Jini™ Technology Core Platform Specification

A system of Jini™ technology-enabled services and/or devices is a Java™ technology-centered, distributed system designed for simplicity, flexibility, and federation. The Jini architecture provides mechanisms for machines or programs to enter into a federation where each machine or program offers resources to other members of the federation and uses resources as needed. The design of the Jini architecture exploits the ability to move Java programming language code from machine to machine and unifies, under the notion of a service, everything from the user of a system of Jini technology-enabled services and/or devices, to the software available on the machines, to the hardware components of the machines themselves.

Version 1.1
October 2000

Copyright © 2000 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, and JavaBeans 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

Jini™ Technology Core Platform Specification

1 Introduction

THIS document is the 1.1 release of the Jini™ Technology Core Platform Specification. We have reordered the specifications with this release, combining the JCP specifications into a single specification. Chapter designations, for example, AR.1 as the Introduction chapter of the Jini™ Architecture Specification, have remained unchanged.

1.1 Dependencies

This document relies on the following other specifications:

- *The Java™ Remote Method Invocation Specification*
- *The Java™ Object Serialization Specification*

Comments

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

DJ

Discovery and Join

DJ.1 Introduction

ENTITIES that wish to start participating in a distributed a system of Jini technology-enabled services and/or devices, known as a *djinn*, must first obtain references to one or more Jini lookup services. The protocols that govern the acquisition of these references are known as the *discovery* protocols. Once these references have been obtained, a number of steps must be taken for entities to start communicating usefully with services in a djinn; these steps are described by the *join* protocol.

DJ.1.1 Terminology

A *host* is a single hardware device that may be connected to one or more networks. An individual host may house one or more Java virtual machines[1] (JVM).

Throughout this document we make reference to a *discovering entity*, a *joining entity*, or simply an *entity.*

- A *discovering entity* is simply one or more cooperating objects in the Java programming language on the same host that are about to start, or are in the process of, obtaining references to Jini lookup services.
- A *joining entity* comprises one or more cooperating objects in the Java programming language on the same host that have just received a reference to the lookup service and are in the process of obtaining services from, and possibly exporting them to, a djinn.

[1] *As used in this document, the terms “Java virtual machine” or “JVM” mean a virtual machine for the Java platform.

- An *entity* may be a discovering entity, a joining entity, or an entity that is already a member of a djinn; the intended meaning should be clear from the context.
- A *group* is a logical name by which a group of djinns is identified.

Since all participants in a djinn are collections of one or more objects in the Java programming language, this document will not make a distinction between an entity that is a dedicated device using Jini technology or something running in a JVM that is hosted on a legacy system. Such distinctions will be made only when necessary.

DJ.1.2 Host Requirements

Hosts that wish to participate in a djinn must have the following properties:

- A functioning JVM, with access to all packages needed to run software written to the Jini specifications
- A properly configured network protocol stack

The properties required of the network protocol stack will vary depending on the network protocol(s) being used. Throughout this document we will assume that IP is being used, and highlight areas that might apply differently to other networking protocols.

DJ.1.2.1 Protocol Stack Requirements for IP Networks

Hosts that make use of IP for networking must have the following properties:

- An IP address. IP addresses may be statically assigned to some hosts, but we expect that many hosts will have addresses assigned to them dynamically. Dynamic IP addresses are obtained by hosts through use of DHCP.
- Support for unicast TCP and multicast UDP. The former is used by subsystems using Jini technology such as Java Remote Method Invocation (RMI); both are used during discovery.
- Provision of some mechanism (for example, a simple HTTP server) that facilitates the downloading of Java RMI stubs and other necessary code by remote parties. This mechanism does not have to be provided by the host itself, but the code must be made available by some cooperating party.

DJ.1.3 Protocol Overview

There are three related discovery protocols, each designed with different purposes:

- The *multicast request protocol* is employed by entities that wish to discover nearby lookup services. This is the protocol used by services that are starting up and need to locate whatever djinns happen to be close. It can also be used to support browsing of local lookup services.
- The *multicast announcement protocol* is provided to allow lookup services to advertise their existence. This protocol is useful in two situations. When a new lookup service is started, it might need to announce its availability to potential clients. Also, if a network failure occurs and clients lose track of a lookup service, this protocol can be used to make them aware of its availability after network service has been restored.
- The *unicast discovery protocol* makes it possible for an entity to communicate with a specific lookup service. This is useful for dealing with non-local djinns and for using services in specific djinns over a long period of time.

The discovery protocols require support for multicast or restricted-scope broadcast, along with support for reliable unicast delivery, in the transport layer. The discovery protocols make use of the Java platform's object serialization to exchange information in a platform-independent manner.

DJ.1.4 Discovery in Brief

This section provides a brief overview of the operation of the discovery protocols. For a detailed description suitable for use by implementors, see Section DJ.2 "The Discovery Protocols".

DJ.1.4.1 Groups

A group is an arbitrary string that acts as a name. Each lookup service has a set of zero or more groups associated with it. Entities using the multicast request protocol specify a set of groups they want to communicate with, and lookup services advertise the groups they are associated with using the multicast announcement protocol. This allows for flexibility in configuring entities: instead of maintaining a set of URLs for specific lookup services to contact, and that need to be changed if any of these services moves, an entity can maintain a set of group names.

Although group names are arbitrary strings, it is recommended that DNS-style names (for example, “eng.sun.com”) be used to avoid name conflicts. One group name, represented by the empty string, is predefined as the *public* group. Unless otherwise configured, lookup services should default to being members of the public group, and discovering entities should attempt to find lookup services in the public group.

DJ.1.4.2 The Multicast Request Protocol

The multicast request protocol, shown in Figure AR.1.1, proceeds as follows:

1. The entity that wishes to discover a djinn establishes a TCP-based server that accepts references to the lookup service. This server is an instance of the *multicast response* service.
2. Lookup services listen for multicast requests for references to lookup services for the groups they manage. These listening entities are instances of the *multicast request* service. This is *not* an RMI-based service; the protocol is described in Section DJ.2 “The Discovery Protocols”.
3. The discovering entity performs a multicast that requests references to lookup services; it provides a set of groups in which it is interested, and enough information to allow listeners to connect to its multicast response server.
4. Each multicast request server that receives the multicast will, if it is a member of a group for which it receives a request, connect to the multicast response server described in the request, and use the unicast discovery protocol to pass an instance of the lookup service’s implementation of `net.jini.core.lookup.ServiceRegistrar`.

At this point, the discovering entity has one or more remote references to lookup services.

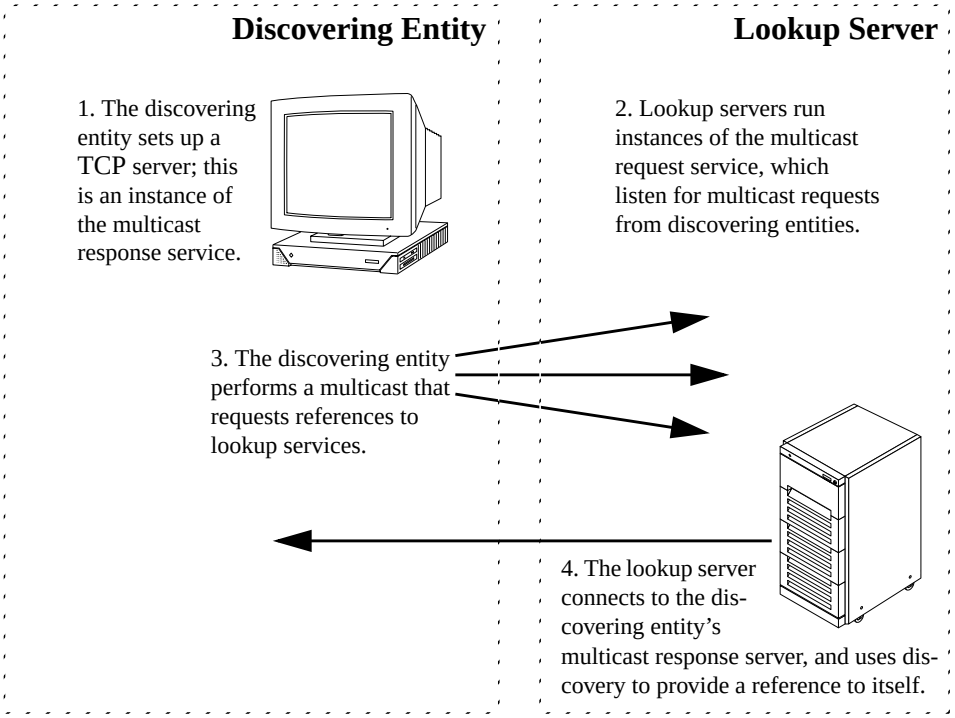


FIGURE DJ.1.1: ***The Multicast Request Protocol***

DJ.1.4.3 The Multicast Announcement Protocol

The multicast announcement protocol follows these steps:

1. Interested entities on the network listen for multicast announcements of the existence of lookup services. If an announcement of interest arrives at such an entity, it uses the unicast discovery protocol to contact the given lookup service.
2. Lookup services prepare to take part in the unicast discovery protocol (see below) and send multicast announcements of their existence at regular intervals.

DJ.1.4.4 The Unicast Discovery Protocol

The unicast discovery protocol works as follows:

1. The lookup service listens for incoming connections and, when a connection is made by a client, decodes the request and, if the request is correct, responds with a marshalled object that implements the `net.jini.core.lookup.ServiceRegistrar` interface.
2. An entity that wishes to contact a particular lookup service uses known host and port information to establish a connection to that service. It sends a discovery request and listens for a marshalled object as above in response.

DJ.2 The Discovery Protocols

THERE are three closely related discovery protocols: one is used to discover one or more lookup services on a local area network (LAN), another is used to announce the presence of a lookup service on a local network, and the last is used to establish communications with a specific lookup service over a wide-area network (WAN).

DJ.2.1 Protocol Roles

The multicast discovery protocols work together over time. When an entity is initially started, it uses the multicast request protocol to actively seek out nearby lookup services. After a limited period of time performing active discovery in this way, it ceases using the multicast request protocol and switches over to listening for multicast lookup announcements via the multicast announcement protocol.

DJ.2.2 The Multicast Request Protocol

The multicast request protocol allows an entity that has just been started, or that needs to provide browsing capabilities to a user, to actively discover nearby lookup services.

DJ.2.2.1 Protocol Participants

Several components take part in the multicast request protocol. Of these, two run on an entity that is performing multicast requests, and two run on the entity that listens for such requests and responds.

On the requesting side live the following components:

- A multicast request client performs multicasts to discover nearby lookup services.

- A multicast response server listens for responses from those lookup services.

These components are paired; they do not occur separately. Any number of pairs of such components may coexist in a single JVM at any given time.

The lookup service houses the other two participants:

- A multicast request server listens for incoming multicast requests.
- A multicast response client responds to callers, passing each a proxy that allows it to communicate with its lookup service.

Although these components are paired, as on the client side, only a single pair will typically be associated with each lookup service.

These local pairings apart, the remote client/server pairings should be clear from the above description and the diagram of protocol participants in Figure AR.2.1.

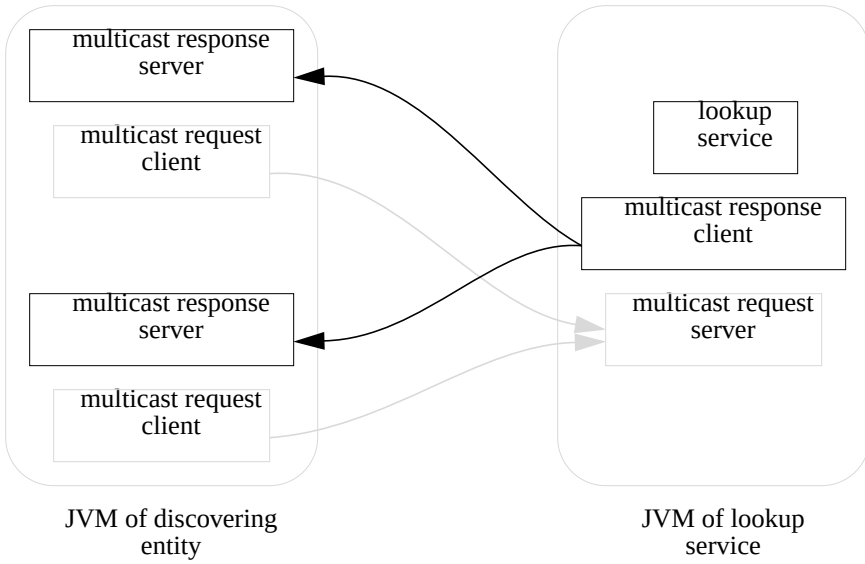


FIGURE DJ.2.1: ***Multicast Request Protocol Participants***

DJ.2.2.2 The Multicast Request Service

The multicast request service is not based on Java RMI; instead, it makes use of the multicast datagram facility of the networking transport layer to request that

lookup services advertise their availability to a requesting host. In a TCP/IP environment the network protocol used is multicast UDP. Request datagrams are encoded as a sequence of bytes, using the data and object serialization facilities of the Java programming language to provide platform independence.

DJ.2.2.3 Request Packet Format

A multicast discovery request packet body must:

- Be 512 bytes in size or less, in order to fit into a single UDP datagram
- Encapsulate its parameters in a platform-independent manner
- Be straightforward to encode and decode

Accordingly, we define the packet format to be a contiguous series of bytes as would be produced by a `java.io.DataOutputStream` object writing into a `java.io.ByteArrayOutputStream` object. The contents of the packet, in order of appearance, are illustrated by the following fragment of pseudocode which generates the appropriate byte array:

```
int protoVersion;                       // protocol version
int port;                               // port to contact
java.lang.String[] groups;              // groups of interest
net.jini.core.lookup.ServiceID[] heard; // known lookups

java.io.ByteArrayOutputStream byteStr =
    new java.io.ByteArrayOutputStream();
java.io.DataOutputStream objStr =
    new java.io.DataOutputStream(byteStr);

objStr.writeInt(protoVersion);
objStr.writeInt(port);
objStr.writeInt(heard.length);
for (int i = 0; i < heard.length; i++) {
    heard[i].writeBytes(objStr);
}
objStr.writeInt(groups.length);
for (int i = 0; i < groups.length; i++) {
    objStr.writeUTF(groups[i]);
```

```
}

byte[] packetBody = byteStr.toByteArray(); // the final result
```

To elaborate on the roles of the variables above:

- The `protoVersion` variable contains an integer that indicates the version of the discovery protocol. This will permit interoperability between different protocol versions. For the current version of the discovery protocol, `protoVersion` must have the value 1.
- The `port` variable contains the TCP port respondents must connect to in order to continue the discovery process.
- The `groups` variable contains a set of strings (organized as an array) naming the groups the entity wishes to discover. This set may be empty, which indicates that all lookup services are being looked for.
- The `heard` variable contains a set of `net.jini.core.lookup.ServiceID` objects (organized as an array) that identify lookup services from which this entity has already heard and that do not need to respond to this request.
- The `packetBody` variable contains the marshalled discovery request in a form that is suitable for putting into a datagram packet or writing to an output stream.

The table below illustrates the contents of a multicast request packet body.

Count	Serialized Type	Description
1	`int`	protocol version
1	`int`	port to connect to
1	`int`	count of lookups heard
variable	`net.jini.core.lookup.ServiceID`	lookups heard
1	`int`	count of groups
variable	`java.lang.String`	groups

If the size of the packet body should exceed 512 bytes, the set of lookups from which an entity has heard must be left incomplete in the packet body, such that the size of the packet body will come to 512 bytes or less. How this is done is not

specified. It is not permissible for implementations to simply truncate packets at 512 bytes.

Similarly, if the number of groups requested causes the size of a packet body to exceed 512 bytes, implementations must perform several separate multicasts, each with a disjoint subset of the full set of groups to be requested, until the entire set has been requested. Each request must contain the largest set of responses heard that will keep the size of the request below 512 bytes.

DJ.2.2.4 The Multicast Response Service

Unlike the multicast request service, the multicast response service is a normal TCP-based service. In this service the multicast response client contacts the multicast response server specified in a multicast request, after which unicast discovery is performed. The multicast response server to contact can be determined by using the source address of the request that has been received, along with the port number encapsulated in that request.

The only difference between the unicast discovery performed in this instance and the normal case is that the entity being connected to initiates unicast discovery, not the connecting entity. An alternative way of looking at this is that in both cases, once the connection has been established, the party that is looking for a lookup service proxy initiates unicast discovery.

DJ.2.3 Discovery Using the Multicast Request Protocol

Now we describe the discovery sequence for local area network (LAN)-based environments that use the multicast request protocol to discover one or more djinns.

DJ.2.3.1 Steps Taken by the Discovering Entity

The entity that wishes to discover a djinn takes the following steps:

1. It establishes a multicast request client, which will send packets to the well-known multicast network endpoint on which the multicast request service operates.
2. It establishes a TCP server socket that listens for incoming connections, over which the unicast discovery protocol is used. This server socket is the multicast response server socket.

3. It creates a set of `net.jini.core.lookup.ServiceID` objects. This set contains service IDs for lookup services from which it has already heard, and is initially empty.
4. It sends multicast requests at periodic intervals. Each request contains connection information for its multicast response server, along with the most recent set of service IDs for lookup services it has heard from.
5. For each response it receives via the multicast response service, it adds the service ID for that lookup service to the set it maintains.
6. The entity continues multicasting requests for some period of time. Once this point has been reached, it unexports its multicast response server and stops making multicast requests.
7. If the entity has received sufficient references to lookup services at this point, it is now finished. Otherwise, it must start using the multicast announcement protocol.

The interval at which requests are performed is not specified, though an interval of five seconds is recommended for most purposes. Similarly, the number of requests to perform is not mandated, but we recommend seven. Since requests may be broken down into a number of separate multicasts, these recommendations do not pertain to the number of packets to be sent.

DJ.2.3.2 Steps Taken by the Multicast Request Server

The system that hosts an instance of the multicast request service takes the following steps:

1. It binds a datagram socket to the well-known multicast endpoint on which the multicast request service lives so that it can receive incoming multicast requests.
2. When a multicast request is received, the discovery request server may use the service ID set from the entity that is sending requests to determine whether it should respond to that entity. If its own service ID is not in the set, and any of the groups requested exactly matches any of the groups it is a member of or the set of groups requested is empty, it must respond. Otherwise, it must not respond.
3. If the entity must be responded to, the request server connects to the other party's multicast response server using the information provided in the

request, and provides a lookup service registrar using the unicast discovery protocol.

DJ.2.3.3 Handling Responses from Multiple Djinns

What happens when there are several djinns on a network, and calls to an entity's discovery response service are made by principals from more than one of those djinns, will depend on the nature of the discovering entity. Possible approaches include the following:

If the entity provides a *finder*-style visual interface that allows a user to choose one or more djinns for their system to join, it should loop at step 4 in section DJ.2.3.1, and provide the ability to:

- Display the names and descriptions of the djinns it has found out about
- Allow the user to select zero or more djinns to join
- Continue to dynamically update its display, until the user has finished their selection
- Attempt to join all of those djinns the user selected

On the other hand, if the behavior of the entity is fully automated, it should follow the join protocol described in Section DJ.3 "The Join Protocol".

DJ.2.4 The Multicast Announcement Protocol

The multicast announcement protocol is used by Jini lookup services to announce their availability to interested parties within multicast radius. Participants in this protocol are the multicast announcement client, which resides on the same system as a lookup service, and the multicast announcement server, at least one instance of which exists on every entity that listens for such announcements.

The multicast announcement client is a long-lived process; it must start at about the same time as the lookup service itself and remain running as long as the lookup service is alive.

DJ.2.4.1 The Multicast Announcement Service

The multicast announcement service uses multicast datagrams to communicate from a single client to an arbitrary number of servers. In a TCP/IP environment the underlying protocol used is multicast UDP.

Multicast announcement packets are constrained by the same requirements as multicast request packets. The fields in a multicast announcement packet body are as follows:

Count	Serialized Type	Description
1	`int`	protocol version
1	`java.lang.String`	host for unicast discovery
1	`int`	port to connect to
1	`net.jini.core.lookup.ServiceID`	service ID of originator
1	`int`	count of groups
variable	`java.lang.String`	groups represented by originator

The fields have the following purposes:

- The protocol version field provides for possible future extensions to the protocol. For the current version of the multicast announcement protocol this field must contain the value 1. This field is written as if using the method `java.io.DataOutput.writeInt`.
- The host field contains the name of a host to be used by recipients to which they may perform unicast discovery. This field is written as if using the method `java.io.DataOutput.writeUTF`.
- The port field contains the TCP port of the above host at which to perform unicast discovery. This field is written as if using the method `java.io.DataOutput.writeInt`.
- The service ID field allows recipients to keep track of the services from which they have received announcements so that they will not need to unnecessarily perform unicast discovery. This field is written as if using the method `net.jini.core.lookup.ServiceID.writeBytes`.
- The count field indicates the number of groups of which the given lookup service is a member. This field is written as if using the method `java.io.DataOutput.writeInt`.
- This is followed by a sequence of strings equal in number to the count field, each of which is a group that the given lookup service is a member of. Each string is written as if using the method `java.io.DataOutput.writeUTF`.

If the size of the set of groups represented by a lookup service causes the size of a multicast announcement packet body to exceed 512 bytes, several separate packets must be multicast, each with a disjoint subset of the full set of groups, such that the full set of groups is represented by all packets.

DJ.2.4.2 The Protocol

The details of the multicast announcement protocol are simple. The entity that runs the lookup service takes the following steps:

1. It constructs a datagram socket object, set up to send to the well-known multicast endpoint on which the multicast announcement service operates.
2. It establishes the server side of the unicast discovery service.
3. It multicasts announcement packets at intervals. The length of the interval is not mandated, but 120 seconds is recommended.

An entity that wishes to listen for multicast announcements performs the following set of steps:

1. It establishes a set of service IDs of lookup services from which it has already heard, using the set discovered by using the multicast request protocol as the initial contents of this set.
2. It binds a datagram socket to the well-known multicast endpoint on which the multicast announcement service operates and listens for incoming multicast announcements.
3. For each announcement received, it determines whether the service ID in that announcement is in the set from which it has already heard. If so, or if the announcement is for a group that is not of interest, it ignores the announcement. Otherwise, it performs unicast discovery using the host and port in the announcement to obtain a reference to the announced lookup service, and then adds this service ID to the set from which it has already heard.

DJ.2.5 Unicast Discovery

While workgroup-level devices need to be able only to discover local djinns, a user might need to be able to access services in djinns that may be dispersed more widely (for example in offices in other cities or on other continents). To this end,

the software at the user's fingertips must be able to obtain a reference to the lookup service of a remote djinn. This is done using the unicast discovery protocol.

The unicast Jini discovery protocol uses the underlying reliable unicast transport protocol provided by the network instead of the unreliable multicast transport. In the case of IP-based networks this means that the unicast discovery protocol uses unicast TCP instead of multicast UDP.

DJ.2.5.1 The Protocol

The unicast discovery protocol is a simple request-response protocol.

If an entity wishes to obtain a reference to a given djinn, the entity has a lookup locator object for that djinn and makes a TCP connection to the host and port specified by that lookup locator. It sends a unicast discovery request (see below), to which the remote host responds.

If a lookup service is responding to a multicast request, the request to which it is responding contains the address and port to respond to, and it makes a TCP connection to that address and port. The respondee sends a unicast discovery request, and the lookup service responds with a proxy.

The protocol diagram in Figure AR.2.2 illustrates the flow when unicast discovery is initiated by a discovering entity.

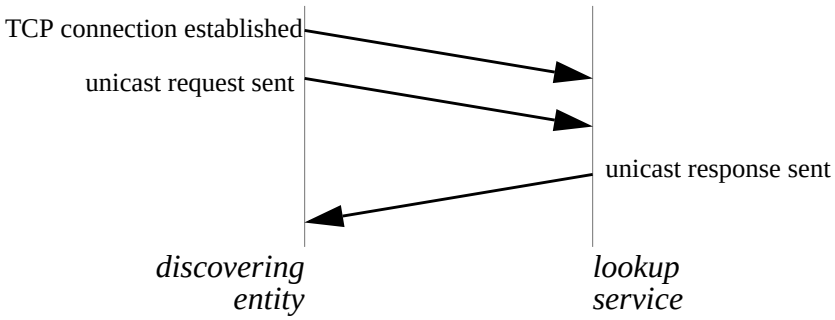


FIGURE DJ.2.2: ***Unicast Discovery Initiated by a Discovering Entity***

The protocol diagram in Figure AR.2.3 indicates the flow when a lookup service initiates unicast discovery in response to a multicast request.

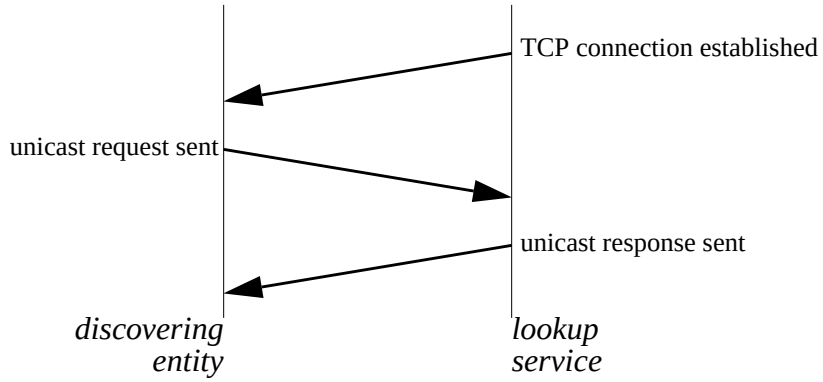


FIGURE DJ.2.3: ***Unicast Discovery Initiated by a Lookup Service***

DJ.2.5.2 Request Format

A discovery request consists of a stream of data as would be obtained by writing code similar to the following:

```
int protoVersion; // protocol version

java.io.ByteArrayOutputStream byteStr =
    new java.io.ByteArrayOutputStream();
java.io.DataOutputStream objStr =
    new java.io.DataOutputStream(byteStr);

objStr.writeInt(protoVersion);

byte[] requestBody = byteStr.toByteArray(); // final result
```

The `protoVersion` variable above must have a value of 1 for the current version of the unicast discovery protocol. The `requestBody` variable contains the discovery request as would be sent to the unicast discovery request service.

DJ.2.5.3 Response Format

The response to the above request consists of a stream of data as would be obtained by writing code similar to the following:

```
net.jini.core.lookup.ServiceRegistrar reg;
    String[] groups; // groups registrar will respond with

java.rmi.MarshalledObject obj =
    new java.rmi.MarshalledObject(reg);
java.io.ByteArrayOutputStream byteStr =
    new java.io.ByteArrayOutputStream();
java.io.ObjectOutputStream objStr = new
    java.io.ObjectOutputStream(byteStr);

objStr.writeObject(obj);
objStr.writeInt(groups.length);
for (int i = 0; i < groups.length; i++) {
    objStr.writeUTF(groups[i]);
}

byte[] responseBody = byteStr.toByteArray(); // final result
```

When the discovering entity receives this data stream, it can deserialize the `MarshalledObject` it has been sent and use the `get` method of that object to obtain a lookup service registrar for that djinn.

DJ.3 The Join Protocol

HAVING covered the discovery protocols, we continue on to describe the join protocol. This protocol makes use of the discovery protocols to provide a standard sequence of steps that services should perform when they are starting up and registering themselves with a lookup service.

DJ.3.1 Persistent State

A service must maintain certain items of state across restarts and crashes. These items are as follows:

- Its service ID. A new service will not have been assigned a service ID, so this will be not be set when a service is started for the first time. After a service has been assigned a service ID, it must continue to use it across all lookup services.
- A set of attributes that describe the service's lookup service entry.
- A set of groups in which the service wishes to participate. For most services this set will initially contain a single entry: the empty string (which denotes the public group).
- A set of specific lookup services to register with. This set will usually be empty for new services.

Note that by "new service" here, we mean one that has never before been started, not one that is being started again or one that has been moved from one network to another.

DJ.3.2 The Join Protocol

When a service initially starts up, it should pause a random amount of time (up to 15 seconds is a reasonable range). This will reduce the likelihood of a packet

storm occurring if power is restored to a network segment that houses a large number of services.

DJ.3.2.1 Initial Discovery and Registration

For each member of the set of specific lookup services to register with, the service attempts to perform unicast discovery of each one and to register with each one. If any fails to respond, the implementor may choose to either retry or give up, but the non-responding lookup service should not be automatically removed from the set if an implementation decides to give up.

Joining Groups

If the set of groups to join is not empty, the service performs multicast discovery and registers with each of the lookup services that either respond to requests or announce themselves as members of one or more of the groups the service should join.

Order of Discovery

The unicast and multicast discovery steps detailed above do not need to proceed in any strict sequence. The registering service must register the same sets of attributes with each lookup service, and must use a single service ID across all registrations.

DJ.3.2.2 Lease Renewal and Handling of Communication Problems

Once a service has registered with a lookup service, it periodically renews the lease on its registration. A lease with a particular lookup service is cancelled only if the registering service is instructed to unregister itself.

If a service cannot communicate with a particular lookup service, the action it takes depends on its relation to that lookup service. If the lookup service is in the persistent set of specific lookup services to join, the service must attempt to reregister with that lookup service. If the lookup service was discovered using multicast discovery, it is safe for the registering service to forget about it and await a subsequent multicast announcement.

DJ.3.2.3 Making Changes and Performing Updates

Attribute Modification

If a service is asked to change the set of attributes with which it registers itself, it saves the changed set in a persistent store, then performs the requested change at each lookup service with which it is registered.

Registering and Unregistering with Lookup Services

If a service is asked to register with a specific lookup service, it adds that lookup service to the persistent set of lookup services it should join, and then registers itself with that lookup service as detailed above.

If a service is asked to unregister from a specific lookup service and that service is in the persistent set of lookup services to join, it should be removed from that set. Whether or not this step needs to be taken, the service cancels the leases for all entries it maintains at that lookup service.

DJ.3.2.4 Joining or Leaving a Group

If a service is asked to join a group, it adds the name of that group to the persistent set of groups to join and either starts or continues to perform multicast discovery using this augmented group.

If the service is requested to leave a group, the steps are a little more complex:

1. It removes that group from the persistent set of groups to join.
2. It removes all lookup services that match only that group in the set of groups it is interested in from the set it has discovered using multicast discovery, and unregisters from those lookup services.
3. It either continues to perform multicast discovery with the reduced set of groups or, if the set has been reduced to empty, ceases multicast discovery.

DJ.4 Network Issues

NOW we will discuss various issues that pertain to the multicast network protocol used by the multicast discovery service. Much of the discussion centers on the Internet protocols, as the lookup discovery protocol is expected to be most heavily used on IP-based internets and intranets.

DJ.4.1 Properties of the Underlying Transport

The network protocol that is used to communicate between a discovering entity and an instance of the discovery request service is assumed to be unreliable and connectionless, and to provide unordered delivery of packets.

This maps naturally onto both IP multicast and local-area IP broadcast, but should work equally well with connection-oriented reliable multicast protocols.

DJ.4.1.1 Limitations on Packet Sizes

Since we assume that the underlying transport does not necessarily deliver packets in order, we must address this fact. Although we could mandate that request packets contain sequence numbers, such that they could be reassembled in order by instances of the discovery request service, this seems excessive. Instead, we require that discovery requests not exceed 512 bytes in size, including headers for lower-level protocols. This squeaks in below the lowest required MTU size that is required to be supported by IP implementations.

DJ.4.2 Bridging Calls to the Discovery Request Service

Whether or not calls to the discovery request service will need to be bridged across LAN or wide area network (WAN) segments will depend on the network protocol being used and the topology of the local network.

In an environment in which every LAN segment happens to host a Jini lookup service, bridging might not be necessary. This does not seem likely to be a typical scenario.

Where the underlying transport is multicast IP, intelligent bridges and routers must be able to forward packets appropriately. This simply requires that they support one of the multicast IP routing protocols; most router vendors already do so.

If the underlying transport were permitted to be local-area IP broadcast, some kind of intelligent broadcast relay would be required, similar to that described in the DHCP and BOOTP specifications. Since this would increase the complexity of the infrastructure needed to support the Jini discovery protocol, we mandate use of multicast IP instead of broadcast IP.

DJ.4.3 Limiting the Scope of Multicasts

In an environment that makes use of IP multicast or a similar protocol, the joining entity should restrict the scope of the multicasts it makes by setting the time-to-live (TTL) field of outgoing packets appropriately. The value of the TTL field is not mandated, but we recommend that it be set to 15.

DJ.4.4 Using Multicast IP as the Underlying Transport

If multicast IP is being used as the underlying transport, request packets are encapsulated using UDP (checksums must be enabled). A combination of a well-known multicast IP address and a well-known UDP port is used by instances of the discovery request service and joining entities.

DJ.4.5 Address and Port Mappings for TCP and Multicast UDP

The port number for Jini lookup discovery requests is `4160`. This applies to both the multicast and unicast discovery protocols. For multicast discovery the IP address of the multicast group over which discovery requests should travel is `224.0.1.85`. Multicast announcements should use the address `224.0.1.84`.

DJ.5 LookupLocator Class

THE `LookupLocator` class provides a simple interface for performing unicast discovery:

```
package net.jini.core.discovery;

import java.io.IOException;
import java.io.Serializable;
import java.net.MalformedURLException;
import net.jini.core.lookup.ServiceRegistrar;

public class LookupLocator implements Serializable {
    public LookupLocator(String host, int port) {…}
    public LookupLocator(String url)
        throws MalformedURLException {…}
    public String getHost() {…}
    public int getPort() {…}
    public ServiceRegistrar getRegistrar()
        throws IOException, ClassNotFoundException {…}
    public ServiceRegistrar getRegistrar(int timeout)
        throws IOException, ClassNotFoundException {…}
}
```

Each constructor takes parameters that allow the object to determine what IP address and TCP port number it should connect to. The first form takes a host name and port number. The second form takes what should be a *jini*-scheme URL. If the URL is invalid, it throws a `java.net.MalformedURLException`. Neither constructor performs the unicast discovery protocol, nor does either resolve the host name passed as argument.

The `getHost` method returns the name of the host with which this object attempts to perform unicast discovery, and the `getPort` method returns the TCP port at that host to which this object connects. The `equals` method returns true if both instances have the same host and port.

There are two forms of `getRegistrar` method. Each performs unicast discovery and returns an instance of the proxy for the specified lookup service, or throws either a `java.io.IOException` or a `java.lang.ClassNotFoundException` if a problem occurs during the discovery protocol. Each method performs unicast discovery every time it is called.

The form of this method that takes a `timeout` parameter will throw a `java.io.InterruptedIOException` if it blocks for more than `timeout` milliseconds while waiting for a response. A similar timeout is implied for the no-arg form of this method, but the value of the timeout in milliseconds may be specified globally using the `net.jini.discovery.timeout` system property, with a default equal to 60 seconds.

DJ.5.1 Jini Technology URL Syntax

While the Uniform Resource Locator (URL) specification merely demands that a URL be of the form `protocol:data`, standard URL syntaxes tend to take one of two forms:

- *protocol*://*host*/*data*
- *protocol*://*host*:*port*/*data*

The protocol component of a Jini technology URL is, not surprisingly, `jini`. The host name component of the URL is an ordinary DNS name or IP address. If the DNS name resolves to multiple IP addresses, it is assumed that a lookup service for the same djinn lives at each address. If no port number is specified, the default is 4160.[2]

The URL has no data component, since the lookup service is generally not searchable by name. As a result, a Jini technology URL ends up looking like

```
jini://example.org
```

with the port defaulting to 4160 since it is not provided explicitly, or, to indicate a non-default port,

```
jini://example.com:4162
```

[2] If you speak hexadecimal, you will notice that 4160 is the decimal representation of (CAFE – BABE).

DJ.5.2 Serialized Form

Class	`serialVersionUID`	Serialized Fields
`LookupLocator`	1448769379829432795L	`String host` `int port`

EN

Entry

EN.1 Entries and Templates

ENTRIES are designed to be used in distributed algorithms for which exact-match lookup semantics are useful. An entry is a typed set of objects, each of which may be tested for exact match with a template.

EN.1.1 Operations

A service that uses entries will support methods that let you use entry objects. In this document we will use the term “operation” for such methods. There are three types of operations:

- *Store operations*—operations that store one or more entries, usually for future matches.
- *Match operations*—operations that search for entries that match one or more templates.
- *Fetch operations*—operations that return one or more entries.

It is possible for a single method to provide more than one of the operation types. For example, consider a method that returns an entry that matches a given template. Such a method can be logically split into two operation types (match and fetch), so any statements made in this specification about either operation type would apply to the appropriate part of the method’s behavior.

EN.1.2 `Entry`

An entry is a typed group of object references represented by a class that implements the marker interface `net.jini.core.entry.Entry`. Two different entries have the same type if and only if they are of the same class.

```
package net.jini.core.entry;

public interface Entry extends java.io.Serializable { }
```

For the purpose of this specification, the term "field" when applied to an entry will mean fields that are public, non-static, non-transient, and non-final. Other fields of an entry are not affected by entry operations. In particular, when an entry object is created and filled in by a fetch operation, only the public non-static, non-transient, and non-final fields of the entry are set. Other fields are not affected, except as set by the class's no-arg constructor.

Each `Entry` class must provide a public no-arg constructor. Entries may not have fields of primitive type (`int`, `boolean`, etc.), although the objects they refer to may have primitive fields and non-public fields. For any type of operation, an attempt to use a malformed entry type that has primitive fields or does not have a no-arg constructor throws `IllegalArgumentException`.

EN.1.3 Serializing `Entry` Objects

`Entry` objects are typically not stored directly by an entry-using service (one that supports one or more entry operations). The client of the service will typically turn an `Entry` into an implementation-specific representation that includes a serialized form of the entry's class and each of the entry's fields. (This transformation is typically not explicit but is done by a client-side proxy object for the remote service.) It is these implementation-specific forms that are typically stored and retrieved from the service. These forms are not directly visible to the client, but their existence has important effects on the operational contract. The semantics of this section apply to all operation types, whether the above assumptions are true or not for a particular service.

Each entry has its fields serialized separately. In other words, if two fields of the entry refer to the same object (directly or indirectly), the serialized form that is compared for each field will have a separate copy of that object. This is true only of different fields of an entry; if an object graph of a particular field refers to the same object twice, the graph will be serialized and reconstituted with a single copy of that object.

A fetch operation returns an entry that has been created by using the entry type's no-arg constructor, and whose fields have been filled in from such a serialized form. Thus, if two fields, directly or indirectly, refer to the same underlying object, the fetched entry will have independent copies of the original underlying object.

This behavior, although not obvious, is both logically correct and practically advantageous. Logically, the fields can refer to object graphs, but the entry is not itself a graph of objects and so should not be reconstructed as one. An entry (relative to the service) is a set of separate fields, not a unit of its own. From a practical standpoint, viewing an entry as a single graph of objects requires a matching service to parse and understand the serialized form, because the ordering of objects in the written entry will be different from that in a template that can match it.

The serialized form for each field is a `java.rmi.MarshalledObject` object instance, which provides an `equals` method that conforms to the above matching semantics for a field. `MarshalledObject` also attaches a codebase to class descriptions in the serialized form, so classes written as part of an entry can be downloaded by a client when they are retrieved from the service. In a store operation, the class of the entry type itself is also written with a `MarshalledObject`, ensuring that it, too, may be downloaded from a codebase.

EN.1.4 `UnusableEntryException`

A `net.jini.core.entry.UnusableEntryException` will be thrown if the serialized fields of an entry being fetched cannot be deserialized for any reason:

```
package net.jini.core.entry;

public class UnusableEntryException extends Exception {
    public Entry partialEntry;
    public String[] unusableFields;
    public Throwable[] nestedExceptions;
    public UnusableEntryException(Entry partial,
        String[] badFields, Throwable[] exceptions) {…}
    public UnusableEntryException(Throwable e) {…}
}
```

The `partialEntry` field will refer to an entry of the type that would have been fetched, with all the usable fields filled in. Fields whose deserialization caused an exception will be `null` and have their names listed in the `unusableFields` string array. For each element in `unusableFields` the corresponding element of

`nestedExceptions` will refer to the exception that caused the field to fail deserialization.

If the retrieved entry is corrupt in such a way as to prevent even an attempt at field deserialization (such as being unable to load the exact class for the entry), `partialEntry` and `unusableFields` will both be `null`, and `nestedExceptions` will be a single element array with the offending exception.

The kinds of exceptions that can show up in `nestedExceptions` are:

- `ClassNotFoundException`: The class of an object that was serialized cannot be found.
- `InstantiationException`: An object could not be created for a given type.
- `IllegalAccessException`: The field in the entry was either inaccessible or `final`.
- `java.io.ObjectStreamException`: The field could not be deserialized because of object stream problems.
- `java.rmi.RemoteException`: When a `RemoteException` is the nested exception of an `UnusableEntryException`, it means that a remote reference in the entry's state is no longer valid (more below). Remote errors associated with a method that is a fetch operation (such as being unable to contact a remote server) are not reflected by `UnusableEntryException` but in some other way defined by the method (typically by the method throwing `RemoteException` itself).

Generally speaking, storing a remote reference to a non-persistent remote object in an entry is risky. Because entries are stored in serialized form, entries stored in an entry-based service will typically not participate in the garbage collection that keeps such references valid. However, if the reference is not persistent because the referenced server does not export persistent references, that garbage collection is the only way to ensure the ongoing validity of a remote reference. If a field contains a reference to a non-persistent remote object, either directly or indirectly, it is possible that the reference will no longer be valid when it is deserialized. In such a case the client code must decide whether to remove the entry from the entry-fetching service, to store the entry back into the service, or to leave the service as it is.

In the Java 2 platform, activatable object references fit this need for persistent references. If you do not use a persistent type, you will have to handle the above problems with remote references. You may choose instead to have your entries store information sufficient to look up the current reference rather than putting actual references into the entry.

EN.1.5 Templates and Matching

Match operations use entry objects of a given type, whose fields can either have *values* (references to objects) or *wildcards* (`null` references). When considering a template *T* as a potential match against an entry *E*, fields with values in *T* must be matched exactly by the value in the same field of *E*. Wildcards in *T* match any value in the same field of *E*.

The type of *E* must be that of *T* or be a subtype of the type of *T*, in which case all fields added by the subtype are considered to be wildcards. This enables a template to match entries of any of its subtypes. If the matching is coupled with a fetch operation, the fetched entry must have the type of *E*.

The values of two fields match if `MarshalledObject.equals` returns `true` for their `MarshalledObject` instances. This will happen if the bytes generated by their serialized form match, ignoring differences of serialization stream implementation (such as blocking factors for buffering). Class version differences that change the bytes generated by serialization will cause objects not to match. Neither entries nor their fields are matched using the `Object.equals` method or any other form of type-specific value matching.

You can store an entry that has a `null`-valued field, but you cannot match explicitly on a `null` value in that field, because `null` signals a wildcard field. If you have a field in an entry that may be variously `null` or not, you can set the field to `null` in your entry. If you need to write templates that distinguish between set and unset values for that field, you can (for example) add a `Boolean` field that indicates whether the field is set and use a `Boolean` value for that field in templates.

An entry that has no wildcards is a valid template.

Serialized Form

Class	`serialVersionUID`	Serialized Fields
`UnusableEntryException`	–2199083666668626172L	*all public fields*

LE

Distributed Leasing

LE.1 Introduction

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

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

LE.1.1 Leasing and Distributed Systems

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

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

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

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

As a consequence of this, distributed systems, especially those with persistent state, are prone to accumulations of outdated and unwanted information. The accumulation of such information, which can include objects stored for future use and subsequently forgotten, may be slow, but the trend is always upward. Over the (comparatively) long life of a distributed system, such unwanted information can grow without upper bound, taking up resources and compromising the performance of the overall system.

A standard way of dealing with these problems is to consider the cleanup of unused resources to be a system administration task. When such resources begin to get scarce, a human administrator is given the task of finding resources that are no longer needed and deleting them. This solution, however, is error prone (since the administrator is often required to judge the use of a resource with no actual

evidence about whether or not the resource is being used) and tends to happen only when resource consumption has gotten out of hand.

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

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

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

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

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

LE.1.2 Goals and Requirements

The requirements of this set of interfaces are:

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

The goals of this chapter are:

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

LE.2 Basic Leasing Interfaces

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

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

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

LE.2.1 Characteristics of a Lease

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

- A lease is a time period during which the grantor of the lease ensures (to the best of the grantor's abilities) that the holder of the lease will have access to some resource. The time period of the lease can be determined solely by the lease grantor, or can be a period of time that is negotiated between the holder of the lease and the grantor of the lease. Duration negotiation need not be multi-round; it often suffices for the requestor to indicate the time desired and the grantor to return the actual time of grant.

- During the period of a lease, a lease can be cancelled by the entity holding the lease. Such a cancellation allows the grantor of the lease to clean up any resources associated with the lease and obliges the grantor of the lease to not take any action involving the lease holder that was part of the agreement that was the subject of the lease.
- A lease holder can request that a lease be renewed. The renewal period can be for a different time than the original lease, and is also subject to negotiation with the grantor of the lease. The grantor may renew the lease for the requested period or a shorter period or may refuse to renew the lease at all. However, when renewing a lease the grantor cannot, unless explicitly requested to do so, shorten the duration of the lease so that it expires before it would have if it had not been renewed. A renewed lease is just like any other lease and is itself subject to renewal.
- A lease can expire. If a lease period has elapsed with no renewals, the lease expires, and any resources associated with the lease may be freed by the lease grantor. Both the grantor and the holder are obliged to act as though the leased agreement is no longer in force. The expiration of a lease is similar to the cancellation of a lease, except that no communication is necessary between the lease holder and the lease grantor.

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

LE.2.2 Basic Operations

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

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

import java.rmi.RemoteException;

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

```
    int DURATION = 1;
    int ABSOLUTE = 2;

    long getExpiration();
    void cancel() throws UnknownLeaseException,
                         RemoteException;
    void renew(long duration) throws LeaseDeniedException,
                                     UnknownLeaseException,
                                     RemoteException;
    void setSerialFormat(int format);
    int getSerialFormat();
    LeaseMap createLeaseMap(long duration);
    boolean canBatch(Lease lease);
}
```

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

The interface defines two constants that can be used when requesting a lease. The first, `FOREVER`, can be used to request a lease that never expires. When granted such a lease, the lease holder is responsible for ensuring that the leased resource is freed when no longer needed. The second constant, `ANY`, is used by the requestor to indicate that no particular lease time is desired and that the grantor of the lease should supply a time that is most convenient for the grantor.

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

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

The first method in the `Lease` interface, `getExpiration`, returns a `long` that indicates the time, relative to the current clock, that the lease will expire. Follow-

ing the usual convention in the Java programming language, this time is represented as milliseconds from the beginning of the epoch and can be used to compare the expiration time of the lease with the result of a call to obtain the current time, `java.lang.System.currentTimeMillis`.

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

The third method, `renew`, is used to renew a lease for an additional period of time. The length of the desired renewal is given, in milliseconds, in the parameter to the call. This duration is not added to the original lease, but is used to determine a new expiration time for the existing lease. This method has no return value; if the renewal is granted, this is reflected in the lease object on which the call was made. If the lease grantor is unable or unwilling to renew the lease, a `LeaseDeniedException` is thrown. If a renewal fails, the lease is left intact for the same duration that was in force prior to the call to `renew`. If the lease being renewed is unknown to the lease grantor, an `UnknownLeaseException` is thrown. The method can also throw a `RemoteException` if the implementation of the method requires calling a remote object that is the lease holder.

As with a call that grants a lease, the duration requested in a `renew` call need not be honored by the entity granting the lease. A renewal may not be for longer than the duration requested, but the grantor may decide to renew a lease for a period of time that is shorter than the duration requested. However, the new lease cannot have a duration that is shorter than the duration remaining on the lease being renewed unless a shorter duration is specifically requested.

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

durational. The second method, `getSerialFormat`, returns an integer indicating the format that will be used to serialize the `Lease` object.

The last two methods are used to aid in the batch renewal or cancellation of a group of `Lease` objects. The first of these, `createLeaseMap`, creates a `Map` object that can contain leases whose renewal or cancellation can be batched and adds the current lease to that map. The current lease will be renewed for the duration indicated by the argument to the method when all of the leases in the `LeaseMap` are renewed. The second method, `canBatch`, returns a boolean value indicating whether or not the lease given as an argument to the method can be batched (in `renew` and `cancel` calls) with the current lease. Whether or not two `Lease` objects can be batched is an implementation detail determined by the objects. However, if a `Lease` object can be batched with any other `Lease` object, the set of objects that can be batched must form an equivalence class. That is, the `canBatch` relationship must be reflexive, symmetric, and associative. This means that, for any three `Lease` objects `x`, `y`, and `z` that return `true` for any instance of the `canBatch` call, it will be the case that:

- `x.canBatch(x)` is `true`
- if `x.canBatch(y)` is `true` then `y.canBatch(x)` is `true`
- if `x.canBatch(y)` is `true` and `y.canBatch(z)` is `true`, then `x.canBatch(z)` is `true`

In addition to the above methods, an object that implements the `Lease` interface will probably need to override the `equals` and `hashcode` methods inherited from `Object`. It is likely that such leases, while appearing as local objects, will in fact contain remote references—either explicitly copied or passed via a method call—to implementation-specific objects in the address space of the lease grantor. These local references may even include their own state (such as the expiration time of the lease) that may, over time, vary from the actual expiration time of the lease to which they refer. Two such references should evaluate as equal (and have the same `hashcode` value) when they refer to the same lease in the grantor, which will not be reflected by the default implementation of the `equals` method.

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

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

The `UnknownLeaseException` is used to indicate that the `Lease` object used is not known to the grantor of the lease. This can occur when a lease expires or

when a copy of a lease has been cancelled by some other lease holder. This exception is defined as:

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

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

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

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

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

The `LeaseException` superclass is defined as:

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

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

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

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

import java.rmi.RemoteException;

public interface LeaseMap extends java.util.Map {
    boolean canContainKey(Object key);
    void renewAll() throws LeaseMapException, RemoteException;
    void cancelAll() throws LeaseMapException,RemoteException;
}
```

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

The first method defined in the `LeaseMap` interface, `canContainKey`, takes a `Lease` object as an argument and returns `true` if that `Lease` object can be added to the `Map` and `false` otherwise. A `Lease` object can be added to a `Map` if that `Lease` object can be renewed in a batch with the other objects in the `LeaseMap`. The requirements for this depend on the implementation of the `Lease` object. However, if a `LeaseMap` object, `m`, contains a `Lease` object, `n`, then for some `Lease` object `o`, `n.canBatch(o)` returns `true` if and only if `m.canContainKey(o)` returns `true`.

The second method, `renewAll`, will attempt to renew all of the `Lease` objects in the `LeaseMap` for the duration associated with the `Lease` object. If all of the `Lease` objects are successfully renewed, the method will return nothing. If some `Lease` objects fail to renew, those objects will be removed from the `LeaseMap` and will be contained in the thrown `LeaseMapException`.

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

The LeaseMapException class is defined as:

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

import java.util.Map;

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

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

LE.2.3 Leasing and Time

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

The difficulty of synchronizing clocks in a distributed system is well known. The problem is somewhat more tractable in the case of leases, which are expected to be for periods of minutes to months, as the accuracy of synchronization required is expected to be in terms of minutes rather than nanoseconds. Over a particular local group of machines, a time service could be used that would allow this level of synchronization.

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

The reasoning behind such a choice is based on the observation that the accuracy of the clocks used in the machines that make up a distributed system is matched much more closely than the clocks on those systems. While there may be minutes of difference in the notion of the absolute time held by widely separated

systems, there is much less likelihood of a significant difference over the rate of change of time in those systems. While there is clearly some difference in the notion of duration between systems (if there were not, synchronization for absolute time would be much easier), that difference is not cumulative in the way errors in absolute time are.

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

LE.2.4 Serialized Forms

Class	`serialVersionUID`	Serialized Fields
`LeaseException`	–7902272546257490469L	*all public fields*
`UnknownLeaseException`	–2921099330511429288L	*none*
`LeaseDeniedException`	5704943735577343495L	*none*
`LeaseMapException`	–4854893779678486122L	*none*

LE.3 Example Supporting Classes

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

However, a number of classes can be supplied that can simplify the handling of leases in some common cases. We will describe examples of these supporting classes and show how these classes can be used with leased resources. Please note that complete specifications for such utilities and services do exist and may differ in some degree from these examples.

LE.3.1 A Renewal Class

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

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

tener of the appropriate type. This functionality can be supplied by a class with an interface like the following:

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

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

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

The duration of a particular lease can be queried by a call to the method `getExpiration`. This method takes a `Lease` object and returns the time at which that lease will be allowed to expire by the `LeaseRenew` object. Note that this is different from the `Lease.getExpiration` method, which tells the time at which the lease will expire if it is not renewed. If there is no `Lease` object corresponding to the argument for this call being handled by the `LeaseRenew` object, an `UnknownLeaseException` will be thrown. This can happen either when no such `Lease` has ever been given to the `LeaseRenew` object, or when a `Lease` object that has been held has already expired or been cancelled. Notice that because this

object is assumed to be in the same address space as the object that acquired the lease, we can also assume that it shares the same clock with that object, and hence can use absolute time rather than a duration-based system.

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

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

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

LE.3.2 A Renewal Service

Objects that hold a lease that needs to be renewed may themselves be activatable, and thus unable to ensure that they will be capable of renewing a lease at some particular time in the future (since they might not be active at that time). For such objects it might make sense to hand the lease renewal duty off to a service that could take care of lease renewal for the object, allowing that object to be deactivated without fear of losing its lease on some other resource.

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

The interface to such an object might look something like:

```
interface LeaseRenewService extends Remote {
    EventRegistration renew(Lease toRenew,
                            long renewTil,
                            RemoteEventListenter notifyBeforeDrop,
                            MarshalledObject returnOnNotify)
```

```
        throws RemoteException;
    void onRenewFailure(Lease toRenew,
                        RemoteEventListenter toNotify,
                        MarshalledObject returnOnNotify)
        throws RemoteException, UnknownLeaseException;
}
```

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

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

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

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

plied in a previous invocation of the method `renew`. If the `Lease` is not known to the `LeaseRenewService` object, an `UnknownLeaseException` will be thrown.

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

EV

Distributed Events

EV.1 Introduction

THE purpose of the distributed event interfaces specified in this document is to allow an object in one Java virtual machine (JVM) to register interest in the occurrence of some event occurring in an object in some other JVM, perhaps running on a different physical machine, and to receive a notification when an event of that kind occurs.

EV.1.1 Distributed Events and Notifications

Programs based on an object that is reacting to a change of state somewhere outside the object are common in a single address space. Such programs are often used for interactive applications in which user actions are modeled as events to which other objects in the program react. Delivery of such *local events* can be assumed to be well ordered, very fast, predictable, and reliable. Further, the entity that is interested in the event can be assumed to always want to know about the event as soon as the event has occurred.

The same style of programming is useful in distributed systems, where the object reacting to an event is in a different JVM, perhaps on a different physical machine, from the one on which the event occurred. Just as in the single-JVM case, the logic of such programs is often reactive, with actions occurring in response to some change in state that has occurred elsewhere.

A distributed event system has a different set of characteristics and requirements than a single-address-space event system. Notifications of events from remote objects may arrive in different orders on different clients, or may not arrive at all. The time it takes for a notification to arrive may be long (in comparison to the time for computation at either the object that generated the notification or the

object interested in the notification). There may be occasions in which the object wishing the event notification does not wish to have that notification as soon as possible, but only on some schedule determined by the recipient. There may even be times when the object that registered interest in the event is not the object to which a notification of the event should be sent.

Unlike the single-address-space notion of an event, a distributed event cannot be guaranteed to be delivered in a timely fashion. Because of the possibilities of network delays or failures, the notification of an event may be delayed indefinitely and even lost in the case of a distributed system.

Indeed, there are times in a distributed system when the object of a notification may actively desire that the notification be delayed. In systems that allow object activation (such as is allowed by Java Remote Method Invocation (RMI) in the Java), an object might wish to be able to find out whether an event occurred but not want that notification to cause an activation of the object if it is otherwise quiescent. In such cases, the object receiving the event might wish the notification to be delayed until the object requests notification delivery, or until the object has been activated for some other reason.

Central to the notion of a distributed notification is the ability to place a third-party object between the object that generates the notification and the party that ultimately wishes to receive the notification. Such third parties, which can be strung together in arbitrary ways, allow ways of off-loading notifications from objects, implementing various delivery guarantees, storing of notifications until needed or desired by a recipient, and the filtering and rerouting of notifications. In a distributed system in which full applications are made up of components assembled to produce an overall application, the third party may be more than a filter or storage spot for a notification; in such systems it is possible that the third party is the final intended destination of the notification.

EV.1.2 Goals and Requirements

The requirements of this set of interfaces are to:

- Specify an interface that can be used to send a notification of the occurrence of the event
- Specify the information that must be contained in such a notification

In addition, the fact that the interfaces are designed to be used by objects in different virtual machines, perhaps separated by a network, imposes other requirements, including:

- Allowing various degrees of assurance on delivery of a notification

- Support for different policies of scheduling notification
- Explicitly allowing the interposition of objects that will collect, hold, filter, and forward notifications

Notice that there is no requirement for a single interface that can be used to register interest in a particular kind of event. Given the wide variety of kinds of events, the way in which interest in such events can be indicated may vary from object to object. This document will talk about a model that lies behind the system's notion of such a registration, but the interfaces that are used to accomplish such a registration are not open to general description.

EV.2 The Basic Interfaces

THE basic interfaces you are about to see define a protocol that can be used by one object to register interest in a kind of state change in another object, and to receive a notification of an occurrence of that kind of state change, either directly or through some third-party, that is specified by the object at the time of registration. The protocol is meant to be as simple as possible. No attempt is made to indicate the reliability or the timeliness of the notifications; such guarantees are not part of the protocol but instead are part of the implementation of the various objects involved.

In particular, the purpose of these interfaces is:

- To show the information needed in any method that allows registration of interest in the occurrence of a kind of event in an object
- To provide an example of an interface that allows the registration of interest in such events
- To specify an interface that can be used to send a notification of the occurrence of the event

Implicit in the event registration and notification is the idea that events can be classified into *kinds*. Registration of interest indicates the kind of event that is of interest, while a notification indicates that an instance of that kind of event has occurred.

EV.2.1 Entities Involved

An *event* is something that happens in an object, corresponding to some change in the abstract state of the object. Events are abstract occurrences that are not directly observed outside of an object, and might not correspond to a change in the *actual* state of the object that advertises the ability to register interest in the event. However, an object may choose to export an identification of a kind of event and allow other objects to indicate interest in the occurrence of events of that kind; this indi-

cates that the *abstract* state of the object includes the notion of this state changing. The information concerning what kinds of events occur within an object can be exported in a number of ways, including identifiers for the various events or methods allowing registration of interest in that kind of event.

An object is responsible for identifying the kinds of events that can occur within that object, allowing other objects to register interest in the occurrence of such events, and generating `RemoteEvent` objects that are sent as notifications to the objects that have registered interest when such events occur.

Registration of interest is not temporally open ended but is limited to a given duration using the notion of a lease. Full specification of the way in which leasing is used is contained in Section LE "Distributed Leasing".

The basic, concrete objects involved in a distributed event system are:

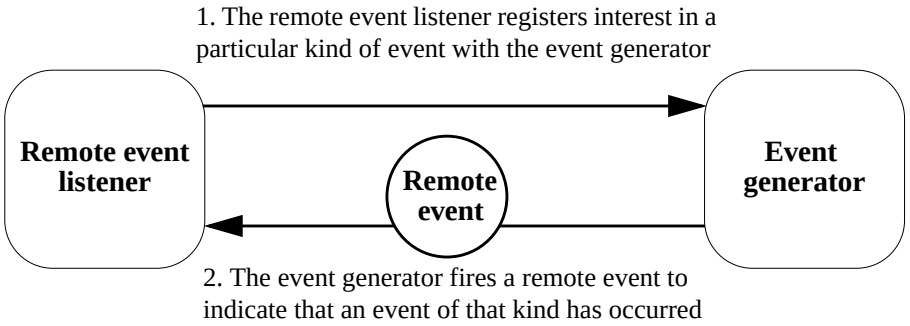
- The object that registers interest in an event
- The object in which an event occurs (referred to as the event generator)
- The recipient of event notifications (referred to as a remote event listener)

An *event generator* is an object that has some kinds of abstract state changes that might be of interest to other objects and allows other objects to register interest in those events. This is the object that will generate notifications when events of this kind occur, sending those notifications to the event listeners that were indicated as targets in the calls that registered interest in that kind of event.

A *remote event listener* is an object that is interested in the occurrence of some kinds of events in some other object. The major function of a remote event listener is to receive notifications of the occurrence of an event in some other object (or set of objects).

A *remote event* is an object that is passed from an event generator to a remote event listener to indicate that an event of a particular kind has occurred. At a minimum, a remote event contains information about the kind of event that has occurred, a reference to the object in which the event occurred, and a sequence number allowing identification of the particular instance of the event. A notifica-

tion will also include an object that was supplied by the object that registered interest in the kind of event as part of the registration call.



EV.2.2 Overview of the Interfaces and Classes

The event and notification interfaces introduced here define a single basic type of entity, a set of requirements on the information that needs to be handed to that entity, and some supporting interfaces and classes. All of the classes and interfaces defined in this specification are in the `net.jini.core.event` package.

The basic type is defined by the interface `RemoteEventListener`. This interface requires certain information to be passed in during the registration of interest in the kind of event that the notification is indicating. There is no single interface that defines how to register interest in such events, but the ways in which such information could be communicated will be discussed.

The supporting interfaces and classes define a `RemoteEvent` object, an `EventRegistration` object used as an identifier for registration, and a set of exceptions that can be generated.

The `RemoteEventListener` is the receiver of `RemoteEvents`, which signals that a particular kind of event has occurred. A `RemoteEventListener` is defined by an interface that contains a single method, `notify`, which informs interested listeners that an event has occurred. This method returns no value, and has parameters that contain enough information to allow the method call to be idempotent. In addition, this method will return information that was passed in during the registration of interest in the event, allowing the *registrant,* the object that registered interest with the event generator, to associate arbitrary information or actions with the notification.

The RemoteEventListener interface extends from the Remote interface, so the methods defined in RemoteEventListener are remote methods and objects supporting these interfaces will be passed by RMI, by reference. Other objects defined by the system will be local objects, passed by value in the remote calls.

The first of these supporting classes is RemoteEvent, which is sent to indicate that an event of interest has occurred in the event generator. The basic form of a RemoteEvent contains:

- An identifier for the kind of event in which interest has been registered
- A reference to the object in which the event occurred
- A sequence number identifying the instance of the event type
- An object that was passed in, as part of the registration of interest in the event by the registrant

These RemoteEvent notification objects are passed to a RemoteEventListener as a parameter to the RemoteEventListener notify method.

The EventRegistration class defines an object that returns the information needed by the registrant and is intended to be the return value of remote event registration calls. Instances of the EventRegistration class contain an identifier for the kind of event, the current sequence number of the kind of event, and a Lease object for the registration of interest.

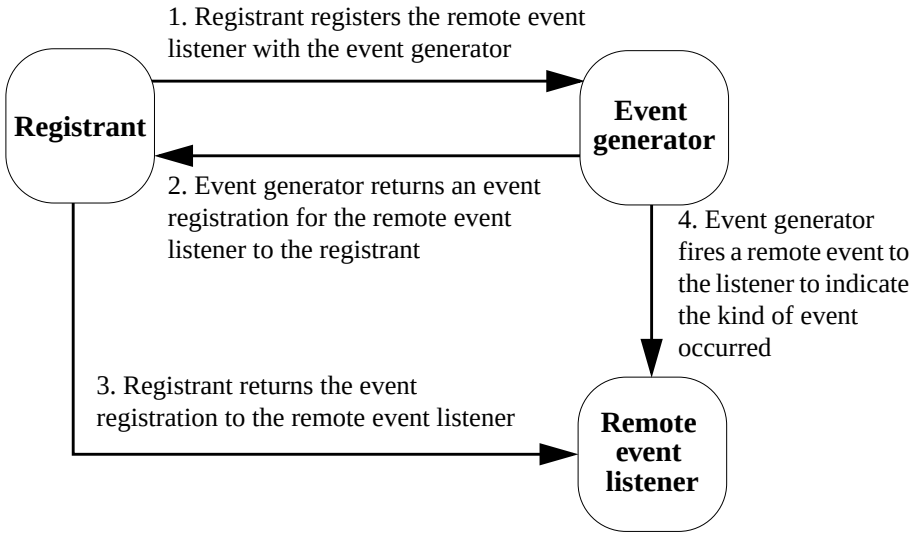
Although there is no single interface that allows for the registration of event notifications, there are a number of requirements that would be put on any such interface if it wished to conform with the remote event registration model. In particular, any such interface should reflect:

- Event registrations are bounded in time in a way that allows those registrations to be renewed when necessary. This can easily be reflected by returning, as part of an event registration, a lease for that registration.
- Notifications need not be delivered to the entity that originally registered interest in the event. The ability to have third-party filters greatly enhances the functionality of the system. The easiest way to allow such functionality is to allow the specification of the RemoteEventListener to receive the notification as part of the original registration call.
- Notifications can contain a MarshalledObject supplied by the original registrant, allowing the passing of arbitrary information (including a closure that is to be run on notification) as part of the event notification, so the registration call should include a MarshalledObject that is to be passed as part of the RemoteEvent.

EV.2.3 Details of the Interfaces and Classes

EV.2.3.1 The `RemoteEventListener` Interface

The `RemoteEventListener` interface needs to be implemented by any object that wants to receive a notification of a `RemoteEvent` from some other object. The object supporting the `RemoteEventListener` interface does not have to be the object that originally registered interest in the occurrence of an event. To allow the notification of an event's occurrence to be sent to an entity other than the one that registered with the event generator, the registration call needs to accept a destination parameter that indicates the object to which the notification should be sent. This destination must be an object that implements the `RemoteEventListener` interface.



The `RemoteEventListener` interface extends the `Remote` interface (indicating that it is an interface to a `Remote` object) and the `java.util.EventListener` interface. This latter interface is used in the Java Abstract Window Toolkit (AWT) and JavaBeans components to indicate that an interface is the recipient of event

notifications. The RemoteEventListener interface consists of a single method, notify:

```
public interface RemoteEventListener extends Remote,
    java.util.EventListener
{
    void notify(RemoteEvent theEvent)
        throws UnknownEventException, RemoteException;
}
```

The notify method has a single parameter of type RemoteEvent that encapsulates the information passed as part of a notification. The RemoteEvent base class extends the class java.util.EventObject that is used in both JavaBeans components and AWT components to propagate event information. The notify method returns nothing but can throw exceptions.

EV.2.3.2 The RemoteEvent Class

The public part of the RemoteEvent class is defined as:

```
public class RemoteEvent extends java.util.EventObject {
    public RemoteEvent(Object source,long eventID,
                       long seqNum, MarshalledObject handback)
    public Object getSource () {…}
    public long getID() {…}
    public long getSequenceNumber() {…}
    public MarshalledObject getRegistrationObject() {…}
}
```

The abstract state contained in a RemoteEvent object includes: a reference to the object in which the event occurred, a long that identifies the kind of event relative to the object in which the event occurred, a long that indicates the sequence number of this instance of the event kind, and a MarshalledObject that is to be handed back when the notification occurs.

The combination of the event identifier and the object reference of the event generator obtained from the RemoteEvent object should uniquely identify the event type. If this type is not one in which the RemoteEventListener has registered interest (or in which someone else has registered interest on behalf of the RemoteEventListener object), an UnknownEventException may be generated as a return from the remote event listener's notify method.[1]

On receipt of an `UnknownEventException`, the caller of the `notify` method is allowed to cancel the lease for the combination of the `RemoteEventListener` instance and the kind of event that was contained in the `notify` call.

The sequence number obtained from the `RemoteEvent` object is an increasing value that can act as a hint to the number of occurrences of this event relative to some earlier sequence number. Any object that generates a `RemoteEvent` is required to ensure that for any two `RemoteEvent` objects with the same event identifier, the sequence number of those events differ if and only if the `RemoteEvent` objects are a response to different events. This guarantee is required to allow notification calls to be idempotent. A further guarantee is that if two `RemoteEvents`, *x* and *y*, come from the same source and have the same event identifier, then *x* occurred before *y* if and only if the sequence number of *x* is lower than the sequence number of *y*.

A stronger guarantee is possible for those generators of `RemoteEvents` that choose to support it. This guarantee states that not only do sequence numbers increase, but they are not skipped. In such a case, if `RemoteEvent` *x* and *y* have the same source and the same event identifier, and *x* has sequence number *m* and *y* has sequence number *n*, then if $m < n$ there were exactly $n–m–1$ events of the same event type between the event that triggered *x* and the event that triggered *y*. Such sequence numbers are said to be "fully ordered."

There are interactions between the generation of sequence numbers for a `RemoteEvent` object and the ability to see events that occur within the scope of a transaction. Those interactions are discussed in Section LEEV.2.4 "Sequence Numbers, Leasing and Transactions".

The common intent of a call to the `notify` method is to allow the recipient to find out that an occurrence of a kind of event has taken place. The call to the `notify` method is synchronous to allow the party making the call to know whether the call succeeded. However, it is not part of the semantics of the call that the notification return can be delayed while the recipient of the call reacts to the occurrence of the event. Simply put, the best strategy on the part of the recipient is to note the occurrence in some way and then return from the `notify` method as quickly as possible.

---

[1] There are cases in which the `UnknownEventException` may not be appropriate, even when the notification is for a combination of an event and a source that is not expected by the recipient. Objects that act as event mailboxes for other objects, for example, may be willing to accept any sort of notification from a particular source until explicitly told otherwise.

EV.2.3.3 The UnknownEventException

The UnknownEventException is thrown when the recipient of a RemoteEvent does not recognize the combination of the event identified and the source of the event as something in which it is interested. Throwing this exception has the effect of asking the sender to not send further notifications of this kind of event from this source in the future. This exception is defined as:

```
public class UnknownEventException extends Exception {
    public UnknownEventException() {
        super();
    }
    public UnknownEventException(String reason){
        super(reason);
    }
}
```

EV.2.3.4 An Example EventGenerator Interface

Registering interest in an event can take place in a number of ways, depending on how the event generator identifies its internal events. There is no single way of identifying the events that are reasonable for all objects and all kinds of events, and so there is no single way of registering interest in events. Because of this, there is no single interface for registration of interest.

However, the interaction between the event generator and the remote event listener does require that some initial information be passed from the registrant to the object that will make the call to its notify method.

The EventGenerator interface is an example of the kind of interface that could be used for registration of interest in events that can (logically) occur within an object. This is a remote interface that contains one method:

```
public interface EventGenerator extends Remote {
    public EventRegistration register(long evId,
                    MarshalledObject handback,
                    RemoteEventListener toInform,
                    long leaseLength)
        throws UnknownEventException, RemoteException;
}
```

The one method, register, allows registration of interest in the occurrence of an event inside the object. The method takes an evID that is used to identify the class of events, an object that is handed back as part of the notification, a reference to an

`RemoteEventListener` object, and a `long` integer indicating the leasing period for the interest registration.

The `evID` is a `long` that is obtained by a means that is not specified here. It may be returned by other interfaces or methods, or be defined by constants associated with the class or some interface implemented by the class. If an `evID` is supplied to this call that is not recognized by the `EventGenerator` object, an `UnknownEventException` is thrown. The use of a `long` to identify kinds of events is used only for illustrative purposes—objects may identify events by any number of mechanisms, including identifiers, using separate methods to allow registration in different events, or allowing various sorts of pattern matching to determine what events are of interest.

The second argument of the `register` method is a `MarshalledObject` that is to be handed back as part of the notification generated when an event of the appropriate type occurs. This object is known to the remote event listener and should contain any information that is needed by the listener to identify the event and to react to the occurrence of that event. This object will be passed back as part of the event object that is passed as an argument to the notify method. By passing a `MarshalledObject` into the `register` method, the re-creation of the object is postponed until the object is needed.

The ability to pass a `MarshalledObject` as part of the event registration should be common to all event registration methods. While there is no single method for identifying events in an object, the use of the pattern in which the remote event listener passes in an object that is passed back as part of the notification is central to the model of remote events presented here.

The third argument of the `EventGenerator` interface's `register` method is a `RemoteEventListener` implementation that is to receive event notifications. The listener may be the object that is registering interest, or it may be some other `RemoteEventListener`, such as a third-party event handler or notification "mailbox." The ability to specify some third-party object to handle the notification is also central to this model of event notification, and the capability of specifying the recipient of the notification is also common to all event registration interfaces.

The final argument to the `register` method is a `long` indicating the requested duration of the registration. This period is a request, and the period of interest actually granted by the event generator may be different. The actual duration of the registration lease is returned as part of the `Lease` object included in the `EventRegistration` object.

The `register` method returns an `EventRegistration` object. This object contains a `long` identifying the kind of event in which interest was registered (relative to the object granting the registration), a reference to the object granting the registration, and a `Lease` object.

EV.2.3.5 The `EventRegistration` Class

Objects of the class `EventRegistration` are meant to encapsulate the information the client needs to identify a notification as a response to a registration request and to maintain that registration request. It is not necessary for a method that allows event interest registration to return an `EventRegistration` object. However, the class does show the kind of information that needs to be returned in the event model.

The public parts of this class look like

```
public class EventRegistration implements java.io.Serializable
{
    public EventRegistration(long eventID,
                             Object eventSource,
                             Lease eventLease,
                             long seqNum) {…}
    public long getID() {…}
    public Object getSource() {…}
    public Lease getLease() {…}
    public long getSequenceNumber() {…}
}
```

The `getID` method returns the identifier of the event in which interest was registered. This, combined with the return value returned by `getSource`, will uniquely identify the kind of event. This information is needed to hand off to third-party repositories to allow them to recognize the event and route it correctly if they are to receive notifications of those events.

The result of the `EventRegistration.getID` method should be the same as the result of the `RemoteEvent.getID` method, and the result of the `EventRegistration.getSource` method should be the same as the `RemoteEvent.getSource` method.

The `getSource` method returns a reference to the event generator, which is used in combination with the result of the `getID` method to uniquely identify an event.

The `getLease` returns the `Lease` object for this registration. It is used in lease maintenance.

The `getSequenceNumber` method returns the value of the sequence number on the event kind that was current when the registration was granted, allowing comparison with the sequence number in any subsequent notifications.

EV.2.4 Sequence Numbers, Leasing and Transactions

There are cases in which event registrations are allowed within the scope of a transaction, in such a way that the notifications of these events can occur within the scope of the transaction. This means that other participants in the transaction may see some events whose visibility is hidden by the transaction from entities outside of the transaction. This has an effect on the generation of sequence numbers and the duration of an event registration lease.

An event registration that occurs within a transaction is considered to be scoped by that transaction. This means that any occurrence of the kind of event of interest that happens as part of the transaction will cause a notification to be sent to the recipients indicated by the registration that occurred in the transaction. Such events must have a separate event identification number (the `long` returned in the `RemoteEvent getID` method) to allow third-party store-and-forward entities to distinguish between an event that happens within a transaction and those that happen outside of the transaction. Notifications of these events will not be sent to entities that registered interest in this kind of event outside the scope of the transaction until and unless the transaction is committed.

Because of this isolation requirement of transactions, notifications sent from inside a transaction will have a different sequence number than the notifications of the same events would have outside of the transaction. Within a transaction, all `RemoteEvent` objects for a given kind of event are given a sequence number relative to the transaction, even if the event that triggered the `RemoteEvent` occurs outside of the scope of the transaction (but is visible within the transaction). One counter-intuitive effect of this is that an object could register for notification of some event *E* both outside a transaction and within a transaction, and receive two distinct `RemoteEvent` objects with different sequence numbers for the same event. One of the `RemoteEvent` objects would contain the event with a sequence number relative to the transaction, while the other would contain the event with a sequence number relative to the source object.

The other effect of transactions on event registrations is to limit the duration of a lease. A registration of interest in some kind of event that occurs within the scope of a transaction should be leased in the same way as other event interest registrations. However, the duration of the registration is the minimum of the length of the lease and the duration of the transaction. Simply put, when the transaction ends (either because of a commit or a rollback), the interest registration also ends. This is true even if the lease for the event registration has not expired and no call has been made to `cancel` the lease.

It is still reasonable to lease event interest registrations, even in the scope of a transaction, because the requested lease may be shorter than the transaction in

question. However, no such interest registration will survive the transaction in which it occurs.

EV.2.5 Serialized Forms

Class	serialVersionUID	Serialized Fields
RemoteEvent	1777278867291906446L	Object source long eventID long seqNum MarshalledObject handback
UnknownEventException	5563758083292687048L	*none*
EventRegistration	4055207527458053347L	Object source long eventID Lease lease long seqNum

EV.3 Third-Party Objects

ONE of the basic reasons for the event design is to allow the production of third-party objects, or "agents," that can be used to enhance a system built using distributed events and notifications. Now we will look at three examples of such agents, which allow various forms of enhanced functionality without changing the basic interfaces. Each of these agents may be thought of as *distributed event adapters.*

The first example we will look at is a *store-and-forward agent.* The purpose of this object is to act on behalf of the event generator, allowing the event generator to send the notification to one entity (the store-and-forward agent) that will forward the notification to all of the event listeners, perhaps with a particular policy that allows a failed delivery attempt to be retried at some later date.

The second example, which we will call a *notification filter,* is an object that may be local to either the event generator or the event listener. This agent gets the notification and spawns a thread that will respond, using a method supplied by the object that originally registered interest in events of that kind.

The final object is a *notification mailbox.* This mailbox will store notifications for another object (a remote event listener) until that object requests that the notifications be delivered. This design allows the listener object that registered interest in the event type to select the times at which a notification can be delivered without losing any notifications that would have otherwise have been delivered. Please note that complete specifications for such services do exist and may differ in some degree from this example.

EV.3.1 Store-and-Forward Agents

A store-and-forward agent enables the object generating a notification to hand off the actual notification of those who have registered interest to a separate object.

This agent can implement various policies for reliability. For example, the agent could try to deliver the notification once (or a small number of times) and, if that call fails, not try again. Or the agent could try and, on notification failure, try again at a preset or computed interval of time for some known period of time. Either way, the object in which the event occurred could avoid worrying about the

delivery of notifications, needing to notify only the store-and-forward agent (which might be on the same machine and hence more reliably available).

From the point of view of the remote event listener, there is no difference between the notification delivered by a store-and-forward agent and one delivered directly from the object in which the event that generated the original notification occurred. This transparency allows the decision to use a store-and-forward agent to be made by the object generating the notification, independent of the object receiving the notification. There is no need for distributed agreement; all that is required is that the object using the agent know about the agent.

A store-and-forward agent is used by an object that generates notifications. When an object registers interest in receiving notifications of a particular event type, the object receiving that registration will pass the registration along to the store-and-forward agent. This agent will keep track of which objects need to be notified of events that occur in the original object.

When an event of interest occurs in the original object, it need send only a single notification to the store-and-forward agent. This notification can return immediately, with processing further happening inside the store-and-forward agent. The object in which the event of interest occurred will now be freed from informing those that registered interest in the event.

Notification is taken over by the store-and-forward agent. This agent will now consult the list of entities that have registered interest in the occurrence of an event and send a notification to those entities. Note that these might not be the same as the objects that registered interest in the event; the object that should receive the event notification is specified during the event interest registration.

The store-and-forward agent might be able to make use of network-level multicast (assuming that the `RemoteEvent` object to be returned is identical for multiple recipients of the `notify` call), or might send a separate notification to each of the entities that have registered interest. Different store-and-forward agents could implement different levels of service, from a simple agent that sends a notification and doesn't care whether the notification is actually delivered (for example, one that simply caught `RemoteExceptions` and discards them) to agents that will repeatedly try to send the notification, perhaps using different fallback strategies, until the notification is known to be successful or some number of tries have been attempted.

The store-and-forward agent does not need to know anything about the kinds of events that are triggering the notifications that it stores and forwards. All that is needed is that the agent implement the `RemoteEventListener` interface and some interface that allows the object producing the initial notification to register with the agent. This combination of interfaces allows such a service to be offered to any number of different objects without having to know anything about the possible changes in abstract state that might be of interest in those objects.

Note that the interface used by the object generating the original notifications to register with the store-and-forward agent does not need to be standard. Different qualities of service concerning the delivery of notifications may require different registration protocols. Whether or not the relationship between the notification originator and the store-and-forward agent is leased or not is also up to the implementation of the agent. If the relationship is leased, lease renewal requests would need to be forwarded to the agent.

In fact, an expected pattern of implementation would be to place a store-and-forward agent on every machine on which objects were running that could produce events. This agent, which could be running in a separate JVM (on hardware that supported multiple processes) could off-load the notification-generating objects from the need to send those notifications to all objects that had registered interest. It would also allow for consistent handling of delivery guarantees across all objects on a particular machine. Since the store-and-forward agent is on the same machine as the objects using the agent, the possibilities of partial failure brought about by network problems (which wouldn't affect communication between objects on the same machine) and server machine failure (which would induce total, rather than partial, failure in this case) are limited. This allows the reliability of notifications to be off-loaded to these agents instead of being a problem that needs to be solved by all of the objects using the notification interfaces.

A store-and-forward agent does require an interface that allows the agent to know what notifications it is supposed to send, the destinations of those notifications, and on whose behalf those notifications are being sent. Since it is the store-and-forward agent that is directing notification calls to the individual recipients, the agent will also need to hold the `Object` (if any) that was passed in during interest registration to be returned as part of the `RemoteEvent` object.

In addition, the store-and-forward agent could be the issuer of `Lease` objects to the object registering interest in some event. This could offload any lease renewal calls from the original recipient of the registration call, which would need to know only when there were no more interest registrations of a particular event kind remaining in the store-and-forward agent.

EV.3.2 Notification Filters

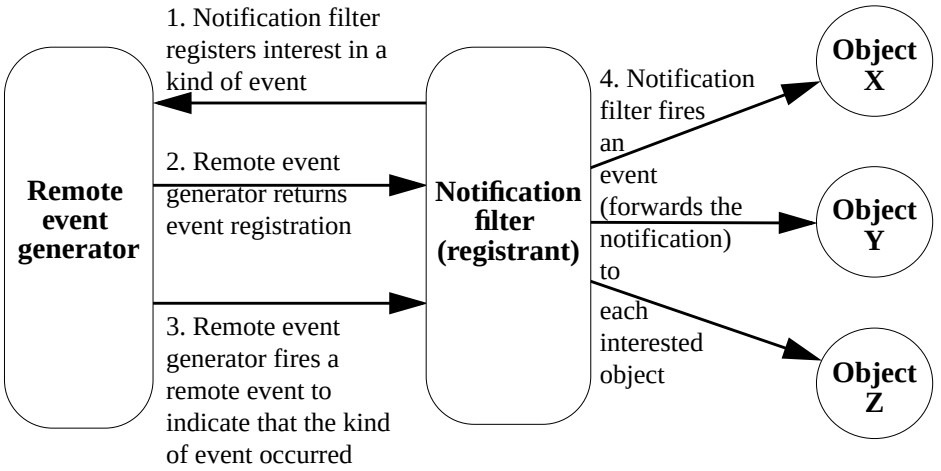
Similar to a store-and-forward agent is a notification filter, which can be used by either the generator of a notification or the recipient to intercept notification calls, do processing on those calls, and act in accord with that processing (perhaps forwarding the notification, or even generating new notifications).

Again, such filters are made possible because of the uniform signature of the method used to send all notifications and because of the ability of an object to

indicate the recipient of a notification when registering for a notification. This uniformity and indirection allow the composition of third-party entities. A filter could receive events from a store-and-forward agent without the client of the original registration knowing about the store-and-forward agent or the server in which the notifications are generated knowing about the filter. This composition can be extended further; store-and-forward agents could use other store-and-forward agents, and filters can themselves receive notifications from other filters.

EV.3.2.1 Notification Multiplexing

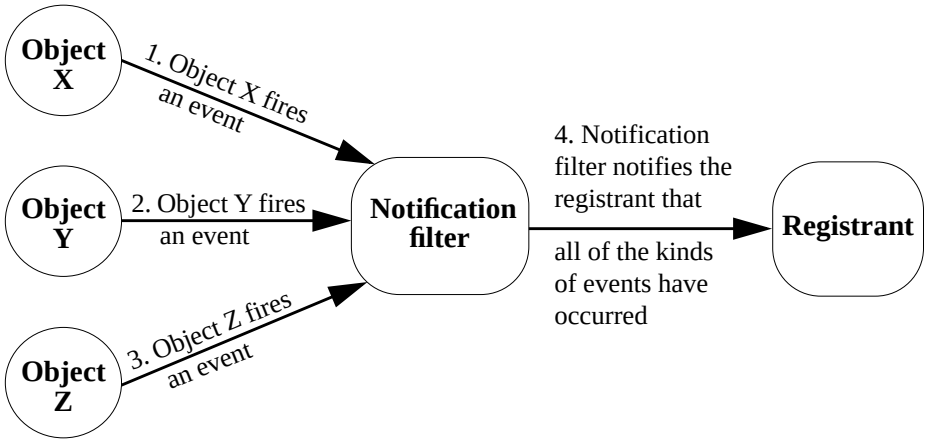
One example of such a filter is one that can be used to concentrate notifications in a way to help minimize network traffic. If a number of different objects on a single machine are all interested in some particular kind of event, it could make sense to create a notification filter that would register interest in the event. When a notification was received by the filter, it would forward the notification to each of the (machine local) objects that had expressed interest.



EV.3.2.2 Notification Demultiplexing

Another example of such a filter is an object that generates an event in response to a series of events that it has received. There might be an object that is interested only in some particular sequence of events in some other object or group of objects. This object could register interest in all of the different kinds of events, asking that the notifications be sent to a filter. The purpose of the filter is to receive

the notifications and, when the notifications fit the desired pattern (as determined by some class passed in from the object that has asked the notifications be sent to the filter), generate some new notification that is delivered to the client object.



EV.3.3 Notification Mailboxes

The purpose of a notification mailbox is to store the notifications sent to an object until such time as the object for which the notifications were intended desires delivery.

Such delivery can be in a single batch, with the mailbox storing any notifications received after the last request for delivery until the next request is received. Alternatively, a notification mailbox can be viewed as a faucet, with notifications turned on (delivering any that have arrived since the notifications were last turned off) and then delivering any subsequent notifications to an object immediately, until told by that object to hold the notifications.

The ability to have notification mailboxes is important in a system that allows objects to be deactivated (for example, to be saved to stable storage in such a way that they are no longer taking up any computing resource) and re-activated. The usual mechanism for activating an object is a method call. Such activation can be expensive in both time and computing resources; it is often too expensive to be justified for the receipt of what would otherwise be an asynchronous event notification. An event mailbox can be used to ensure that an object will not be activated merely to handle an event notification.

Use of a mailbox is simple; the object registering interest in receiving an event notification simply gives the mailbox as the place to send the notifications. The mailbox can be made responsible for renewing leases while an object is inactive,

and for storing all (or the most recent, or the most recent and the count of other) notifications for each type of event of interest to the object. When the object indicates that it wishes to receive any notifications from the mailbox, those notifications can be delivered. Delivery can continue until the object requests storage to occur again, or storage can resume automatically.

Such a mailbox is a type of filter. In this case, however, the mailbox filters over time rather than over events. A pure mailbox need not be concerned with the kinds of notifications that it stores. It simply holds the `RemoteEvent` objects until they are wanted.

It is because of mailboxes and other client-side filters that the information returned from an event registration needs to include a way of identifying the event and the source of the event. Such client-side agents need a way of distinguishing between the events they are expected to receive and those that should generate an exception to the sender. This distinction cannot be made without some simple way of identifying the event and the object of origin.

EV.3.4 Compositionality

All of the above third-party entities work because of two simple features of the `RemoteEventListener` interface:

- There is a single method, `notify`, that passes a single type of object, `RemoteEvent` (or a subtype of that object) for all notifications
- There is a level of indirection in delivery allowed by the separate specification of a recipient in the registration method that allows the client of that call to specify a third-party object to contact for notifications

The first of these features allows the composition of notification handlers to be chained, beginning with the object that generates the notification. Since the ultimate recipient of the event is known to be expecting the event through a call to the single `notify` method, other entities can be composed and interposed in the call chain as long as they produce this call with the right `RemoteEvent` object (which will include a field indicating the object at which the notification originated). Because there is a single method call for all notifications, third-party handlers can be produced to accept notifications of events without having to know the kind of event that has occurred or any other detail of the event.

Compositionality in the other direction (driven by the recipient of the notification) is enabled by allowing the object registering interest to indicate the first in an arbitrary chain of third parties to receive the notification. Thus the recipient can

build a chain of filters, mailboxes, and forwarding agents to allow any sort of delivery policy that object desires, and then register interest with an indication that all notifications should be delivered to the beginning of that chain. From the point of view of the object in which the notification originates, the series of objects the notification then goes through is unknown and irrelevant.

EV.4 Integration with JavaBeans Components

As we noted previously, distributed notification differs from local notification (such as the notification used in user interface programming) in a number of ways. In particular, a distributed notification may be delayed, dropped, or otherwise fail between the object in which the event occurred and the object that is the ultimate recipient of the notification of that event. Additionally, a distributed event notification may require handling by a number of third-party objects between the object that is interested in the notification and the object that generates the notification. These third-party objects need to be able to handle arbitrary events, and so from the point of view of the type system, all of the events must be delivered in the same fashion.

Although this model differs from the event model used for user interface tools such as the AWT or Java Foundation Classes (JFC), such a difference in model is to be expected. The event model for such user interface toolkits was never meant to allow the components that communicate using these local event notifications to be distributed across virtual or physical machines; indeed, such systems assume that the event delivery will be fast, reliable, and not open to the kinds of partial failures or delays that are common in the distributed case.

In between the requirements of a local event model and the distributed event model presented here is the event model used by software components to communicate changes in state. The delegation event model, which is the event model for JavaBeans components, written in the Java programming language, is built as an extension of the event model used for AWT and JFC. This is completely appropriate, as most JavaBeans components will be located in a single address space and can assume that the communication of events between components will meet the reliability and promptness requirements of that model.

However, it is also possible that JavaBeans components will be distributed across virtual, and even physical, machines. The assumption that the event propagation will be either fast or reliable can lead to subtle program errors that will not be found until the components are deployed (perhaps on a slow or unreliable network). In such case, an event and notification model such as that found in this specification is more appropriate.

One approach would be to add a second event model to the JavaBeans component specification that dealt only with distributed events. While this would have the advantage of exporting the difference between local and remote components to the component builder, it would also complicate the JavaBeans component model unnecessarily.

We will show how the current distributed event model can be fit into the existing Java platform's event model. While the mapping is not perfect (nor can it be, since there are essential differences between the two models), it will allow the current tools used to assemble JavaBeans components to be used when those components are distributed.

EV.4.1 Differences with the JavaBeans Component Event Model

The JavaBeans component event model is derived from the event model used in the AWT in the Java platform. The model is characterized by:

- Propagation of event notifications from sources to listeners by Java technology method invocations on the target listener objects
- Identification of the kind of event notification by using a different method in the listener being called for each kind of event
- Encapsulation of any state associated with an event notification in an object that inherits from `java.util.EventObject` and that is passed as the sole argument of the notification method
- Identification of event sources by the convention of those sources defining registration methods, one for each kind of event in which interest can be registered, that follow a particular design pattern

The distributed event and notification model that we have defined is similar in a number of ways:

- Distributed event propagation is accomplished by the use of `Remote` methods.
- State passed as part of the notification is encapsulated in an object that is derived from `java.util.EventObject` and is passed as the sole argument of the notification method.
- The `RemoteEventListener` interface extends the more basic interface `java.util.EventListener`.

However, there are also differences between the JavaBeans component event model and the distributed event model proposed here:

- Identification of the kind of event is accomplished by passing an identifier from the source of the notification to the listener; the combination of the object in which the event occurred and the identifier uniquely identifies the kind of event.
- Notifications are accomplished through a single method, `notify`, defined in the `RemoteEventListener` interface rather than by a different method for each kind of event.
- Registration of interest in a kind of event is for a (perhaps renewable) period of time, rather than being for a period of time bound by the active cancellation of interest.
- Objects registering interest in an event can, as part of that registration, include an object that will be passed back to the recipient of the notification when an event of the appropriate type occurs.

Most of these differences in the two models can be directly traced to the distributed nature of the events and notifications defined in this specification.

For example, as you have seen, reliability and recovery of the distributed notification model is based on the ability to create third-party objects that can provide those guarantees. However, for those third-party objects to be able to work in general cases, the signature for a notification must be the same for all of the event notifications that are to be handled by that third party. If we were to follow the JavaBeans component model of having a different method for each kind of event notification, third party objects would need to support every possible notification method, including those that had not yet been defined when the third-party object was implemented. This is clearly impossible.

Note that this is not a weakness in the JavaBeans component event model, merely a difference required by the different environments in which the event models are assumed to be used. The JavaBeans component event model, like the AWT model on which it is based, assumes that the event notification is being passed between objects in the same address space. Such notifications do not need various delivery and reliability guarantees—delivery can be considered to be (virtually) instantaneous and can be assumed to be fully reliable.

Being able to send event notifications through a single `Remote` method also requires that the events be identified in some way other than the signature of the notification delivery method. This leads to the inclusion of an event identifier in the event object. Since the generation of these event identifiers cannot be guaranteed to be globally unique across all of the objects in a distributed system, they

must be made relative to the object in which they are generated, thus requiring the combination of the object of origin and the event identifier to completely identify the kind of event.

The sequence number being included in the event object is also an outgrowth of the distributed nature of the interfaces. Since no distributed mechanism can guarantee reliability, there is always the possibility that a particular notification will not be delivered, or could be delivered more than once by some notification agent. This is not a problem in the single-address-space environment of AWT and JavaBeans components, but requires the inclusion of a sequence number in the distributed case.

EV.4.2 Converting Distributed Events to JavaBeans Component Events

Translating between the event models is fairly straightforward. All that is required is:

- Allow an event listener to map from a distributed event listener to the appropriate call to a notification method
- Allow creation of a `RemoteEvent` from the event object passed in the JavaBeans component event notification method
- Allow creation of a JavaBeans component event object from a `RemoteEvent` object without loss of information

Each of these is fairly straightforward and can be accomplished in a number of ways.

More complex matings of the two systems could be undertaken, including third-party objects that keep track of the interest registrations made by remote objects and implement the corresponding JavaBeans component event notification methods by making the remote calls to the `RemoteEventListener notify` method with properly constructed `RemoteEvent` objects. Such objects would need to keep track of the event sequence numbers and would need to deal with the additional failure modes that are inherent in distributed calls. However, their implementation would be fairly straightforward and would fit into the JavaBeans component model of event adapters.

TX

Transaction

TX.1 Introduction

TRANSACTIONS are a fundamental tool for many kinds of computing. A transaction allows a set of operations to be grouped in such a way that they either all succeed or all fail; further, the operations in the set appear from outside the transaction to occur simultaneously. Transactional behaviors are especially important in distributed computing, where they provide a means for enforcing consistency over a set of operations on one or more remote participants. If all the participants are members of a transaction, one response to a remote failure is to abort the transaction, thereby ensuring that no partial results are written.

Traditional transaction systems often center around transaction processing monitors that ensure that the correct implementation of transactional semantics is provided by all of the participants in a transaction. Our approach to transactional semantics is somewhat different. Within our system we leave it to the individual objects that take part in a transaction to implement the transactional semantics in the way that is best for that kind of object. What the system primarily provides is the coordination mechanism that those objects can use to communicate the information necessary for the set of objects to agree on the transaction. The goal of this system is to provide the *minimal* set of protocols and interfaces that *allow* objects to implement transaction semantics rather than the *maximal* set of interfaces, protocols, and policies that *ensure* the correctness of any possible transaction semantics. So the completion protocol is separate from the semantics of particular transactions.

This document presents this completion protocol, which consists of a two-phase commit protocol for distributed transactions. The two-phase commit protocol defines the communication patterns that allow distributed objects and resources to wrap a set of operations in such a way that they appear to be a single operation. The protocol requires a manager that will enable consistent resolution

of the operations by a guarantee that all participants will eventually know whether they should commit the operations (roll forward) or abort them (roll backward). A participant can be any object that supports the participant contract by implementing the appropriate interface. Participants are not limited to databases or other persistent storage services.

Clients and servers will also need to depend on specific transaction semantics. The default transaction semantics for participants is also defined in this document.

The two-phase commit protocol presented here, while common in many traditional transaction systems, has the potential to be used in more than just traditional transaction processing applications. Since the semantics of the individual operations and the mechanisms that are used to ensure various properties of the meta-operation joined by the protocol are left up to the individual objects, variations of the usual properties required by transaction processing systems are possible using this protocol, as long as those variances can be resolved by this protocol. A group of objects could use the protocol, for example, as part of a process allowing synchronization of data that have been allowed to drift for efficiency reasons. While this use is not generally considered to be a classical use of transactions, the protocol defined here could be used for this purpose. Some variations will not be possible under these protocols, requiring subinterfaces and subclasses of the ones provided or entirely new interfaces and classes.

Because of the possibility of application to situations that are beyond the usual use of transactions, calling the two-phase commit protocol a transaction mechanism is somewhat misleading. However, since the most common use of such a protocol is in a transactional setting, and because we do define a particular set of default transaction semantics, we will follow the usual naming conventions used in such systems rather than attempting to invent a new, parallel vocabulary.

The classes and interfaces defined by this specification are in the packages `net.jini.core.transaction` and `net.jini.core.transaction.server`. In this document you will usually see these types used without a package prefix; as each type is defined, the package it is in is specified.

TX.1.1 Model and Terms

A transaction is created and overseen by a *manager*. Each manager implements the interface `TransactionManager`. Each *transaction* is represented by a `long` identifier that is unique with respect to the transaction's manager.

Semantics are represented by *semantic* transaction objects, such as the ones that represent the default semantics for services. Even though the manager needs to know only how to complete transactions, clients and participants need to share a common view of the semantics of the transaction. Therefore clients typically

create, pass, and operate on semantic objects that contain the transaction identifier instead of using the transaction's identifier directly, and transactable services typically accept parameters of a particular semantic type, such as the `Transaction` interface used for the default semantics.

As shown in Figure AR.1.1, a *client* asks the manager to create a transaction, typically by using a semantic factory class such as `TransactionFactory` to create a semantic object. The semantic object created is then passed as a parameter when performing operations on a service. If the service is to accept this transaction and govern its operations thereby, it must *join* the transaction as a *participant*. Participants in a transaction must implement the `TransactionParticipant` interface. Particular operations associated with a given transaction are said to be *performed under* that transaction. The client that created the transaction might or might not be a participant in the transaction.

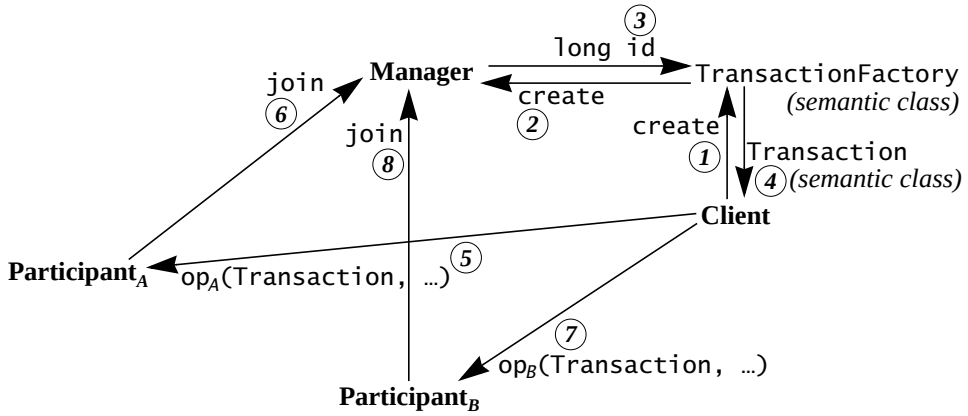


FIGURE TX.1.1: ***Transaction Creation and Use***

A transaction *completes* when any entity either *commits* or *aborts* the transaction. If a transaction commits successfully, then all operations performed under that transaction will complete. Aborting a transaction means that all operations performed under that transaction will appear never to have happened.

Committing a transaction requires each participant to *vote*, where a vote is either *prepared* (ready to commit), *not changed* (read-only), or *aborted* (the transaction should be aborted). If all participants vote "prepared" or "not changed," the transaction manager will tell each "prepared" participant to *roll forward*, thus committing the changes. Participants that voted "not changed" need do nothing more. If the transaction is ever aborted, the participants are told to *roll back* any changes made under the transaction.

TX.1.2 Distributed Transactions and ACID Properties

The two-phase commit protocol is designed to enable objects to provide ACID properties. The default transaction semantics define one way to preserve these properties. The ACID properties are:

- *Atomicity:* All the operations grouped under a transaction occur or none of them do. The protocol allows participants to discover which of these alternatives is expected by the other participants in the protocol. However, it is up to the individual object to determine whether it wishes to operate in concert with the other participants.
- *Consistency:* The completion of a transaction must leave the system in a consistent state. Consistency includes issues known only to humans, such as that an employee should always have a manager. The enforcement of consistency is outside of the realm of the transaction itself—a transaction is a tool to allow consistency guarantees and not itself a guarantor of consistency.
- *Isolation:* Ongoing transactions should not affect each other. Participants in a transaction should see only intermediate states resulting from the operations of their own transaction, not the intermediate states of other transactions. The protocol allows participating objects to know what operations are being done within the scope of a transaction. However, it is up to the individual object to determine if such operations are to be reflected only within the scope of the transaction or can be seen by others who are not participating in the transaction.
- *Durability:* The results of a transaction should be as persistent as the entity on which the transaction commits. However, such guarantees are up to the implementation of the object.

The dependency on the participant's implementation for the ACID properties is the greatest difference between this two-phase commit protocol and more traditional transaction processing systems. Such systems attempt to ensure that the ACID properties are met and go to considerable trouble to ensure that no participant can violate any of the properties.

This approach differs for both philosophical and practical reasons. The philosophical reason is centered on a basic tenet of object-oriented programming, which is that the implementation of an object should be hidden from any part of the system outside the object. Ensuring the ACID properties generally requires that an object's implementation correspond to certain patterns. We believe that if these properties are needed, the object (or, more precisely, the programmer imple-

menting the object) will know best how to guarantee the properties. For this reason, the manager is solely concerned with completing transactions properly. Clients and participants must agree on semantics separately.

The practical reason for leaving the ACID properties up to the object is that there are situations in which only some of the ACID properties make sense, but that can still make use of the two-phase commit protocol. A group of transient objects might wish to group a set of operations in such a way that they appear atomic; in such a situation it makes little sense to require that the operations be durable. An object might want to enable the monitoring of the state of some long-running transactions; such monitoring would violate the isolation requirement of the ACID properties. Binding the two-phase commit protocol to all of these properties limits the use of such a protocol.

We also know that particular semantics are needed for particular services. The default transaction semantics provide useful general-purpose semantics built on the two-phase commit completion protocol.

Distributed transactions differ from single-system transactions in the same way that distributed computing differs from single-system computing. The clearest difference is that a single system can have a single view of the state of several services. It is possible in a single system to make it appear to any observer that all operations performed under a transaction have occurred or none have, thereby achieving isolation. In other words, no observer will ever see only part of the changes made under the transaction. In a distributed system it is possible for a client using two servers to see the committed state of a transaction in one server and the pre-committed state of the same transaction in another server. This can be prevented only by coordination with the transaction manager or the client that committed the transaction. Coordination between clients is outside the scope of this specification.

TX.1.3 Requirements

The transaction system has the following requirements:

- Define types and contracts that allow the two-phase commit protocol to govern operations on multiple servers of differing types or implementations.
- Allow participation in the two-phase commit protocol by any object in the Java programming language, where “participation” means to perform operations on that object under a given transaction.
- Each participant may provide ACID properties with respect to that participant to observers operating under a given transaction.

- Use standard Java programming language techniques and tools to accomplish these goals. Specifically, transactions will rely upon Java Remote Method Invocation (RMI) to communicate between participants.
- Define specific default transaction semantics for use by services.

TX.2 The Two-Phase Commit Protocol

THE two-phase commit protocol is defined using three primary types:

- `TransactionManager`: A transaction manager creates new transactions and coordinates the activities of the participants.
- `NestableTransactionManager`: Some transaction managers are capable of supporting nested transactions.
- `TransactionParticipant`: When an operation is performed under a transaction, the participant must join the transaction, providing the manager with a reference to a `TransactionParticipant` object that will be asked to vote, roll forward, or roll back.

The following types are imported from other packages and are referenced in unqualified form in the rest of this specification:

```
java.rmi.Remote
java.rmi.RemoteException
java.rmi.NoSuchObjectException
java.io.Serializable
net.jini.core.lease.LeaseDeniedException
net.jini.core.lease.Lease
```

All the methods defined to throw `RemoteException` will do so in the circumstances described by the RMI specification.

Each type is defined where it is first described. Each method is described where it occurs in the lifecycle of the two-phase commit protocol. All methods, fields, and exceptions that can occur during the lifecycle of the protocol will be specified. The section in which each method or field is specified is shown in a comment, using the § abbreviation for the word “section.”

TX.2.1 Starting a Transaction

The `TransactionManager` interface is implemented by servers that manage the two-phase commit protocol:

```
package net.jini.core.transaction.server;

public interface TransactionManager
    extends Remote, TransactionConstants // §TX.2.4
{
    public static class Created implements Serializable {
        public final long id;
        public final Lease lease;
        public Created(long id, Lease lease) {…}
    }
    Created create(long leaseFor) // §TX.2.1
        throws LeaseDeniedException, RemoteException;
    void join(long id, TransactionParticipant part,
              long crashCount) // §TX.2.3
        throws UnknownTransactionException,
               CannotJoinException, CrashCountException,
               RemoteException;
    int getState(long id) // §TX.2.7
        throws UnknownTransactionException, RemoteException;
    void commit(long id) // §TX.2.5
        throws UnknownTransactionException,
               CannotCommitException,
               RemoteException;
    void commit(long id, long waitFor) // §TX.2.5
        throws UnknownTransactionException,
               CannotCommitException,
               TimeoutExpiredException, RemoteException;
    void abort(long id) // §TX.2.5
        throws UnknownTransactionException,
               CannotAbortException,
               RemoteException;
    void abort(long id, long waitFor) // §TX.2.5
        throws UnknownTransactionException,
               CannotAbortException,
               TimeoutExpiredException, RemoteException;
}
```

A client obtains a reference to a `TransactionManager` object via a lookup service or some other means. The details of obtaining such a reference are outside the scope of this specification.

A client creates a new transaction by invoking the manager's `create` method, providing a desired `leaseFor` time in milliseconds. This invocation is typically indirect via creating a semantic object. The time is the client's expectation of how long the transaction will last before it completes. The manager may grant a shorter lease or may deny the request by throwing `LeaseDeniedException`. If the granted lease expires or is cancelled before the transaction manager receives a `commit` or `abort` of the transaction, the manager will abort the transaction.

The purpose of the `Created` nested class is to allow the `create` method to return two values: the transaction identifier and the granted lease. The constructor simply sets the two fields from its parameters.

TX.2.2 Starting a Nested Transaction

The `TransactionManager.create` method returns a new *top-level* transaction. Managers that implement just the `TransactionManager` interface support only top-level transactions. *Nested* transactions, also known as *subtransactions*, can be created using managers that implement the `NestableTransactionManager` interface:

```
package net.jini.core.transaction.server;

public interface NestableTransactionManager
    extends TransactionManager
{
    TransactionManager.Created
        create(NestableTransactionManager parentMgr,
               long parentID, long leaseFor) // §TX.2.2
        throws UnknownTransactionException,
               CannotJoinException, LeaseDeniedException,
               RemoteException;
    void promote(long id, TransactionParticipant[] parts,
                 long[] crashCounts,
                 TransactionParticipant drop)
        throws UnknownTransactionException,
               CannotJoinException, CrashCountException,
               RemoteException; // §TX.2.7
}
```

The `create` method takes a *parent* transaction—represented by the manager for the parent transaction and the identifier for that transaction—and a desired lease time in milliseconds, and returns a new *nested* transaction that is *enclosed by* the specified parent along with the granted lease.

When you use a nested transaction you allow changes to a set of objects to abort without forcing an abort of the parent transaction, and you allow the commit of those changes to still be conditional on the commit of the parent transaction.

When a nested transaction is created, its manager joins the parent transaction. When the two managers are different, this is done explicitly via `join` (see Section TX.2.3 "Joining a Transaction"). When the two managers are the same, this may be done in a manager-specific fashion.

The `create` method throws `UnknownTransactionException` if the parent transaction is unknown to the parent transaction manager, either because the transaction ID is incorrect or because the transaction is no longer active and its state has been discarded by the manager.

```
package net.jini.core.transaction;

public class UnknownTransactionException
    extends TransactionException
{
    public UnknownTransactionException() {…}
    public UnknownTransactionException(String desc) {…}
}

public class TransactionException extends Exception {
    public TransactionException() {…}
    public TransactionException(String desc) {…}
}
```

The `create` method throws `CannotJoinException` if the parent transaction is known to the manager but is no longer active.

```
package net.jini.core.transaction;

public class CannotJoinException extends TransactionException
{
    public CannotJoinException() {…}
    public CannotJoinException(String desc) {…}
}
```

TX.2.3 Joining a Transaction

The first time a client tells a participant to perform an operation under a given transaction, the participant must invoke the transaction manager's `join` method with an object that implements the `TransactionParticipant` interface. This object will be used by the manager to communicate with the participant about the transaction.

```
package net.jini.core.transaction.server;

public interface TransactionParticipant
    extends Remote, TransactionConstants // §TX.2.4
{
    int prepare(TransactionManager mgr, long id) // §TX.2.6
        throws UnknownTransactionException, RemoteException;
    void commit(TransactionManager mgr, long id) // §TX.2.6
        throws UnknownTransactionException, RemoteException;
    void abort(TransactionManager mgr, long id) // §TX.2.6
        throws UnknownTransactionException, RemoteException;
    int prepareAndCommit(TransactionManager mgr, long id)
                                                  // §TX.2.7
        throws UnknownTransactionException, RemoteException;
}
```

If the participant's invocation of the `join` method throws `RemoteException`, the participant should not perform the operation requested by the client and should rethrow the exception or otherwise signal failure to the client.

The `join` method's third parameter is a *crash count* that uniquely defines the version of the participant's storage that holds the state of the transaction. Each time the participant loses the state of that storage (because of a system crash if the storage is volatile, for example) it must change this count. For example, the participant could store the crash count in stable storage.

When a manager receives a `join` request, it checks to see if the participant has already joined the transaction. If it has, and the crash count is the same as the one specified in the original `join`, the `join` is accepted but is otherwise ignored. If the crash count is different, the manager throws `CrashCountException` and forces the transaction to abort.

```
package net.jini.core.transaction.server;

public class CrashCountException extends TransactionException
{
```

```
    public CrashCountException() {…}
    public CrashCountException(String desc) {…}
}
```

The participant should reflect this exception back to the client. This check makes `join` idempotent when it should be, but forces an abort for a second `join` of a transaction by a participant that has no knowledge of the first `join` and hence has lost whatever changes were made after the first `join`.

An invocation of `join` can throw `UnknownTransactionException`, which means the transaction is unknown to the manager, either because the transaction ID was incorrect, or because the transaction is no longer active and its state has been discarded by the manager. The `join` method throws `CannotJoinException` if the transaction is known to the manager but is no longer active. In either case the `join` has failed, and the method that was attempted under the transaction should reflect the exception back to the client. This is also the proper response if `join` throws a `NoSuchObjectException`.

TX.2.4 Transaction States

The `TransactionConstants` interface defines constants used in the communication between managers and participants.

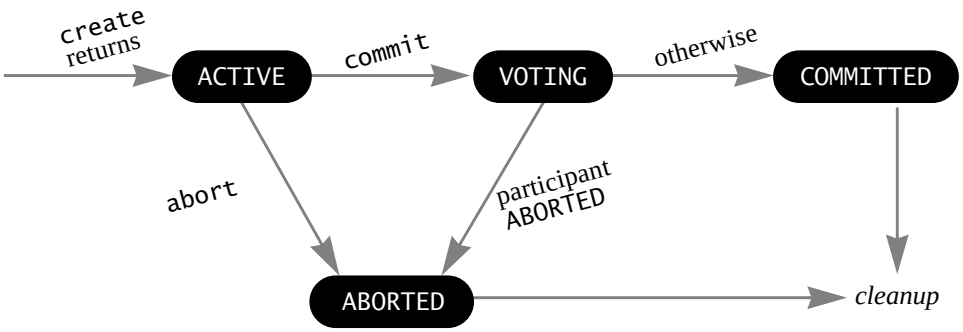
```
package net.jini.core.transaction.server;

public interface TransactionConstants {
    int ACTIVE = 1;
    int VOTING = 2;
    int PREPARED = 3;
    int NOTCHANGED = 4;
    int COMMITTED = 5;
    int ABORTED = 6;
}
```

These correspond to the states and votes that participants and managers go through during the lifecycle of a given transaction.

TX.2.5 Completing a Transaction: The Client's View

In the client's view, a transaction goes through the following states:



For the client, the transaction starts out `ACTIVE` as soon as `create` returns. The client drives the transaction to completion by invoking `commit` or `abort` on the transaction manager, or by cancelling the lease or letting the lease expire (both of which are equivalent to an `abort`).

The one-parameter `commit` method returns as soon as the transaction successfully reaches the `COMMITTED` state, or if the transaction is known to have previously reached that state due to an earlier `commit`. If the transaction reaches the `ABORTED` state, or is known to have previously reached that state due to an earlier `commit` or `abort`, then `commit` throws `CannotCommitException`.

```
package net.jini.core.transaction;

public class CannotCommitException
    extends TransactionException
{
    public CannotCommitException() {…}
    public CannotCommitException(String desc) {…}
}
```

The one-parameter `abort` method returns as soon as the transaction successfully reaches the `ABORTED` state, or if the transaction is known to have previously reached that state due to an earlier `commit` or `abort`. If the transaction is known to have previously reached the `COMMITTED` state due to an earlier `commit`, then `abort` throws `CannotAbortException`.

```
package net.jini.core.transaction;

public class CannotAbortException extends TransactionException
{
    public CannotAbortException() {…}
    public CannotAbortException(String desc) {…}
}
```

Both `commit` and `abort` can throw `UnknownTransactionException`, which means the transaction is unknown to the manager. This may be because the transaction ID was incorrect, or because the transaction has proceeded to *cleanup* due to an earlier commit or abort, and has been forgotten.

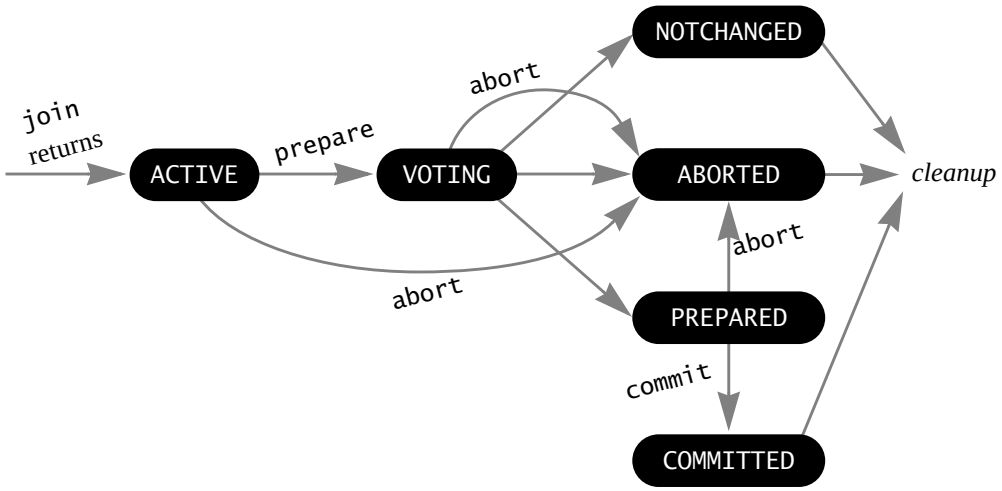
Overloads of the `commit` and `abort` methods take an additional `waitFor` timeout parameter specified in milliseconds that tells the manager to wait until it has successfully notified all participants about the outcome of the transaction before the method returns. If the timeout expires before all participants have been notified, a `TimeoutExpiredException` will be thrown. If the timeout expires before the transaction reaches the `COMMITTED` or `ABORTED` state, the manager must wait until one of those states is reached before throwing the exception. The `committed` field in the exception is set to `true` if the transaction committed or to `false` if it aborted.

```
package net.jini.core.transaction;

public class TimeoutExpiredException extends
             TransactionException
{
    public boolean committed;
    public TimeoutExpiredException(boolean committed) {…}
    public TimeoutExpiredException(String desc,
                                   boolean committed) {…}
}
```

TX.2.6 Completing a Transaction: A Participant's View

In a participant's view, a transaction goes through the following states:



For the participant, the transaction starts out `ACTIVE` as soon as `join` returns. Any operations attempted under a transaction are valid only if the participant has the transaction in the `ACTIVE` state. In any other state, a request to perform an operation under the transaction should fail, signaling the invoker appropriately.

When the manager asks the participant to `prepare`, the participant is `VOTING` until it decides what to return. There are three possible return values for `prepare`:

- The participant had no changes to its state made under the transaction—that is, for the participant the transaction was read-only. It should release any internal state associated with the transaction. It must signal this with a return of `NOTCHANGED`, effectively entering the `NOTCHANGED` state. As noted below, a well-behaved participant should stay in the `NOTCHANGED` state for some time to allow idempotency for `prepare`.
- The participant had its state changed by operations performed under the transaction. It must attempt to prepare to roll those changes forward in the event of a future incoming `commit` invocation. When the participant has successfully prepared itself to roll forward (see Section TX.2.8 "Crash Recovery"), it must return `PREPARED`, thereby entering the `PREPARED` state.
- The participant had its state changed by operations performed under the transaction but is unable to guarantee a future successful roll forward. It

must signal this with a return of ABORTED, effectively entering the ABORTED state.

For top-level transactions, when a participant returns PREPARED it is stating that it is ready to roll the changes forward by saving the necessary record of the operations for a future commit call. The record of changes must be at least as durable as the overall state of the participant. The record must also be examined during recovery (see Section TX.2.8 “Crash Recovery”) to ensure that the participant rolls forward or rolls back as the manager dictates. The participant stays in the PREPARED state until it is told to commit or abort. It cannot, having returned PREPARED, drop the record except by following the “roll decision” described for crash recovery (see Section TX.2.8.1 “The Roll Decision”).

For nested transactions, when a participant returns PREPARED it is stating that it is ready to roll the changes forward into the parent transaction. The record of changes must be as durable as the record of changes for the parent transaction.

If a participant is currently executing an operation under a transaction when prepare is invoked for that transaction, the participant must either: wait until that operation is complete before returning from prepare; know that the operation is guaranteed to be read-only, and so will not affect its ability to prepare; or abort the transaction.

If a participant has not received any communication on or about a transaction over an extended period, it may choose to invoke getState on the manager. If getState throws UnknownTransactionException or NoSuchObjectException, the participant may safely infer that the transaction has been aborted. If getState throws a RemoteException the participant may choose to believe that the manager has crashed and abort its state in the transaction—this is not to be done lightly, since the manager may save state across crashes, and transient network failures could cause a participant to drop out of an otherwise valid and committable transaction. A participant should drop out of a transaction only if the manager is unreachable over an extended period. However, in no case should a participant drop out of a transaction it has PREPARED but not yet rolled forward.

If a participant has joined a nested transaction and it receives a prepare call for an enclosing transaction, the participant must complete the nested transaction, using getState on the manager to determine the proper type of completion.

If a participant receives a prepare call for a transaction that is already in a post-VOTING state, the participant should simply respond with that state.

If a participant receives a prepare call for a transaction that is unknown to it, it should throw UnknownTransactionException. This may happen if the participant has crashed and lost the state of a previously active transaction, or if a previous NOTCHANGED or ABORTED response was not received by the manager and the participant has since forgotten the transaction.

Note that a return value of NOTCHANGED may not be idempotent. Should the participant return NOTCHANGED it may proceed directly to clean up its state. If the manager receives a RemoteException because of network failure, the manager will likely retry the prepare. At this point a participant that has dropped the information about the transaction will throw UnknownTransactionException, and the manager will be forced to abort. A well-behaved participant should stay in the NOTCHANGED state for a while to allow a retry of prepare to again return NOTCHANGED, thus keeping the transaction alive, although this is not strictly required. No matter what it voted, a well-behaved participant should also avoid exiting for a similar period of time in case the manager needs to re-invoke prepare.

If a participant receives an abort call for a transaction, whether in the ACTIVE, VOTING, or PREPARED state, it should move to the ABORTED state and roll back all changes made under the transaction.

If a participant receives a commit call for a PREPARED transaction, it should move to the COMMITTED state and roll forward all changes made under the transaction.

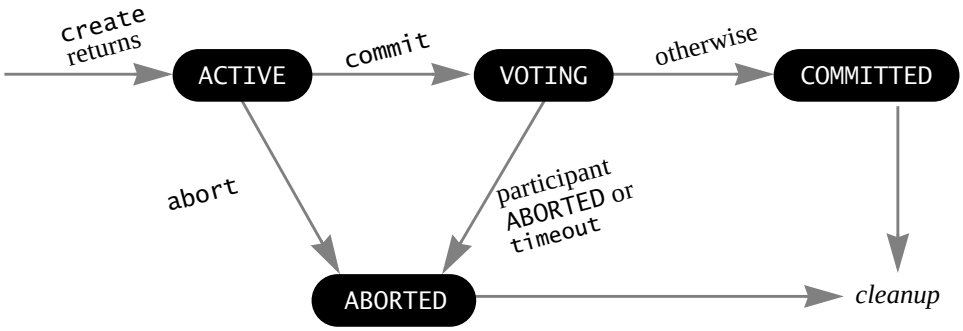
The participant's implementation of prepareAndCommit must be equivalent to the following:

```
public int prepareAndCommit(TransactionManager mgr, long id)
    throws UnknownTransactionException, RemoteException
{
    int result = prepare(mgr, id);
    if (result == PREPARED) {
        commit(mgr, id);
        result = COMMITTED;
    }
    return result;
}
```

The participant can often implement prepareAndCommit much more efficiently than shown, but it must preserve the above semantics. The manager's use of this method is described in the next section.

TX.2.7 Completing a Transaction: The Manager's View

In the manager's view, a transaction goes through the following states:



When a transaction is created using `create`, the transaction is `ACTIVE`. This is the only state in which participants may `join` the transaction. Attempting to join the transaction in any other state throws a `CannotJoinException`.

Invoking the manager's `commit` method causes the manager to move to the `VOTING` state, in which it attempts to complete the transaction by rolling forward. Each participant that has joined the transaction has its `prepare` method invoked to vote on the outcome of the transaction. The participant may return one of three votes: `NOTCHANGED`, `ABORTED`, or `COMMITTED`.

If a participant votes `ABORTED`, the manager must abort the transaction. If `prepare` throws `UnknownTransactionException` or `NoSuchObjectException`, the participant has lost its state of the transaction, and the manager must abort the transaction. If `prepare` throws `RemoteException`, the manager may retry as long as it wishes until it decides to abort the transaction.

To abort the transaction, the manager moves to the `ABORTED` state. In the `ABORTED` state, the manager should invoke `abort` on all participants that have voted `PREPARED`. The manager should also attempt to invoke `abort` on all participants on which it has not yet invoked `prepare`. These notifications are not strictly necessary for the one-parameter forms of `commit` and `abort`, since the participants will eventually abort the transaction either by timing out or by asking the manager for the state of the transaction. However, informing the participants of the abort can speed up the release of resources in these participants, and so attempting the notification is strongly encouraged.

If a participant votes `NOTCHANGED`, it is dropped from the list of participants, and no further communication will ensue. If all participants vote `NOTCHANGED` then the entire transaction was read-only and no participant has any changes to roll forward. The transaction moves to the `COMMITTED` state and then can immediately

move to *cleanup*, in which resources in the manager are cleaned up. There is no behavioral difference to a participant between a `NOTCHANGED` transaction and one that has completed the notification phase of the `COMMITTED` state.

If no participant votes `ABORTED` and at least one participant votes `PREPARED`, the transaction also moves to the `COMMITTED` state. In the `COMMITTED` state the manager must notify each participant that returned `PREPARED` to roll forward by invoking the participant's `commit` method. When the participant's `commit` method returns normally, the participant has rolled forward successfully and the manager need not invoke `commit` on it again. As long as there exists at least one participant that has not rolled forward successfully, the manager must preserve the state of the transaction and repeat attempts to invoke `commit` at reasonable intervals. If a participant's `commit` method throws `UnknownTransactionException`, this means that the participant has already successfully rolled the transaction forward even though the manager did not receive the notification, either due to a network failure on a previous invocation that was actually successful or because the participant called `getState` directly.

If the transaction is a nested one and the manager is prepared to roll the transaction forward, the members of the nested transaction must become members of the parent transaction. This *promotion* of participants into the parent manager must be atomic—all must be promoted simultaneously, or none must be. The multi-participant `promote` method is designed for this use in the case in which the parent and nested transactions have different managers.

The `promote` method takes arrays of participants and crash counts, where `crashCounts[i]` is the crash count for `parts[i]`. If any crash count is different from a crash count that is already known to the parent transaction manager, the parent manager throws `CrashCountException` and the parent transaction must abort. The `drop` parameter allows the nested transaction manager to drop itself out of the parent transaction as it promotes its participants into the parent transaction if it no longer has any need to be a participant itself.

The manager for the nested transaction should remain available until it has successfully driven each participant to completion and promoted its participants into the parent transaction. If the nested transaction's manager disappears before a participant is positively informed of the transaction's completion, that participant will not know whether to roll forward or back, forcing it to vote `ABORTED` in the parent transaction. The manager may cease `commit` invocations on its participants if any parent transaction is aborted. Aborting any transaction implicitly aborts any uncommitted nested transactions. Additionally, since any committed nested transaction will also have its results dropped, any actions taken on behalf of that transaction can be abandoned.

Invoking the manager's `abort` method, cancelling the transaction's lease, or allowing the lease to expire also moves the transaction to the `ABORTED` state as

described above. Any transactions nested inside that transaction are also moved directly to the ABORTED state.

The manager may optimize the VOTING state by invoking a participant's prepareAndCommit method if the transaction has only one participant that has not yet been asked to vote and all previous participants have returned NOTCHANGED. (Note that this includes the special case in which the transaction has exactly one participant.) If the manager receives an ABORTED result from prepareAndCommit, it proceeds to the ABORTED state. In effect, a prepareAndCommit moves through the VOTING state straight to operating on the results.

A getState call on the manager can return any of ACTIVE, VOTING, ABORTED, NOTCHANGED, or COMMITTED. A manager is permitted, but not required, to return NOTCHANGED if it is in the COMMITTED state and all participants voted NOTCHANGED.

TX.2.8 Crash Recovery

Crash recovery ensures that a top-level transaction will consistently abort or roll forward in the face of a system crash. Nested transactions are not involved.

The manager has one *commit point*, where it must save state in a durable fashion. This is when it enters the COMMITTED state with at least one PREPARED participant. The manager must, at this point, commit the list of PREPARED participants into durable storage. This storage must persist until all PREPARED participants successfully roll forward. A manager may choose to also store the list of PREPARED participants that have already successfully rolled forward or to rewrite the list of PREPARED participants as it shrinks, but this optimization is not required (although it is recommended as good citizenship). In the event of a manager crash, the list of participants must be recovered, and the manager must continue acting in the COMMITTED state until it can successfully notify all PREPARED participants.

The participant also has one commit point, which is prior to voting PREPARED. When it votes PREPARED, the participant must have durably recorded the record of changes necessary to successfully roll forward in the event of a future invocation of commit by the manager. It can remove this record when it is prepared to successfully return from commit.

Because of these commitments, manager and participant implementations should use durable forms of RMI references, such as the Activatable references introduced in the Java 2 platform. An unreachable manager causes much havoc and should be avoided as much as possible. A vanished PREPARED participant puts a transaction in an untenable permanent state in which some, but not all, of the participants have rolled forward.

TX.2.8.1 The Roll Decision

If a participant votes `PREPARED` for a top-level transaction, it must guarantee that it will execute a recovery process if it crashes between completing its durable record and receiving a `commit` notification from the manager. This recovery process must read the record of the crashed participant and make a *roll decision*—whether to roll the recorded changes forward or roll them back.

To make this decision, it invokes the `getState` method on the transaction manager. This can have the following results:

- `getState` returns `COMMITTED`: The recovery should move the participant to the `COMMITTED` state.
- `getState` throws either an `UnknownTransactionException` or a `NoSuchObjectException`: The recovery should move the participant to the `ABORTED` state.
- `getState` throws `RemoteException`: The recovery should repeat the attempt after a pause.

TX.2.9 Durability

Durability is a commitment, but it is not a guarantee. It is impossible to guarantee that any given piece of stable storage can *never* be lost; one can only achieve decreasing probabilities of loss. Data that is force-written to a disk may be considered durable, but it is less durable than data committed to two or more separate, redundant disks. When we speak of "durability" in this system it is always used relative to the expectations of the human who decided which entities to use for communication.

With multi-participant transactions it is entirely possible that different participants have different durability levels. The manager may be on a tightly replicated system with its durable storage duplicated on several host systems, giving a high degree of durability, while a participant may be using only one disk. Or a participant may always store its data in memory, expecting to lose it in a system crash (a database of people currently logged into the host, for example, need not survive a system crash). When humans make a decision to use a particular manager and set of participants for a transaction they must take into account these differences and be aware of the ramifications of committing changes that may be more durable on one participant than another. Determining, or even defining and exposing, varying levels of durability is outside the scope of this specification.

TX.3 Default Transaction Semantics

THE two-phase commit protocol defines how a transaction is created and later driven to completion by either committing or aborting. It is neutral with respect to the semantics of locking under the transaction or other behaviors that impart semantics to the use of the transaction. Specific clients and servers, however, must be written to expect specific transaction semantics. This model is to separate the completion protocol from transaction semantics, where transaction semantics are represented in the parameters and return values of methods by which clients and participants interact.

This chapter defines the default transaction semantics of services. These semantics preserve the traditional ACID properties (you will find a brief description of the ACID properties in Section TX.1.2 “Distributed Transactions and ACID Properties”). The semantics are represented by the `Transaction` and `NestableTransaction` interfaces and their implementation classes `ServerTransaction` and `NestableServerTransaction`. Any participant that accepts as a parameter or returns any of these types is promising to abide by the following definition of semantics for any activities performed under that transaction.

TX.3.1 `Transaction` and `NestableTransaction` Interfaces

The client’s view of transactions is through two interfaces: `Transaction` for top-level transactions and `NestableTransaction` for transactions under which nested transactions can be created. First, the `Transaction` interface:

```
package net.jini.core.transaction;

public interface Transaction {
    public static class Created implements Serializable {
        public final Transaction transaction;
        public final Lease lease;
        Created(Transaction transaction, Lease lease) {…}
```

```
        }
        void commit() // §TX.2.5
            throws UnknownTransactionException,
                   CannotCommitException,
                   RemoteException;
        void commit(long waitFor) // §TX.2.5
            throws UnknownTransactionException,
                   CannotCommitException,
                   TimeoutExpiredException, RemoteException;
        void abort() // §TX.2.5
            throws UnknownTransactionException,
                   CannotAbortException,
                   RemoteException;
        void abort(long waitFor) // §TX.2.5
            throws UnknownTransactionException,
                   CannotAbortException,
                   TimeoutExpiredException, RemoteException;
    }
```

The `Created` nested class is used in a factory `create` method for top-level transactions (defined in the next section) to hold two return values: the newly created `Transaction` object and the transaction's lease, which is the lease granted by the transaction manager. The `commit` and `abort` methods have the same semantics as discussed in Section TX.2.5 "Completing a Transaction: The Client's View".

Nested transactions are created using `NestableTransaction` methods:

```
package net.jini.core.transaction;

public interface NestableTransaction extends Transaction {
    public static class Created implements Serializable {
        public final NestableTransaction transaction;
        public final Lease lease;
        Created(NestableTransaction transaction, Lease lease)
            {…}
    }
    Created create(long leaseFor) // §TX.2.2
        throws UnknownTransactionException,
               CannotJoinException, LeaseDeniedException,
               RemoteException;
    Created create(NestableTransactionManager mgr,
                   long leaseFor) // §TX.2.2
        throws UnknownTransactionException,
```

```
            CannotJoinException, LeaseDeniedException,
            RemoteException;
    }
```

The `Created` nested class is used to hold two return values: the newly created `Transaction` object and the transaction's lease, which is the lease granted by the transaction manager. In both `create` methods, `leaseFor` is the requested lease time in milliseconds. In the one-parameter `create` method the nested transaction is created with the same transaction manager as the transaction on which the method is invoked. The other `create` method can be used to specify a different transaction manager to use for the nested transaction.

TX.3.2 `TransactionFactory` Class

The `TransactionFactory` class is used to create top-level transactions.

```
package net.jini.core.transaction;

public class TransactionFactory {
    public static Transaction.Created
        create(TransactionManager mgr, long leaseFor)
                                                  // §TX.2.1
        throws LeaseDeniedException, RemoteException {…}
    public static NestableTransaction.Created
        create(NestableTransactionManager mgr,long leaseFor)
                                                  // §TX.2.2
        throws LeaseDeniedException, RemoteException {…}
}
```

The first `create` method is usually used when nested transactions are not required. However, if the manager that is passed to this method is in fact a `NestableTransactionManager`, then the returned `Transaction` can in fact be cast to a `NestableTransaction`. The second `create` method is used when it is known that nested transactions need to be created. In both cases, a `Created` instance is used to hold two return values: the newly created transaction object and the granted lease.

TX.3.3 ServerTransaction and NestableServerTransaction Classes

The ServerTransaction class exposes functionality necessary for writing participants that support top-level transactions. Participants can cast a Transaction to a ServerTransaction to obtain access to this functionality.

```
public class ServerTransaction
    implements Transaction, Serializable
{
    public final TransactionManager mgr;
    public final long id;
    public ServerTransaction(TransactionManager mgr, long id)
        {…}
    public void join(TransactionParticipant part,
                     long crashCount) // §TX.2.3
        throws UnknownTransactionException,
               CannotJoinException, CrashCountException,
               RemoteException {…}
    public int getState() // §TX.2.7
        throws UnknownTransactionException, RemoteException
        {…}
    public boolean isNested() {…} // §TX.3.3
}
```

The mgr field is a reference to the transaction manager that created the transaction. The id field is the transaction identifier returned by the transaction manager's create method.

The constructor should not be used directly; it is intended for use by the TransactionFactory implementation.

The methods join, commit, abort, and getState invoke the corresponding methods on the manager, passing the transaction identifier. They are provided as a convenience to the programmer, primarily to eliminate the possibility of passing an identifier to the wrong manager. For example, given a ServerTransaction object tr, the invocation

```
tr.join(participant, crashCount);
```

is equivalent to

```
tr.mgr.join(tr.id, participant, crashCount);
```

The isNested method returns true if the transaction is a nested transaction (that is, if it is a NestableServerTransaction with a non-null parent) and

`false` otherwise. It is provided as a method on `ServerTransaction` for the convenience of participants that do not support nested transactions.

The `hashCode` method returns the `id` cast to an `int` XORed with the result of `mgr.hashCode()`. The `equals` method returns `true` if the specified object is a `ServerTransaction` object with the same manager and transaction identifier as the object on which it is invoked.

The `NestableServerTransaction` class exposes functionality that is necessary for writing participants that support nested transactions. Participants can cast a `NestableTransaction` to a `NestableServerTransaction` to obtain access to this functionality.

```
package net.jini.core.transaction.server;

public class NestableServerTransaction
    extendsServerTransaction implements NestableTransaction
{
    public final NestableServerTransaction parent;
    public NestableServerTransaction(
           NestableTransactionManager mgr, long id,
           NestableServerTransaction parent) {…}
    public void promote(TransactionParticipant[] parts,
                        long[] crashCounts,
                        TransactionParticipant drop)
                                                  // §TX.2.7
        throws UnknownTransactionException,
               CannotJoinException, CrashCountException,
               RemoteException {…}
    public boolean enclosedBy(NestableTransaction enclosing)
        {…}
}
```

The `parent` field is a reference to the parent transaction if the transaction is nested (see Section TX.2.2 “Starting a Nested Transaction”) or `null` if it is a top-level transaction.

The constructor should not be used directly; it is intended for use by the `TransactionFactory` and `NestableServerTransaction` implementations.

Given a `NestableServerTransaction` object `tr`, the invocation

```
tr.promote(parts, crashCounts, drop)
```

is equivalent to

```
((NestableTransactionManager)tr.mgr).promote(tr.id, parts,
                                             crashCounts, drop)
```

The `enclosedBy` method returns `true` if the specified transaction is an enclosing transaction (parent, grandparent, etc.) of the transaction on which the method is invoked; otherwise it returns `false`.

TX.3.4 `CannotNestException` Class

If a service implements the default transaction semantics but does not support nested transactions, it usually needs to throw an exception if a nested transaction is passed to it. The `CannotNestException` is provided as a convenience for this purpose, although a service is not required to use this specific exception.

```
package net.jini.core.transaction;

public class CannotNestException extends TransactionException
{
    public CannotNestException() {…}
    public CannotNestException(String desc) {…}
}
```

TX.3.5 Semantics

Activities that are performed as pure transactions (all access to shared mutable state is performed under transactional control) are subject to sequential ordering, meaning the overall effect of executing a set of sibling (all at the same level, whether top-level or nested) pure transactions concurrently is always equivalent to some sequential execution.

Ancestor transactions can execute concurrently with child transactions, subject to the locking rules below.

Transaction semantics for objects are defined in terms of strict two-phase locking. Every transactional operation is described in terms of acquiring locks on objects; these locks are held until the transaction completes. The most typical locks are read and write locks, but others are possible. Whatever the lock types are, conflict rules are defined such that if two operations do not commute, then they acquire conflicting locks. For objects using standard read and write locks, read locks do not conflict with other read locks, but write locks conflict with both

read locks and other write locks. A transaction can acquire a lock if the only conflicting locks are those held by ancestor transactions (or itself). If a necessary lock cannot be acquired and the operation is defined to proceed without waiting for that lock, then serializability might be violated. When a subtransaction commits, its locks are inherited by the parent transaction.

In addition to locks, transactional operations can be defined in terms of object creation and deletion visibility. If an object is defined to be created under a transaction, then the existence of the object is visible only within that transaction and its inferiors, but will disappear if the transaction aborts. If an object is defined to be deleted under a transaction, then the object is not visible to any transaction (including the deleting transaction) but will reappear if the transaction aborts. When a nested transaction commits, visibility state is inherited by the parent transaction.

Once a transaction reaches the `VOTING` stage, if all execution under the transaction (and its subtransactions) has finished, then the only reasons the transaction can abort are:

- The manager crashes (or has crashed)
- One or more participants crash (or have crashed)
- There is an explicit abort

Transaction deadlocks are not guaranteed to be prevented or even detected, but managers and participants are permitted to break known deadlocks by aborting transactions.

An active transaction is an *orphan* if it or one of its ancestors is guaranteed to abort. This can occur because an ancestor has explicitly aborted or because some participant or manager of the transaction or an ancestor has crashed. Orphans are not guaranteed to be detected by the system, so programmers using transactions must be aware that orphans can see internally inconsistent state and take appropriate action.

Causal ordering information about transactions is not guaranteed to be propagated. First, given two sibling transactions (at any level), it is not possible to tell whether they were created concurrently or sequentially (or in what order). Second, if two transactions are causally ordered and the earlier transaction has completed, the outcome of the earlier transaction is not guaranteed to be known at every participant used by the later transaction, unless the client is successful in using the variant of `commit` or `abort` that takes a timeout parameter. Programmers using non-blocking forms of operations must take this into account.

As long as a transaction persists in attempting to acquire a lock that conflicts with another transaction, the participant will persist in attempting to resolve the

outcome of the transaction that holds the conflicting lock. Attempts to acquire a lock include making a blocking call, continuing to make non-blocking calls, and registering for event notification under a transaction.

TX.3.6 Serialized Forms

Class	serialVersionUID	Serialized Fields
Transaction.Created	–5199291723008952986L	*all public fields*
NestableTransaction.Created	–2979247545926318953L	*all public fields*
TransactionManager.Created	–4233846033773471113L	*all public fields*
ServerTransaction	4552277137549765374L	*all public fields*
NestableServerTransaction	–3438419132543972925L	*all public fields*
TransactionException	–5009935764793203986L	*none*
CannotAbortException	3597101646737510009L	*none*
CannotCommitException	–4497341152359563957L	*none*
CannotJoinException	5568393043937204939L	*none*
CannotNestException	3409604500491735434L	*none*
TimeoutExpiredException	3918773760682958000L	*all public fields*
UnknownTransactionException	443798629936327009L	*none*
CrashCountException	4299226125245015671L	*none*

LU
Lookup Service

LU.1 Introduction

THE Jini lookup service is a fundamental part of the federation infrastructure for a *djinn,* the group of devices, resources, and users that are joined by the Jini technology infrastructure. The *lookup service* provides a central registry of services available within the djinn. This lookup service is a primary means for programs to find services within the djinn, and is the foundation for providing user interfaces through which users and administrators can discover and interact with services in the djinn.

Although the primary purpose of this specification is to define the interface to the djinn's central service registry, the interfaces defined here can readily be used in other service registries.

LU.1.1 The Lookup Service Model

The lookup service maintains a flat collection of *service items*. Each service item represents an instance of a service available within the djinn. The item contains the RMI stub (if the service is implemented as a remote object) or other object (if the service makes use of a local proxy) that programs use to access the service, and an extensible collection of attributes that describe the service or provide secondary interfaces to the service.

When a new service is created (for example, when a new device is added to the djinn), the service registers itself with the djinn's lookup service, providing an initial collection of attributes. For example, a printer might include attributes indicating speed (in pages per minute), resolution (in dots per inch), and whether duplex printing is supported. Among the attributes might be an indicator that the service is new and needs to be configured.

An administrator uses the event mechanism of the lookup service to receive notifications as new services are registered. To configure the service, the administrator might look for an attribute that provides an applet for this purpose. The administrator might also use an applet to add new attributes, such as the physical location of the service and a common name for it; the service would receive these attribute change requests from the applet and respond by making the changes at the lookup service.

Programs (including other services) that need a particular type of service can use the lookup service to find an instance. A match can be made based on the specific data types for the Java programming language implemented by the service as well as the specific attributes attached to the service. For example, a program that needs to make use of transactions might look for a service that supports the type `net.jini.core.transaction.server.TransactionManager` and might further qualify the match by desired location.

Although the collection of service items is flat, a wide variety of hierarchical views can be imposed on the collection by aggregating items according to service types and attributes. The lookup service provides a set of methods to enable incremental exploration of the collection, and a variety of user interfaces can be built by using these methods, allowing users and administrators to browse. Once an appropriate service is found, the user might interact with the service by loading a user interface applet, attached as another attribute on the item.

If a service encounters some problem that needs administrative attention, such as a printer running out of toner, the service can add an attribute that indicates what the problem is. Administrators again use the event mechanism to receive notification of such problems.

LU.1.2 Attributes

The attributes of a service item are represented as a set of attribute sets. An individual *attribute set* is represented as an instance of some class for the Java platform, each attribute being a public field of that class. The class provides strong typing of both the set and the individual attributes. A service item can contain multiple instances of the same class with different attribute values, as well as multiple instances of different classes. For example, an item might have multiple instances of a `Name` class, each giving the common name of the service in a different language, plus an instance of a `Location` class, an `Owner` class, and various service-specific classes. The schema used for attributes is not constrained by this specification, but a standard foundation schema for Jini technology-enabled systems is defined in the *Jini Lookup Attribute Schema Specification*.

Concretely, a set of attributes is implemented with a class that correctly implements the interface `net.jini.core.entry.Entry`, as described in Section DJ "Entry". Operations on the lookup service are defined in terms of template matching, using the same semantics as in Section DJ "Entry", but the definition is augmented to deal with sets of entries and sets of templates. A set of entries matches a set of templates if there is at least one matching entry for every template (with every entry usable as the match for more than one template).

LU.2 The ServiceRegistrar

THE types defined in this specification are in the net.jini.core.lookup package. The following types are imported from other packages and are referenced in unqualified form in the rest of this specification:

```
java.rmi.MarshalledObject
java.rmi.RemoteException
java.rmi.UnmarshalException
java.io.Serializable
java.io.DataInput
java.io.DataOutput
java.io.IOException
net.jini.core.discovery.LookupLocator
net.jini.core.entry.Entry
net.jini.core.lease.Lease
net.jini.core.event.RemoteEvent
net.jini.core.event.EventRegistration
net.jini.core.event.RemoteEventListener
```

LU.2.1 ServiceID

Every service is assigned a universally unique identifier (UUID), represented as an instance of the ServiceID class.

```
public final class ServiceID implements Serializable {
    public ServiceID(long mostSig, long leastSig) {…}
    public ServiceID(DataInput in) throws IOException {…}
    public void writeBytes(DataOutput out) throws IOException
        {…}
    public long getMostSignificantBits() {…}
    public long getLeastSignificantBits() {…}
}
```

A service ID is a 128-bit value. Service IDs are equal (using the `equals` method) if they represent the same 128-bit value. For simplicity and reliability, service IDs are intended to be generated only by lookup services, not by clients. As such, the `ServiceID` constructor merely takes 128 bits of data, to be computed in an implementation-dependent manner by the lookup service. The `writeBytes` method writes out 16 bytes in standard network byte order. The second constructor reads in 16 bytes in standard network byte order.

The most significant long can be decomposed into the following unsigned fields:

```
0xFFFFFFFF00000000        time_low
0x00000000FFFF0000        time_mid
0x000000000000F000        version
0x0000000000000FFF        time_hi
```

The least significant long can be decomposed into the following unsigned fields:

```
0xC000000000000000        variant
0x3FFF000000000000        clock_seq
0x0000FFFFFFFFFFFF        node
```

The `variant` field must be 0x2. The `version` field must be either 0x1 or 0x4. If the `version` field is 0x4, then the most significant bit of the `node` field must be set to 1, and the remaining fields are set to values produced by a cryptographically strong pseudo-random number generator. If the `version` field is 0x1, then the `node` field is set to an IEEE 802 address, the `clock_seq` field is set to a 14-bit random number, and the `time_low`, `time_mid`, and `time_hi` fields are set to the least, middle, and most significant bits (respectively) of a 60-bit timestamp measured in 100-nanosecond units since midnight, October 15, 1582 UTC.

The `toString` method returns a 36-character string of six fields separated by hyphens, each field represented in lowercase hexadecimal with the same number of digits as in the field. The order of fields is: `time_low`, `time_mid`, `version` and `time_hi` treated as a single field, `variant` and `clock_seq` treated as a single field, and `node`.

LU.2.2 `ServiceItem`

Items are stored in the lookup service using instances of the `ServiceItem` class.

```
public class ServiceItem implements Serializable {
    public ServiceItem(ServiceID serviceID,
                       Object service,
```

```
                        Entry[] attributeSets) {…}
    public ServiceID serviceID;
    public Object service;
    public Entry[] attributeSets;
}
```

The constructor simply assigns each parameter to the corresponding field.

Each `Entry` represents an attribute set. The class must have a public no-arg constructor, and all non-static, non-final, non-transient public fields must be declared with reference types, holding serializable objects. Each such field is serialized separately as a `MarshalledObject`, and field equality is defined by `MarshalledObject.equals`. The only relationship constraint on attribute sets within an item is that exact duplicates are eliminated; other than that, multiple attribute sets of the same type are permitted, multiple attribute set types can have a common superclass, and so on.

The `net.jini.core.entry.UnusableEntryException` is not used in the lookup service; alternate semantics for individual operations are defined later in this section.

LU.2.3 `ServiceTemplate` and Item Matching

Items in the lookup service are matched using instances of the `ServiceTemplate` class.

```
public class ServiceTemplate implements Serializable {
    public ServiceTemplate(ServiceID serviceID,
                           Class[] serviceTypes,
                           Entry[] attributeSetTemplates) {…}
    public ServiceID serviceID;
    public Class[] serviceTypes;
    public Entry[] attributeSetTemplates;
}
```

The constructor simply assigns each parameter to the corresponding field. A service item (`item`) matches a service template (`tmpl`) if:

- `item.serviceID` equals `tmpl.serviceID` (or if `tmpl.serviceID` is `null`), and
- `item.service` is an instance of every type in `tmpl.serviceTypes`, and
- `item.attributeSets` contains at least one matching entry for each entry template in `tmpl.attributeSetTemplates`.

An entry matches an entry template if the class of the template is the same as, or a superclass of, the class of the entry, and every non-`null` field in the template equals the corresponding field of the entry. Every entry can be used to match more than one template. For both service types and entry classes, type matching is based simply on fully qualified class names. Note that in a service template, for `serviceTypes` and `attributeSetTemplates`, a `null` field is equivalent to an empty array; both represent a wildcard.

LU.2.4 Other Supporting Types

The `ServiceMatches` class is used for the return value when looking up multiple items.

```
public class ServiceMatches implements Serializable {
    public ServiceMatches(ServiceItem[] items,
                          int totalMatches) {…}
    public ServiceItem[] items;
    public int totalMatches;
}
```

The constructor simply assigns each parameter to the corresponding field.

A `ServiceEvent` extends `RemoteEvent` with methods to obtain the service ID of the matched item, the transition that triggered the event, and the new state of the matched item.

```
public abstract class ServiceEvent extends RemoteEvent {
    public ServiceEvent(Object source,
                        long eventID,
                        long seqNum,
                        MarshalledObject handback,
                        ServiceID serviceID,
                        int transition) {…}
    public ServiceID getServiceID() {…}
    public int getTransition() {…}
    public abstract ServceItem getServiceItem() {…}
}
```

The `getServiceID` and `getTransition` methods return the value of the corresponding constructor parameter. The remaining constructor parameters are the same as in the `RemoteEvent` constructor.

The rest of the semantics of both these classes is explained in the next section.

LU.2.5 ServiceRegistrar

The ServiceRegistrar defines the interface to the lookup service. The interface is not a remote interface; each implementation of the lookup service exports proxy objects that implement the ServiceRegistrar interface local to the client, using an implementation-specific protocol to communicate with the actual remote server. All of the proxy methods obey normal RMI remote interface semantics except where explicitly noted. Two proxy objects are equal (using the equals method) if they are proxies for the same lookup service.

Methods are provided to register service items, find items that match a template, receive event notifications when items are modified, and incrementally explore the collection of items along the three major axes: entry class, attribute value, and service type.

```
public interface ServiceRegistrar {
    ServiceRegistration register(ServiceItem item,
                                 long leaseDuration)
        throws RemoteException;

    Object lookup(ServiceTemplate tmpl)
        throws RemoteException;

    ServiceMatches
        lookup(ServiceTemplate tmpl, int maxMatches)
        throws RemoteException;

    int TRANSITION_MATCH_NOMATCH = 1 << 0;
    int TRANSITION_NOMATCH_MATCH = 1 << 1;
    int TRANSITION_MATCH_MATCH = 1 << 2;

    EventRegistration notify(ServiceTemplate tmpl,
                             int transitions,
                             RemoteEventListener listener,
                             MarshalledObject handback,
                             long leaseDuration)
        throws RemoteException;

    Class[] getEntryClasses(ServiceTemplate tmpl)
        throws RemoteException;

    Object[] getFieldValues(ServiceTemplate tmpl,
```

```
                                int setIndex,
                                String field)
            throws NoSuchFieldException, RemoteException;

        Class[] getServiceTypes(ServiceTemplate tmpl,
                                String prefix)
            throws RemoteException;

        ServiceID getServiceID();
        LookupLocator getLocator() throws RemoteException;

        String[] getGroups() throws RemoteException;
    }
```

Every method invocation on `ServiceRegistrar` and `ServiceRegistration` is atomic with respect to other invocations.

The `register` method is used to register a new service and to re-register an existing service. The method is defined so that it can be used in an idempotent fashion. Specifically, if a call to `register` results in a `RemoteException` (in which case the item might or might not have been registered), the caller can simply repeat the call to `register` with the same parameters, until it succeeds.

To register a new service, `item.serviceID` should be `null`. In that case, if `item.service` does not equal (using `MarshalledObject.equals`) any existing item's service object, then a new service ID will be assigned and included in the returned `ServiceRegistration` (described in the next section). The service ID is unique over time and space with respect to all other service IDs generated by all lookup services. If `item.service` does equal an existing item's service object, the existing item is first deleted from the lookup service (even if it has different attributes) and its lease is cancelled, but that item's service ID is reused for the newly registered item.

To re-register an existing service, or to register the service in any other lookup service, `item.serviceID` should be set to the same service ID that was returned by the initial registration. If an item is already registered under the same service ID, the existing item is first deleted (even if it has different attributes or a different service instance) and its lease is cancelled by the lookup service. Note that service object equality is not checked in this case, to allow for reasonable evolution of the service (for example, the serialized form of the stub changes or the service implements a new interface).

Any duplicate attribute sets that are included in a service item are eliminated in the stored representation of the item. The lease duration request (specified in milliseconds) is not exact; the returned lease is allowed to have a shorter (but not

longer) duration than what was requested. The registration is persistent across restarts (crashes) of the lookup service until the lease expires or is cancelled.

The single-parameter form of `lookup` returns the service object (that is, just `ServiceItem.service`) from an item matching the template or `null` if there is no match. If multiple items match the template, it is arbitrary as to which service object is returned by the invocation. If the returned object cannot be deserialized, an `UnmarshalException` is thrown with the standard RMI semantics.

The two-parameter form of `lookup` returns at most `maxMatches` items matching the template and the total number of items that match the template. The return value is never `null`, and the returned items array is `null` only if `maxMatches` is zero. For each returned item, if the service object cannot be deserialized, the `service` field of the item is set to `null` and no exception is thrown. Similarly, if an attribute set cannot be deserialized, that element of the `attributeSets` array is set to `null` and no exception is thrown.

The `notify` method is used to register for event notification. The registration is leased; the lease duration request (specified in milliseconds) is not exact. The registration is persistent across restarts (crashes) of the lookup service until the lease expires or is cancelled. The event ID in the returned `EventRegistration` is unique at least with respect to all other active event registrations at this lookup service with different service templates or transitions.

While the event registration is in effect, a `ServiceEvent` is sent to the specified listener whenever a `register`, lease cancellation or expiration, or attribute change operation results in an item changing state in a way that satisfies the template and transition combination. The `transitions` parameter is the bitwise OR of any non-empty set of transition values:

- `TRANSITION_MATCH_NOMATCH`: An event is sent when the changed item matches the template before the operation, but doesn't match the template after the operation (this includes deletion of the item).
- `TRANSITION_NOMATCH_MATCH`: An event is sent when the changed item doesn't match the template before the operation (this includes not existing), but does match the template after the operation.
- `TRANSITION_MATCH_MATCH`: An event is sent when the changed item matches the template both before and after the operation.

The `getTransition` method of `ServiceEvent` returns the singleton transition value that triggered the match.

The `getServiceItem` method of `ServiceEvent` returns the new state of the item (the state after the operation) or `null` if the item was deleted by the operation. Note that this method is declared `abstract`; a lookup service uses a subclass of `ServiceEvent` to transmit the new state of the item however it chooses.

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. For example, a gap might occur if the lookup service crashes, even if no events are lost due to the crash.

As mentioned earlier, users are allowed to explore a collection of items down each of the major axes: entry class, attribute value, and service type.

The `getEntryClasses` method looks at all service items that match the specified template, finds every entry (among those service items) that either doesn't match any entry templates or is a subclass of at least one matching entry template, and returns the set of the (most specific) classes of those entries. Duplicate classes are eliminated, and the order of classes within the returned array is arbitrary. A `null` reference (not an empty array) is returned if there are no such entries or no matching items. If a returned class cannot be deserialized, that element of the returned array is set to `null` and no exception is thrown.

The `getFieldValues` method looks at all service items that match the specified template, finds every entry (among those service items) that matches `tmpl.attributeSetTemplates[setIndex]`, and returns the set of values of the specified field of those entries. Duplicate values are eliminated, and the order of values within the returned array is arbitrary. a `null` reference (not an empty array) is returned if there are no matching items. If a returned value cannot be deserialized, that element of the returned array is set to `null` and no exception is thrown. `NoSuchFieldException` is thrown if `field` does not name a field of the entry template.

The `getServiceTypes` method looks at all service items that match the specified template and, for every service item, finds the most specific type (class or interface) or types the service item is an instance of that are neither equal to, nor a superclass of, any of the service types in the template and that have names that start with the specified prefix, and returns the set of all such types. Duplicate types are eliminated, and the order of types within the returned array is arbitrary. A `null` reference (not an empty array) is returned if there are no such types. If a returned type cannot be deserialized, that element of the returned array is set to `null` and no exception is thrown.

Every lookup service assigns itself a service ID when it is first created; this service ID is returned by the `getServiceID` method. (Note that this does not make a remote call.) A lookup service is always registered with itself under this service ID, and if a lookup service is configured to register itself with other lookup services, it will register with all of them using this same service ID.

The `getLocator` method returns a `LookupLocator` that can be used if necessary for unicast discovery of the lookup service. The definition of this class is given in Section DJ "Discovery and Join".

The `getGroups` method returns the set of groups that this lookup service is currently a member of. The semantics of these groups is defined in Section DJ "Discovery and Join".

LU.2.6 `ServiceRegistration`

A registered service item is manipulated using a `ServiceRegistration` instance.

```
public interface ServiceRegistration {
    ServiceID getServiceID();
    Lease getLease();
    void addAttributes(Entry[] attrSets)
        throws UnknownLeaseException, RemoteException;
    void modifyAttributes(Entry[] attrSetTemplates,
                          Entry[] attrSets)
        throws UnknownLeaseException, RemoteException;
    void setAttributes(Entry[] attrSets)
        throws UnknownLeaseException, RemoteException;
}
```

Like `ServiceRegistrar`, this is not a remote interface; each implementation of the lookup service exports proxy objects that implement this interface local to the client. The proxy methods obey normal RMI remote interface semantics.

The `getServiceID` method returns the service ID for this service. (Note that this does not make a remote call.)

The `getLease` method returns the lease that controls the service registration, allowing the lease to be renewed or cancelled. (Note that `getLease` does not make a remote call.)

The `addAttributes` method adds the specified attribute sets (those that aren't duplicates of existing attribute sets) to the registered service item. Note that this operation has no effect on existing attribute sets of the service item and can be repeated in an idempotent fashion. `UnknownLeaseException` is thrown if the registration lease has expired or been cancelled.

The `modifyAttributes` method is used to modify existing attribute sets. The lengths of the `attrSetTemplates` and `attrSets` arrays must be equal, or `IllegalArgumentException` is thrown. The service item's attribute sets are modified as follows. For each array index `i`: if `attrSets[i]` is `null`, then every entry that matches `attrSetTemplates[i]` is deleted; otherwise, for every non-`null` field in `attrSets[i]`, the value of that field is stored into the corresponding field of every entry that matches `attrSetTemplates[i]`. The class of `attrSets[i]` must be the same as, or a superclass of, the class of `attrSetTemplates[i]`, or

IllegalArgumentException is thrown. If the modifications result in duplicate entries within the service item, the duplicates are eliminated. An UnknownLeaseException is thrown if the registration lease has expired or been cancelled.

Note that it is possible to use modifyAttributes in ways that are not idempotent. The attribute schema should be designed in such a way that all intended uses of this method can be performed in an idempotent fashion. Also note that modifyAttributes does not provide a means for setting a field to null; it is assumed that the attribute schema is designed in such a way that this is not necessary.

The setAttributes method deletes all of the service item's existing attributes and replaces them with the specified attribute sets. Any duplicate attribute sets are eliminated in the stored representation of the item. UnknownLeaseException is thrown if the registration lease has expired or been cancelled.

LU.2.7 Serialized Forms

Class	serialVersionUID	Serialized Fields
ServiceID	–7803375959559762239L	long mostSig long leastSig
ServiceItem	717395451032330758L	*all public fields*
ServiceTemplate	7854483807886483216L	*all public fields*
ServiceMatches	–5518280843537399398L	*all public fields*
ServiceEvent	1304997274096842701L	ServiceID serviceID int transition