A Collection of Jini™ Technology Helper Utilities and Services Specifications

This Collection of Jini™ Technology Helper Utilities and Services Specifications defines a set of standard helper utilities and services which extend the Jini Technology Core Platform. These helper utilities and services encapsulate desirable behaviors in the form of a set of reusable components that can be used to help simplify the process of developing Jini technology-enabled clients and services (*Jini clients and services*) for the Jini technology application environment. 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.

Version 1.1
October 2000

Contents

x

US

Introduction to Helper Utilities and Services

US.1 Summary

WHEN developing clients and services that will participate in the application environment for Jini™ technology, 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 protocols 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 protocols 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 technology-enabled client (*Jini client*) or service will wish to regularly renew leases that it holds. For example, when a Jini technology-enabled service (*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.

To help simplify the process of developing clients and services for the application environment for Jini technology (*Jini application environment*), several specifications in this document collection define reusable components that encapsulate behaviors such as those outlined above. Employing such 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.

What is presented first is terminology that may be helpful when analyzing these specifications. Following the section on terminology, brief summaries of the content of each of the current helper utilities and services specifications are provided. Finally, the other specifications on which these specifications depend are listed for reference.

US.2 Terminology

THIS section defines terms and discusses concepts that may be referenced throughout the helper utilities and services specifications. While the terms and concepts that appear in this section are general in nature and may apply to multiple components specified in this collection, each specification may define additional terms and concepts to further facilitate the understanding of a particular component. Each specification may also present supplemental information about some of the terms defined in this section and their relationship with the component being specified.

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

In addition to the glossary, the *Jini™ Technology Core Platform Specification* (referred to as the *core specification*) presents detailed definitions of a number of terms and concepts appearing both in this section and throughout the helper utilities and services specifications. When appropriate, the relevant specification will be referenced.

US.2.1 Terms Related to Discovery and Join

The Jini Technology Core Platform Specification, "Discovery and Join", 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 Jini technology-enabled services and/or devices and Jini 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 Technology Core Platform Specification, "Discovery and Join" 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 Technology Core Platform Specification, "Discovery and Join"*. 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.*

US.2.2 Jini Clients and Services

For the purposes of the helper utilities and services specifications, 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 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 that 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.

US.2.3 Helper Service

A Jini technology-enabled *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.

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

US.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.

US.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 via group discovery. The managed set of groups is typically represented as a `String` array, or a `Collection` of `String` elements.

Each element of the managed set of locators corresponds to a specific lookup service that the entity wishes to be discovered via locator discovery. Typically, this set is represented as an array of `net.jini.core.discovery.LookupLocator` objects or some other `Collection` type whose elements are `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.

US.2.6 What Exceptions Imply about Future Behavior

When interacting with a remote object, an entity may call methods that result in exceptions. The specification of those methods should define what each possible exception implies (if anything) about the current state of the object. One important

aspect of an object's state is whether or not further interactions with the object are likely to be fruitful. Throughout the helper utilities and services specifications, the following general terms may be used to classify what a given exception implies about the probability of success of future operations on the object that threw the exception:

- Bad object exception: If a method invocation on an object throws a *bad object exception,* it can be assumed that any further operations on that object will also fail.
- Bad invocation exception: If a method invocation on an object throws a *bad invocation exception,* it can be assumed that any retries of the *same* method with the *same* arguments that are expected to return the *same* value will also fail. No new assertions can be made about the probability of success of any future invocation of that method with different arguments or if a different return value is expected, nor can any new assertions be made about the probability of success of invocations of the object's other methods.
- Indefinite exception: If a method invocation on an object throws an *indefinite exception*, no new assertions can be made about the probability of success of any future invocation of that method, regardless of the arguments used or return value expected, nor can any new assertions be made about the probability of success of any *other* operation on the same object.

Unless otherwise noted, the throwing of a bad object, bad invocation, or indefinite exception by one object does not imply anything about the state of another object, even if both objects are associated with the same remote entity.

These terms can be used in the specification of a method to describe the meaning of exceptions that might be thrown, as well as in the specification of what a given utility or service will, may, or should do when it receives an exception in the course of interacting with a given object.

If a specification does not say otherwise, the following classification is used to categorize each `RuntimeException`, `Error`, or `java.rmi.RemoteExceptions` as a bad object, bad invocation, or indefinite exception:

- Bad object exceptions:
  - Any `java.lang.RuntimeException`
  - Any `java.lang.Error` *except* one that is a `java.lang.LinkageError` or `java.lang.OutOfMemoryError`
  - Any `java.rmi.NoSuchObjectException`

- Any `java.rmi.ServerError` with a `detail` field that is a bad object exception
- Any `java.rmi.ServerException` with a `detail` field that is a bad object exception

◆ Bad invocation exceptions:

- Any `java.rmi.MarshalException` with a `detail` field that is a `java.io.ObjectStreamException`
- Any `java.rmi.UnmarshalException` with a `detail` field that is a `java.io.ObjectStreamException`
- Any `java.rmi.ServerException` with a `detail` field that is a bad invocation exception

◆ Indefinite exceptions

- Any `java.lang.OutOfMemoryError`
- Any `java.lang.LinkageError`
- Any `java.rmi.RemoteException` *except* those that can be classified as either a bad invocation or bad object exception

US.2.7 Unavailable Lookup Services

While interacting (or attempting to interact) with a lookup service, an entity may encounter one of the exception types described in the previous section. When the entity does receive such an exception, what may be concluded about the state of the lookup service is dependent on the type of exception encountered.

If an entity encounters a bad object exception while interacting with a lookup service, the entity can usually conclude that the associated proxy it holds can no longer be used to interact with the lookup service. This can be due to any number of reasons. For example, if the lookup service is administratively destroyed, the old proxy will never be capable of communicating with any new incarnations of the lookup service, allowing the entity to dispose of the old proxy since it is no longer of any use to the entity.

If an indefinite exception occurs while interacting with a lookup service, the entity can interpret such an occurrence as a communication failure that may or may not be only temporary.

Finally, entities that encounter a bad invocation exception while interacting with a lookup service should view the lookup service as being in an unknown,

possibly corrupt state, and should discontinue further interaction with that lookup service until the problem is resolved.

Whenever an entity receives any of these exceptions while interacting with a lookup service, the affected lookup service is referred to as *unavailable* or *unreachable*. For most entities the unavailability of a particular lookup service should not prevent the entity from continuing its processing, although in other situations an entity might consider at least some of these exceptional conditions unrecoverable. In general, when an entity encounters an unreachable lookup service, the exception or error indicating that the lookup service is unavailable should be caught and handled, usually by requesting that the lookup service be *discarded* (see the next section), and the entity should continue its processing.

US.2.8 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 itself or by the utility that the entity employs to perform the actual discovery duties.

Whenever a lookup service is discarded by a utility employed by the entity, the utility sends to all of the entity's discovery listeners, a notification event referencing both the discarded lookup service and the member groups to which the lookup service belongs. This event is referred to as a *discarded event*. It may be useful to note that discarded events can be classified by two characteristics:

- Whether the event was generated as a direct consequence of an explicit request made by the entity itself (*active*) or as a consequence of a determination made by some utility employed by the entity (*passive*)
- Whether the event is related to communication problems or to the entity losing interest in discovering the affected lookup services

US.2.8.1 Active Communication Discarded Event

When the occurrence of exceptional conditions causes an entity to conclude that a lookup service is unreachable, the entity typically will request that the lookup service be discarded. When the entity itself requests that such an unreachable lookup service be discarded, the resulting discarded event may be referred to as an *active communication discarded event*. The term *active* is used because the entity takes specific action to request that the lookup service be discarded. Because the entity

cannot communicate with the unreachable lookup service, the event is associated with *communication*.

US.2.8.2 Active No-Interest Discarded Event

Whenever the entity makes a request that results in the removal of an element from the relevant managed set of groups or locators, one or more of the lookup services associated with the removed groups or locators may be discarded—even though the lookup services are still reachable. The lookup services may be discarded in this situation because the contents of the sets of groups and locators the entity wishes to discover may have changed in such a way that one or more of the previously discovered lookup services are no longer of interest to the entity. In this case, if any already discovered lookup service is found to belong to none of the groups in the new managed set of groups or if its locator no longer belongs to the entity's new managed set of locators, a discarded event is generated and sent to all of the entity's discovery listeners. This type of discarded event may be referred to as an *active no-interest discarded event* (active because the entity itself executed an action that resulted in the discarding of one or more lookup services).

US.2.8.3 Passive Communication Discarded Event

If the utility that the entity uses to perform group (multicast) discovery determines that one of the previously discovered lookup services has stopped sending multicast announcements, that utility may discard the lookup service. That is, the utility may remove the lookup service from the managed set and send a discarded event to notify the entity that the lookup service is unavailable. The discarded event sent in this situation is often referred to as a *passive communication discarded event*.

US.2.8.4 Passive No-Interest Discarded Event

If the utility that the entity uses to perform group discovery determines that the member groups of one of the previously discovered lookup services has changed, the utility may discard that lookup service. The lookup service may be discarded in this situation because the lookup service may no longer be a member of any of the groups the entity wishes to discover; that is, the lookup service is no longer of interest to the entity. In this case, the utility sends a discarded event to all of the entity's discovery listeners. This type of discarded event may be referred to as a *passive no-interest discarded event* (passive because the entity itself did not explicitly request that the lookup service be discarded).

If a lookup service is discarded because it was found to be unreachable (associated with a communication discarded event), that lookup service will be made eligible for rediscovery. In this case, the process of discarding a lookup service—either actively or passively—can be viewed as a mechanism for the removal of stale entries in the managed set of lookup services. Discarding such a lookup service removes the need for operations such as lease renewal attempts on a lookup service that is currently unavailable. Upon rediscovery of the discarded lookup service, the entity typically processes the rediscovered lookup service as if it were discovered for the first time.

Any lookup service corresponding to a no-interest discarded event is no longer eligible for discovery until one of the following occurs:

- The entity changes its managed set of locators or its managed set of groups to include, either the discarded lookup service's locator or at least one of its member groups respectively.
- The set of member groups of the discarded lookup service is changed to include one or more of the groups the entity is currently interested in discovering.

US.2.8.5 Changed Event

An event related to the discarded event is referred to as a *changed event*. This event notifies the entity of changes in the contents of the member groups of one or more of the lookup services in the managed set. If the entity registers interest in such an event and if the utility that the entity uses to perform group discovery determines that one or more of those member group sets has indeed changed, then a changed event is sent.

US.2.8.6 Remote Objects, Stubs, and Proxies

The *"Jini™ Technology Glossary"* defines a *remote object* as an object whose methods can be invoked from a Java virtual machine (JVM)[1], 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

[1] The terms "Java virtual machine" or "JVM" mean a virtual machine for the Java platform.

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 in discussing concepts related to RMI, the term *server* (or *remote server*) is sometimes used in place of the term *remote object*.

To initiate an 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 the helper utilities and services specifications. 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. A *smart 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 that has 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, but ultimately occurs in the JVM of the service.

Note that the term *front-end proxy* (or simply *front end*) is often used interchangeably with the term *proxy.* Similarly, the term *back-end server* (or simply, *back end*) is often 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.

US.2.9 Activation

The glossary defines *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* or *deactivation*, respectively. Many Jini services that wish 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.

US.3 Introduction to the Helper Utilities

US.3.1 The Discovery Utilities

THE *Jini Discovery Utilities Specification* defines a set of general-purpose utility interfaces collectively referred to as the discovery management interfaces. Those interfaces define the policies to apply when implementing helper utilities that manage an entity's discovery duties. Currently, the set of discovery management interfaces consists of the following three interfaces:

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

Because the discovery management interfaces provide a uniform way to define utility classes that perform discovery-related management duties on behalf of an entity, the discovery utilities specification defines a number of helper utility classes that implement one or more of these interfaces. Those classes are:

- `LookupDiscovery`
- `LookupLocatorDiscovery`
- `LookupDiscoveryManager`

The discovery utilities specification closes with a discussion of a set of low-level utility classes that can be useful when applying the discovery management policies to build higher-level helper utilities for discovery. Those classes are:

- `Constants`
- `OutgoingMulticastRequest`
- `IncomingMulticastRequest`
- `OutgoingMulticastAnnouncement`

- `IncomingMulticastAnnouncement`
- `OutgoingUnicastRequest`
- `IncomingUnicastRequest`
- `OutgoingUnicastResponse`
- `IncomingUnicastResponse`

US.3.1.1 The `DiscoveryManagement` Interface

The `DiscoveryManagement` interface defines methods related to the discovery event mechanism and discovery process termination. Through this interface an entity can register or unregister `DiscoveryListener` objects to receive discovery events, retrieve proxies to the currently discovered lookup services, discard a lookup service so that it is eligible for rediscovery, or terminate the discovery process.

US.3.1.2 The `DiscoveryGroupManagement` Interface

The `DiscoveryGroupManagement` interface defines methods and constants related to the management of the set containing the names of the groups whose members are the lookup services that are to be discovered via group discovery. The methods of this interface define how an entity retrieves or modifies the managed set of groups to discover.

US.3.1.3 The `DiscoveryLocatorManagement` Interface

The `DiscoveryLocatorManagement` interface defines methods related to the management of the set of `LookupLocator` objects corresponding to the specific lookup services that are to be discovered via locator discovery. The methods of this interface define how an entity retrieves or modifies the managed set of locators to discover.

US.3.1.4 The `LookupDiscovery` Helper Utility

The `LookupDiscovery` helper utility encapsulates the functionality required of an entity that wishes to employ multicast discovery to discover a lookup service located within the entity's *multicast radius*. This utility provides an implementation that makes the process of acquiring lookup service instances, based on no

information other than group membership, which is much simpler for both services and clients.

US.3.1.5 The `LookupLocatorDiscovery` Helper Utility

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

US.3.1.6 The `LookupDiscoveryManager` Helper Utility

The `LookupDiscoveryManager` is a helper utility class 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, locators, and listeners, such an entity can employ this class to provide those facilities on its behalf.

US.3.1.7 The `Constants` Class

The `Constants` class provides easy access to defined constants that may be useful when participating in the discovery process.

US.3.1.8 The `OutgoingMulticastRequest` Utility

The `OutgoingMulticastRequest` class provides facilities for marshalling multicast discovery requests into a form suitable for transmission over a network to announce one's interest in discovering a lookup service.

US.3.1.9 The `IncomingMulticastRequest` Utility

The facilities provided by the `IncomingMulticastRequest` class encapsulate the details of the process of unmarshalling received multicast discovery requests into a form in which the individual parameters of the request may be easily accessed.

US.3.1.10 The `OutgoingMulticastAnnouncement` Utility

The `OutgoingMulticastAnnouncement` class encapsulates the details of the process of marshalling multicast discovery announcements into a form suitable for transmission over a network to announce the availability of a lookup service to interested parties.

US.3.1.11 The `IncomingMulticastAnnouncement` Utility

The `IncomingMulticastAnnouncement` class encapsulates the details of the process of unmarshalling multicast discovery announcements into a form in which the individual parameters of the announcement may be easily accessed.

US.3.1.12 The `OutgoingUnicastRequest` Utility

The `OutgoingUnicastRequest` class encapsulates the details of the process of marshalling unicast discovery requests into a form suitable for transmission over a network to attempt discovery of a specific lookup service.

US.3.1.13 The `IncomingUnicastRequest` Utility

The `IncomingUnicastRequest` class encapsulates the details of the process of unmarshalling unicast discovery requests into a form in which the individual parameters of the request may be easily accessed.

US.3.1.14 The `OutgoingUnicastResponse` Utility

The `OutgoingUnicastResponse` class encapsulates the details of the process of marshalling a unicast discovery response into a form suitable for transmission over a network to respond to a unicast discovery request.

US.3.1.15 The `IncomingUnicastResponse` Utility

The `IncomingUnicastResponse` class encapsulates the details of the process of unmarshalling a unicast discovery response into a form in which the individual parameters of the request may be easily accessed.

US.3.2 The Lease Utilities

The *Jini Lease Utilities Specification* defines helper utility classes, along with supporting interfaces and supporting classes, that encapsulate 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. Currently, this specification defines only one helper utility class:

- `LeaseRenewalManager`

US.3.2.1 The `LeaseRenewalManager` Helper Utility

The `LeaseRenewalManager` is a helper utility class that organizes and manages all of the activities related to the renewal of the leases granted to a Jini client or service by another Jini service. Rather than providing its own facility for coordinating and maintaining all of the necessary state information related to lease renewal, such an entity can employ this class to provide those facilities on its behalf.

US.3.3 The Join Utilities

The *Jini Join Utilities Specification* defines helper utility classes, supporting interfaces, and supporting classes, that encapsulate functionality related to discovery and registration interactions that a well-behaved Jini service will typically have with a lookup service. Currently, this specification defines only one helper utility class:

- `JoinManager`

US.3.3.1 The `JoinManager` Helper Utility

The `JoinManager` is a helper utility class that performs all of the functions related to lookup service discovery, joining, lease renewal, and attribute management, functions that the programming model requires of a well-behaved Jini service. Rather than providing its own facility for providing such functions, a Jini service can employ this class to provide those facilities on its behalf.

US.3.4 The Service Discovery Utilities

The *Jini Service Discovery Utilities Specification* defines helper utility classes (with supporting interfaces and classes) that encapsulate functionality that aids a Jini service or client in acquiring services of interest, registered with the various lookup services with which the service or client wishes to interact. Currently, the service discovery utilities specification defines only one helper utility class:

- `ServiceDiscoveryManager`

US.3.4.1 The `ServiceDiscoveryManager` Helper Utility

The `ServiceDiscoveryManager` class is a helper utility class that any 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. Although the `ServiceDiscoveryManager` performs lookup discovery event handling for clients and services, the primary functionality the `ServiceDiscoveryManager` provides is service discovery and management.

The `ServiceDiscoveryManager` 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 `ServiceDiscoveryManager` 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 `ServiceDiscoveryManager` 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 Technology Core Platform Specification, "Lookup Service".* 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 (for example, because of 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 Technology Core Platform Specification, "Lookup Service".* 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.

US.4 Introduction to the Helper Services

US.4.1 The Lookup Discovery Service

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 separate helper service to perform discovery duties on the entity's behalf. Such an entity may wish to deactivate for various reasons, one being to conserve computational resources. While the entity is deactivated, the helper 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 becomes available. Such a helper service is referred to as a *lookup discovery service*.

The `LookupDiscoveryService` interface defines the lookup discovery helper service. Through that interface, other Jini services and clients may request that discovery processing be performed on their behalf.

US.4.2 The Lease Renewal Service

The *lease renewal service*—defined by the `net.jini.lease.LeaseRenewalService` interface—is a helper 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 they are needed may find the lease renewal service quite useful. Such a service can request that the lease renewal service 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.

Entities that have continuous *access* to a network but that cannot be continuously *connected* to that network (for example, a cell phone), may also find this service useful. By allowing a lease renewal service (which can be continuously connected) to renew the leases on the resources acquired by the entity, the entity

may remain disconnected until needed. This lease renewal service removes the need to perform the discovery and lookup process each time the entity reconnects to the network, potentially resulting in a significant increase in efficiency.

US.4.3 The Event Mailbox Service

The *event mailbox service* defined by the `net.jini.event.EventMailbox` interface is a helper service that can be employed by entities to store event notifications on their behalf. When an entity registers with the event mailbox service, that service will collect events intended for the registered entity until the entity initiates delivery of the events.

A service such as the event mailbox service 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 event mailbox service useful in achieving their respective event delivery goals.

US.5 Dependencies

THE helper utilities and services specifications rely on one or more of the following specifications:

- *Java™ Remote Method Invocation Specification*
- *Java™ Object Serialization Specification*
- *Jini™ Technology Glossary*
- *Jini™ Technology Core Platform Specification*
  - Section DJ “Discovery and Join”
  - Section LE “Distributed Leasing”
  - Section TX “Transaction”
  - Section LU “Lookup Service”
- *Jini Lookup Attribute Schema Specification*

DU

Jini Discovery Utilities Specification

DU.1 Introduction

EACH discovering entity in a Java virtual machine (JVM)[1] on a given host is independently responsible for obtaining references to lookup services. In this specification we first cover a set of *discovery management interfaces* that define the policies to apply when implementing helper utilities that manage an entity's discovery duties: in particular, the management of multicast (group) discovery and unicast (locator) discovery. After the discovery management interfaces are defined, a set of standard helper utility classes that implement one or more of those interfaces is presented. This specification closes with a discussion of a set of lower-level utility classes that can be useful when applying the discovery management policies to build higher-level helper utilities for discovery.

DU.1.1 Dependencies

This specification relies on the following other specifications:

- *Java Object Serialization Specification*
- *The Jini Technology Core Platform Specification,* "Lookup Service"
- *The Jini Technology Core Platform Specification,* "Discovery and Join"

[1] The terms "Java virtual machine" and "JVM" mean a virtual machine for the Java platform.

DU.2 The Discovery Management Interfaces

DU.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, the entity must first discover a lookup service, before the entity can begin interacting with that lookup service.

The interfaces collectively referred to 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 an application environment for Jini technology (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 protocols (see *The Jini Technology Core Platform Specification,* "Discovery and Join"). The methods of this interface define how an entity accesses or modifies the set of groups whose members are lookup services that the entity is interested in discovering through group discovery.

The `DiscoveryLocatorManagement` interface defines methods related to the management of the set of lookup services that are to be discovered using the uni-

cast discovery protocol (as defined in the *Jini Discovery and Join Specification*). The methods of this interface define how an entity accesses or modifies the contents of the set of `LookupLocator` objects corresponding to the specific lookup services the entity has targeted for locator discovery.

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. Such a class would implement both `DiscoveryGroupManagement` and `DiscoveryLocatorManagement`, but not `DiscoveryManagement`.

A specific example of a class that implements only a subset of the set of interfaces specified here is the `LookupDiscovery` utility class defined later in this specification. That class implements both the `DiscoveryManagement` and `DiscoveryGroupManagement` interfaces, but not the `DiscoveryLocatorManagement` interface.

Throughout this discussion of the discovery management interfaces, the phrase *implementation class* refers to any concrete class that implements one or more of those interfaces. The phrase *implementation object* should be understood to mean an instance of such an implementation class. Additionally, whenever a description refers to the *discovering entity* (or simply, the *entity),* that phrase is intended to be interpreted as the object (the client or service) that has created an implementation object, and which wishes to use the public methods specified by these interfaces and provided by that object.

DU.2.2 Other Types

The types defined in the specification of the discovery management interfaces are in the `net.jini.discovery` package. The following additional types may also be

referenced in this specification. 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.DiscoveryChangeListener
net.jini.discovery.LookupDiscovery
net.jini.discovery.LookupDiscoveryManager
java.io.IOException
java.security.Permission
java.util.EventListener
java.util.EventObject
java.util.Map
```

DU.2.3 The `DiscoveryManagement` Interface

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

```
package net.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();
}
```

DU.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`), retrieve proxies to the currently discovered lookup services, 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 should 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.

The `DiscoveryEvent`, `DiscoveryListener`, and `DiscoveryChangeListener` classes are defined later in this specification.

The `addDiscoveryListener` method adds a listener to the set of objects listening for discovery events. This method takes a single argument as input: an instance of `DiscoveryListener` corresponding to the listener 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 the added listener is also an instance of `DiscoveryChangeListener` (a subclass of `DiscoveryListener`), then in addition to receiving events related to discovered and discarded lookup services, that listener will also be notified of group membership changes that occur in any of the lookup services targeted for at least group discovery.

If `null` is input to this method, a `NullPointerException` is thrown. If the listener input to this method duplicates (using the `equals` method) another element in the set of listeners, no action is taken.

Implementations of the `DiscoveryManagement` interface must guarantee reentrancy with respect to `DiscoveryListener` objects registered through this method. Should the instance of `DiscoveryManagement` invoke a method on a registered listener (a local call), calls from that method to any method of the `DiscoveryManagement` instance are guaranteed not to result in a deadlock condition.

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

The `getRegistrars` method returns an array consisting of instances of the `ServiceRegistrar` interface. Each element in the returned set is a proxy to one of the currently discovered lookup services. Each time this method is invoked, a new array is returned. If no lookup services have been discovered, an empty 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: an instance of the `ServiceRegistrar` interface corresponding to the proxy to the lookup service to discard.

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

Currently, there exist utilities such as the `LookupDiscovery` and `LookupDiscoveryManager` helper utilities that will, on behalf of a discovering entity, automatically discard a lookup service upon determining that the lookup service has become unreachable or uninteresting. Although most entities will typically employ such a utility to help with both its discovery as well as its discard duties, it is important to note that if the entity itself determines that the lookup service is unavailable, it is the responsibility of the entity to invoke the `discard` method. This scenario usually happens when the entity attempts to interact with a lookup service, but encounters an exceptional condition (for example, a communication failure). When the entity actively discards a lookup service, the discarded lookup service becomes eligible to be re-discovered. Allowing unreachable lookup services to remain in the managed set can result in repeated and unnecessary attempts to interact with lookup services with which the entity can no longer communicate. Thus, the mechanism provided by this method is intended to provide a way to remove such "stale" lookup service references from the managed set.

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—referred to as a *discarded event*—to be sent to all listeners registered with the implementation object. When this method completes successfully, the lookup service is guaranteed to have been removed from the managed set, and the lookup service is then said to have been "discarded". No such guarantee is made with respect to when the discarded event is sent to the registered listeners. That is, the event notifying the listeners that the lookup service has been discarded may or may not be sent asynchronously.

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.

DU.2.4 The `DiscoveryGroupManagement` Interface

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

```
package net.jini.discovery;

public interface DiscoveryGroupManagement {
    public static final String[] ALL_GROUPS = null;
    public static final String[] NO_GROUPS = new String[0];

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

DU.2.4.1 The Semantics

The `DiscoveryGroupManagement` interface defines methods and constants related to the management of the set containing the names of the groups whose members are the lookup services that are to be discovered using the multicast discovery protocols; 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 managed set of groups to discover, where phrases such as "the groups to discover" or "discovering the desired groups" refer to the discovery of the lookup services that are members of those groups.

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`. Each of these methods throws a `NullPointerException` when at least one element of the input array is `null`.

The empty set is denoted by an empty array, and "no set" is indicated by `null`. 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 `ALL_GROUPS` and the `NO_GROUPS` constants are defined for convenience, and represent no set and the empty set respectively.

The `getGroups` method returns an array consisting of the names of the groups in the managed set; that is, the names of the groups the implementation object is currently configured to discover.