Jini™ Technology Helper Utilities And Services Specification

June 15, 1999

The Jini™ system is a distributed system based on Java™ technology and designed around the goals of simplicity, flexibility, and federation. Jini technology helper utility and service classes are intended to aid in the process of building clients and services that will participate in a Jini application environment. This document describes the functionality and operational requirements of the current set of classes that satisfy the definition of a Jini technology helper utility or a Jini technology helper service.

alpha draft

Copyright © 1999 Sun Microsystems, Inc.
901 San Antonio Road, Palo Alto, CA 94303 USA.
All rights reserved.

Sun Microsystems, Inc. has intellectual property rights (“Sun IPR”) relating to implementations of the technology described in this publication (“the Technology”). In particular, and without limitation, Sun IPR may include one or more patents or patent applications in the U.S. or other countries. Your limited right to use this publication does not grant you any right or license to Sun IPR nor any right or license to implement the Technology. Sun may, in its sole discretion, make available a limited license to Sun IPR and/or to the Technology under a separate license agreement. Please visit http://www.sun.com/software/communitysource/.

Sun, the Sun logo, Sun Microsystems, Jini, the Jini logo, JavaSpaces, Java, JavaBeans, Solaris, NFS, PC-NFS, EmbeddedJava, PersonalJava, and Solstice are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

THIS SPECIFICATION IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE SPECIFICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN ANY TECHNOLOGY, PRODUCT, OR PROGRAM DESCRIBED IN THIS SPECiFICATION AT ANY TIME.

Contents

0. About This Document ix

0.1 Status ix

0.2 Annotations ix

0.3 Comments x

1. Introduction 1

1.1 Overview 1

1.2 Terminology 2

1.2.1 Terms Related to Discovery and Join 3

1.2.2 Jini Clients and Services 3

1.2.3 Helper Service 4

1.2.4 Helper Utility 4

1.2.5 Managed Sets 4

1.2.6 Unavailable Lookup Services 5

1.2.7 Discarding a Lookup Service 6

1.2.8 Activation 8

1.3 Utility Interfaces 8
1.4 Helper Utilities 9
1.4.1 The LookupLocatorDiscovery Utility 9
1.4.2 The LookupDiscoveryManager 9
1.4.3 The LeaseRenewalManager 10
1.4.4 The JoinManager 10
1.4.5 The ClientLookupManager 10
1.5 Helper Services 11
1.5.1 The LookupDiscoveryService 11
1.5.2 The LeaseRenewalService 12
1.5.3 The EventMailbox 12
1.6 Dependencies 13
2. The Discovery Management Interfaces 15
2.1 Overview 15
2.2 Other Types 17
2.3 The `DiscoveryManagement` Interface 18
2.3.1 The Semantics 18
2.4 The `DiscoveryGroupManagement` Interface 21
2.4.1 The Semantics 21
2.5 The `DiscoveryLocatorManagement` Interface 24
2.5.1 The Semantics 24
3. The `LookupLocatorDiscovery` Utility 27
3.1 Overview 27
3.2 Other Types 28

3.3 The Interface 29
3.4 The Semantics 29
3.5 Supporting Interfaces 30
3.5.1 The `DiscoveryManagement` Interfaces 30
4. The `LookupDiscoveryManager` 31
4.1 Overview 31
4.2 Other Types 32
4.3 The Interface 33
4.4 The Semantics 33
4.5 Supporting Interfaces and Classes 34
4.5.1 The `DiscoveryManagement` Interfaces 35
4.5.2 The `DiscoveryListener` Interface 35
4.5.3 The `DiscoveryEvent` Class 36
5. The `LeaseRenewalManager` 37
5.1 Overview 37
5.2 Other Types 38
5.3 The Interface 39
5.4 The Semantics 39
5.5 Supporting Interfaces and Classes 43
5.5.1 The `LeaseListener` Interface 43
5.5.2 The `LeaseRenewalEvent` Class 44
6. The `JoinManager` 47
6.1 Overview 47
6.2 Other Types 48

6.3 The Interface 50
6.4 The Semantics 51
6.5 Supporting Interfaces and Classes 56
6.5.1 The `DiscoveryManagement` Interface 56
6.5.2 The `ServiceIDListener` Interface. 57
7. The `ClientLookupManager` 59
7.1 Overview 59
7.2 The Object Types. 62
7.3 The Interface 64
7.4 The Semantics 65
7.5 Supporting Interfaces and Classes 77
7.5.1 The `DiscoveryManagement` Interface 77
7.5.2 The `ServiceItemFilter` Interface. 78
7.5.3 The `ServiceDiscoveryEvent` Class 79
7.5.4 The `ServiceDiscoveryListener` Interface . . . 81
7.5.5 The `LookupCache` Interface. 83
8. The `LookupDiscoveryService` 89
8.1 Overview 89
8.1.1 Goals & Requirements 92
8.2 Other Types 92
8.3 The Interface 94
8.4 The Semantics 95
8.5 Supporting Interfaces and Classes 99
8.5.1 The `LookupDiscoveryRegistration` Interface 100

8.5.2 The `RemoteDiscoveryEvent` Class 108
9. The `LeaseRenewalService` 111
9.1 Overview 111
9.1.1 Goals & Requirements 112
9.2 Other Types 113
9.3 The Interface 114
10. The `EventMailbox` 123
10.1 Overview 123
10.2 Requirements 124
10.3 Other Types 124
10.4 Model and Terms 125
10.5 The `EventMailbox` Interface 126
10.6 The `MailboxRegistration` Interface 127
10.7 Typical Mailbox Interactions 129

About This Document 0

0.1 Status

This document is the 1.1 Alpha draft version of the Jini™ Technology Helper Utilities And Services Specification.The primary purpose of publishing at this time is to gather comments from a wider audience on the design and usefulness of these utilities. As a welcome side effect, we will also gather data on the utility of this document itself—we encourage editorial as well as technical comment.

This specification is only one of many Jini specifications that together would describe the functionality of a 1.1 Jini technology release. Please note that this specification may change in any way, including its title, organization, or content, without notice. Any part of this specification may at a later date be deleted, subsumed by an existing Jini specification, or become a separate Jini specification. As with any draft, all details are subject to change without notice.

0.2 Annotations

Note – In this document you will see several paragraphs in this style. These are areas where we specifically invite comment. Consider them as "notes to reviewers."

0.3 Comments

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

Introduction 1

1.1 Overview

When developing clients and services that will participate in a Jini™ application environment, there are a number of behaviors that the developer may find desirable to incorporate in the client or service. Some of these behaviors may satisfy requirements described in the specifications of various Jini technology components; some behaviors may simply represent design practices that are desirable and should be encouraged. Examples of the sort of behavior that is required or desirable include the following:

- It is a requirement of the Jini Discovery protocol that a service must continue to listen for and act on announcements from lookup services in which the service has registered interest.
- It is a requirement of the Jini Discovery protocol that, until successful, a service must continue to attempt to join the specific lookup services with which it has been configured to join.
- Under many conditions, a Jini client or service will wish to regularly renew leases that it holds. For example, when a Jini service registers with a Jini Lookup service, the service is requesting residency in the lookup service. Residency in a lookup service is a leased resource. Thus, when the requested residency is granted, the lookup service also imposes a lease on that residency. Typically, such a registered service will wish to extend the lease on its residency beyond the original expiration time; resulting in a need to renew the lease on a regular basis.

- Many Jini services will need to maintain a dormant (inactive) state; becoming active only when needed.
- Many Jini clients and services will need to have a mechanism for finding and managing Jini services.
- Many Jini clients and services will find it desirable to employ a separate service that will handle events, in some useful way, on behalf of the participant.

In order to help simplify the process of developing clients and services for the Jini application environment, a set of reusable components, which encapsulate behaviors such as those outlined above, will be specified in this document. These components will take the form of *helper* utilities and services. Employing these utilities and services to build such desirable behavior into a Jini client or service can help to avoid poor design and implementation decisions, greatly simplifying the development process.

Note that when this document uses either the word “must” or the word “will” to describe a behavior of one of the helper utility or service classes, that behavior is considered a requirement. When this document describes a behavior using the word “should” — or otherwise indicates that the behavior is desirable — the behavior is highly recommended but not required.

1.2 Terminology

This section defines terms and discusses concepts that may be referenced throughout the chapters of this document. While the terms and concepts appearing in this section are general in nature and may apply to multiple components specified in this document, each chapter may define additional terms and concepts to further facilitate the understanding of a particular component’s specification. Each chapter may also present supplemental information about some of the terms defined in this section and their relationship with the component being specified.

Because this specification makes use of a number of terms defined in the *Jini™ Technology Glossary*, reviewing that document is recommended. A number of the terms defined in the glossary are also defined in this section to provide easy reference, and because those terms are used extensively in this document. Additionally, this section augments the definitions of some of the terms from the glossary with details relevant to this specification.

In addition to the glossary, the Jini technology specifications present detailed definitions of a number of terms and concepts appearing both in this section and throughout this document. When appropriate, the relevant Jini technology specification will be referenced.

1.2.1 Terms Related to Discovery and Join

The *Jini™ Discovery and Join Specification* defines a *discovering entity* as one or more cooperating software objects written in the Java™ programming language (*Java software objects*), executing on the same host, that are in the process of obtaining references to Jini Lookup services. That specification also defines a *joining entity* as one or more cooperating Java software objects, on the same host, that have received a reference to a lookup service and are in the process of obtaining services from, and possibly exporting services to, a federation of lookup services referred to as a *djinn*. The lookup services comprising a djinn may be organized into one or more sets known as *groups*. Multiple groups may or may not be disjoint. Each group of lookup services is identified by a logical name represented by a `String` object.

The *Jini™ Discovery and Join Specification* defines two protocols used in the discovery process: the *multicast discovery protocol* and the *unicast discovery protocol*.

When a discovering entity employs the multicast discovery protocol to discover lookup services that are members of one or more groups belonging to a set of groups, that discovery process is referred to as *group discovery*.

The utility class `net.jini.core.discovery.LookupLocator` is defined in the *Jini™ Discovery Utilities Specification*. Any instance of that class is referred to as a *locator*. When a discovering entity employs the unicast discovery protocol to discover specific lookup services, each corresponding to an element in a set of locators, that discovery process is referred to as *locator discovery*.

1.2.2 Jini Clients and Services

For the purposes of this document, a Jini *client* is defined as a discovering entity that can retrieve a service (or a remote reference to a service) registered with a discovered lookup service; and invoke the methods of the service so as to meet the entity's requirements. An entity that acts only as a client never registers with (requests residency in) a lookup service.

A Jini *service* is defined as both a discovering and a joining entity containing methods which may be of use to some other Jini client or service, and which registers with discovered lookup services to provide access to those methods. Note that a Jini service can also act as a Jini client.

The term *client-like entity* may be used, in general, when referring to Jini clients and Jini services that act as clients.

Note that when the term *entity* is used, that term may be referring to a discovering entity, a joining entity, a client-like entity, a service, or some combination of these types of entities. Whenever that general term is used, it should be clear from the context what type of entity is being discussed.

1.2.3 Helper Service

A Jini technology *helper service* is defined in this document as an interface or set of interfaces, with an associated implementation, that encapsulates behavior that is either required or highly desirable in service entities that adhere to the Jini technology programming model (or simply, the *Jini programming model*). A helper service is a Jini service that can be registered with any number of lookup services, and whose methods can execute on remote hosts.

A helper service should be of use to more than one type of entity participating in a Jini application environment, and should provide a significant reduction in development complexity for developers of such entities.

1.2.4 Helper Utility

This document distinguishes between a helper *utility* and a helper *service*. Helper utilities are programming components that can be used during the construction of Jini services and/or clients. Helper utilities are *not* remote and do not register with a lookup service. Helper utilities are instantiated locally by entities wishing to employ them.

1.2.5 Managed Sets

When performing discovery duties, entities will often maintain references to discovered lookup services in a set referred to as the *managed set of lookup services*. The entity may also maintain two other notable sets: the *managed set of groups* and the *managed set of locators*.

Each element of the managed set of groups is a name of a group whose members are lookup services that the entity wishes to be discovered. These interfaces represent this set as a `String` array.

Each element of the managed set of locators corresponds to a specific lookup service that the entity wishes to be discovered. Typically, this set is represented as an array of `net.jini.core.discovery.LookupLocator` objects.

Note that when the general term *managed set* is used, it should be clear from the context whether groups, locators, or lookup services are being discussed.

1.2.6 Unavailable Lookup Services

While interacting (or attempting to interact) with a lookup service, an entity may encounter an exception or error condition. Depending on the nature of the exception or error encountered, the entity may interpret the situation to mean that the lookup service is simply no longer available; that is, the lookup service is *unavailable*.

For most entities, the unavailability of a particular lookup service should not prevent the entity from continuing its processing. Although there are a number of exception and error conditions that in other situations might be considered unrecoverable (or fatal) to an entity, when the condition indicates an unavailable lookup service, the entity should catch and handle the exception or error and continue processing.

The set of exceptions and errors that are non-fatal to an entity when interacting with a lookup service is dependent on the nature of the entity. Thus, each entity must specify its own set of non-fatal exceptions and errors for identifying unavailable lookup services.

Although this document cannot specify a single, definitive set of non-fatal exceptions and errors that meets the needs of all entities, a set that may meet the needs of *most* entities is presented below:

- `java.lang.Exception`
- `java.lang.LinkageError`
- `java.lang.OutOfMemoryError`
- `java.lang.StackOverflowError`

Thus, when any of the conditions in the list above (or the conditions defined by the entity itself) occur while the entity is interacting with a lookup service, the lookup service should be considered unavailable. Whenever an entity encounters an unavailable lookup service, the entity should catch and handle the condition; usually by requesting that the unavailable lookup service be discarded (see below). For all other exceptions and errors considered fatal, the entity should not attempt to recover.

1.2.7 Discarding a Lookup Service

When an already-discovered lookup service is removed from the managed set of lookup services, it is said to be *discarded*. The process of discarding a lookup service is initiated, either directly or indirectly, by the discovering entity.

When the entity encounters an unavailable lookup service, the entity typically will request that the unavailable lookup service be discarded. Additionally, whenever the entity requests the removal of an element from the managed set of groups or locators, one or more of the lookup services associated with the removed elements may be discarded. For this case, whether or not the discard process is executed is dependent on the state of the managed sets after the removal request has been processed.

Whenever a lookup service is discarded, a notification event referencing the discarded lookup service will be sent to all of the entity's discovery listeners. This event is referred to as a *discard event*.

If a lookup service is discarded because it was found to be unavailable, that lookup service will be made eligible for re-discovery. In this case, the process of discarding a lookup service can be viewed as a mechanism for the removal of stale entries in the managed set of lookup services. The discarded lookup service is effectively "marked" for re-discovery, removing the need for operations such as lease renewal attempts on a lookup service that is currently unavailable. Upon re-discovery of the discarded lookup service, the implementation object will process the re-discovered lookup service as if it were discovered for the first time.

If a lookup service is discarded because the group(s) or locator with which it is associated have been removed from their respective managed set, the lookup service is no longer eligible for discovery until those group(s) or that locator are again added to the appropriate managed set.

Remote Objects, Stubs and Proxies

The *Jini™ Technology Glossary* states that a *remote object* is an object whose methods can be invoked from a Java virtual machine (JVM), potentially on a different host. Furthermore, the glossary states that such an object is described by one or more *remote interfaces*.

When invoking methods remotely through Java Remote Method Invocation (RMI), it is useful to think of the invocation as consisting of two components: a client component and a server component. When the client component initiates a remote method call, the server component carries out the execution of the remote method, and RMI facilitates the necessary communication between the two parties. Note that when discussing concepts related to RMI, the term *server* (or *remote server*) is sometimes used in place of the term *remote object*.

In order to initiate the invocation of a remote method, the client must have access to an object referred to as the *stub* of the remote object. The stub is an object local to the client that acts as the "representative" of the remote object. The stub implements the same set of remote interfaces that the remote object implements. From the point of view of the client, the stub *is* the remote object. When the client invokes a method on the local stub, communication with the remote object occurs, resulting in the execution of the corresponding method in the remote object's JVM.

The term *proxy* is used extensively throughout this document. With respect to remote objects in general, and entities operating within a Jini application environment in particular, a proxy is simply an intermediary object through which one entity (the client) may request the invocation of the methods provided by another entity (the remote object or the service).

Proxies can take a number of different forms. Some proxies take the form of what is often referred to as a *smart proxy*. This type of proxy typically consists of a set of local methods and a set of one or more remote object references (stubs). Clients invoke one or more of the local methods to access the methods of the remote objects referenced in the proxy.

Another form that a proxy can take is that of the stub of a remote object. That is, all stubs are simply proxies to their corresponding remote objects. Except for the local methods `equals` and `hashCode`, this type of proxy consists of remote methods only.

Some proxies are implemented as *strictly local*. Proxies of this form consist of only local methods, each executing in the client's JVM. Unlike smart proxies, no remote invocations result when any method of a strictly local proxy is invoked.

Typically, Jini services provide a proxy having one of the forms described above. When a service registers with a lookup service, the service's proxy is copied (through serialization) into the lookup service. When a client looks up the service, the service's proxy is downloaded to the client. The client can then invoke the methods contained in the service's proxy. If the invoked method is a local method, then execution will occur in the JVM of the client. If the invoked method is a remote method (or results in a remote invocation), then execution is initiated in the client's JVM and, ultimately occurs in the JVM of the service.

Note that for the purposes of this document, the term *front-end proxy* (or simply, *front-end*) may be used interchangeably with the term proxy. Similarly, the term *back-end server* (or simply, *back-end*) may be used interchangeably with the term remote object. Thus, the back-end of a service is the part of the service's implementation that satisfies the contract advertised in the service's remote interface.

1.2.8 Activation

The glossary defines the term *active object* as a remote object that is instantiated and exported in a JVM on some system. Remote objects can be implemented with the ability to change their state from inactive to active, or from active to inactive; the process of doing so is referred to as *activation* and *deactivation*, respectively. Many Jini services wishing to conserve computational resources may find this capability desirable. When the back-end of any Jini service is implemented with the ability to activate and deactivate, the service is referred to as an *activatable service*. Refer to the *Java Remote Method Invocation Specification* for the details of activation.

1.3 Utility Interfaces

This document specifies a set of general purpose utility interfaces collectively referred to as the *Discovery Management Interfaces*. This set currently consists of the following three interfaces:

- `DiscoveryManagement`

- `DiscoveryGroupManagement`
- `DiscoveryLocatorManagement`

These interfaces specify sets of methods that define a mechanism for managing various aspects of an entity's discovery duties. Because these interfaces provide a uniform way to define utility classes that perform discovery-related management duties on behalf of a client or service, a number of the helper utility classes specified in this document employ one or more of these interfaces.

1.4 Helper Utilities

This document specifies the following Jini technology helper utility classes:

- `LookupLocatorDiscovery`
- `LookupDiscoveryManager`
- `LeaseRenewalManager`
- `JoinManager`
- `ClientLookupManager`

1.4.1 The `LookupLocatorDiscovery` Utility

The `LookupLocatorDiscovery` helper utility encapsulates the functionality required of an entity that wishes to employ the unicast discovery protocol to discover a Jini Lookup service. This utility provides an implementation that makes the process of finding specific instances of the Jini Lookup service much simpler for both services and clients.

1.4.2 The `LookupDiscoveryManager`

The `LookupDiscoveryManager` is a helper utility class that organizes and manages all discovery-related activities on behalf of a Jini client or service. That is, rather than providing its own facility for coordinating and maintaining all of the necessary state information related to group names, locators, and `net.jini.discovery.DiscoveryListener` objects, the entity can employ this class to provide those facilities on its behalf.

1.4.3 The `LeaseRenewalManager`

The `LeaseRenewalManager` helper utility class encapsulates functionality that provides for the coordination, systematic renewal, and overall management of a set of leases associated with some object on behalf of another object.

1.4.4 The `JoinManager`

The `JoinManager` is a helper utility class that performs all of the functions related to discovery, joining, service lease renewal, and attribute management which the Jini technology programming model requires of a well-behaved Jini service.

1.4.5 The `ClientLookupManager`

The `ClientLookupManager` class is a helper utility class that any client-like entity can use to create and populate a cache of service references, and with which the entity can register for notification of the availability of services of interest. Like the `JoinManager` utility class, this class needs to be notified when a desired lookup service is discovered. But unlike the `JoinManager`, the `ClientLookupManager` does not register the entity as a service with discovered lookup services. Although both the `JoinManager` and the `ClientLookupManager` perform lookup discovery event handling for the entities that employ them, the `JoinManager` performs join processing for Jini services, while the `ClientLookupManager` performs service discovery and management processing both for clients and for services.

The `ClientLookupManager` class can be asked to "discover" services an entity is interested in using, and to cache the references to those services as each is found. The cache can be viewed as a set of services that the entity can access through a set of public, non-remote methods.

The `ClientLookupManager` class also provides a mechanism for an entity to request notification when a service of interest is discovered for the first time or has encountered a state change (such as removal from all lookup services or attribute set changes).

For convenience, the `ClientLookupManager` class also provides versions of a method named `lookup`, which employs invocation semantics similar to the semantics of the `lookup` method of the `ServiceRegistrar` interface,

specified in the *Jini™ Lookup Service Specification*. Entities needing to find services on only an infrequent basis, or in which the cost of making a remote call is outweighed by the overhead of maintaining a local cache (e.g., due to limited resources), may find this method useful.

All three mechanisms described above — local queries on the cache, service discovery notification, and remote lookups — employ the same template-matching scheme as that described in the *Jini™ Lookup Service Specification*. Additionally, each mechanism allows the entity to supply an action object referred to as a *filter*. Such an object is a non-remote object that defines additional matching criteria that will be applied when searching for the entity's services of interest. This filtering facility is particularly useful to entities that wish to extend the capabilities of the standard template-matching scheme.

1.5 Helper Services

This document specifies the following Jini technology helper services:

- `LookupDiscoveryService`
- `LeaseRenewalService`
- `EventMailbox`

1.5.1 The `LookupDiscoveryService`

Under certain circumstances, a discovering entity may find it useful to allow a third party to perform the entity's discovery duties. For example, an activatable entity that wishes to deactivate may wish to employ a special Jini service — referred to as the `LookupDiscoveryService` — to perform discovery duties on behalf of the entity. Such an entity may wish to deactivate for various reasons, one being to conserve computational resources. While the entity is deactivated, the lookup discovery service, running on the same or a separate host, would employ the Jini Discovery protocols to find lookup services in which the entity has expressed interest, and would notify the entity when a previously unavailable lookup service becomes available.

1.5.2 The `LeaseRenewalService`

The `LeaseRenewalService` is a Jini service that can be employed by both Jini clients and services to perform all lease renewal duties on their behalf. Services that wish to remain inactive until needed may find the `LeaseRenewalService` quite useful. Such a service can request that the `LeaseRenewalService` take on the responsibility of renewing the leases granted to the service, and then safely deactivate without risking the loss of access to the resources corresponding to the leases being renewed.

Clients that have continuous *access* to a network, but which cannot be continuously *connected* to that network (e.g., a cell phone), may also find this service useful. By allowing a `LeaseRenewalService` (which can be continuously connected) to renew the leases on the resources acquired by the client, the client may remain disconnected until needed. This removes the need to perform the discovery and lookup process each time the client re-connects to the network; possibly resulting in a significant increase in efficiency.

1.5.3 The `EventMailbox`

The `EventMailbox` service is a Jini service that can be employed by Jini clients and services to store event notifications on their behalf. When an entity registers with the `EventMailbox` service, that service will collect events intended for the registered entity until the entity initiates delivery of the events.

A service such as the `EventMailbox` can be particularly useful to entities that desire more control over the delivery of the events sent to them. Some entities operating in a distributed system may find it undesirable or inefficient to be contacted solely for the purpose of having an event delivered; preferring to defer the delivery to a time that is more convenient, as determined by the entity itself.

For example, an entity wishing to deactivate or detach from a network may wish to have its events stored until the entity is available to retrieve them. Additionally, some entities may wish to batch process event notifications for efficiency. In both scenarios, the entities described may find the `EventMailbox` service useful in achieving the respective event delivery goals.

1.6 Dependencies

This specification relies on the following specifications:

- Java™ Remote Method Invocation Specification
- Java™ Object Serialization Specification
- Jini™ Technology Glossary
- Jini™ Distributed Event Specification
- Jini™ Distributed Leasing Specification
- Jini™ Discovery and Join Specification
- Jini™ Discovery Utilities Specification
- Jini™ Lookup Service Specification
- Jini™ Lookup Attribute Schema Specification
- Jini™ Transaction Specification

1

The Discovery Management Interfaces 2

2.1 Overview

Discovery is one behavior that is common to all entities wishing to interact with a Jini Lookup service. Whether an entity is a client, a service, or a service acting as a client, before the entity can begin interacting with a Jini Lookup service, the entity must first discover that lookup service.

The interfaces referred to collectively as the *discovery management* interfaces specify sets of methods that define a mechanism that may be used to manage various aspects of the discovery duties of entities that wish to participate in a Jini application environment. These interfaces provide a uniform way to define utility classes that perform the necessary discovery-related management duties on behalf of a client or service. Currently, there are three discovery management interfaces (belonging to the package `net.jini.discovery`):

- `DiscoveryManagement`
- `DiscoveryGroupManagement`
- `DiscoveryLocatorManagement`

The `DiscoveryManagement` interface defines semantics for methods related to the discovery event mechanism and discovery process termination. Through this interface an entity can register or un-register for discovery events, discard a lookup service, or terminate the discovery process.

The `DiscoveryGroupManagement` interface defines methods related to the management of the sets of lookup services that are to be discovered using the multicast discovery protocol (as defined in the *Jini™ Discovery and Join Specification*). The methods of this interface define how an entity retrieves or modifies the set of groups associated with those lookup services.

The `DiscoveryLocatorManagement` interface defines methods related to the management of the set of lookup services that are to be discovered using the unicast discovery protocol (as defined in the *Jini™ Discovery and Join Specification*). The methods of this interface define how an entity retrieves or modifies the set of `LookupLocator` objects associated with those lookup services.

Although each interface defines semantics for methods involved in the management of the discovery process, the individual roles each interface plays in that process are independent of each other. Because of this independence, there may be scenarios where it is desirable to implement some subset of these interfaces.

For example, a class may wish to implement the functionality defined in `DiscoveryManagement`, but may not wish to allow entities to modify the groups and locators associated with the lookup services to be discovered. Such a class may have a "hard-coded" list of the groups and locators that it internally registers with the discovery process. For this case, the class would implement only `DiscoveryManagement`.

Alternatively, another class may not wish to allow the entity to register more than one listener with the discovery event mechanism; nor may it wish to allow the entity to terminate discovery. It may simply wish to allow the entity to modify the sets of lookup services that will be discovered. A class such as this would implement both `DiscoveryGroupManagement` and `DiscoveryLocatorManagement`, but not `DiscoveryManagement`.

Throughout this chapter, the phrase *implementation class* refers to any concrete class which implements one or more of the discovery management interfaces, and the phrase *implementation object* should be understood as an instance of such an implementation class. Additionally, whenever a description refers to the *discovering entity* (or simply, the *entity),* that phrase should be taken to mean the object (the client or service) that has created an implementation object; and which wishes to avail itself of the public methods specified by these interface(s) and provided by that object.

2.2 Other Types

The types defined in the specification of the discovery management interfaces are in the `net.jini.discovery` package. The following 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.lookup.ServiceRegistrar
java.io.IOException
```

2.3 The `DiscoveryManagement` Interface

The public methods specified by the `DiscoveryManagement` interface are as follows:

```
package com.sun.jini.discovery;

public interface DiscoveryManagement
{
    public void addDiscoveryListener
                                    (DiscoveryListener listener);
    public void removeDiscoveryListener
                                    (DiscoveryListener listener);

    public ServiceRegistrar[] getRegistrars();

    public void discard(ServiceRegistrar proxy);
    public void terminate();
}
```

2.3.1 The Semantics

The `DiscoveryManagement` interface defines methods related to the discovery event mechanism and discovery process termination. Through this interface an entity can register or un-register `DiscoveryListener` objects to receive discovery events (instances of `DiscoveryEvent`), discard a lookup service so that it is eligible for re-discovery, or terminate the discovery process.

Implementation classes of this interface may impose additional semantics on any method. For example, such a class may choose to require that rather than simply terminate discovery processing, the `terminate` method additionally must cancel all leases held by the implementation object and terminate all lease management being performed on behalf of the entity.

For information on any additional semantics imposed on a method of this interface, refer to the specification of the particular implementation class.

Note that `DiscoveryEvent` and `DiscoveryListener` are both defined in the *Jini™ Discovery Utilities Specification*.

- The `addDiscoveryListener` method adds a listener to the set of objects listening for discovery events (`DiscoveryEvent`). This method takes a single argument as input: the listener (`DiscoveryListener`) to add to the set.

  Once a listener is registered, it will be notified of all lookup services discovered to date, and will then be notified as new lookup services are discovered or existing lookup services are discarded.

  If `null` is input to this method, no action will be taken.

- The `removeDiscoveryListener` method removes a listener from the set of objects listening for discovery events. This method takes a single argument as input: the listener to remove from the set.

  If the listener object input to this method does not exist in the set of listeners maintained by the implementation class, then this method will take no action.

Note – Should pending events be addressed with respect to invocations of `removeDiscoveryListener`? For example, should a requirement like the following be made: "Once an invocation of `removeDiscoveryListener` returns, it is guaranteed that none of the removed listener(s) will receive any pending notification events." This question is asked because some of the current implementations of the discovery management interfaces (e.g., `LookupDiscovery`, `LookupLocatorDiscovery`, `LookupDiscoveryManager`) queue the references to the listeners along with the pending events (for efficiency). So, even if a listener is removed, a reference to it still exists in the event queue; and the listener may be notified when the event finally gets to the front of the queue. So should this requirement be specified — forcing existing implementations to be changed — or is it too implementation specific to include here?

- The `getRegistrars` method returns an array of `ServiceRegistrar` objects, each corresponding to a lookup service that has been discovered. Each time this method is invoked, a new array is returned. This method takes no arguments as input.
- The `discard` method removes a particular lookup service from the managed set of lookup services, and makes that lookup service eligible to be re-discovered. This method takes a single argument as input: the proxy to the lookup service to discard.

If the proxy input to this method is `null`, or if it matches none of the lookup services in the managed set, this method takes no action.

Note that once a lookup service is discovered, there is no requirement for on-going communication between the discovered lookup service and the implementation object. This means that if a lookup service goes away, there may be no automatic notification of the occurrence of such an event. Thus, if an entity encounters an unavailable lookup service, it is the responsibility of the entity to invoke the `discard` method.

Invoking the `discard` method defined by the `DiscoveryManagement` interface will result in the flushing of the lookup service from the appropriate cache, ultimately causing a discard notification to be sent to all `DiscoveryListener` objects registered with the implementation object.

- The `terminate` method ends all discovery processing being performed on behalf of the entity. This method takes no input arguments.

  After this method has been invoked, no new lookup services will be discovered, and the effect of any new operations performed on the current implementation object are undefined.

  Any additional termination semantics must be defined by the implementation class.

2.4 The `DiscoveryGroupManagement` Interface

The public methods specified by the `DiscoveryGroupManagement` interface are as follows:

```
package com.sun.jini.discovery;

public interface DiscoveryGroupManagement
{
    public String[] getGroups();
    public void addGroups(String[] groups) throws IOException;
    public void setGroups(String[] groups) throws IOException;
    public void removeGroups(String[] groups);
}
```

2.4.1 The Semantics

The `DiscoveryGroupManagement` interface defines methods related to the management of the sets of lookup services that are to be discovered using the multicast discovery protocol; that is, lookup services that are discovered by way of group discovery. The methods of this interface define how an entity retrieves or modifies the set of groups associated with those lookup services.

The methods that modify the managed set of groups each take a single input parameter: a `String` array, none of whose elements may be `null`. The empty set is denoted by the empty array, and "no set" is indicated by `null`. No set indicates that all lookup services within the *multicast radius* of the implementation object should be discovered. Invoking any of these methods with an input array that contains duplicate group names is equivalent to performing the invocation with the duplicates removed from the array.

- The `getGroups` method returns an array consisting of the names of the groups in the managed set. If the managed set of groups is empty, this method will return the empty array. This method takes no arguments as input, and will return a new array upon each invocation.
- The `addGroups` method adds a set of group names to the managed set. The array input to this method contains the group names that are to be added.

Once a new name is added to the managed set, attempts will be made to discover all (as yet) undiscovered lookup services that are members of the group having that name.

The entity must have `DiscoveryPermission` on each of the groups in the new set or a `SecurityException` will be propagated through this method.

This method throws `IOException`. This is because an invocation of this method may result in the re-initiation of the discovery process, a process that can throw `IOException` when socket allocation occurs.

This method throws an `UnsupportedOperationException` if there is no managed set of groups to augment. If `null` is input, this method throws a `NullPointerException`. If the empty array is input, the managed set of groups will not change.

- The `setGroups` method replaces all of the group names in the managed set with names from a new set. The array input to this method contains the group names that will replace the current names in the managed set.

  Once a new group name has been placed in the managed set, the entity will not be notified for lookup services belonging to that group that have already been discovered. Attempts to discover all (as yet) undiscovered lookup services belonging to that group will continue to be made.

  If `null` is input to `setGroups`, then attempts will be made to discover all (as yet) undiscovered lookup services located within the multicast radius of the implementation object. If the empty array is input, then group discovery will cease.

  The entity must have `DiscoveryPermission` on each of the groups in the new set or a `SecurityException` will be propagated through this method.

  This method throws `IOException`. This is because an invocation of this method may result in the re-initiation of the discovery process, a process that can throw `IOException` when socket allocation occurs.

- The `removeGroups` method deletes a set of group names from the managed set of groups. The array input to this method contains the group names that will be removed from the managed set.

This method throws an `UnsupportedOperationException` if there is no managed set of groups from which to remove elements. If `null` is input to `removeGroups`, a `NullPointerException` will be thrown. If the empty array is input, the managed set of groups will not change.

Any already-discovered lookup service that is a member of one or more group(s) removed from the managed set by either `setGroups` or `removeGroups` will be discarded and will not be eligible for discovery; but only if it satisfies both of the following conditions:

- the lookup service is not a member of any group in the new managed set resulting from the invocation of `setGroups` or `removeGroups`, and
- the lookup service is not currently eligible for discovery through other means (such as locator discovery).

2.5 The `DiscoveryLocatorManagement` Interface

The public methods specified by the `DiscoveryLocatorManagement` interface are as follows:

```
package com.sun.jini.discovery;

public interface DiscoveryLocatorManagement
{
    public LookupLocator[] getLocators();
    public void addLocators(LookupLocator[] locators);
    public void setLocators(LookupLocator[] locators);
    public void removeLocators(LookupLocator[] locators);
}
```

2.5.1 The Semantics

The `DiscoveryLocatorManagement` interface defines methods related to the management of the set of lookup services that are to be discovered using the unicast discovery protocol; that is, lookup services that are discovered by way of locator discovery. The methods of this interface define how an entity retrieves or modifies the set of locators associated with those lookup services.

The methods that modify the managed set of locators each take a single input parameter: an array of locators, none of whose elements may be `null`. Invoking any of these methods with an input array that contains duplicate locators (as determined by `LookupLocator.equals`) is equivalent to performing the invocation with the duplicates removed from the array.

- The `getLocators` method returns an array containing the set of `LookupLocator` objects in the managed set of locators. The returned set will include both the set of `LookupLocator` objects corresponding to lookup services that have already been discovered as well as the set of those that have not yet been discovered. If the managed set is empty, this method will return the empty array. This method takes no arguments as input, and will return a new array upon each invocation.
- The `addLocators` method adds a set of locators to the managed set. The array input to this method contains the set of `LookupLocator` objects that will be added to the managed set.

This method throws an `UnsupportedOperationException` if there is no managed set of locators to augment. If `null` is input to `addLocators`, a `NullPointerException` will be thrown. If the empty array is input, the managed set of locators will not change.

- The `setLocators` method replaces all of the locators in the managed set with `LookupLocator` objects from a new set. The array input to this method contains the set of `LookupLocator` objects that will replace the locators in the managed set.

  If `null` is input to `setLocators`, a `NullPointerException` will be thrown. If the empty array is input, locator discovery will cease.

- The `removeLocators` method deletes a set of locators from the managed set. The array input to this method contains the set of `LookupLocator` objects that will be removed from the managed set.

  This method throws an `UnsupportedOperationException` if there is no managed set of locators from which to remove elements. If `null` is input to `removeLocators`, a `NullPointerException` will be thrown. If the empty array is input, the managed set of locators will not change.

Any already-discovered lookup service corresponding to a locator that is removed from the managed set by either `setLocators` or `removeLocators` will be discarded and will not be eligible for discovery; but only if it is not currently eligible for discovery through other means (such as group discovery).

The `LookupLocatorDiscovery` Utility

3

3.1 Overview

The *Jini™ Discovery and Join Specification* states that the "unicast discovery protocol is a simple request-response protocol". In a Jini application environment, the entities that participate in this protocol are a Jini client or service acting as a discovering entity and a Jini Lookup service which acts as the entity that is to be discovered. The client or service sends unicast discovery requests to the lookup service; and the lookup service reacts to those requests by sending unicast discovery responses to the interested client or service.

The `LookupLocatorDiscovery` helper utility (belonging to the package `com.sun.jini.discovery`) encapsulates the functionality required of an entity that wishes to employ the unicast discovery protocol to discover a Jini Lookup service. This utility provides an implementation that makes the process of finding specific instances of the Jini Lookup service much simpler for both services and clients.

Because the `LookupLocatorDiscovery` helper utility class will participate in only the unicast discovery protocol, and because the unicast discovery protocol imposes no restriction on the physical location of a service or client relative to a lookup service, this utility can be used to discover lookup services running on hosts that are located far from, or near to, the hosts on which the service is running. This lack of a restriction on location brings with it a requirement that the discovering entity supply specific information about the desired lookup services to the `LookupLocatorDiscovery` utility; namely, the location of the

device(s) hosting each lookup service. This information is supplied through an instance of the `LookupLocator` utility, defined in the *Jini™ Discovery and Join Specification*.

It may be of value to note the difference between `LookupLocatorDiscovery` and the utility `LookupDiscovery` defined in the *Jini™ Discovery Utilities Specification*. Although both are non-remote utility classes that entities can use to discover at least one lookup service, the `LookupLocatorDiscovery` utility is designed to provide discovery capabilities that satisfy different needs than those satisfied by the `LookupDiscovery` utility. These two utilities differ in the following ways:

- Whereas the `LookupLocatorDiscovery` utility is used to discover lookup services by their *locators*, employing the unicast discovery protocol, the `LookupDiscovery` utility uses the multicast discovery protocol to discover lookup services by the *groups* to which the lookup services belong.
- Whereas the `LookupLocatorDiscovery` utility requires that the discovering entity supply the specific location — or address — of the desired lookup service(s) in the form of a `LookupLocator` object, the `LookupDiscovery` utility imposes no such restriction on the discovering entity.
- Whereas the `LookupLocatorDiscovery` utility can be used by a discovering entity to discover lookup services that are both “near” and “far”, the `LookupDiscovery` utility can be used to discover only those lookup services that are located within the same *multicast radius* as that of the discovering entity.

3.2 Other Types

The types defined in the specification of the `LookupLocatorDiscovery` utility class are in the `com.sun.jini.discovery` package. The following 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.discovery.DiscoveryManagement
net.jini.discovery.DiscoveryLocatorManagement
net.jini.discovery.LookupDiscovery
```

3.3 The Interface

The public methods provided by the `LookupLocatorDiscovery` class are as follows.

```
package com.sun.jini.discovery;

public final class LookupLocatorDiscovery
                        implements DiscoveryManagement
                                   DiscoveryLocatorManagement
{
    public LookupLocatorDiscovery(LookupLocator[] locators);

    public LookupLocator[] getDiscoveredLocators();
    public LookupLocator[] getUndiscoveredLocators();
}
```

3.4 The Semantics

Each instance of the `LookupLocatorDiscovery` class must behave as if it operates independently of all other instances.

The `equals` method for this class returns true if and only if two instances of this class refer to the same object. That is, x and y are equal instances of this class if and only if x == y has the value `true`.

- The constructor of the `LookupLocatorDiscovery` class takes a single input parameter: a set of locators, none of whose elements may be `null`. Each element in the input set corresponds to a specific lookup service the discovering entity wishes to be discovered. This set is represented as an array of `LookupLocator` objects.

  Invoking the constructor with an input array that contains duplicate locators (as determined by `LookupLocator.equals`) is equivalent to performing the invocation with the duplicates removed from the array.

  Discovery typically starts as soon as an instance of this class is created, and ends when the `terminate` method is called. However, if `null` or the empty set is passed to the constructor, discovery will not be started until either the `setLocators` or `addLocators` method is called with a non-empty set.

- The `getDiscoveredLocators` method returns the set of `LookupLocator` objects representing the desired lookup services that have already been discovered. If the set is empty, this method will return the empty array. This method takes no arguments as input, and will return a new array upon each invocation.
- The `getUndiscoveredLocators` method returns the set of `LookupLocator` objects representing the desired lookup services that have not yet been discovered. If the set is empty, this method will return the empty array. This method takes no arguments as input, and will return a new array upon each invocation.

3.5 Supporting Interfaces

The `LookupLocatorDiscovery` utility class depends on the following interfaces: `DiscoveryManagement` and `DiscoveryLocatorManagement`.

3.5.1 The `DiscoveryManagement` Interfaces

The `LookupLocatorDiscovery` class implements both the `DiscoveryManagement` and the `DiscoveryLocatorManagement` interfaces, which together define methods related to the coordination and managment of all discovery processing. Refer to the chapter of this specification titled *The Discovery Management Interfaces*.

The `LookupLocatorDiscovery` class defines no additional semantics for any method specified by either of these interfaces.

The LookupDiscoveryManager 4

4.1 Overview

Although the goals of any well-behaved Jini client or service are application-specific, the goals of such entities with respect to their interaction with Jini Lookup services generally begin with employing the Jini Discovery protocols (defined in the *Jini™ Discovery and Join Specification*) to obtain a reference to at least one Jini Lookup service. Because the discovery duties performed by such entities may require the management of significant amounts of state information, those duties can become quite tedious.

The `LookupDiscoveryManager` is a helper utility class (belonging to the package `com.sun.jini.discovery`) that organizes and manages all discovery-related activities on behalf of a Jini client or service. Rather than providing its own facility for coordinating and maintaining all of the necessary state information related to group names, `LookupLocator` objects, and `DiscoveryListener` objects, such entities can employ this class to provide those facilities on its behalf.

Note that throughout this specification, two terms will be used interchangeably when referring to Jini clients and services that create an instance of the `LookupDiscoveryManager` and avail themselves of that class' methods. Those terms are: the *discovering entity* or simply, the *entity.*

4.2 Other Types

The types defined in the specification of the `LookupDiscoveryManager` utility class are in the `com.sun.jini.discovery` package. The following 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.lookup.ServiceRegistrar
net.jini.discovery.DiscoveryEvent
net.jini.discovery.DiscoveryListener
net.jini.discovery.DiscoveryManagement
net.jini.discovery.DiscoveryGroupManagement
net.jini.discovery.DiscoveryLocatorManagement
java.io.IOException
java.util.EventListener
java.util.EventObject
```

4.3 The Interface

The only new public method of the `LookupDiscoveryManager` utility class is the constructor. All other public methods implemented by this class are specified in the discovery management interfaces.

```
package com.sun.jini.discovery;

public class LookupDiscoveryManager
                        implements DiscoveryManagement,
                                   DiscoveryGroupManagement,
                                   DiscoveryLocatorManagement
{
    public LookupDiscoveryManager(String[] groups,
                                  LookupLocator locators,
                                  DiscoveryListener listener)
                                           throws IOException;
}
```

4.4 The Semantics

The `equals` method for this class returns true if and only if two instances of this class refer to the same object. That is, x and y are equal instances of this class if and only if x == y has the value `true`.

- The constructor for the `LookupDiscoveryManager` takes the following arguments as input:
  - A `String` array, none of whose elements may be `null`, and in which each element is the name of a group whose members are lookup services the entity wishes to be discovered
  - An array of `LookupLocator` objects, none of whose elements may be `null`, and in which each element corresponds to a specific lookup service the entity wishes to be discovered
  - A reference to a `DiscoveryListener` object that will be notified when a targeted lookup service is discovered or discarded

The LookupDiscoveryManager will, on behalf of the client, employ the Jini Discovery protocols defined in the *Jini™ Discovery and Join Specification* to find the lookup services associated with the first two arguments, and will maintain and manage the discovered lookup services from both the input set of groups and the input set of locators.

If an empty or null array is input to the groups argument, no lookup service will be discovered using the Jini multicast discovery protocol.

If an empty or null array is input to the locators argument, no lookup service will be discovered using the Jini unicast discovery protocol.

If the constructor 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 calling LookupLocator.equals), the invocation is equivalent to constructing an instance of LookupDiscoveryManager with no duplicates in either set.

The last argument to the constructor is a reference to a listener object that will be registered to receive discovery event notifications. If a null reference is input to this argument, then the client will receive no discovery events. It is the responsibility of the client to create and pass into the LookupDiscoveryManager an object that implements the DiscoveryListener interface. That implementation must provide the definition of the actions to take upon receipt of any notifications.

This constructor throws IOException. This is because construction of a LookupDiscoveryManager may initiate the multicast discovery process, a process that can throw IOException.

4.5 Supporting Interfaces and Classes

The LookupDiscoveryManager depends on the following interfaces: DiscoveryManagement, DiscoveryGroupManagement, DiscoveryLocatorManagement, and DiscoveryListener. This class also depends (indirectly) on one concrete class: DiscoveryEvent.

4.5.1 The `DiscoveryManagement` Interfaces

The `LookupDiscoveryManager` implements the `DiscoveryManagement`, `DiscoveryGroupManagement`, and `DiscoveryLocatorManagement` interfaces which together define methods related to the coordination and managment of all discovery processing. Those interfaces are defined in the chapter of this specification titled *The Discovery Management Interfaces*. The `LookupDiscoveryManager` class defines no additional semantics for any method specified by the discovery management interfaces.

4.5.2 The `DiscoveryListener` Interface

The `DiscoveryListener` interface defines the mechanism through which an entity receives notification from discovery utility objects such as `LookupDiscovery` or `LookupLocatorDiscovery` that a lookup service has been discovered or discarded. This interface is specified in the *Jini™ Discovery Utilities Specification*. It is presented here for reference.

```
package net.jini.discovery;

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

4.5.3 The `DiscoveryEvent` Class

The `DiscoveryEvent` class defines the object passed to interested entities to indicate that one or more `ServiceRegistrar` objects (lookup services) have been discovered or discarded by a discovery utility object such as `LookupDiscovery` or `LookupLocatorDiscovery`. This interface is specified in the *Jini™ Discovery Utilities Specification*. It is presented here for reference.

```
package net.jini.discovery;

public class DiscoveryEvent extends EventObject
{
    public DiscoveryEvent(Object source,
                          ServiceRegistrar[] regs);

    public ServiceRegistrar[] getRegistrars();
}
```

The `LeaseRenewalManager` 5

5.1 Overview

The `LeaseRenewalManager` helper utility class (belonging to the package `com.sun.jini.lease`) encapsulates functionality which provides for the coordination, systematic renewal, and overall management of a set of leases associated with some object on behalf of another object.

The concept of *leased* resources is fundamental to the Jini technology programming model. Providing a leasing mechanism helps to prevent the accumulation of outdated and unwanted resources in time-based distributed systems such as the Jini system. The Jini technology leasing model, defined in the *Jini™ Distributed Leasing Specification*, requires renewed proof of interest to continue the existence of a leased resource. Thus, any Jini client or service that registers with another Jini service providing leased resources may be granted access to those resources for a negotiated period of time; and must continue to request renewal of the lease on each resource for as long as the client or service wishes to have access to the resource.

For example, the Jini Lookup service leases two resources: residency in its database and registration with its event notification mechanism. Thus, if a service that is registered with a Jini Lookup service wishes to continue its residency beyond the length of the negotiated lease, the service must request a lease renewal from that lookup service. This renewal process must be repeated for as long as the service wishes to maintain its residency in the lookup. Similarly, if a registered service has requested that the lookup service notify it of events of interest, then prior to the expiration of the lease on the event

mechanism, the service must request that the lookup service continue to send such events. As with residency in the lookup service, these renewal requests must be repeated for as long as the service wishes to receive event notifications.

Another example of a Jini service providing leased resources would be a service that implements the *Jini™ Transaction Specification* to manage transactions on behalf of registered participants. That specification requires that a transaction must be a leased resource. Therefore, any entity that creates such a transaction object is required to negotiate (with an entity referred to as a *transaction manager*) a lease on that object; repeatedly requesting lease renewals prior to the lease's expiration, for as long as the transaction is to remain in effect.

The `LeaseRenewalManager` class is designed to be a simple mechanism that provides for the systematic renewal and overall management of leases placed on resources that are advertised by Jini services, for which a Jini client or service has registered interest. This class is a utility class, not a remote service. In order to use this utility, an entity must create, in its own address space, an instance of the `LeaseRenewalManager` to manage the entity's leases locally.

5.2 Other Types

The types defined in the specification of the `LeaseRenewalManager` utility class are in the `com.sun.jini.lease` package. The following types may be referenced in this chapter. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.lease.Lease
net.jini.core.lease.UnknownLeaseException
net.jini.core.lease.LeaseDeniedException
java.rmi.RemoteException
java.rmi.NoSuchObjectException
java.util.EventObject
java.util.EventListener
```

5.3 The Interface

The public methods provided by the `LeaseRenewalManager` class are as follows.

```
package com.sun.jini.lease;

public class LeaseRenewalManager
{
    public LeaseRenewalManager();
    public LeaseRenewalManager(Lease lease,
                               long expiration,
                               LeaseListener listener);

    public void renewUntil(Lease lease,
                           long expiration,
                           LeaseListener listener);
    public void renewFor(Lease lease,
                         long duration,
                         LeaseListener listener);

    public long getExpiration(Lease lease)
                throws UnknownLeaseException;
    public void setExpiration(Lease lease, long expiration)
                throws UnknownLeaseException;
    public void remove(Lease lease) throws UnknownLeaseException;
    public void cancel(Lease lease)
                throws UnknownLeaseException, RemoteException;
    public void clear();
}
```

5.4 The Semantics

This class distinguishes between two time values associated with lease expiration: the *desired* time of expiration for the lease; and the *actual* time of expiration granted when the lease is created or renewed. Both time values are absolute times, not relative time durations. The actual expiration time of a lease object can be retrieved by invoking the `getExpiration` method on the lease (see the `Lease` interface defined in the *Jini™ Distributed Leasing Specification*).

Methods of this class that accept `Lease.FOREVER` or `Lease.ANY` as the desired expiration time will treat `Lease.FOREVER` and `Lease.ANY` as semantically identical. That is, if either `Lease.FOREVER` or `Lease.ANY` is requested, renewal requests will be made indefinitely until a cancellation request is made. Although these values are not distinguished in the semantics of the `LeaseRenewalManager` itself, they may ultimately be distinguished in the semantics of the entity that granted the lease. Whenever the `LeaseRenewalManager` requests that a lease-granting entity renew a lease to expire at some absolute time, the value of the desired lease expiration time (whatever that value may be — `Lease.FOREVER`, `Lease.ANY`, or any other value) will be passed on to that lease-granting entity; leaving it to the lease-granting entity to define the semantics of the requested lease expiration time.

The `LeaseRenewalManager` makes certain concurrency guarantees. When the `LeaseRenewalManager` makes a remote call (for example, when requesting the renewal of a lease), any invocations made on the methods of the `LeaseRenewalManager` will not be blocked.

Similarly, the `LeaseRenewalManager` makes a re-entrancy guarantee with respect to `LeaseListener` objects registered with the `LeaseRenewalManager`. Should the `LeaseRenewalManager` invoke a method on a registered listener (a local call), calls from that method to any method of the `LeaseRenewalManager` are guaranteed not to result in a deadlock condition.

The `equals` method for this class returns true if and only if two instances of this class refer to the same object. That is, x and y are equal instances of this class if and only if x == y has the value `true`.

The individual semantics of each method of this class are described below.

- The constructor has two forms:
  - The first form of the constructor takes no arguments. This form of the constructor instantiates a `LeaseRenewalManager` object that initially manages no leases.
  - The second form of the constructor creates a `LeaseRenewalManager` that initially manages a single lease. This form of the constructor requires that a reference to the initial lease be supplied as an argument.

This constructor also takes an `expiration` argument that represents the *desired* (absolute) time of expiration for the lease, and a reference to a `LeaseListener` object that receives notifications of exceptional conditions occurring during renewal attempts.

Creating a `LeaseRenewalManager` using this form of the constructor is equivalent to invoking the no-argument constructor followed by an invocation of the `renewUntil` method (described below).

- The `renewUntil` method adds a lease to the set of leases being managed by the `LeaseRenewalManager`. This method takes as arguments a reference to the lease to manage, the desired time of expiration of the lease, and a reference to the `LeaseListener` object that will receive notification of exceptional conditions when attempting lease renewal. A value of `Lease.ANY` or `Lease.FOREVER` may be passed as the value of the `expiration` argument, and the `LeaseListener` argument may be `null`.

  If the lease input to this method is already in the set of managed leases, the listener object and the desired expiration associated with that lease will be replaced with the new listener and expiration. The lease will remain in the set of managed leases until one of the following occurs:

  - The lease expires
  - The lease is cancelled
  - An explicit removal of the lease from the set is requested
  - An `UnknownLeaseException`, a `LeaseDeniedException`, or any other non-remote exception is received during a lease renewal attempt
  - A (remote) `NoSuchObjectException` is received during a lease renewal attempt

  This method will interpret the expiration time argument as the *desired* absolute system time after which the lease is no longer valid. This argument provides the ability to indicate an expiration time that extends beyond the actual expiration of the lease. If the value input to this argument does indeed extend beyond the actual lease expiration time, then the lease will be systematically renewed at appropriate times until one of the conditions listed above occurs. If the value input is less than or equal to the actual expiration time, nothing will be done to modify the time when the lease actually expires. That is, the lease will *not* be renewed with an expiration time that is less than the originally granted expiration.

If a non-`null` object reference is passed in for the `LeaseListener` argument, the object will receive notification of exceptional conditions occurring upon a renewal attempt. In particular, exceptional conditions may include `UnknownLeaseException`, `LeaseDeniedException`, and `RemoteException`, as well as any other non-`RemoteException`. If a `RemoteException` occurs during a renewal request for a lease, renewal requests will continue to be made until either the lease is renewed, or the lease expiration time has been exceeded. In the latter case, the first `RemoteException` received will be passed on to the `LeaseListener`.

- The `renewFor` method adds a lease to the set of leases being managed by the `LeaseRenewalManager`. This method takes as input a reference to the lease to manage, a `long` value representing the desired duration of the lease, and a reference to a `LeaseListener` object that will receive notifications of exceptional conditions when attempting lease renewal.

  This method will test for arithmetic overflow in the duration parameter. Should such overflow be present, the duration will be set to `Lease.FOREVER`.

  The semantics of this method are identical to those of the `renewUntil` method described above, with the expiration parameter of `renewUntil` set to a value of *duration + current time (*in milliseconds).

- The `getExpiration` method returns the desired time of expiration requested for a particular lease, not the actual expiration that was originally granted when the lease was created. The only argument to this method is the reference to the lease object. If the lease is not in the set of managed leases, an `UnknownLeaseException` will be thrown.

- The `setExpiration` method replaces the current desired expiration of a given lease contained in the set of managed leases with a new desired expiration time. The only arguments to this method are the reference to the lease object and the new expiration time. If the lease is not in the set of managed leases, an `UnknownLeaseException` will be thrown.

- The `remove` method removes a given lease from the set of managed leases. The only argument to this method is the reference to the lease object. If the lease is not in the set of managed leases, an `UnknownLeaseException` will be thrown.

- The `cancel` method both removes a given lease from the set of managed leases and cancels the given lease. The only argument to this method is the reference to the lease object. If the lease is not in the set of managed leases, an `UnknownLeaseException` will be thrown.

  Any exception (remote or non-remote) occurring during the cancellation of the lease will have no effect on the removal of the lease from the managed set. That is, even if an exception occurs during the `cancel` operation, the lease must have been removed from the managed set upon return from this method.

  Any exception thrown by the `cancel` method of the lease object itself may also be thrown by this method.

- The `clear` method removes all leases from the set of managed leases. It does not request the cancellation of those leases. This method takes no arguments.

5.5 Supporting Interfaces and Classes

The `LeaseRenewalManager` utility class depends on one interface: `LeaseListener`, which references one class, `LeaseRenewalEvent`.

5.5.1 The `LeaseListener` Interface

The `LeaseListener` interface defines the mechanism through which an entity receives notification from the `LeaseRenewalManager` that either of the following events has occurred:

- A lease has expired
- A lease renewal request has been denied before the desired absolute expiration time of the lease was reached

It is the responsibility of the entity to pass into the `LeaseRenewalManager` a reference to an object that implements the `LeaseListener` interface, which defines the actions to take upon receipt of a notification. When either of the above events occurs, the `LeaseRenewalManager` will send an instance of `LeaseRenewalEvent` to that listener object.

```
package com.sun.jini.lease;

public interface LeaseListener extends EventListener
{
    void notify(LeaseRenewalEvent e);
}
```

With respect to the lease referenced by the event sent to the listener, the `LeaseRenewalManager` will remove that lease prior to sending the `LeaseRenewalEvent`.

5.5.2 The `LeaseRenewalEvent` Class

When a managed lease expires or has been denied renewal prior to its expiration time, the `LeaseRenewalManager` will send to the listener registered by the entity, an event represented by an instance of the `LeaseRenewalEvent` class. The `LeaseRenewalEvent` class encapsulates the information the `LeaseRenewalManager` associates with the affected lease.

```
package com.sun.jini.lease;

public interface LeaseRenewalEvent extends EventObject
{
    public LeaseRenewalEvent(LeaseRenewalManager source,
                             Lease lease,
                             long expiration,
                             Exception ex);

    public Lease getLease();
    public long getExpiration();
    public Exception getException();
}
```

The LeaseRenewalEvent class is a subclass of the class EventObject, adding the following additional items of abstract state: a reference to the affected Lease object; a long value representing the desired absolute expiration of the affected lease; and the exception (if any) that caused the event to be sent. In addition to the methods of the EventObject class, this class defines methods through which this additional state may be retrieved.

The Semantics

The getLease method returns a reference to the Lease object associated with the event. This method takes no arguments.

The getExpiration method returns a long value representing the desired absolute expiration of the Lease object associated with the event. This method takes no input arguments.

The getException method returns the exception that caused the event to be sent. This method takes no input arguments. The conditions under which a LeaseRenewalEvent may be sent, and the related values returned by this method, are as follows:

- If the lease's expiration time is reached before the LeaseRenewalManager can request renewal, a LeaseRenewalEvent will be sent with no associated exception. In this case, invoking this method will return null.
- If the LeaseRenewalManager sends a LeaseRenewalEvent because a renewal request was denied, this method will return LeaseDeniedException.
- If the LeaseRenewalManager sends a LeaseRenewalEvent because a renewal request could not be completed before the expiration of the lease, this method will return UnknownLeaseException.

Note that both LeaseDeniedException and UnknownLeaseException are defined in the *Jini™ Distributed Leasing Specification*.

The `JoinManager` 6

6.1 Overview

A goal of any well-behaved Jini service, implemented within the bounds defined by the Jini technology programming model, is to advertise the service it provides by requesting residency within at least one Jini Lookup service. Making such a request of a Jini Lookup service is known as registering with, or *joining*, a lookup service. To demonstrate this good behavior, a service must comply with both the *multicast discovery protocol* and the *unicast discovery protocol* in order to discover the lookup services it is interested in joining. The service must also comply with the *join protocol* to register with the desired lookup services. The details of the discovery and join protocols are described in the *Jini™ Discovery and Join Specification*.

In order for the service to maintain its residency in the lookup services it has joined, the service must provide for the coordination, systematic renewal, and overall management of all leases on that residency. In addition to handling all discovery and join duties, as well as managing all leases on lookup residency, the service must provide for the coordination and management of any attribute sets with which it may have registered.

With respect to the duties described above, a Jini service may perform all but the attribute set management duties through the use of the `LookupDiscoveryManager` and `LeaseRenewalManager` helper utility classes (for information on these classes, refer to the chapters in this specification titled *The* LookupDiscoveryManager and *The* LeaseRenewalManager).

Rather than writing a service to use these classes in a coordinated fashion (in addition to providing for attribute management), the service may be written to employ the `JoinManager` class from the `com.sun.jini.lookup` package. This utility class performs all of the functions related to discovery, joining, service lease renewal, and attribute management which the Jini technology programming model requires of a well-behaved Jini service. Each of these activities is intimately involved with the maintenance of a service's residency in one or more lookup services (the service's *join state*), thus the name `JoinManager`.

The `JoinManager` class provides an implementation of the functionality described above. The use of this class in a wide variety of services can help minimize the work resulting from having to repeatedly implement this required functionality in each service.

The `JoinManager` is a utility class, not a remote service. Jini services that wish to use this utility will create an instance of the `JoinManager` in the service's address space to manage the entity's join state locally.

Note that throughout this specification, when the term *service* is used, it refers to the object that has created an instance of the `JoinManager` and avails itself of the public methods of that utility class.

6.2 Other Types

The types defined in the specification of the `JoinManager` utility class are in the `com.sun.jini.lookup` package. The following types may be referenced in this chapter. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.lease.Lease
net.jini.core.entry.Entry
net.jini.core.lookup.ServiceID
net.jini.core.lookup.ServiceRegistrar
net.jini.core.lookup.ServiceRegistration
net.jini.discovery.DiscoveryListener
net.jini.discovery.DiscoveryManagement
net.jini.lookup.entry.ServiceControlled
com.sun.jini.lease.LeaseRenewalManager
com.sun.jini.discovery.LookupLocatorDiscovery
```

```
com.sun.jini.discovery.LookupDiscoveryManager
java.io.IOException
java.util.EventListener
```

6.3 The Interface

The public methods provided by the `JoinManager` class are as follows.

```
package com.sun.jini.lookup;

public class JoinManager
{
    public JoinManager(Object obj,
                       Entry[] attrSets,
                       ServiceIDListener callback,
                       DiscoveryManagement discoveryMgr,
                       LeaseRenewalManager leaseMgr)
                                        throws IOException;
    public JoinManager(Object obj,
                       Entry[] attrSets,
                       ServiceID serviceID,
                       DiscoveryManagement discoveryMgr,
                       LeaseRenewalManager leaseMgr)
                                        throws IOException;

    public DiscoveryManagement getDiscoveryManager();
    public LeaseRenewalManager getLeaseRenewalManager();
    public ServiceRegistrar[]  getJoinSet();

    public Entry[] getAttributes();
    public void addAttributes(Entry[] attrSets);
    public void addAttributes(Entry[] attrSets, boolean checkSC);
    public void setAttributes(Entry[] attrSets);
    public void modifyAttributes(Entry[] attrSetTemplates,
                                 Entry[] attrSets);
    public void modifyAttributes(Entry[] attrSetTemplates,
                                 Entry[] attrSets,
                                 boolean checkSC);
    public void terminate();
}
```

6.4 The Semantics

The `equals` method for this class returns true if and only if two instances of this class refer to the same object. That is, x and y are equal instances of this class if and only if x == y has the value `true`.

The individual semantics of each method of this class are described below.

- The constructor of the `JoinManager` class has two forms. Each form of the constructor throws `IOException`. This is because construction of a `JoinManager` may initiate the multicast discovery process, a process that can throw `IOException`.

  The first form of the constructor takes the following parameters as input:
  - A reference to the service requesting the services of the `JoinManager`
  - An array containing the service's attributes
  - A reference to an object that implements the `ServiceIDListener` interface (belonging to the package `com.sun.jini.lookup`)
  - A reference to an object that implements the `DiscoveryManagement` interface
  - An instance of the `LeaseRenewalManager` utility class

  The assignment of a service ID to the service will result in an event notification being sent to the listener object that was passed as the `ServiceIDListener` argument (`callback`). If a `null` value is passed in through this argument, then no such notification will be sent.

  In order to use the `JoinManager`, the service must supply an object through which notifications that indicate a lookup service has been discovered or discarded will be received. At a minimum, this object must satisfy the contract defined in the `DiscoveryManagement` interface. That is, this object must provide the `JoinManager` with the ability to set discovery listeners and to discard previously discovered lookup services when they are found to be unavailable.

  The `DiscoveryManagement` argument may be set to a value of `null`. If `null` is the value of this argument, then an instance of the `LookupDiscoveryManager` utility class will be constructed to listen for events announcing the discovery of only those lookup services that are members of the public group.

The LeaseRenewalManager argument may be set to a value of null. If null is the value of this argument, an instance of the LeaseRenewalManager class will be created, initially managing no Lease objects. This feature allows a service that employs the JoinManager to use either a single entity to manage all of its leases, or to use separate entities: one to manage the leases unrelated to the join process, and one to manage the leases that result from the join process and which are accessible only within the JoinManager.

The first form of the constructor is typically used by services that have not yet been assigned a service ID, but which have been pre-configured to join lookup services that the service identifies through the initialization of a discovery manager.

The second form of the constructor takes the same arguments as the first, except that an instance of the ServiceID replaces an instance of the ServiceIDListener interface. Note that the ServiceID class is defined in the *Jini™ Lookup Service Specification*, and the ServiceIDListener interface is described later in this document.

The second form of the constructor applies the same semantics to the discoveryMgr and leaseMgr arguments as is applied by the first form of the constructor.

The second form of the constructor should be used by services that have already been assigned a service ID (possibly by the service provider or as a result of a prior registration with some lookup service), and which may or may not have been pre-configured to join lookup services identified by group or by specific location.

- The getDiscoveryManager method returns the instance of DiscoveryManagement that was passed into the constructor by the entity, or which was created as a result of null being input to that parameter. This method takes no arguments as input.

  The object returned by this method provides the JoinManager with the ability to set discovery listeners and to discard previously discovered lookup services when they are found to be unavailable.

- The getLeaseRenewalManager method returns an instance of the LeaseRenewalManager class. This method takes no arguments as input.

The object returned by this method manages the leases requested and held by the `JoinManager`. Although it may also manage leases unrelated to the join process that are requested and held by the service itself, the leases with which the `JoinManager` is concerned are the leases that correspond to the service registration requests the `JoinManager` has made with each lookup service the service wishes to join.

- The `getJoinSet` method returns an array of `ServiceRegistrar` objects, each corresponding to a lookup service with which the service has registered (joined). Each time this method is invoked, a new array is returned. This method takes no arguments.
- The `getAttributes` method returns an array containing the set of attributes currently associated with the service. This method takes no arguments as input, and will return a new array upon each invocation.
- The `addAttributes` method associates a new set of attributes with the service, in addition to the service's current set of attributes. The association of this new set of attributes with the service will be propagated to each lookup service with which the service is registered. This propagation must be performed asynchronously. Because of this, there is no guarantee that the propagation of the attributes to all lookup services with which the service is registered will have completed upon return from this method.

  The set of attributes consisting of the union of the new set with the old set will be associated with the service in all future join processing.

  There are two forms of the `addAttributes` method. Both forms of this method take as input an argument representing the set of attributes to associate with the service. This set is represented as an array of `Entry` objects, none of whose elements may be `null`. An invocation of this method with this array containing duplicate elements is equivalent to performing the invocation with the duplicates removed from the array. If `null` is passed in as the value of this parameter, a `NullPointerException` will be thrown.

  The second form of this method also takes as input a flag indicating whether or not this method should determine if the attributes in the input set are instances of the `ServiceControlled` interface, which is a marker interface that is used to control which entities may modify a service's attribute set. For more information on this interface, refer to section 4.1 of the *Jini™ Lookup Attribute Schema Specification*. If the value of this flag is `true` and at

least one of the attributes to be added is an instance of the `ServiceControlled` interface, a `SecurityException` will be thrown and propagated through this method.

Note that because there is no guarantee that attribute propagation will have completed upon return from this method, services that invoke this method must take care not to modify the contents of the input array. Doing so could cause the service's attribute state reflected on a subset of the lookup services with which the service is registered to be corrupted or inconsistent with the state reflected on the remaining lookup services. It is for this reason that the effects of modifying the contents of the input array, after this method is invoked, are undefined.

- The `setAttributes` method replaces the service's current set of attributes with the given new set of attributes. This method takes a single argument as input: an array of `Entry` objects, none of whose elements may be `null`, that represents the set of attributes that will replace the current set of attributes.

  The replacement of the service's current set of attributes with the new set of attributes will be propagated to each lookup service with which the service is registered. This propagation must be performed asynchronously. Because of this, there is no guarantee that the propagation of the attributes to all lookup services with which the service is registered will have completed upon return from this method.

  The service's new set of attributes will be associated with the service in all future join processing.

  Invoking this method with an input array that contains duplicate elements is equivalent to performing the invocation with the duplicates removed from the array. If `null` is input to `setAttributes`, a `NullPointerException` will be thrown.

  For the same reason as that noted above in the description of the `addAttributes` method, the effects of modifying the contents of the input array, after `setAttributes` is invoked, are undefined.

- The `modifyAttributes` method changes the service's current set of attributes using the same semantics as the `modifyAttributes` method of the `ServiceRegistration` class (see the *Jini™ Lookup Service Specification*). This method has two forms. The first form takes two arguments, the second form takes three arguments. Both forms will take an array of templates in the first argument, and an array of attributes in the

second argument. The templates are used to identify which elements to modify from the service's current set of attributes. The attribute array contains the actual modifications to be made. The additional argument in the signature of the second form of `modifyAttributes` is a flag indicating whether or not this method should determine if the attributes in the input set are instances of the `ServiceControlled` interface, which is a marker interface used to control which entities may modify a service's attribute set (see section 4.1 of the *Jini™ Lookup Attribute Schema Specification*). If the value of this flag is `true` and at least one of the attributes to be modified is an instance of the `ServiceControlled` interface, a `SecurityException` will be thrown and propagated through this method.

The association of the new set of attributes with the service will be propagated to each lookup service with which the service is registered. This propagation must be performed asynchronously. Because of this asynchronous behavior, there is no guarantee that the propagation of the attributes to all lookup services with which the service is registered will have completed upon return from this method.

The set of attributes that results after the modifications have been applied will be associated with the service in all future join processing.

If the length of the array containing the templates does not equal the length of the array containing the attributes, an `IllegalArgumentException` will be thrown and propagated through this method.

For the same reason as that noted above in the description of the `addAttributes` method, the effects of modifying the contents of the input array, after `modifyAttributes` is invoked, are undefined.

- The `terminate` method performs cleanup duties related to the termination of the *lookup service* discovery event mechanism, as well as to the lease and thread management performed by the `JoinManager`.

  This method will cancel all leases managed on behalf of the service, and will terminate all threads that have been created. Additionally, if the discovery manager employed by the `JoinManager` was created by the `JoinManager` itself, this method will terminate *all* discovery processing being performed by that manager object on behalf of the service.

  The `JoinManager` makes certain concurrency guarantees with respect to an invocation of the `terminate` method while other method invocations are in progress. The termination process described above will not begin until

completion of all invocations of the methods defined in the public interface of the `JoinManager`. Furthermore, once the termination process has begun, no remote method invocations will be made by the `JoinManager`, and all other method invocations on the `JoinManager` will not return until the termination process has completed. Upon completion of the termination process, the semantics of all current and future method invocations on the current instance of the `JoinManager` are undefined.

6.5 Supporting Interfaces and Classes

The `JoinManager` utility class depends on the following interfaces: `DiscoveryManagement` and `ServiceIDListener` discussed below.

This class also references the following concrete classes: `LookupDiscoveryManager` and `LeaseRenewalManager`, each described in a separate chapter of this document.

6.5.1 The `DiscoveryManagement` Interface

Although it is not necessary for the `JoinManager` itself to execute the discovery process, it does need to be notified when one of the lookup services it wishes to join is discovered or discarded. Thus, at a minimum, the `JoinManager` requires access to the discovery events sent to the listeners registered with the discovery process' event mechanism. The instance of `DiscoveryManagement` that is passed as an argument to the constructor of the `JoinManager` provides a mechanism for acquiring access to those events. For a complete description of the semantics of the methods of this interface, refer to the chapter of this specification titled *The Discovery Management Interfaces*.

One noteworthy item about the semantics of the `JoinManager` is the effect that invocations of the `discard` method of `DiscoveryManagement` will have on any discovery listeners created by the `JoinManager`. The `DiscoveryManagement` interface specifies that the `discard` method will remove a particular lookup service from the managed set of lookup services that have already been discovered, allowing that lookup service to be re-discovered. Invoking this method will result in the flushing of the lookup service from the appropriate cache, ultimately causing a discard notification to

be sent to all `DiscoveryListener` objects registered with the event mechanism of the discovery process, including all listeners registered by the `JoinManager`.

The receipt of an event notification indicating that a lookup service has been discarded ultimately results in the removal (but not cancellation) of the registration lease that was granted by the discarded lookup service, and which is managed by the `LeaseRenewalManager` on behalf of the `JoinManager`. After removal occurs, the lease will eventually expire.

6.5.2 *The `ServiceIDListener` Interface*

The `ServiceIDListener` interface defines the methods used by a service to register a request for notification from the `JoinManager` upon the assignment of a `serviceID` by a lookup service. It is the responsibility of the service to create and pass into the `JoinManager` an object that implements this interface. That implementation must provide the definition of the actions to take upon receipt of the notification. Typically, the action taken will be to persist the assigned `serviceID` reference.

```
package com.sun.jini.lookup;

public interface ServiceIDListener extends EventListener
{
    public void serviceIDNotify(ServiceID serviceID);
}
```

The `ClientLookupManager` 7

7.1 Overview

The interactions of an entity that operates in a client-like fashion within a Jini application environment are generally distinguished by the fact that the entity first *discovers* one or more Jini Lookup services, then *queries* one or more of the discovered lookup services for references to Jini services that the entity may employ in some task. Although one of the characteristics that distinguishes a Jini service from a Jini client is the service's ability to be registered with a Jini Lookup service, both Jini services and clients can demonstrate the client-like characteristics just described. Thus, the interactions that occur between Jini Lookup services and these so-called client-like entities include discovery processing as well as service queries (lookups).

Note that throughout this specification one of two terms will be employed interchangeably when referring to Jini clients and services that interact with a lookup service in such a client-like fashion, and which create an instance of the `ClientLookupManager` (from the package `com.sun.jini.lookup`) and avail themselves of the public methods of that class. Those terms are: *entity* and *client-like entity*.

Once a client-like entity discovers a set of lookup services and retrieves references to desired services from those lookup services, the entity may choose to discontinue query-related discovery processing. That is, having obtained references to all of the services it wishes to employ, the entity may view the references it holds to the lookup services as no longer necessary.

But over the execution life of any such entity, partial failures such as crashes or network outages may intermittently affect the availability of some of those services of interest. This results in a need to re-query the lookup services to find references to new instances of the service that can replace the unavailable instance. Such scenarios make it desirable for a client-like entity to maintain its references to the lookup services it queries. If an instance of a service is found to be unavailable, the entity can query those lookup services to obtain an instance of the service which is available.

Since a query on a lookup service is a remote call, such calls are much more costly in terms of overhead and failure risk than are local calls. This cost is magnified when an entity must make frequent queries for multiple services. Because of this, an entity may find it desirable to cache the services it obtains from the original queries on the lookup services. Furthermore, by populating the cache with multiple instances of the desired services, redundancy in the availability of those services can be provided. Thus, if an instance of a service is found to be unavailable when needed, the entity can execute a local query on the cache rather than one or more remote queries on the lookup services to obtain an instance which is available.

Typically, an entity will request the creation of a separate cache for each service type of interest. The cache provides a method with which the entity can retrieve an element of the cache. In general, the particular service reference that is returned should not matter to the entity. It should only matter that *a* service reference has been returned, not *which* service reference. If for some reason it does matter to an entity which service reference is returned, then the cache also provides a mechanism which will allow the entity to retrieve all elements of the cache. The entity can then iterate through each element, selecting the particular reference it desires.

Although interacting with a local cache of services in this way can be very useful to entities that need frequent access to multiple services, some client-like entities may wish to interact with the cache in a reactive manner. For example, an entity such as a service browser typically wishes to be *notified* of the arrival of new services of interest as well as any changes in the state of the current services in the cache. Polling for such changes is usually viewed as undesirable. If the cache were to also provide an event mechanism with notification semantics, the needs of both types of entity could be satisfied.

From the scenarios discussed above, one could conclude that when acting in a client-like fashion, it is desirable for an entity to maintain, as much as possible, up-to-date knowledge of the availability of the *lookup* services of interest as

well as the state information associated with all other types of services in which the entity is interested. By maintaining current service state information, the entity can implement efficient mechanisms for service access and usage.

The `ClientLookupManager` class is a helper utility class that any client-like entity can use to create and populate a cache such as that described previously, and with which the entity can register for notification of the availability of services of interest. Like the `JoinManager` utility class, this class needs to be notified when a desired lookup service is discovered. For information on the `JoinManager` utility class, refer to the chapter of this document titled *The* JoinManager.

Unlike the `JoinManager`, the `ClientLookupManager` does not register the entity as a service with discovered lookup services. Although both the `JoinManager` and the `ClientLookupManager` perform lookup discovery event handling for the entities that employ them, the `JoinManager` performs *join* processing for Jini services, while the `ClientLookupManager` performs *service discovery and management* processing both for clients and for services. Thus, typical usage patterns for Jini services wishing to find and use other Jini services generally indicate the employment of both the `JoinManager` and the `ClientLookupManager` utilities, whereas Jini clients would typically use only the `ClientLookupManager`.

The `ClientLookupManager` class can be asked to "discover" services an entity is interested in using, and to cache the references to those services as each is found. The cache can be viewed as a set of service references that the entity can access locally as needed through one of the public, non-remote methods provided in the cache's interface. A service reference added to the cache will be removed from the cache when all of the lookup services with which that service is registered have been discarded.

The `ClientLookupManager` class also provides a mechanism for an entity to request that it be notified when a service of interest is discovered for the first time or has encountered a state change such as removal from all lookup services or attribute set changes.

For convenience, this class also provides versions of a method named `lookup`, which employs invocation semantics similar to the semantics of the `lookup` method of the `ServiceRegistrar` interface defined in the *Jini™ Lookup Service Specification*. This method may be useful to entities needing to find

services on an infrequent basis, or when the cost of making a remote call is outweighed by the overhead of maintaining a local cache (for example, due to limited resources).

All three mechanisms described above — local queries on the cache, service discovery notification, and remote lookups — employ the same template-matching scheme as that described in the *Jini™ Lookup Service Specification*. Additionally, each mechanism allows the entity to supply an object referred to as a *filter*. Such an object is a non-remote object that defines additional matching criteria that the `ClientLookupManager` applies when searching for the entity's services of interest. This filtering facility is particularly useful to entities that wish to extend the capabilities of the standard template-matching scheme.

The `ClientLookupManager` is a utility class, not a remote service. Client-like entities that wish to use this utility will create an instance of the `ClientLookupManager` in the entity's address space so as to manage the entity's "lookup state" locally.

7.2 The Object Types

The types defined in the specification of the `ClientLookupManager` utility class are in the `com.sun.jini.lookup` package. The following 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.lease.Lease
net.jini.core.lookup.ServiceItem
net.jini.core.lookup.ServiceMatches
net.jini.core.lookup.ServiceRegistrar
net.jini.core.lookup.ServiceTemplate
net.jini.discovery.DiscoveryListener
net.jini.discovery.DiscoveryManagement
com.sun.jini.discovery.LookupDiscoveryManager
com.sun.jini.event.ServiceDiscoveryEvent
com.sun.jini.lease.LeaseRenewalManager
com.sun.jini.lookup.LookupCache
com.sun.jini.lookup.ServiceDiscoveryListener
com.sun.jini.lookup.ServiceItemFilter
java.io.IOException
```

```
java.rmi.server.UnicastRemoteObject
java.rmi.MarshalledObject
java.rmi.RemoteException
java.util.EventListener
java.util.EventObject
java.util.Set
```

7.3 The Interface

The public interface provided by the `ClientLookupManager` class defines methods that allow an entity to request that references to services matching criteria defined by the entity be found in discovered lookup services and cached for local retrieval. This interface also defines methods for retrieving the manager objects employed by this utility, and for performing termination processing.

```
package com.sun.jini.lookup;

public class ClientLookupManager {
    public ClientLookupManager
                            (DiscoveryManagement discoveryMgr,
                             LeaseRenewalManager leaseMgr)
                                          throws IOException;
    public LookupCache createLookupCache
                            (ServiceTemplate template,
                             ServiceItemFilter filter,
                               ServiceDiscoveryListener listener)
                                        throws RemoteException;
    public ServiceItem lookup(ServiceTemplate tmpl,
                              ServiceItemFilter filter);
    public ServiceItem lookup(ServiceTemplate tmpl,
                               ServiceItemFilter filter,
                               long waitDur)
                                  throws InterruptedException,
                                         RemoteException;
    public ServiceItem[] lookup(ServiceTemplate tmpl,
                                int maxMatches,
                                ServiceItemFilter filter);
    public ServiceItem[] lookup(ServiceTemplate tmpl,
                                int minMatches,
                                int maxMatches,
                                ServiceItemFilter filter,
                                long waitDur)
                                 throws InterruptedException,
                                         RemoteException;
    public DiscoveryManagement getDiscoveryManager();
    public LeaseRenewalManager getLeaseRenewalManager();
    public void terminate();
}
```

7.4 The Semantics

The `ClientLookupManager` makes certain concurrency guarantees with respect to the methods it defines. When a method of the `ClientLookupManager` invokes a remote method, although such an invocation may block other remote calls made in the `ClientLookupManager`, invocations of local methods will not be blocked.

The `equals` method for this class returns true if and only if two instances of this class refer to the same object. That is, x and y are equal instances of this class if and only if x == y has the value `true`.

Exporting `RemoteEventListener` Objects

A subset of the methods on the `ClientLookupManager`, when invoked, will result in a request for registration with the event mechanism of one or more lookup services. Those methods are `createLookupCache` and the so-called *blocking* versions of the `lookup` method described below.

Any entity that invokes one of these methods must export to each lookup service with which a registration occurs, the stub classes of the `RemoteEventListener` object through which instances of `RemoteEvent` will be received. Furthermore, each of these methods must throw `RemoteException`. The reasons that a `RemoteException` can occur fall into one of the following categories:

- Each of these methods attempts to export a remote object, a process that can throw `RemoteException`
- Each of these methods attempts to register with the event mechanism of at least one lookup service, a process that can throw `RemoteException`

How this set of `ClientLookupManager` methods handles the `RemoteException` is dependent on the reason for the exception. If a `RemoteException` (or any other non-fatal exception or error) is thrown during an attempt to register for events from a lookup service, that lookup service will be discarded and made eligible for re-discovery. On the other hand, if the `RemoteException` is thrown during an attempt to export the listener, the method will exit and the exception will be propagated.

The potential for `RemoteException` during the export process imposes the following requirement: the *same* instance of the listener must be exported to each lookup service from which events will be requested. Furthermore, the

creation and export of the listener must occur prior to the event registration process. This requirement guarantees that should a `RemoteException` occur after the registration process has begun, the exception will not be propagated and event processing will continue.

To understand the significance of this requirement, consider the scenario where a different instance of the listener is exported to each lookup service. If a new lookup service is discovered after the event process has begun for the other lookup services in the managed set, a new instance of the listener must be created and exported. Should a `RemoteException` occur during the export process, the exception will be propagated and all event processing will stop — a result that many entities may view as undesirable.

In order to facilitate exporting the listener, the entity — whether it is a Jini client or a Jini service — is responsible for providing and advertising a mechanism through which each lookup service will acquire the listener's stub classes.

For example, one implementation of the `ClientLookupManager` might provide a special jar file containing only the listener stub classes (to optimize download time). By including this jar file in the entity's `java.rmi.server.codebase` property (in the appropriate format specifying transport protocol and location), the entity *advertises* the mechanism that lookup services can employ to acquire the stub classes. By executing a process to serve up the jar file (for example, an HTTP server), the mechanism through which each lookup service acquires those stub classes is *provided*.

It is important to note that should such a mechanism not be made available to each lookup service with which event registration will be requested, a "silent failure" can occur repeatedly. If the mechanism is not available, each lookup service cannot acquire the exported listener. Because each lookup service cannot acquire the exported listener, any attempts to register for events will fail. Whenever an attempt to register for events fails, the associated lookup service will be discarded and made eligible for re-discovery. Upon re-discovery of the discarded lookup service, the cycle repeats when a new attempt to register for events is made.

The individual semantics of each method of this class are described below.

- The constructor of the `ClientLookupManager` takes two arguments: an object that implements the `DiscoveryManagement` interface and a reference to a `LeaseRenewalManager` object. The constructor throws an

IOException because construction of a ClientLookupManager may initiate the multicast discovery process, a process that can throw IOException.

In order to use the ClientLookupManager, an entity must supply an object through which notifications that indicate a lookup service has been discovered or discarded will be received. At a minimum, this object must satisfy the contract defined in the DiscoveryManagement interface. That is, this object must provide the ClientLookupManager with the ability to set discovery listeners and to discard previously discovered lookup services when they are found to be unavailable.

A value of null may be passed as the DiscoveryManagement argument. If the value of the argument is null, then an instance of the LookupDiscoveryManager utility class will be constructed to listen for events announcing the discovery of only those lookup services that are members of the public group.

A value of null may be passed as the LeaseRenewalManager argument. If the value of the argument is null, an instance of the LeaseRenewalManager class will be created, initially managing no Lease objects.

- The createLookupCache method allows the client-like entity to request that the ClientLookupManager create a new managed set (or cache) and populate it with services, which match criteria defined by the entity, and whose references are registered with one or more of the lookup services the entity has targeted for discovery.

  This method returns an object of type LookupCache. Through this return value, the entity can query the cache for services of interest, manage the cache's event mechanism for service discoveries, or terminate the cache. The specification of the LookupCache interface is presented later in this document.

  An entity typically uses the object returned by this method to provide *local* storage of, and access to, references to services that it is interested in using. Entities needing frequent access to numerous services will find the object returned by this method quite useful because acquisition of those service references is provided through local method invocations. Additionally, because the object returned by this method provides an event mechanism, it is also useful to entities wishing to simply monitor, in an event-driven manner, the state changes that occur in the services of interest.

The `createLookupCache` method takes three arguments: an instance of `ServiceTemplate`, an instance of `ServiceItemFilter`, and an instance of `ServiceDiscoveryListener`. Both the `ServiceItemFilter` and `ServiceDiscoveryListener` interfaces are presented later in this document.

Together, the template and the filter arguments define the criteria with which service-matching should be performed. The listener argument references an object that will receive notifications when services matching the input criteria are discovered for the first time, or have encountered a state change such as removal from all lookup services or attribute set changes.

The template argument employs template-matching semantics identical to the semantics described in the *Jini™ Lookup Service Specification* (where the `ServiceTemplate` interface is defined) to identify the service(s) to acquire from lookup services in the managed set. The filter argument is then used to apply additional matching criteria to any service references found through template-matching. The additional matching criteria defined in this argument are application-specific, and, therefore, must be defined by the client-like entity itself. Furthermore, once an instance of the cache is created, any filter associated with that instance will not change during the life of that particular cache.

As a convenience, a `null` reference input to the template argument is treated as equivalent to inputting a `ServiceTemplate` constructed with all `null` arguments (all *wildcards*). That is, the cache will attempt to discover all services contained in each lookup service in the managed set. If a `null` value is passed as the filter argument, then only template-matching will be employed to find the desired services.

Entities that invoke this method must take care not to modify the contents of the input template after the cache has been created. Doing so could cause the state of the cache to become corrupted or inconsistent. It is for this reason that the effects of modifying the contents of the input template, after this method is invoked, are undefined.

Events and the Cache

In order to keep its contents up-to-date, the cache must register with the event mechanism of each lookup service in the managed set. From the point of view of the cache, a service is “discovered” when it receives a remote

event from one of those lookup services notifying the cache of the existence of a service matching the input criteria. In addition, whenever one of the cache's discovered services experiences a state change in one of the lookup services in which it is registered, the cache will receive a remote event identifying that state change whenever the change satisfies the matching criteria.

For a number of reasons the cache may receive multiple events corresponding to the same Jini service. For example, a particular Jini service may be registered with more than one lookup service from the managed set. Also, multiple configurations of the service may be registered with one or more of those lookup services. If the cache requests events from each lookup service using a template configured with no restriction along the service ID search axis and little or no restriction along the attribute search axis, the cache will receive a notification each time one of the following events occurs:

- Some configuration of the service, matching the template, is registered with one of the lookup services
- The lease of one of the service references matching the template is cancelled or expires
- An attribute set associated with one of the service references matching the template is modified in some way

Just as the cache requests that it be notified of state changes in matching services occurring within each lookup service, an entity may request that the cache deliver events that indicate analogous state changes in the service references stored in the cache.

There are two significant differences in the event mechanism between the lookup services and the cache, and the event mechanism between the cache and the client-like entity. First and foremost, the events sent from the lookup services to the cache are *remote* events, whereas the events sent from the cache to the entity are *local* events. Secondly, each registration or state-change event sent from the cache to the entity may actually have been a result of multiple corresponding events received by the cache from a set of lookup services. Thus, there is a "many-to-one" relationship between the events received by the cache and the events sent by the cache.

For many entities that use the cache's event mechanism to interact with the cache's discovered services, knowledge of the number of distinct service references, as well as identification of the lookup services with which those

references are registered, is of no interest. Such entities typically are interested only in acquiring *a* reference — not *all* references — to the desired services. Thus, the relationship between the two event mechanisms described above allows the `ClientLookupManager` to hide the lookup services with which the cache interacts from the entity. For entities that are interested in the additional information, the cache provides methods separate from the event mechanism for obtaining such information.

To summarize, although the cache may receive *multiple* events signaling a state change related to a particular matching service, the cache will typically send only a *single* corresponding event to the entity. That is, for any matching service:

- The cache will send a *service discovery event* to the entity only once: after the cache acquires the *first* reference to the matching service.
- The cache will send a *service removal event* to the entity only once: after every reference to the service has had its lease expire or cancelled; that is, only after all references to the matching service have been removed from every lookup service in the cache's managed set.
- For each set of event(s) notifying the cache that a particular modification has been made to the attribute set associated with one of the service references, one *service modification event* will be sent to the entity; but *only if* the attribute set state reflected in the received event represents an actual change in the service's current attribute set state (as maintained by the cache).

With respect to the state of the attribute sets associated with the service references stored in the cache, the cache should be viewed as maintaining a single attribute set state for each collection of service references that represent the same service. That single state will always be equivalent to the state reflected in the last attribute set modification event received by the cache.

For example, suppose three different references to a service which matches the input criteria are each registered with three lookup services in the managed set. Suppose the attribute sets associated with each service reference are modified in exactly the same way. For this specific case, the cache would receive three events — one from each lookup service — signaling these modifications. Upon receipt of the first event, the cache modifies its current notion of the service's attribute set state, and then notifies the entity of the change; but only if the state reflected in the event represents a change in the current state.

Next, suppose a second modification, different from the first, is made on only two of the service references, and a third unique modification is made on the remaining service reference. In this case, the cache will still receive three events, but how the cache handles the events is dependent on the order of arrival of the events. For simplicity, call the three events e_1, e_2, and e_3. Use ***S*** to represent the cache's current notion of the service's attribute set state, and use s_1 and s_2 to represent the states resulting after each attribute modification has occurred. In this example, e_1 and e_2 will be sent to the cache after the service's attribute sets are each modified to s_1 in their respective lookup services. Event e_3 is sent after the service's attribute sets are modified to s_2 in the remaining lookup service.

If the order of arrival is e_1, e_2, and then e_3, the cache will change ***S*** into s_1 and notify the entity after the arrival of e_1, but will do nothing upon the arrival of e_2. Upon the arrival of e_3, the cache will change ***S*** (which is now s_1) into s_2. If the order of arrival of the events is e_1, e_3, and then e_2, the cache will first change ***S*** into s_1, then into s_2, and then back into s_1 again. Furthermore, for each state change made, the cache will send a notification to the entity.

Thus, the events generated by the cache's event mechanism and sent by the cache to the entity, are more representative of the state changes that occur in the cache than in the lookup services.

An entity may register for events from the cache in one of two ways. The entity may supply an instance of `ServiceDiscoveryListener` to the listener argument of the `createLookupCache` method, or it may invoke a method on the cache to add a listener to the cache. Thus, an entity may register for events from the cache at any time during the execution life of the cache.

Similarly, the cache provides a method that an entity,which is currently registered for events from the cache, may use at any time to un-register with the cache's event mechanism.

- The `lookup` method queries each available lookup service in the managed set for service reference(s) that match criteria defined by the entity that invokes this method. Entities typically employ this method when they need infrequent access to services, and when the cost of making remote queries is outweighed by the overhead of maintaining a local cache (for example, because of resource limitations).

Although the `lookup` method has four versions, each version falls into one of two categories: those versions of this method that return a single instance of `ServiceItem`, and those versions that return a set of service references as an array of `ServiceItem` objects.

Two arguments are common to all versions of this method: an instance of `ServiceTemplate` and an instance of `ServiceItemFilter`.

The difference between the two versions of `lookup` in each of the two categories differ only in whether or not a particular version provides what is referred to as a "wait" (or blocking) feature. That is, within each category, one version of `lookup` will return immediately upon failure (or success), whereas the blocking version will declare failure only after waiting (and failing) a specified amount of time. The particular version of this method that an entity employs is typically determined by the entity's intended usage pattern.

The descriptions below refer to all versions of this method, except where explicitly noted.

The `template` argument and the `filter` argument both have semantics identical to that defined for these arguments in the description of the `createLookupCache` method above. In particular,

- A `null` reference value for the `template` argument is treated as the equivalent of a "wildcarded" `ServiceTemplate`.
- If `null` is the value for the `filter` argument, then only template-matching will be employed to find the desired services.
- The effects of modifying the contents of the `template` while the invocation is in progress are unpredictable and undefined

If no service can be found that matches the desired criteria, then the versions of `lookup` from the first category — those that return a single instance of `ServiceItem` — will return `null`; whereas the versions from the second category — those that return an array of `ServiceItem` instances — will return an empty array.

The versions of this method from the first category can be used in a fashion similar to the first form of the `lookup` method defined in the `ServiceRegistrar` interface described in the *Jini™ Lookup Service Specification*. That is, an entity would typically invoke one of these versions

of `lookup` when it wishes to find a *single* service reference, and the particular lookup service with which that service reference is registered is unimportant to the entity.

These versions of `lookup` differ with the corresponding version of `lookup` in `ServiceRegistrar` in the following ways:

- These versions of `lookup` query *multiple* lookup services (the order in which the lookup services are queried is defined by the implementation).
- These versions of `lookup` can apply additional matching criteria, in the form of a filter object, when deciding whether a service reference found through standard template-matching should be returned to the entity.

The two versions of this method that return an array of `ServiceItem` objects can be used in a fashion similar to the second form of `lookup` defined in the `ServiceRegistrar` interface. That is, an entity would typically invoke these versions of `lookup` when it wishes to find *multiple* service references matching the input criteria. Each of the versions of `lookup` that return return an array of `ServiceItem` objects takes as one of arguments an `int` that represents the maximum number of matches that should be returned. The array returned by these methods will contain no more than that number of service references, although it may contain fewer than that number.

As with the versions of `lookup` that return a single instance of `ServiceItem`, multiple queries and filtering are also notable differences between the second-category versions of this method and their counterpart in `ServiceRegistrar`.

For all of the versions of `lookup`, whenever a lookup service query returns a `null` service reference, the filter will be by-passed. On the other hand, whenever a lookup service query returns a non-`null` service reference in which one or more elements of the associated `attributeSets` array is `null`, the filter will be applied, but with the `null` elements of the `attributeSets` array removed. Upon finding a successful match, the service item will be returned in the appropriate way, with the `null` elements of the `attributeSets` array removed.

Each version of `lookup` may be confronted with *duplicate* references during a search for a service of interest. This is because the same service may register with more than one lookup service in the managed set. As with the

cache, when a set of service references is returned by `lookup`, each service reference in the return set will be unique with respect to all other service references in the set.

If it is determined that a lookup service is unavailable (due to an exception or some other non-fatal error) while interacting with a lookup service from the managed set, all versions of `lookup` will invoke the `discard` method on the instance of `DiscoveryManagement` being employed by the `ClientLookupManager`. Doing so will result in the unavailable lookup service being discarded and made eligible for re-discovery.

Recall that the propagation of modifications to a service's attributes across a set of lookup services typically occurs asynchronously. It is for this reason that while invoking `lookup` to find a set of matching services, it is possible that the set returned may reflect multiple references having the same service ID with different attributes. Note that although this sort of inconsistent state can also occur if the entity employs a cache, the cache will eventually reflect the correct state.

The Blocking Feature of Lookup

As noted above, each category contains a version of `lookup` that provides a feature in which the entity can request that if the number of service references found throughout the available lookup services does not fall into a desired range, the method will wait a finite period of time until either an acceptable minimum number of service references are discovered, or the specified time period has passed.

The versions of `lookup` providing this blocking feature each takes as one of its parameters a value of type `long` that represents the number of milliseconds to wait for the service to be discovered. In addition, each of these versions of `lookup` may throw an `InterruptedException`.

One of these blocking versions of `lookup` implicitly uses a value of one for both the acceptable minimum and the allowable maximum number of service references to discover. The other blocking version requires that the entity specify the range through the `minMatches` and `maxMatches` parameters respectively.

Prior to blocking, each of these versions of `lookup` first queries each available lookup service in an attempt to retrieve a satisfactory number of matching services. Whether or not the method actually blocks is dependent

on how many matching service references are found during the query process. Blocking occurs only if after querying *all* of the available lookup services, the number of matching services found is less than the acceptable minimum. If the waiting period passes before that minimum number of service references is found, the method will return the service references that have been discovered up to that point. If the waiting period passes and no services have been found, `null` or an empty array (depending on the version of `lookup`) will be returned.

If after querying all of the available lookup services the number of matching services found is greater than or equal to the specified minimum, but less than the specified maximum, the method will return the currently discovered service references without blocking. If the initial query process produces the desired maximum number of service references, the method will return the results immediately.

The blocking versions of `lookup` are quite useful to entities that cannot proceed until such a service of interest is found. If a non-positive value is input to the `waitDur` argument, then the method will not wait. It will simply query the available lookup services and employ the return semantics described above.

The values of the `minMatches` and `maxMatches` arguments must both be positive, and `maxMatches` must be greater than or equal to `minMatches`; otherwise, an `IllegalArgumentException` will be thrown.

The blocking versions of `lookup` make a concurrency guarantee with respect to the discovery of new lookup services during the wait period. That is, while waiting for matching service reference(s) to be discovered, if one or more of the desired — but previously unavailable — lookup services is discovered and added to the managed set, those new lookup services will also be queried for the service(s) of interest.

Additionally, both blocking versions of `lookup` throw an `InterruptedException`. When an entity invokes either version with valid parameters, the entity may decide during the wait period that it no longer wishes to wait the entire period for the method to return. Thus, while the method is blocking on the discovery of matching service(s), it may be interrupted by invoking the `interrupt` method from the `Thread` class. The intent of this mechanism is to allow the entity to interrupt a blocking `lookup` in the same way it would a sleeping thread.

- The `getDiscoveryManager` method will return an instance of the `DiscoveryManagement` interface. The object returned by this method provides the `ClientLookupManager` with the ability to set discovery listeners and to discard previously discovered lookup services when they are found to be unavailable. This method takes no arguments.
- The `getLeaseRenewalManager` method will return an instance of the `LeaseRenewalManager` class. The object returned by this method manages the leases requested and held by the `ClientLookupManager`. In general, these leases correspond to the registrations made by the `ClientLookupManager` with the event mechanism of each lookup service in the managed set. This method takes no arguments.
- The `terminate` method performs cleanup duties related to the termination of the event mechanism for *lookup service* discovery, the event mechanism for *service* discovery, and the cache management duties of the `ClientLookupManager`.

  For each instance of `LookupCache` created and managed by the `ClientLookupManager`, the `terminate` method will do the following:
  - Either remove all listener objects registered for receipt of `DiscoveryEvent` objects or, if the discovery manager employed by the `ClientLookupManager` was created by the `ClientLookupManager` itself, terminate *all* discovery processing being performed by that manager object on behalf of the entity.
  - Cancel all event leases granted by each lookup service in the managed set of lookup services.
  - Un-export all remote listener objects registered with each lookup service in the managed set.
  - Terminate all threads involved in the process of retrieving and storing references to discovered services of interest.

  The `ClientLookupManager` makes certain concurrency guarantees with respect to an invocation of the `terminate` method while other method invocations are in progress. The termination process described above will not begin until completion of all invocations of the methods defined in the public interface of the `ClientLookupManager`; that is, until completion of invocations of `createLookupCache`, `lookup`, `getDiscoveryManager`, and `getLeaseRenewalManager`.

Additionally, once the termination process has begun, no remote method invocations will be made by the `ClientLookupManager`, and all other method invocations on the `ClientLookupManager` will not return until the termination process has completed. Upon completion of the termination process, the semantics of all current and future method invocations on the current instance of the `ClientLookupManager` are undefined.

7.5 Supporting Interfaces and Classes

The `ClientLookupManager` utility class depends on the following interfaces defined in the *Jini™ Lookup Service Specification*: `ServiceTemplate`, `ServiceItem`, and `ServiceMatches`. This class also depends on a number of interfaces defined below; those interfaces are: `DiscoveryManagement`, `ServiceItemFilter`, `ServiceDiscoveryListener`, and `LookupCache`.

The `ClientLookupManager` class references the following concrete classes: `LookupDiscoveryManager` and `LeaseRenewalManager`, each described in a separate chapter of this document, and `ServiceDiscoveryEvent` defined in this chapter.

7.5.1 The `DiscoveryManagement` Interface

Although it is not necessary for the `ClientLookupManager` itself to execute the discovery process, it does need to be notified when one of the lookup services it wishes to query is discovered or discarded. Thus, at a minimum, the `ClientLookupManager` requires access to the instances of `DiscoveryEvent` sent to the listeners registered with the event mechanism of the discovery process. The instance of `DiscoveryManagement` passed to the constructor of the `ClientLookupManager` provides a mechanism for acquiring access to those events. For a complete description of the semantics of the methods of this interface, refer to the chapter of this specification titled *The Discovery Management Interfaces*.

One noteworthy item about the semantics of the `ClientLookupManager` is the effect that invocations of the `discard` method of `DiscoveryManagement` have on any cache objects created by the `ClientLookupManager`. The `DiscoveryManagement` interface specifies that the `discard` method will remove a particular lookup service from the managed set of lookup services that have already been discovered, allowing that lookup service to be re-discovered. Invoking this method will result in the flushing of the lookup

service from the appropriate cache. This effect ultimately causes a discard notification to be sent to all `DiscoveryListener` objects registered with the event mechanism of the discovery process (including all listeners registered by the `ClientLookupManager`).

The receipt of an event notification indicating that a lookup service from the managed set has been discarded must ultimately result in the cancellation and removal of all event leases that were granted by the discarded lookup service, and that are managed by the `LeaseRenewalManager` on behalf of the `ClientLookupManager`.

Furthermore, every service reference stored in the cache that is registered with the discarded lookup service but is not registered with any of the remaining lookup services in the managed set, will be "discarded" as well. That is, all previously discovered service references that are registered with only unavailable lookup services, will be removed from the cache and made eligible for service re-discovery.

7.5.2 The `ServiceItemFilter` Interface

The `ServiceItemFilter` interface defines the methods used by an object such as the `ClientLookupManager` or the `LookupCache` to apply additional matching criteria when searching for services in which an entity has registered interest. It is the responsibility of the entity requesting the application of additional criteria to construct an implementation of this interface that defines the additional criteria, and to pass the resulting object (referred to as a *filter*) into the object that will apply it.

The filtering mechanism provided by implementations of this interface is particularly useful to entities that wish to extend the capabilities of the standard template matching scheme. For example, since template matching does not allow one to search for services based on a range of attribute values,

this additional matching mechanism can be exploited by the entity to ask the managing object to find all registered printer services that have a resolution attribute *between* say, 300 dpi and 1200 dpi.

```
package com.sun.jini.lookup;

public interface ServiceItemFilter
{
    public boolean check(ServiceItem item);
}
```

The Semantics

The `check` method defines the implementation of the additional matching criteria to apply to a `ServiceItem` object found through standard template matching. This method takes one argument: the `ServiceItem` object to test against the additional criteria. This method returns `true` if the input object satisfies the additional criteria, and `false` otherwise.

Neither a `null` reference nor a `ServiceItem` object containing `null` fields will be passed into this method by the `ClientLookupManager`.

If the parameter input to this method is a `ServiceItem` object that has non-`null` fields, but is associated with attribute sets containing `null` entries, this method must process that parameter in a reasonable manner.

Should an exception occur during an invocation of this method, the semantics of how that exception is handled are undefined.

This method must not modify the contents of the input `ServiceItem` object. Doing so can result in unpredictable and undesirable effects on future processing by the `ClientLookupManager`. That is why the effects of any such modification to the contents of that input parameter are undefined.

7.5.3 *The `ServiceDiscoveryEvent` Class*

The `ServiceDiscoveryEvent` class encapsulates the service discovery information made available by the event mechanism of the `LookupCache`. All listeners that an entity has registered with the cache's event mechanism will

receive an event of type `ServiceDiscoveryEvent` upon the discovery, removal, or modification of one of the cache's services, as described previously in the section titled *Events and the Cache*.

This class is a sub-class of the class `EventObject`. In addition to the methods of the `EventObject` class, this class provides two additional accessor methods which can be used to retrieve the additional state associated with the event: `getPreEventServiceItem` and `getPostEventServiceItem`.

The `getSource` method of the `EventObject` class returns the instance of `LookupCache` from which the given event originated.

```
package com.sun.jini.event;

public class ServiceDiscoveryEvent extends EventObject
{
    public ServiceDiscoveryEvent(Object source,
                                 ServiceItem preEventItem,
                                 ServiceItem postEventItem);

    public ServiceItem getPreEventServiceItem();
    public ServiceItem getPostEventServiceItem();
}
```

The Semantics

The constructor of `ServiceDiscoveryEvent` takes three arguments:

- An instance of `Object` corresponding to the instance of `LookupCache` from which the given event originated
- A `ServiceItem` reference representing the state of the service (associated with the given event) *prior to* the occurrence of the event
- A `ServiceItem` reference representing the state of the service *after* the occurrence of the event

If `null` is passed as the `source` parameter for the constructor, a `NullPointerException` will be thrown.

Depending on the nature of the discovery event, a `null` reference may be passed as one or the other of the remaining parameters, but never both. If `null` is passed as both the `preEventItem` and the `postEventItem` parameters, a `NullPointerException` will be thrown.

Note that the constructor will not modify the contents of either `ServiceItem` argument. Doing so can result in unpredictable and undesirable effects on future processing by the `ClientLookupManager`. That is why the effects of any such modification to the contents of either input parameter are undefined.

The `getPreEventServiceItem` method returns an instance of `ServiceItem` containing the service reference corresponding to the given event. The service state reflected in the returned service item is the state of the service *prior* to the occurrence of the event.

If the event is a discovery event (as opposed to a removal or modification event), then this method will return `null` because the discovered service had no state in the cache prior to its discovery.

The `getPostEventServiceItem` method returns an instance of `ServiceItem` containing the service reference corresponding to the given event. The service state reflected in the returned service item is the state of the service *after* the occurrence of the event.

If the event is a removal event, then this method will return `null` because the discovered service has no state in the cache after it is removed from the cache.

Because making a copy can be a very expensive process, neither accessor method returns a copy of the service reference associated with the given event. Rather, each method returns the appropriate service reference from the cache itself. Because of this cost, listeners (see `ServiceDiscoveryListener` next) that receive a `ServiceDiscoveryEvent` must not modify the contents of the object returned by these methods. Doing so could cause the state of the cache to become corrupted or inconsistent because the objects returned by these methods are also members of the cache. That is why the effects of modifying the object returned by either accessor method are undefined.

7.5.4 *The `ServiceDiscoveryListener` Interface*

The `ServiceDiscoveryListener` interface defines the methods used by objects such as a `LookupCache` to notify an entity that events of interest related to the elements of the cache have occurred. It is the responsibility of the

entity wishing to be notified of the occurrence of such events to construct an object that implements the `ServiceDiscoveryListener` interface and then use that implementation to register with the cache's event mechanism. Any implementation of this interface must define the actions to take upon receipt of an event notification. The action taken is dependent on both the application and the particular event that has occurred.

```
package com.sun.jini.lookup;

public interface ServiceDiscoveryListener extends EventObject
{
    public void serviceAdded(ServiceDiscoveryEvent event);
    public void serviceRemoved(ServiceDiscoveryEvent event);
    public void serviceChanged(ServiceDiscoveryEvent event);
}
```

The Semantics

As described previously in the section titled *Events and the Cache*: when the cache receives from one of the managed lookup services, an event signaling the *registration* of a service of interest for the *first time* (or for the first time since the service has been discarded), the cache invokes the `serviceAdded` method on all instances of `ServiceDiscoveryListener` that are registered with the cache. Doing so notifies the entity that a service of interest has been discovered. This method takes one argument: an instance of `ServiceDiscoveryEvent` containing references to the service item corresponding to the event, including representations of the service's state both before and after the event.

When the cache receives, from a managed lookup service, an event signaling the *removal* of a service of interest from the *last* such lookup service with which it was registered, the cache invokes the `serviceRemoved` method on all instances of `ServiceDiscoveryListener` that are registered with the cache. Doing so notifies the entity that a service of interest has been discarded. This method takes one argument: a `ServiceDiscoveryEvent` object containing references to the service item corresponding to the event, including representations of the service's state both before and after the event.

When the cache receives, from a managed lookup service, an event signaling the unique *modification* of the attributes of a service of interest (across the attribute sets of all references to the service), the cache invokes the `serviceChanged` method on all instances of `ServiceDiscoveryListener` that are registered with the cache. Doing so notifies the entity that the state of a service of interest has changed. The `serviceChanged` method takes one argument: a `ServiceDiscoveryEvent` object containing references to the service item corresponding to the event, including representations of the service's state both before and after the event.

Should an exception occur during an invocation of any of the methods defined by this interface, the semantics of how that exception is handled are undefined.

Each method defined by this interface must not modify the contents of the `ServiceDiscoveryEvent` parameter. Doing so can result in unpredictable and undesirable effects on future processing by the `ClientLookupManager`. It is for this reason that if one of these methods modifies the contents of the parameter, the effects are undefined.

This interface makes the following concurrency guarantee. For any given listener object that implements this interface, no two methods (either the same two methods or different methods) defined by the interface can be invoked at the same time by the same cache. For example, the `serviceRemoved` method must not be invoked while the invocation of another listener's `serviceAdded` method is in progress.

7.5.5 *The `LookupCache` Interface*

The `LookupCache` interface defines the methods provided by the object created and returned by the `ClientLookupManager` when a client-like entity invokes the `createLookupCache` method. It is within that object that discovered service references, which match criteria defined by the entity, are

stored. Through this interface the entity may retrieve one or more of the stored service references, register and un-register with the cache's event mechanism, and terminate all of the cache's processing.

```
package com.sun.jini.lookup;

public interface LookupCache
{
    public ServiceItem   lookup(ServiceItemFilter filter);
    public ServiceItem[] lookup(ServiceItemFilter filter,
                                int maxMatches);

    public void addListener(ServiceDiscoveryListener listener);
    public void removeListener
                          (ServiceDiscoveryListener listener);

    public void discard(Object serviceReference);
    public void terminate();
}
```

The Semantics

Depending on which version is invoked, the `lookup` method of the `LookupCache` interface returns one or more elements, which match the input criteria, and that were stored in the associated cache. The object returned is either a single instance of `ServiceItem`, or a set of service references in the form of an array of `ServiceItem` objects. Each service item returned by either form of this method must have been previously discovered to be both registered with one or more of the lookup services in the managed set, and to match criteria defined by the entity.

One argument is common to both forms of `lookup`: an instance of `ServiceItemFilter`. The semantics of the filter argument are identical to those of the filter argument specified for a number of the methods defined in the interface of the `ClientLookupManager` utility class. This argument is intended to allow an entity to separate its filtering into two steps: an initial filter applied during the discovery phase, and then a finer resolution filter applied upon retrieval from the cache. As with the methods of the `ClientLookupManager`, if `null` is the value of this argument, then no additional filtering will be performed.

The second form of the `lookup` method of the `LookupCache` interface takes an additional argument: a parameter of type `int` that represents the maximum number of matches that should be returned. The array returned by this form of `lookup` will contain no more than the requested number of service references, although it may contain less than that number. The value input to this argument must be positive, otherwise an `IllegalArgumentException` will be thrown.

If the cache is empty, or if no service can be found that matches the input criteria, then the first from of `lookup` will return `null`; whereas the second form of `lookup` will return an empty array. The algorithm used to select the return element(s) from the set of matching service references is implementation dependent.

Neither form of the `lookup` method of the `LookupCache` interface returns a copy of the matching service reference(s) that were selected; rather, each form returns the actual service reference(s) from the cache itself. Because of this, entities that invoke either form of this method must not modify the contents of the service reference(s) returned. Doing so could cause the state of the cache to become corrupted or inconsistent. That is why the effects of modifying the service reference(s) returned by either form of `lookup` is undefined.

Typically an entity will request the creation of a separate cache for each service type of interest. When the entity simply needs a reference to a service of a particular type, the entity should invoke the first form of `lookup` to retrieve one element from the cache; in this case, which particular service reference that is returned will not, in general, matter to the entity. If for some reason it does matter to an entity which service reference is returned, then the entity can invoke the second form of `lookup` requesting that `Integer.MAX_VALUE` service references be returned. Doing so will return all elements of the cache that match the input criteria. The entity can then iterate through each element, selecting the desired reference.

The `addListener` method will register a `ServiceDiscoveryListener` object with the event mechanism of a `LookupCache`. This listener object will receive a `ServiceDiscoveryEvent` upon the discovery, removal, or modification of one of the cache's services, as described previously in the section of this chapter titled *Events and the Cache*. This method takes one argument: a reference to the `ServiceDiscoveryListener` object to register.

Once a listener is registered, it will be notified of all service references discovered to date, and will be notified as new services are discovered and existing services are modified or discarded. If the parameter value is `null`, no action will be taken.

The `LookupCache` makes a re-entrancy guarantee with respect to any `ServiceDiscoveryListener` objects registered with it. Should the `LookupCache` invoke a method on a registered listener (a local call), any call from that method to a local method of the `LookupCache` is guaranteed not to result in a deadlock condition.

The `removeListener` method will remove a `ServiceDiscoveryListener` object from the set of listeners currently registered with a `LookupCache`. Once all listeners are removed from the cache's set of listeners, the cache will send no more `ServiceDiscoveryEvent` notifications. This method takes one argument: a reference to the `ServiceDiscoveryListener` object to remove.

If the parameter value to `removeListener` is `null`, or if the listener passed to this method does not exist in the set of listeners maintained by the implementation class, then this method will take no action.

If an entity determines that a service reference retrieved from the cache is no longer available, the entity should request the removal of that reference from the cache. The mechanism for discarding an unavailable service from the cache is provided by the `discard` method of the `LookupCache` interface. The discard method takes one argument: an instance of `Object` referencing the service reference to remove from the cache.

The `discard` method not only deletes the service reference from the cache, but also sends a notification to all registered listeners indicating that the service has been discarded. Whenever an invocation of this method results in success, the service is said to have been "discarded".

Note that after the service has been discarded, there is no guarantee that the service will again become eligible for discovery. If the service's residency in at least one lookup service from the managed set ends (usually because the service's lease with that lookup service has expired), the service will be re-discovered when it becomes available and re-registers with that lookup service.

The `terminate` method performs cleanup duties related to the termination of the processing being performed by a particular instance of `LookupCache`. For that instance, this method cancels all event leases granted by the lookup services that supplied the contents of the cache, and un-exports all remote

listener objects registered with those lookup services. The `terminate` method is typically called when the entity is no longer interested in the contents of the `LookupCache`.

The `LookupDiscoveryService` 8

8.1 Overview

Part of the *Jini™ Discovery and Join Specification* is devoted to defining the discovery requirements for well-behaved Jini clients and services. These so-called *discovering entities* are required to participate in the multicast discovery protocol. Such entities are required to send multicast discovery requests to lookup services with which the entities wish to interact. In addition, the entities must continuously listen for and act on announcements from the desired lookup services. Interactions with a discovered lookup service may involve registration with that lookup service, or it may simply involve querying the lookup service for services of interest (or both). In order to find *specific* lookup services, discovering entities also need to be able to participate in the unicast discovery protocol.

Under certain circumstances, a discovering entity may find it useful to allow a third party to perform the entity's discovery duties. For example, an activatable entity that wishes to deactivate may wish to employ a special Jini service — referred to as a *lookup discovery service* — to perform discovery duties on behalf of the entity. Such an entity may wish to deactivate for various reasons, one being to conserve computational resources. While the entity is inactive, the lookup discovery service, running on the same or a separate host, would employ the discovery protocols to find lookup services in which the entity has expressed interest, and would notify the entity when a previously unavailable lookup service has become available.

The facilities of the lookup discovery service are of particular value in a scenario where a new lookup service is added to a long-lived djinn containing multiple inactive services. Without the use of a lookup discovery service, the time frame over which the new lookup service is fully populated can be unpredictable and unbounded.

Because an inactive service has no way of discovering a new lookup service, each inactive service wishing to discover and join a new lookup service must first activate. Since activation of an inactive service occurs when some client attempts to use the service, the amount of time that passes between the arrival of the new lookup service and the activation of the service can vary greatly over the range of services. Thus, the time frame over which the lookup service becomes fully populated is unpredictable since it could take arbitrarily long before all of the services activate and then discover and join the new lookup service.

The time frame over which the lookup service becomes fully populated is not only unpredictable, it is also unbounded because there is no guarantee that the lookup service will send multicast announcements between the time the service activates and the time it deactivates. If the timing is right, it is possible that one or more of the services may never discover and join the new lookup service. Thus, without the use of the lookup discovery service, the new lookup service may never fully populate.

As another example of a discovering entity that may find it useful to allow a third party to perform the entity's discovery duties, consider an entity that exists in an environment with one of the following characteristics:

- The environment does not support multicast.
- The environment contains no lookup services within the entity's *multicast radius*.
- The environment does contain lookup service(s) within the entity's multicast radius, but at least one service needed by the entity is not registered with any lookup service within that radius.

If such an entity was provided with references to lookup services — located outside of the entity's multicast radius — which contain services needed by the entity, the entity could contact each lookup service and retrieve the desired service references. One way to achieve this might be to configure the entity to find and use a lookup discovery service which would employ multicast discovery to find nearby lookup services belonging to groups in which the

entity has expressed interest. 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 protocol through a proxy relationship with the lookup discovery service, gaining access not only to each discovered lookup service but also to all of their registered services.

Note that the scenario just described does not come without restrictions. In order for the lookup discovery service to be able to "link" an entity with lookup services in the way described above, the lookup discovery service must be registered with a lookup service having a location that is either 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 scenarios such as those described above, the lookup discovery service participates in both the multicast discovery protocol 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™ Distributed Event Specification*. 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 entity 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 Jini

distributed leasing model as defined in the *Jini™ Distributed Leasing Specification*. That is, the lookup discovery service will only grant its services for a limited period of time without an active expression of continuing interest on the part of the client.

8.1.1 Goals & Requirements

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

- 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 also which 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 embodied in 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 its related classes and interfaces

8.2 Other Types

The types defined in the specification of the `LookupDiscoveryService` interface are in the `com.sun.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.lookup.ServiceRegistrar
net.jini.discovery.DiscoveryEvent
net.jini.discovery.DiscoveryListener
```

```
com.sun.jini.discovery.LookupLocatorDiscovery
java.rmi.RemoteException
java.rmi.NoSuchObjectException
```

8.3 The Interface

The `LookupDiscoveryService` interface defines the service introduced in the previous sections of this chapter. Through this interface, other Jini services and clients may request that discovery processing be performed on their behalf. This interface belongs to the `com.sun.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 front-end proxy objects that implement 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 `RemoteException`, and only requires the default serialization semantics so that this interface can be implemented directly using Java RMI.

```
package com.sun.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. For the purposes of this document, 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 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 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 protocol or the unicast discovery protocol to discover lookup services that correspond to members of the appropriate managed set.

8.4 The Semantics

In order 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.

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 object produced contains methods through which the object may be changed. Because the returned object is 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 (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 names of the groups to which the desired lookup services belong
- An array of zero or more non-`null` `LookupLocator` objects, each corresponding to a specific lookup service the client wishes to be discovered
- A non-`null` `RemoteEventListener` object which specifies the entity that will receive events notifying the registration of the discovery of lookup services of interest
- 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 instance of 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 requesting the re-discovery of discarded lookup services. The semantics of the methods of the `LookupDiscoveryRegistration` interface are defined in the next section.

The `groups` argument takes a `String` array, none of whose elements may be `null`. A `null` value (`LookupDiscovery.ALL_GROUPS`) is acceptable for this argument. 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 (`LookupDiscovery.NO_GROUPS`) is passed in, then no group discovery will be performed for the associated registration until the client, through one of the registration's methods, populates the managed set of groups.

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 one of the registration's methods, populates the managed set of locators.

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 *discovery 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 state of the multicast announcements from that lookup service. Depending on the state of those announcements, the lookup discovery service may send either a discovery event or an event referred to as a *discard event*. The conditions under which either a discovery event or a discard 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 discovery event will be sent to all registrations that have registered interest in lookup services belonging to that group.
- If the multicast announcements from an already-discovered lookup service indicate that the lookup service is no longer a member of one or more of the groups that had been reflected in previous multicast announcements, a discard event will be sent to all registrations that have registered interest in at least one of the missing groups but in none of the remaining groups.
- If the multicast announcements from an already-discovered lookup service are no longer being received, a discard event will be sent to all registrations that have registered interest in that lookup service.

Note – The requirement that the lookup discovery service monitor the state of the multicast announcements from already-discovered lookup services is very desirable because it allows for the maintenance of consistent state. But if we do include such a requirement, then the `LookupDiscovery` utility will have to be changed in a similar fashion.

A more detailed discussion of the event semantics of the lookup discovery service is presented next in the section titled *Event Semantics*.

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

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.

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 and to the set of groups and locators submitted through the registration request. 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 sub-class 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 discovery events for that lookup service will be sent to a registration's listener until the lookup service is discarded (through that registration) and then re-discovered. Note that a detailed definition of what it means for the lookup discovery service to discard a lookup service is presented later in this document.

If, between the time a lookup service is discarded (through any registration) and the time it is re-discovered, a new registration having parameters referencing that lookup service is requested, upon re-discovery 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™ Distributed Event Specification*), which means that when any two such successive events have sequence numbers differing by only a value of 1, then no events have been missed. On the other hand, when viewing the set of received events 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 related to the discovered lookup services may or may not fall "out of sync" with the corresponding remote state. 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 local state will not. To allow clients to identify and correct such a situation, each registration object provides a method which returns the references to that registration's currently discovered lookup services. With this information, the client can update its local state.

When requesting a registration with the lookup discovery service, a client may also supply a reference to an object (as a parameter to the registration method), wrapped in a `MarshalledObject`, referred to as a *handback*. When the lookup discovery service sends an event to a registration's listener, the event sent will also contain a reference to this handback object. 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™ Distributed Event Specification*.

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.

8.5 Supporting Interfaces and Classes

The `LookupDiscoveryService` interface references the following interface: `LookupDiscoveryRegistration`. This class also depends on the following concrete class: `RemoteDiscoveryEvent`.

8.5.1 The `LookupDiscoveryRegistration` Interface

When a client requests a registration with a lookup discovery service, an instance of the `LookupDiscoveryRegistration` interface is returned. It is through this interface that the client manages the parameters reflected in the registration with the lookup discovery service.

```
package com.sun.jini.discovery;

public interface LookupDiscoveryRegistration
{
    public EventRegistration getEventRegistration();
    public Lease getLease();
    public ServiceRegistrar[] getRegistrars()
                                        throws 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, this 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, through the registration process, requests 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, 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 which allow the client to retrieve its event registration information. Additional methods defined by this interface allow the client to retrieve references to the set of 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 `NoSuchObjectException` if the registration on which the method is invoked no longer exists. It should be noted that if a client receives a `NoSuchObjectException` as a result of an invocation of such a method, although the client can assume that the registration no longer exists, the client cannot assume that the lookup discovery service itself no longer exists.

Each remote method of this interface may throw `RemoteException`. Typically, this exception occurs when there is a communications 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.

The Semantics

The methods defined by this interface are organized into a set of accessor methods, a set of group modification methods, a set of locator modification methods, and the `discard` method. Through the accessor methods, various elements of a registration's state can be retrieved. The modification 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 re-discovery.

The Accessor Methods

The `getEventRegistration` method returns an `EventRegistration` object that encapsulates the information needed by the client 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 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 an array containing references to each lookup service that has already been discovered for the registration. This method can be used to maintain synchronization between the set of discovered lookup services represented in the client's local state and the corresponding set represented in the remote state. The local state can become un-synchronized with the remote state when a gap occurs in the events received by a registration's listener.

According to the event semantics previously described, 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. Thus, upon finding gaps in the sequence of events, the client can invoke this method and use the returned information to synchronize its local state with the remote state.

The `getGroups` method returns an array consisting of the group names from the registration's managed set. 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, then `null` is returned.

The `getLocators` method returns an array consisting of the `LookupLocator` objects from the registration's managed set. If the managed set of locators is empty, this method returns the empty array. If there is no managed set of locators associated with the registration, then `null` is returned.

The Group Modification 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`. The empty set is denoted by the empty array (`LookupDiscovery.NO_GROUPS`), and "no set" is indicated by `null` (`LookupDiscovery.ALL_GROUPS`). No set indicates that all lookup services within the multicast radius should be discovered. 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 managed set.

Elements in the array that duplicate elements already in the managed set will be ignored. Once a new name has been added to the managed set, the lookup discovery service will attempt to discover all as yet undiscovered lookup services that are members of the group having that name.

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 new group names that will replace the set of names in the managed set.

Once a new group name has been placed in the managed set, if there are lookup services belonging to that group that have already been discovered, no event will be sent to the registration's listener for those particular lookup services. Attempts to discover all as yet undiscovered lookup services belonging to that group will continue to be made for the registration.

If `null` is passed to `setGroups`, then the lookup discovery service will attempt to discover all as yet undiscovered lookup services located within the multicast radius and, upon discovery of any lookup service, will send to the registration's listener an event signaling that discovery. If the parameter value is the empty array, then group discovery for the registration will cease.

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.

If the registration has no current managed set of groups from which to remove elements, the `removeGroups` method throws an `UnsupportedOperationException`. If `null` is input, this method throws a `NullPointerException`. If the empty array is input, then the registration's managed set of groups will not change.

After a set of groups has been removed from the managed set because of an invocation of either `setGroups` or `removeGroups`, attempts to discover any lookup service that satisfies each of the following characteristics will cease to be made for the registration:

- the lookup service is a member of one or more of the groups that was removed from the registration's managed set, and
- the lookup service is not a member of any group in the new managed set resulting from the invocation of `setGroups` or `removeGroups`, and
- the lookup service does not correspond to any element in the registration's managed set of locators.

The Locator Modification 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`. Invoking any of these methods with a set of locators that contains duplicate locators (as determined by the `equals` method of `LookupLocator`) 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 managed set. Elements in the input set that duplicate (using `LookupLocator.equals`) elements already in the managed set will be ignored.

This method throws an `UnsupportedOperationException` if the registration has no managed set of locators to augment. If `null` is passed to `addLocators`, a `NullPointerException` will be thrown. If the empty array is the parameter value, 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 that will replace the locators in the managed set.

If `null` is passed to `setLocators`, a `NullPointerException` will be thrown. If the parameter value is the empty array, all locator discovery performed by the lookup discovery service, for the registration, will cease.

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

If the registration has no managed set of locators from which to remove elements, this method throws an `UnsupportedOperationException`. If `null` is passed to `removeLocators`, a `NullPointerException` will be thrown. If the parameter value is the empty array, the registration's 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 modification methods, and that new locator equals none of the locators corresponding to the previously discovered lookup services (across all registrations), the lookup discovery service will attempt unicast discovery of the lookup service associated with the new locator. Note that locator equality is determined by the `equals` method of `LookupLocator`.

If locator discovery is attempted, the discovery attempt will be repeated until one of the following events occurs:

- The lookup service is discovered
- The client's lease expires
- The client explicitly removes the locator from the 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, an event signaling a discovery will be sent to the registration's listener.

Whenever an existing locator is removed from the managed set as a result of an invocation of one of the methods above, the action taken by the lookup discovery service depends on whether the lookup service corresponding to that locator had been previously discovered for the registration. Furthermore, if it was previously discovered, the action taken is also dependent on the discovery protocol through which the discovery occurred. With respect to the lookup service corresponding to such a removed locator, the action taken by the lookup discovery service can be described as follows:

- If the lookup service has yet to be discovered for the registration, attempts to perform locator discovery of that lookup service will cease.
- If the lookup service has already been discovered for the registration through locator discovery, but not through group discovery, the lookup service will be *discarded* (as defined below).

Note – The semantics of the locator methods above were specified to be consistent with the group methods of this service as well as the methods of the `LookupDiscovery` utility. We need to modify the methods of the `LookupLocatorDiscovery` utility to be similarly consistent.

Discarding Lookup Services

When the lookup discovery service removes an already-discovered lookup service from a registration's managed set(s) and makes the lookup service eligible for re-discovery, the lookup service is considered to be *discarded*.

There are a number of situations where 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
- When a lookup service — previously discovered through *locator* discovery — is removed from a registration's managed set, in response to an invocation of either the `setLocators` method or the `removeLocators` method
- When 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 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 lookup service, and to request that the lookup discovery service discard that lookup service.

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™ Lookup Service Specification*).

Note that if a lookup service crashes or is unavailable for some reason, there will be no automatic notification of the occurrence of such an event. This means that for each of the registration's targeted lookup services, after a lookup service is initially discovered, the lookup discovery service will not attempt to discover that lookup service again (for that registration) until that lookup service is discarded.

When a client determines that a lookup service discovered for a registration is no longer available, it is the responsibility of the client to 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 re-discover 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 a *non-fatal* exception or error (e.g., `RemoteException`) as a result of the attempt.

Note that the lookup discovery service may be acting on behalf of numerous clients having 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 re-discovered. 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 re-discovery, the registrations through which the invocation is made are added to the list of registrations that will be notified when re-discovery of the lookup service does occur. That is, upon re-discovery of the lookup service, only those registrations through which this method is invoked will be notified.

8.5.2 The `RemoteDiscoveryEvent` Class

Instances of the `RemoteDiscoveryEvent` class are passed to the `RemoteEvent` listener implemented by the client and registered with the lookup discovery service.

```
package com.sun.jini.discovery;

public class RemoteDiscoveryEvent extends RemoteEvent
{
    public RemoteDiscoveryEvent(Object source,
                                long eventID,
                                long seqNum,
                                MarshalledObject handback,
                                boolean discarded,
                                ServiceRegistrar[] registrars);

    public boolean isDiscarded();
    public ServiceRegistrar[] getRegistrars();
}
```

`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,

and a set of proxies corresponding to recently discovered or discarded lookup services. Methods are defined through which this additional state may be retrieved upon receipt of an event that implements this interface.

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 client's registration(s) 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 *discard event*; when `false`, it is referred to as a *discovery event*.

The Semantics

The `isDiscarded` method returns a `boolean` that indicates whether the event is a discovery event or a discard event.

The `getRegistrars` method returns an array of `ServiceRegistrar` objects. The elements contained in the returned array are the proxies that correspond to lookup services that were either newly discovered, re-discovered, discarded, or had their group sets modified.

Rather than a single proxy, an array, is returned so that implementations of the lookup discovery service can choose to "batch" the information sent to a registration. With respect to discoveries, this 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. Afterwards, 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 this chapter in the section titled *Event Semantics*, are strictly increasing — even when the information is batched.

The `getRegistrars` method does not make a remote call. The same array is returned on multiple invocations of this method. That is, this method does not return a copy of the array containing the effected proxies returned in the event.

The `LeaseRenewalService` 9

9.1 Overview

Leasing is a key concept in the Jini architecture. In general, Jini services only grant access to a resource for as long as the clients of those Jini services actively express interest in the resource being maintained. This is in contrast to many other systems where access to a resource is granted until the client explicitly releases the resource, although using the leasing model generally makes a distributed system more robust by allowing stale information and services to be cleaned up. It also places additional requirements on clients and services.

A client of a leased service may run into difficulties if it deactivates. Unless that client takes care to make sure some other process renews the client's leases while it is inactive or the client ensures that it is activated before its lease begins to expire, the client will lose access to the resources it has acquired. This 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 in order to conserve computational resources on its host) and it does not take appropriate steps, its registrations with lookups will expire and before long it will be inaccessible. If it only becomes active when clients require its services, it may never become active again, since 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, 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 who always have the ability to access the network, but for various reasons 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 all of these problems. Services that wish to go quiescent (become inactive) can pass the responsibility for renewing the leases they have been granted to a renewal service. The service may then deactivate without risk of losing access to resources it has acquired. Clients that have continuous access to the network but cannot be continuously connected, such as the cell phone described above, can similarly 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 only grant its services for a limited period of time without an active expression of continuing interest. In order 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 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 services, 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.

A renewal service is 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 an activatable client that deactivates only rarely needs to reactivate solely to renew its lease with the renewal service. In any case, the leases the renewal service grants are likely to be sufficiently long such that the actual renewal calls do not place an excessive load on the client, the renewal service, or the network.

9.1.1 Goals & Requirements

The requirements of the set of classes and interfaces in this chapter 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 these services in a way that does not overly weaken the leasing model

The goals of this chapter are

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

9.2 Other Types

The types defined in the specification of the `LeaseRenewalService` interface are in the `com.sun.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.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
```

9.3 The Interface

`LeaseRenewalService` (in the `com.sun.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` local to the client and use 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 of the methods of `LeaseRenewalService` throw `RemoteException` and only require the default serialization semantics. Therefore `LeaseRenewalService` can be implemented directly using RMI.

```
package com.sun.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 defined on the sets themselves. Two leases in the same set need not be granted by the same host or have the same expiration time; in addition, they can be added or removed from the set independently.

When adding a lease to the set, the client specifies how long it should remain in the set by specifying a *membership duration*. After the membership duration expires, the lease will be removed from the set without further client intervention. The renewal service will never renew a lease for a period that extends beyond its current membership duration expiration.

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 leases in the set. There is an event associated with each set that occurs at a client-specified time before the lease on the set expires. Clients can register for this event using methods provided by the set. A registration for this event does not have its own lease, but instead is bundled into the same lease under which the set was granted.

We use the term *definite exception* to refer to an exception that could be thrown by an operation (such as a remote method call) that would be indicative of a permanent failure. For purposes of this document, `NoSuchObjectException` and all non-`RemoteException` subtypes of `Throwable` are considered to be definite exceptions.

Conversely we use the term *indefinite exception* to refer to an exception that could be thrown by an operation that would be indicative of a transient failure. For purposes of this document all subtypes of `RemoteException` excluding `NoSuchObjectException` are considered indefinite exceptions.

Each lease renewal set has a renewal failure event associated with it that will occur if any lease in the set expires before its membership duration runs out, or if the renewal service attempts to renew a lease and gets a definite exception. Clients can register for this event using methods provided by the set. A registration for this event does not have its own lease, but instead is bundled into the same lease under which the set was granted.

Once placed in a set, a 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 lease is removed by the client
- The lease expires
- The lease's membership duration expires
- A renewal call results in a definite exception

Each lease in a set will be renewed as long as it is in the set. If a renewal call throws an indefinite exception, the renewal service should retry the lease renewal until the lease would otherwise be removed from the set. The renewal service will never cancel a lease. The preferred method of canceling a lease that has been placed in a set 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. The `leaseDuration` argument specifies how long (in milliseconds) the client wants the set's initial lease duration to be. The initial duration of the set's lease will be equal to or shorter than this request; it will not be longer. This duration

must be positive, `Lease.FOREVER`, or `Lease.ANY`; otherwise an `IllegalArgumentException` must be thrown. The set's lease is obtained through a method provided by the set.

The `LeaseRenewalSet` interface defines the interface to the sets created by

```
package com.sun.jini.lease;

public interface LeaseRenewalSet {
    public void addLease(Lease leaseToRenew,
                         long membershipDuration)
        throws RemoteException;

    public EventRegistration setExpirationWarningListener(
            RemoteEventListener listener,
            long                minWarning,
            MarshalledObject    handback)
        throws RemoteException;

    public void clearExpirationWarningListener()
        throws RemoteException;

    public EventRegistration setRenewalFailureListener(
            RemoteEventListener listener,
            MarshalledObject    handback)
        throws RemoteException;

    public void clearRenewalFailureListener()
        throws RemoteException;

    public Lease removeLease(Lease leaseToRemove)
        throws RemoteException;

    public Lease getLease();
}
```

the lease renewal service. The interface is not a remote interface. Each implementation of the renewal service exports proxy objects that implement the `LeaseRenewalSet` local to the client and use an implementation-specific protocol to communicate with the actual remote server. All of the proxy methods obey normal RMI remote interface semantics except where explicitly noted. The proxy objects for two sets are equal (using the `equals` method) if

they are proxies for the same set created by the same renewal service. Any method that communicates with the remote server should throw a `NoSuchObjectException` if the set no longer exists. If a client receives a `NoSuchObjectException` from one of the operations on a lease renewal set the client can infer that the set has been destroyed; however, it should *not* infer that the renewal service has been destroyed.

Leases can be added to the set through the `addLease` method. The `leaseToRenew` argument specifies the lease to be renewed. An `IllegalArgumentException` must be thrown if the lease was granted by the renewal service itself. An `IllegalArgumentException` must also be thrown if the lease is currently a member of another set allocated by the same renewal service.

Note – There seem to be two other viable possibilities when a client attempts to add a lease to one set when that lease is already in another set. Adding a lease to a second set can implicitly remove it from the first. This has the nice property of ensuring that for a given lease and lease removal service, the lease is in at most one set allocated by that lease removal service. On the other hand I am not wild about the idea of implicitly removing leases from sets when adds are performed. If we decide it is important to support atomically moving a lease from one set to another, I would rather have a `move` method. The second possibility is to allow the same lease in multiple sets and force lease removal service implementation to coordinate the renewal of a lease in multiple sets. This makes the implementation of a lease removal service more complex. It also makes the specification more complex, especially with respect to lease renewal length. For now I am going to go with the `IllegalArgumentException` because I think it will be easier to go from there to the move or multiple set semantics than the other way around.

Note – We probably need to revisit this decision if we change the lease spec to allow for uncoordinated renewals of the same lease (as opposed to the current idea of generating new leases to the same resource).

The `membershipDuration` is the initial membership duration for the lease. Unlike a lease duration, the membership duration is unilaterally specified by the client, not negotiated between the client and the service. The duration must have a value between 1 and `Long.MAX_VALUE`, inclusive; otherwise the renewal service must throw an `IllegalArgumentException`.

A `membershipDuration` of `Long.MAX_VALUE` does not imply that the lease will remain in the set forever. The lease will be ejected from the set if the set is destroyed, the lease itself expires, the lease is removed from the set, or any renewal attempt made by the renewal service results in a definite exception.

Calling `addLease` with a lease that is already in the set will associate the existing lease in the set with the new membership duration. The lease is not replaced because it is more likely that the renewal service, rather than the client, has an up-to-date lease expiration. This is because the client should not be renewing a lease that it has passed to a lease renewal service unless the lease is removed first. These semantics also allow `addLease` to be used in an idempotent fashion.

Note – These semantics for `addLease` (and for `removeLease`, next) require that all leases have a reasonable definition for `equals`. This still needs to be formalized in the lease spec.

Leases are removed from the set using the `removeLease` method. Removal from the set will not cause the lease to be cancelled. The method will return the lease that is being removed. The expiration time of the returned lease will reflect either:

- The result of the last successful renewal call that the renewal service made

  or

- The expiration time the lease originally had when it was added, if the renewal service has not yet successfully renewed the lease

If the lease is not in the set, `null` will be returned.

The `getLease` method returns the lease associated with the set itself. (Note this method does not make a remote call.)

The lease renewal service does not support multiple simultaneous registrations for the same kind of event. Although it would be useful in some limited circumstances, to do so would require event registrations to be leased separately from the set they are associated with. For the average client of the lease renewal service this ability would increase the number of leases that it would have to manage. Since the renewal service is based on the premise that some clients have difficulty managing their own leases, increasing the number of leases that a client would need to manage could significantly complicate the

implementation of those clients. Because there can be at most one listener for each kind of event, a given set provides a `set/clear` interface instead of the more common `addListener/removeListener` interface.

The `setExpirationWarningListener` method allows the client to register for notification of the approaching expiration of the set's lease. The `listener` argument specifies what listener should be notified when the lease is about to expire. The `minWarning` argument specifies in milliseconds how long before lease expiration the event should be generated. This must be zero or a positive number. If it is not, an `IllegalArgumentException` must be thrown. If the current expiration of the set's lease is sooner than `minWarning`, the event will occur immediately (though it will take time to propagate to the handler).

The `handback` argument to `setExpirationWarningListener` specifies an object that will be part of the expiration warning event notification. This mechanism is detailed in the *Jini™ Distributed Event Specification*.

The `setExpirationWarningListener` method returns the event registration for this event. This registration has the same lease as the lease renewal set. The event ID returned by the event registration is unique, at least with respect to all other active event registrations created by the given renewal service.

The method must throw a `NullPointerException` if `listener` is `null`.

If an event handler has already been specified for this event the current registration is replaced with the new one. The returned event registration must have the same event ID as the replaced registration. Because both registrations are for the same kind of event, the events sent to the new registration must be in the same sequence as the events sent to the old registration.

The `clearExpirationWarningListener` method removes the event registration currently associated with the approaching expiration of the set's lease. It is acceptable to call this method even if there is no active registration.

The `setRenewalFailureListener` method allows the client to register for the event associated with the failure to renew a lease in the set. These events are generated when a lease expires while it is still in the set, or the service attempted to renew the lease and gets a definite exception. The `listener` argument specifies the listener to be notified if a lease could not be renewed.

The handback argument to setRenewalFailureListener specifies an object that will be part of the renewal failure event notification. This mechanism is detailed in the *Jini™ Distributed Event Specification*.

The setRenewalFailureListener method returns the event registration for this event. This registration has the same lease as the set. The event ID returned by the event registration is unique at least with respect to all other active event registrations created by the given renewal service.

The method must throw NullPointerException if listener is null.

If an event handler has already been specified for this event the current registration is replaced with the new one. The returned event registration must have the same event ID as the replaced registration. Because both registrations are for the same kind of event, the events sent to the new registration must be in the same sequence as the events sent to the old registration.

The clearRenewalFailureListener method removes the event registration currently associated with the event for the failure to renew a lease in the set. It is acceptable to call this method even if there is no active registration.

ExpirationWarningEvent objects are passed to the event handlers specified

```
package com.sun.jini.lease;

public class ExpirationWarningEvent extends RemoteEvent {
    public ExpirationWarningEvent(Object            source,
                                  long              eventID,
                                  long              seqNum,
                                  MarshalledObject  handback,
                                  Lease             lease);
    public Lease getLease();
}
```

in calls to the LeaseRenewalSet method, setExpirationWarningListener. The ExpirationWarningEvent is a subclass of RemoteEvent, adding one additional item of abstract state—the lease which is about to expire. This state is returned by the getLease method. Its expiration will reflect the expiration the lease had when the event occurred.

Renewal calls may have changed the actual expiration between the time the event was generated and delivered. The event's other state is described in the *Jini™ Distributed Event Specification*.

Sequence numbers for a given event ID are strictly increasing. If there is no gap between two sequence numbers, no events have been missed; if there is a gap, events might (but might not) have been missed.

`RenewalFailureEvent` objects are passed to the event handlers specified in

```
package com.sun.jini.lease;

public class RenewalFailureEvent extends RemoteEvent {
    public RenewFailureEvent(Object            source,
                             long              eventID,
                             long              seqNum,
                             MarshalledObject  handback,
                             Lease             lease,
                             Throwable         throwable);
    public Lease getLease(){};
    public Throwable getThrowable();
}
```

calls to the `LeaseRenewalSet` method, `setRenewalFailureListener`. The `RenewalFailureEvent` is a subclass of `RemoteEvent`, adding two additional items of abstract state—the lease which could not be renewed before expiration and the `Throwable` that was thrown by the first failed renewal attempt. The lease is returned by the `getLease` method, and the `Throwable` by the `getThrowable` method. If the `Throwable` is `null` it can be assumed that the renewal service was unable to call `renew` before the lease expired.

If the renewal service was able to renew the lease before the event occurred, the lease's expiration will reflect the result of the last successful renewal call. When a renewal failure event is generated for a given lease, that lease is removed from the set.

The event's other state is as described in the *Jini™ Distributed Event Specification*. Sequence numbers for a given event ID are strictly increasing. If there is no gap between two sequence numbers, no events have been missed; if there is a gap, events might (but might not) have been missed.

The `EventMailbox` 10

10.1 Overview

The *Jini™ Distributed Event Specification* describes a notification mailbox object for storing event notifications on behalf of other objects. This is particularly important for objects that want more control over event notifications. In a distributed system, it may not be desirable for an object to be contacted solely for the purpose of having an event delivered. Objects owned by a mobile node which may detach from a Jini system is a good example. In this scenario, for the purpose of event notifications, it may be both undesirable as well as not possible to contact such objects. In addition, an object preferring to batch process event notifications may designate a third party to accept events on its behalf. The third party collects events over time until an object initiates their delivery. This chapter defines interfaces and protocols which allow Jini services and clients to interact with a type of third-party event store.

The Jini technology programming model is designed to allow the building of a distributed system that is flexible and robust. The event mailbox service provides a mechanism for alternate delivery semantics for systems built using asynchronous event notifications. While distributed events facilitate the construction of reactive programs, the event mailbox adds the ability for a service to determine how and when events are received. In the Jini Distributed Event model, the `RemoteEventListener` has no control over how or when an event notification is accepted. Events are delivered by the object that generates the event, referred to as the *generator*. Such events may be delivered directly from the generator to the listener, or indirectly through an arbitrary

chain of third-party objects. This delivery is initiated by the generator. Through the use of an event mailbox service, the `RemoteEventListener` may accept event notifications at its convenience.

The `EventMailbox` (in the package `com.sun.jini.event.mailbox`) will store notifications for other objects until the delivery is requested. Services "advertise" their capabilities by publishing a service interface to a well-known location. Clients that need the service consult the well-known location, download code for the service interface, and initiate requests against it. An interface to an event mailbox service can be acquired from any well-known location capable of storing objects written in the Java programming language. Entities wishing to control how and when event notifications are received may then use this interface to specify the mailbox as a third party for event acceptance and delivery.

10.2 Requirements

The requirements of this set of interfaces are:

- To specify an interface for objects to request use of the mailbox
- To specify the information that must be returned as a result of such registration
- To specify client interactions with the mailbox
- To specify how clients retrieve events from the mailbox

10.3 Other Types

The types defined in the specification of the event mailbox service are in the `com.sun.jini.event.mailbox` package. This specification assumes knowledge of the *Jini™ Distributed Event Specification*. The following object types may be referenced in this chapter. Whenever referenced, these object types will be referenced in unqualified form:

```
java.rmi.RemoteException
java.io.IOException
net.jini.core.event.EventRegistration
net.jini.core.event.RemoteEvent
net.jini.core.event.RemoteEventListener
```

10.4 Model and Terms

Objects contain data and methods that operate on the data. As methods are invoked, the object changes state. One of the defining characteristics of object-oriented programming is that the state of an object is not directly visible by entities other than the object itself. However, there are times that an object may be interested in the occurrence of a particular event within another object. These events are changes in the abstract state of the object, which may or may not be directly reflected in the actual state of the object.

An object may export a set of events which other objects may find of interest. These exported events express abstract state changes about which external objects can ask to be informed. External objects may register interest in such events. Registration for event notification includes specifying the object that will be informed when an instance of an event of interest occurs. This entity may be the object that performed the registration (referred to as the *registrant*) or some other object chosen by the registrant. All that is required of the entity is that it supports the `RemoteEventListener` interface. When an event of interest occurs, the object which experienced the abstract state change sends a notification to all listener objects designated to receive such notifications.

A *reactive object*, which takes action based on the occurrence of abstract state changes in other objects on the network, may want more control over how and when events are received. To achieve such controlled delivery, the object could employ the use of a third party to store events on its behalf. At its own convenience, the object may then instruct the third party to forward the collected events. Objects may express their intent to use the third party by registering with it. Registration results in a registration object through which the client of the event mailbox service may instruct the mailbox to forward events to a specified recipient referred to as the *forwarding target* object. The client of the event mailbox service can then dictate when and how events are received by informing the third party when and how to deliver events to the forwarding target.

The event mailbox serves this purpose. The event mailbox service (or simply the *mailbox*) is a third-party object which accepts events on behalf of other objects on the network. When client objects register with the event mailbox, they are, in effect, asking it to provide an object supporting the `RemoteEventListener` interface which can be specified with generator objects to receive events. For any generator, notifying a listener that was

obtained from a mailbox results in the event being sent to the mailbox. Over time, the mailbox collects events on behalf of its clients. These events will be presented to forwarding target objects at a later time.

The term *mailbox resources* is used to refer to the `RemoteEventListener` and associated space for collected events allocated by the event mailbox for use by client objects. Mailbox resources are leased resources and, as such, will be maintained as long as the associated lease is valid. Since no two environments may be served by a single design trade-off, operational parameters—controls for how the event mailbox deals with issues such as compaction, defragmentation, and low space behavior—may be exposed through an administration interface which can vary across different event mailbox implementations.

10.5 The `EventMailbox` Interface

The `EventMailbox` is an interface implemented by an object that wants to receive and forward `RemoteEvent` notifications on behalf of another object. The `EventMailbox` interface contains a single `register` method.

```
package com.sun.jini.event;

public interface EventMailbox
{
    MailboxRegistration register(long leaseDuration)
        throws RemoteException;
}
```

The `register` method has a single parameter of type `long`. This represents the lease duration for which use of the event mailbox is requested. The `register` method returns an object of type `MailboxRegistration`. The value passed in as the requested lease duration must be a positive `long` value. This value represents the duration, in milliseconds, for use of the event mailbox. The special values defined in the *Jini™ Distributed Leasing Specification*, namely `Lease.FOREVER` and `Lease.ANY`, are also valid values. `Lease.FOREVER` represents a duration of maximum value while `Lease.ANY` represents a duration whose length is preferred and chosen by the event mailbox. The granted duration for use of the event mailbox service resources is represented as part of a `Lease` object included in the returned

MailboxRegistration object. In granting a lease to the caller of the register method, the event mailbox may grant either the requested duration or a shorter duration. If the register method throws a RemoteException, then registration is not guaranteed to have happened. Each successful invocation of the register method produces a new registration object. Thus, the register method is not idempotent.

10.6 The MailboxRegistration Interface

The MailboxRegistration interface abstracts the set of client interactions with the event mailbox service. An object implementing the MailboxRegistration interface is returned to the client as the result of calling the register method of the EventMailbox interface. The details of how a MailboxRegistration object interacts with the underlying event mailbox service is hidden from a holder of the event mailbox registration. Thus, the MailboxRegistration acts like a proxy object.

The encapsulation of what is needed by a client to interact with the event mailbox includes a Lease object, a RemoteEventListener object and methods to disable and enable event delivery. The Lease object represents the usage duration for the resources associated with a RemoteEventListener handed out by the event mailbox. The listener object represents an object capable of being informed of events of interest as specified by the *Jini™ Distributed Event Specification*. To use the event mailbox to accept events on its behalf, an object must specify the listener obtained from the MailboxRegistration object when registering interest in events exported by generator objects. The enableDelivery method is used by objects in possession of the registration object to instruct the event mailbox to deliver events to the specified target listener. The disableDelivery method is used to instruct the event mailbox to stop delivering events.

The getLease method takes no parameters and returns an object of type Lease. The lease on the event mailbox resources is retrieved from the MailboxRegistration by calling getLease, which is a local method call. Should the lease expire or be cancelled, the event mailbox resources associated with the lease are purged.

The getListener method takes no parameters and returns an object of type RemoteEventListener. The returned object, referred to as a *mailbox listener*, is retrieved from the MailboxRegistration by calling the getListener

method, which is a local method call. A holder of a mailbox listener object is free to register it with generator objects when expressing interest in a distributed event.

Events are sent by the generator objects through the mailbox listener and

```
package com.sun.jini.event.mailbox;

public interface MailboxRegistration
{
    public Lease getLease();
    public RemoteEventListener getListener();
    public void enableDelivery(RemoteEventListener target)
        throws RemoteException;
    public void disableDelivery() throws RemoteException;
}
```

collected by the mailbox. The `enableDelivery` and `disableDelivery` methods of the `MailboxRegistration` are used for forwarding collected events. The `enableDelivery` method takes a single parameter of type `RemoteEventListener`. The `disableDelivery` method takes no parameters.

A forwarding target object may have events delivered as a continuous sequence of events that may be started or halted. This is analogous to turning a faucet on and off. Initially, the faucet is off. When the `enableDelivery` method is called, notification of the specified target listener object with the sequence of events collected by the event mailbox is commenced. Making an `enableDelivery` method call while event delivery has already been enabled updates the target listener as specified in the forwarding target object parameter. An `enableDelivery` call that throws a `RemoteException` is not guaranteed to have successfully enabled delivery of the events stored in the mailbox. Any new events, arriving at the mailbox while the faucet is on, will be accumulated by the mailbox and eventually delivered to the current forwarding target object. An event that is successfully delivered to the forwarding target object is removed from event mailbox storage. The forwarding target object is a `RemoteEventListener`. The event mailbox service delivers events to the forwarding target object by calling its `notify` method. If this call throws a `RemoteException`, the event is not guaranteed to have been sent. An event mailbox implementation may choose to retry such failed delivery attempts. If this call throws a `NoSuchObjectException`,

event delivery is halted. The process of sending events continues until the event mailbox has either exhausted its supply of collected events from storage or is instructed to halt delivery. Calling `enableDelivery` with a `null` forwarding target object halts event delivery.

Calling the `disableDelivery` method instructs the mailbox to halt the delivery of events. Once the `disableDelivery` method has been successfully called, the event mailbox will cease sending events. If `disableDelivery` throws a `RemoteException`, the halting of event delivery is not guaranteed to have succeeded. Making a `disableDelivery` method call after delivery has already been disabled has no effect.

10.7 Typical Mailbox Interactions

The following scenarios are examples of typical client interactions with the event mailbox service.

Scenario 1

A client object performs the following actions when choosing an event mailbox service:

- Acquires a reference to an `EventMailbox`
- Registers with the event mailbox and receives a registration object
- Stores the event mailbox registration object in stable storage
- Retrieves the listener from the registration object
- Registers with event generators specifying the listener obtained from the event mailbox as the place to send events
- Becomes unable to receive events directly

Suppose a client object intends to register its listener interface with a number of event generators across the network, but wants more control over how and when events are delivered. Because it already knows how to contact these generators, the client is able to register the listener obtained from the event mailbox with the same set of generators. This results in events being sent to the mailbox. At this point, all events of interest to the client will be sent through the event mailbox listener registered by the client. The client can control when events are delivered by instructing the event mailbox service to commence

delivery at a desirable time. Likewise, the client can have more control over how events are delivered by turning on the event faucet, receiving events, and turning off the faucet after the number of events received reaches a desirable threshold.

Scenario 2

A client object performs the following actions when initiating delivery of its event notifications:

- Becomes able to receive events directly
- Rebuilds mailbox registration object from stable storage if necessary
- Instructs the event mailbox service to deliver events specifying a valid forwarding target object
- Instructs the event mailbox service to stop delivering events

A client object that becomes available to receive events prepares to receive them as soon as possible. During the period of time the client was unable to receive them directly, events meant for the client were delivered to and stored by the event mailbox. The client then retrieves from the event mailbox service, all of the events it would have missed had it not employed this service. Use of the event mailbox service is ended when the lease granted by the event mailbox is cancelled or expires. The determination of a reasonable time to terminate this relationship is left to the client object.