If the renewal duration is greater than the remaining desired duration, then the remaining desired duration will be requested when renewing the client lease. One exception is that when the desired expiration is `Lease.FOREVER`, the renewal duration may be `Lease.ANY`, in which case `Lease.ANY` will be requested when renewing the client lease, regardless of the value of the remaining desired duration.

For example, if the renewal duration associated with a given client lease is 360,000 milliseconds, then when the renewal service renews the client lease, it will ask for a new duration of 360,000 milliseconds—unless the client lease is going to reach its desired expiration in less than 360,000 milliseconds. If the client lease's desired expiration is within 360,000 milliseconds, the renewal service will ask for the difference between the current time and the desired expiration. If the renewal duration had been `Lease.ANY`, the renewal service would have asked for a new duration of `Lease.ANY`.

If a lease's actual expiration is later than the lease's desired expiration, the renewal service will not renew the lease; the lease will remain in the set until its desired expiration is reached, the set is destroyed, or it is removed by the client.

Each set is leased from the renewal service. If the lease on a set expires or is cancelled, the renewal service will destroy the set and take no further action with regard to the client leases in the set. Each lease renewal set has associated with it an expiration warning event that occurs at a client-specified time before the lease

on the set expires. Clients can register for warning events using methods provided by the set. A registration for warning events does not have its own lease, but instead is covered by the same lease under which the set was granted.

The term *definite exception* is used to refer to an exception that could be thrown by an operation on a client lease (such as a remote method call) that would be indicative of a permanent failure of the client lease. In this specification, all bad object exceptions, bad invocation exceptions, and `LeaseExceptions` are considered to be definite exceptions (see *Introduction to Helper Utilities and Services*, Section US.2.6, "What Exceptions Imply about Future Behavior").

Each lease renewal set has associated with it a renewal failure event that will occur in either of two cases: if any client lease in the set reaches its actual expiration before its desired expiration is reached, or if the renewal service attempts to renew a client lease and gets a definite exception. Clients can register for failure events using methods provided by the set. A registration for failure events does not have its own lease but instead is covered by the same lease under which the set was granted.

Once placed in a set, a client lease will stay there until one or more of the following occurs:

- The lease on the set itself expires or is cancelled, causing destruction of the set.
- The client lease is removed by the client.
- The client lease's desired expiration is reached.
- The client lease's actual expiration is reached; this will generate a renewal failure event.
- A renewal attempt on the client lease results in a definite exception; this will generate a renewal failure event.

Each client lease in a set will be renewed as long as it is in the set. If a renewal call throws an indefinite exception (see *Introduction to Helper Utilities and Services,* Section US.2.6, "What Exceptions Imply about Future Behavior"), the renewal service should retry the lease renewal until the lease would otherwise be removed from the set. The preferred method of cancelling a client lease is for the client to first remove the lease from the set and then call `cancel` on it. It is also permissible for the client to cancel the lease without first removing the lease from the set, although this is likely to result in additional network traffic.

The client creates a set by calling the `createLeaseRenewalSet` method of a `LeaseRenewalService`. The `leaseDuration` argument specifies how long (in milliseconds) the client wants the set's initial lease duration to be. The duration initially granted for the set's lease will be equal to or shorter than this request; it