as printers, displays, or disks; software such as applications or utilities; information such as databases and files; and users of the system.

Services in a Jini system communicate with each other by using a *service protocol*, which is a set of interfaces written in the Java programming language. The set of such protocols is open ended. The base Jini system defines a small number of such protocols that define critical service interactions.

AR.2.1.2 Lookup Service

Services are found and resolved by a *lookup service*. The lookup service is the central bootstrapping mechanism for the system and provides the major point of contact between the system and users of the system. In precise terms, a lookup service maps interfaces indicating the functionality provided by a service to sets of objects that implement the service. In addition, descriptive entries associated with a service allow more fine-grained selection of services based on properties understandable to people.

Objects in a lookup service may include other lookup services; this provides hierarchical lookup. Further, a lookup service may contain objects that encapsulate other naming or directory services, providing a way for bridges to be built between a Jini Lookup service and other forms of lookup service. Of course, references to a Jini Lookup service may be placed in these other naming and directory services, providing a means for clients of those services to gain access to a Jini system.

A service is added to a lookup service by a pair of protocols called *discovery* and *join*—first the service locates an appropriate lookup service (by using the *discovery* protocol), and then it joins it (by using the *join* protocol).

AR.2.1.3 Java Remote Method Invocation (RMI)

Communication between services can be accomplished using *Java Remote Method Invocation* (RMI). The infrastructure to support communication between services is not itself a service that is discovered and used but is, rather, a part of the Jini technology infrastructure. RMI provides mechanisms to find, activate, and garbage collect object groups.

Fundamentally, RMI is a Java programming language-enabled extension to traditional remote procedure call mechanisms. RMI allows not only data to be passed from object to object around the network but full objects, including code. Much of the simplicity of the Jini system is enabled by this ability to move code around the network in a form that is encapsulated as an object.

AR.2.1.4 Security

The design of the security model for Jini technology is built on the twin notions of a *principal* and an *access control list*. Jini services are accessed on behalf of some entity—the principal— which generally traces back to a particular user of the system. Services themselves may request access to other services based on the identity of the object that implements the service. Whether access to a service is allowed depends on the contents of an access control list that is associated with the object.

AR.2.1.5 Leasing

Access to many of the services in the Jini system environment is *lease* based. A lease is a grant of guaranteed access over a time period. Each lease is negotiated between the user of the service and the provider of the service as part of the service protocol: A service is requested for some period; access is granted for some period, presumably taking the request period into account. If a lease is not renewed before it is freed—either because the resource is no longer needed, the client or network fails, or the lease is not permitted to be renewed—then both the user and the provider of the resource may conclude that the resource can be freed.

Leases are either exclusive or non-exclusive. Exclusive leases ensure that no one else may take a lease on the resource during the period of the lease; non-exclusive leases allow multiple users to share a resource.

AR.2.1.6 Transactions

A series of operations, either within a single service or spanning multiple services, can be wrapped in a *transaction*. The Jini Transaction interfaces supply a service protocol needed to coordinate a *two-phase commit*. How transactions are implemented—and indeed, the very semantics of the notion of a transaction—is left up to the service using those interfaces.

AR.2.1.7 Events

The Jini architecture supports distributed *events*. An object may allow other objects to register interest in events in the object and receive a notification of the occurrence of such an event. This enables distributed event-based programs to be written with a variety of reliability and scalability guarantees.

AR.2.2 Component Overview

The components of the Jini system can be segmented into three categories: *infrastructure*, *programming model*, and *services*. The infrastructure is the set of components that enables building a federated Jini system, while the services are the entities within the federation. The programming model is a set of interfaces that enables the construction of reliable services, including those that are part of the infrastructure and those that join into the federation.

These three categories, though distinct and separable, are entangled to such an extent that the distinction between them can seem blurred. Moreover, it is possible to build systems that have some of the functionality of the Jini system with variants on the categories or without all three of them. But a Jini system gains its full power because it is a *system* built with the particular infrastructure and programming models described, based on the notion of a service. Decoupling the segments within the architecture allows legacy code to be changed minimally to take part in a Jini system. Nevertheless, the full power of a Jini system will be available only to new services that are constructed using the integrated model.

A Jini system can be seen as a network extension of the infrastructure, programming model, and services that made Java technology successful in the single-machine case. These categories along with the corresponding components in the familiar Java application environment are shown in Figure AR.2.1:

	Infrastructure	**Programming Model**	**Services**
Base Java	Java VM RMI Java Security	Java APIs JavaBeans . . .	JNDI Enterprise Beans JTS . . .
Java + Jini	Discovery/Join Distributed Security Lookup	Leasing Transactions Events	Printing Transaction Manager JavaSpaces Service . . .

FIGURE AR.2.1: ***Jini Architecture Segmentation***

AR.2.2.1 Infrastructure

The Jini technology infrastructure defines the minimal Jini technology core. The infrastructure includes the following:

- A distributed security system, integrated into RMI, that extends the Java platform's security model to the world of distributed systems.
- The discovery and join protocols, service protocols that allow services (both hardware and software) to discover, become part of, and advertise supplied services to the other members of the federation.
- The lookup service, which serves as a repository of services. Entries in the lookup service are objects written in the Java programming language; these objects can be downloaded as part of a lookup operation and act as local proxies to the service that placed the code into the lookup service.

The discovery and join protocols define the way a service of any kind becomes part of a Jini system; RMI defines the base language within which the Jini services communicate; the distributed security model and its implementation define how entities are identified and how they get the rights to perform actions on their own behalf and on the behalf of others; and the lookup service reflects the current members of the federation and acts as the central marketplace for offering and finding services by members of the federation.

AR.2.2.2 Programming Model

The infrastructure both enables the programming model and makes use of it. Entries in the lookup service are leased, allowing the lookup service to reflect accurately the set of currently available services. When services join or leave a lookup service, events are signaled, and objects that have registered interest in such events get notifications when new services become available or old services cease to be active. The programming model rests on the ability to move code, which is supported by the base infrastructure.

Both the infrastructure and the services that use that infrastructure are computational entities that exist in the physical environment of the Jini system. However, services also constitute a set of interfaces which define communication protocols that can be used by the services and the infrastructure to communicate between themselves.

These interfaces, taken together, make up the distributed extension of the standard Java programming language model that constitutes the Jini programming

model. Among the interfaces that make up the Jini programming model are the following:

- The leasing interface, which defines a way of allocating and freeing resources using a renewable, duration-based model
- The event and notification interfaces, which are an extension of the event model used by JavaBeans components to the distributed environment, enable event-based communication between Jini services
- The transaction interfaces, which enable entities to cooperate in such a way that either all of the changes made to the group occur atomically or none of them occur

The lease interface extends the Java programming language model by adding time to the notion of holding a reference to a resource, enabling references to be reclaimed safely in the face of network failures.

The event and notification interfaces extend the standard event models used by JavaBeans components and the Java application environment to the distributed case, enabling events to be handled by third-party objects while making various delivery and timeliness guarantees. The model also recognizes that the delivery of a distributed notification may be delayed.

The transaction interfaces introduce a lightweight, object-oriented protocol enabling Jini applications to coordinate state changes. The transaction protocol provides two steps to coordinate the actions of a group of distributed objects. The first step is called the *voting phase,* in which each object "votes" whether it has completed its portion of the task and is ready to commit any changes it made. In the second step, a coordinator issues a "commit" request to each object.

The Jini Transaction protocol differs from most transaction interfaces in that it does not assume that the transactions occur in a transaction processing system. Such systems define mechanisms and programming requirements that guarantee the correct implementation of a particular transaction semantics. The Jini Transaction protocol takes a more traditional object-oriented view, leaving the correct implementation of the desired transaction semantics up to the implementor of the particular objects that are involved in the transaction. The goal of the transaction protocol is to define the interactions that such objects must have to coordinate such groups of operations.

The interfaces that define the Jini programming model are used by the infrastructure components where appropriate and by the initial Jini services. For example, the lookup service makes use of the leasing and event interfaces. Leasing ensures that services registered continue to be available, and events help administrators discover problems and devices that need configuration. The JavaSpaces

service, one example of a Jini service, utilizes leasing and events, and also supports the Jini Transaction protocol. The transaction manager can be used to coordinate the voting phase of a transaction for those objects that support transaction protocol.

The implementation of a service is not required to use the Jini programming model, but such services need to use that model for their interaction with the Jini technology infrastructure. For example, every service interacts with the Jini Lookup service by using the programming model; and whether a service offers resources on a leased basis or not, the service's registration with the lookup service will be leased and will need to be periodically renewed.

The binding of the programming model to the services and the infrastructure is what makes such a federation a Jini system not just a collection of services and protocols. The combination of infrastructure, service, and programming model, all designed to work together and constructed by using each other, simplifies the overall system and unifies it in a way that makes it easier to understand.

AR.2.2.3 Services

The Jini technology infrastructure and programming model are built to enable services to be offered and found in the network federation. These services make use of the infrastructure to make calls to each other, to discover each other, and to announce their presence to other services and users.

Services appear programmatically as objects written in the Java programming language, perhaps made up of other objects. A service has an interface that defines the operations that can be requested of that service. Some of these interfaces are intended to be used by programs, while others are intended to be run by the receiver so that the service can interact with a user. The type of the service determines the interfaces that make up that service and also define the set of methods that can be used to access the service. A single service may be implemented by using other services.

Example Jini services include the following:

- A printing service, which can print from Java applications and legacy applications
- A JavaSpaces service, which can be used for simple communication and for storage of related groups of objects written in the Java programming language
- A transaction manager, which enables groups of objects to participate in the Jini Transaction protocol defined by the programming model

service, one example of a Jini service, utilizes leasing and events, and also supports the Jini Transaction protocol. The transaction manager can be used to coordinate the voting phase of a transaction for those objects that support transaction protocol.

The implementation of a service is not required to use the Jini programming model, but such services need to use that model for their interaction with the Jini technology infrastructure. For example, every service interacts with the Jini Lookup service by using the programming model; and whether a service offers resources on a leased basis or not, the service's registration with the lookup service will be leased and will need to be periodically renewed.

The binding of the programming model to the services and the infrastructure is what makes such a federation a Jini system not just a collection of services and protocols. The combination of infrastructure, service, and programming model, all designed to work together and constructed by using each other, simplifies the overall system and unifies it in a way that makes it easier to understand.

AR.2.2.3 Services

The Jini technology infrastructure and programming model are built to enable services to be offered and found in the network federation. These services make use of the infrastructure to make calls to each other, to discover each other, and to announce their presence to other services and users.

Services appear programmatically as objects written in the Java programming language, perhaps made up of other objects. A service has an interface that defines the operations that can be requested of that service. Some of these interfaces are intended to be used by programs, while others are intended to be run by the receiver so that the service can interact with a user. The type of the service determines the interfaces that make up that service and also define the set of methods that can be used to access the service. A single service may be implemented by using other services.

Example Jini services include the following:

- A printing service, which can print from Java applications and legacy applications
- A JavaSpaces service, which can be used for simple communication and for storage of related groups of objects written in the Java programming language
- A transaction manager, which enables groups of objects to participate in the Jini Transaction protocol defined by the programming model

AR.2.3 Service Architecture

Services form the interactive basis for a Jini system, both at the programming and user interface levels. The details of the service architecture are best understood once the Jini Discovery and Jini Lookup protocols are presented.

AR.2.3.1 Discovery and Lookup Protocols

The heart of the Jini system is a trio of protocols called *discovery*, *join*, and *lookup*. A pair of these protocols—discovery and join—occur when a device is plugged in. Discovery occurs when a service is looking for a lookup service with which to register. Join occurs when a service has located a lookup service and wishes to join it. Lookup occurs when a client or user needs to locate and invoke a service described by its interface type (written in the Java programming language) and possibly other attributes. Figure AR.2.2 outlines the discovery process.

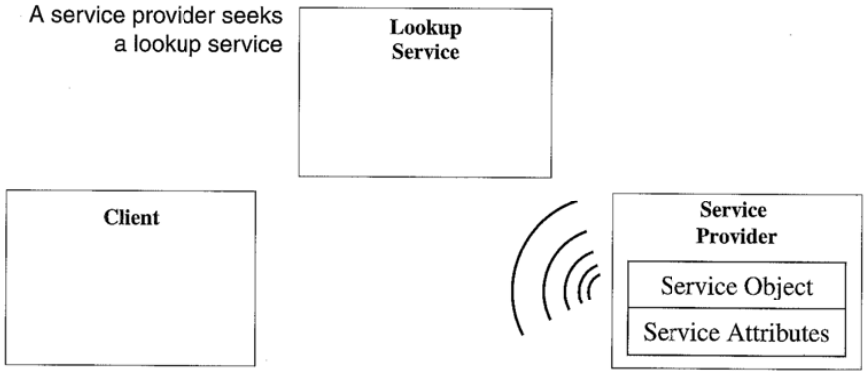

FIGURE AR.2.2: ***Discovery***

Jini Discovery/Join is the process of adding a service to a Jini system. A service provider is the originator of the service—a device or software, for example. First, the service provider locates a lookup service by multicasting a request on the local network for any lookup services to identify themselves (discovery, see Figure AR.2.2). Then, a service object for the service is loaded into the lookup service (join, see Figure AR.2.3). This service object contains the Java programming language interface for the service, including the methods that users and applications will invoke to execute the service along with any other descriptive attributes.

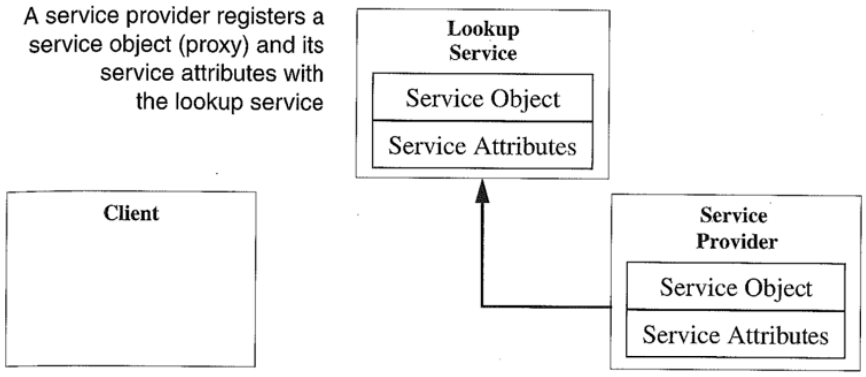

FIGURE AR.2.3: ***Join***

Services must be able to find a lookup service; however, a service may delegate the task of finding a lookup service to a third party. The service is now ready to be looked up and used, as shown in the following diagram (Figure AR.2.4).

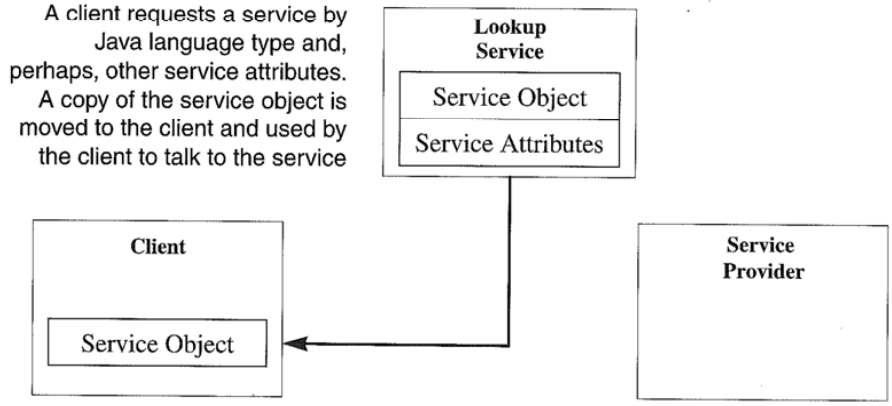

FIGURE AR.2.4: ***Lookup***

A client locates an appropriate service by its type—that is, by its interface written in the Java programming language—along with descriptive attributes that

are used in a user interface for the lookup service. The service object is loaded into the client.

The final stage is to invoke the service, as shown in the following diagram (Figure AR.2.5).

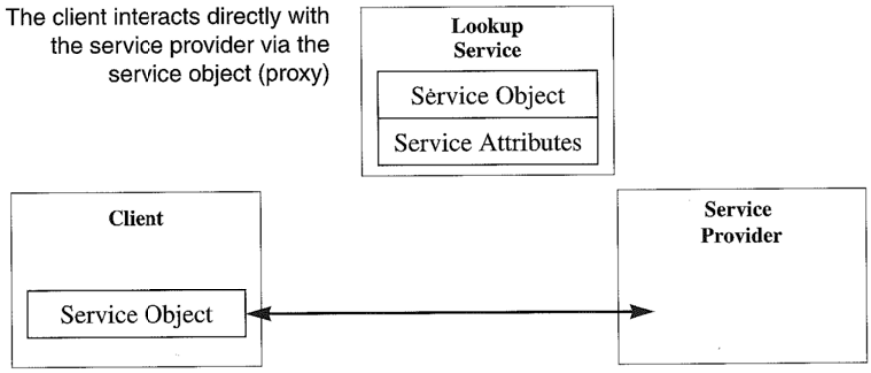

FIGURE AR.2.5: ***Client Uses Service***

The service object's methods may implement a private protocol between itself and the original service provider. Different implementations of the same service interface can use completely different interaction protocols.

The ability to move objects and code from the service provider to the lookup service and from there to the client of the service gives the service provider great freedom in the communication patterns between the service and its clients. This code movement also ensures that the service object held by the client and the service for which it is a proxy are always synchronized because the service object is supplied by the service itself. The client knows only that it is dealing with an implementation of an interface written in the Java programming language, so the code that implements the interface can do whatever is needed to provide the service. Because this code came originally from the service itself, the code can take advantage of implementation details of the service that are known only to the code.

The client interacts with a service via a set of interfaces written in the Java programming language. These interfaces define the set of methods that can be used to interact with the service. Programmatic interfaces are identified by the type system of the Java programming language, and services can be found in a lookup service by asking for those that support a particular interface. Finding a service this way ensures that the program looking for the service will know how to

use that service, because that use is defined by the set of methods that are defined by the type.

Programmatic interfaces may be implemented either as RMI references to the remote object that implements the service, as a local computation that provides all of the service locally, or as some combination. Such combinations, called *smart proxies*, implement some of the functions of a service locally and the remainder through remote calls to a centralized implementation of the service.

A user interface can also be stored in the lookup service as an attribute of a registered service. A user interface stored in the lookup service by a Jini service is an implementation that allows the service to be directly manipulated by a user of the system.

In effect, a user interface for a service is a specialized form of the service interface that enables a program, such as a browser, to step out of the way and let the human user interact directly with a service.

In situations in which no lookup service can be found, a client could use a technique called *peer lookup* instead. In such situations, the client can send out the same identification packet that is used by a lookup service to request service providers to register. Service providers will then attempt to register with the client as though it were a lookup service. The client can select the services it needs from the registration requests it receives in response and drop or refuse the rest.

AR.2.3.2 Service Implementation

Objects that implement a service may be designed to run in a single address space with other, helper, objects especially when there are certain location or security-based requirements. Such objects make up an *object group*. An object group is guaranteed to always reside in a single address space or virtual machine when those objects are running. Objects that are not in the same object group are isolated from each other, typically by running them in a different virtual machine or address space.

A service may be implemented directly or indirectly by specialized hardware. Such devices can be contacted by the code associated with the interface for the service.

From the service client's point of view, there is no distinction between services that are implemented by objects on a different machine, services that are downloaded into the local address space, and services that are implemented in hardware. All of these services will appear to be available on the network, will appear to be objects written in the Java programming language, and, only as far as correct functioning is concerned, one kind of implementation could be replaced

by another kind of implementation without change or knowledge by the client. (Note that security permissions must be properly granted.)

AR.3 An Example

THIS example shows how a Jini printing service might be used by a digital camera to print a high-resolution color image. It will start with the printer joining an existing Jini system, continue with its being configured, and end with printing the image.

AR.3.1 Registering the Printer Service

A printer that is either freshly connected to a Jini system or is powered up once it has been connected to a Jini system grouping needs to discover the appropriate lookup service and register with it. This is the *discovery* and *join* phase.

AR.3.1.1 Discovering the Lookup Service

The basic operations of discovering the lookup service are implemented by a Jini software class. An instance of this class acts as a mediator between devices and services on one hand and the lookup service on the other. In this example the printer first registers itself with a local instance of this class. This instance then multicasts a request on the local network for any lookup services to identify themselves. The instance listens for replies and, if there are any, passes to the printer an array of objects that are proxies for the discovered lookup services.

AR.3.1.2 Joining the Lookup Service

To register itself with the lookup service, the printer needs first to create a service object of the correct type for printing services. This object provides the methods that users and applications will invoke to print documents. Also needed is an array of `LookupEntry` instances to specify the attributes that describe the printer, such as that it can print in color or black and white, what document formats it can print, possible paper sizes, and printing resolution.

The printer then calls the `register` method of the lookup service object that it received during the discovery phase, passing it the printer service object and the array of attributes. The printing service is now registered with the lookup service.

AR.3.1.3 Optional Configuration

At this point the printing service can be used, but the local system administrator might want to add additional information about the printer in the form of additional attributes, such as a local name for the service, information about its physical location, and a list of who may access the service. The system administrator might also want to register with the device to receive notifications for any errors that arise, such as when the printer is out of paper.

One way the system administrator could do this would be to use a special utility program to pass this additional information to the service. In fact this program might have received notification from the lookup service that a new service was being added and then alerted the system administrator.

AR.3.1.4 Staying Alive

When the printer registers with the Jini Lookup service it receives a *lease*. Periodically, the printer will need to renew this lease with the lookup service. If the printer fails to renew the lease, then when the lease expires, the lookup service will remove the entry for it, and the printer service will no longer be available.

AR.3.2 Printing

Some services provide a user interface for interaction with them; others rely on an application to mediate such interaction. This example assumes that a person has a digital camera that has taken a picture they want to print on a high-resolution printer. The first thing that the camera needs to do after it is connected to the network is locate a Jini printing service. Once a printing service has been located and selected, the camera can invoke methods to print the image.

AR.3.2.1 Locate the Lookup Service

Before the camera can use a Jini service, it must first locate the Jini Lookup service, just as the print service needed to do to register itself. The camera registers

itself with a local instance of the Jini software class `LookupDiscovery`, which will notify the camera of all discovered lookup services.

AR.3.2.2 Search for Printing Services

Finding an appropriate service requires passing a template that is used to match and filter the set of existing services. The template specifies both the type of the required service, which is the first filter on possible services, and a set of attributes which is used to reduce the number of matching services if there are several of the right type. In this example, the camera supplies a template specifying the printer type and an array of attribute objects. The type of each object specifies the attribute type, and its fields specify values to be matched. For each attribute, fields that should be matched, such as color printing, are filled in; ones that don't matter are left null. The Jini Lookup service is passed this template and returns an array of all of the printing services that match it. If there are several matching services, the camera may further filter them—in this case perhaps to ensure high print resolution—and present the user with the list of possible printers for choice. The final result is a single service object for the printing service.

At this point the printing service has been selected, and the camera and the printer service communicate directly with each other; the lookup service is no longer involved.

AR.3.2.3 Configuring the Printer

Before printing the image, the user might wish to configure the printer. This might be done directly by the camera invoking the service object's `configure` method; this method may display a dialog box on the camera's display with which the user may specify printer settings. When the image is printed, the service object sends the configuration information to the printer service.

AR.3.2.4 Requesting That the Image Be Printed

To print the image, the camera calls the print method of the service object, passing it the image as an argument. The service object performs any necessary preprocessing and sends the image to the printer service to be printed.

AR.3.2.5 Registering for Notification

If the user wishes to be notified when the image has been printed, the camera needs to register itself with the printer service using the service object. The camera might also wish to register to be notified if the printer encounters any errors.

AR.3.2.6 Receiving Notification

When the printer has finished printing the image or encounters an error, it signals an event to the camera. When the camera receives the event, it may notify the user that the image has been printed or that an error has occurred.

AR.4 For More Information

THIS document does not provide a full specification of Jini technology. Each of the Jini technology components is specified in a companion document. In particular, the reader is directed to the following documents:

- *The Java Remote Method Invocation Specification*
- *The Java Object Serialization Specification*
- *The Jini Discovery and Join Specification*
- *The Jini Device Architecture Specification*
- *The Jini Distributed Events Specification*
- *The Jini Distributed Leasing Specification*
- *The Jini Lookup Service Specification*
- *The Jini Lookup Attribute Schema Specification*
- *The Jini Entry Specification*
- *The Jini Transaction Specification*

THE JINI DISCOVERY AND JOIN SPECIFICATION defines how a service should behave when it first starts up to find the local lookup services with which it should register, and how lookups should advertise their availability. The discovery protocol lets a service find "discoverable" lookup services. A service may also be configured to register with specific lookup services or to register only with particular lookup services. Most services will use discovery, since most will want to be available to local clients. Clients will use discovery to find local services, but use explicit denotation to contact specific lookups that are useful even if they are far away.

This discovery protocol is designed for discovery on IP networks. IP networks are widespread and so was the first discovery protocol designed. Other networks will require different discovery protocols that will be designed for their distinct characteristics.

DJ

The Jini Discovery and Join Specification

DJ.1 Introduction

ENTITIES that wish to start participating in a distributed Jini system, 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 (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 technology 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.

- 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 Jini software
- 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.

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 DJ.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.
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.

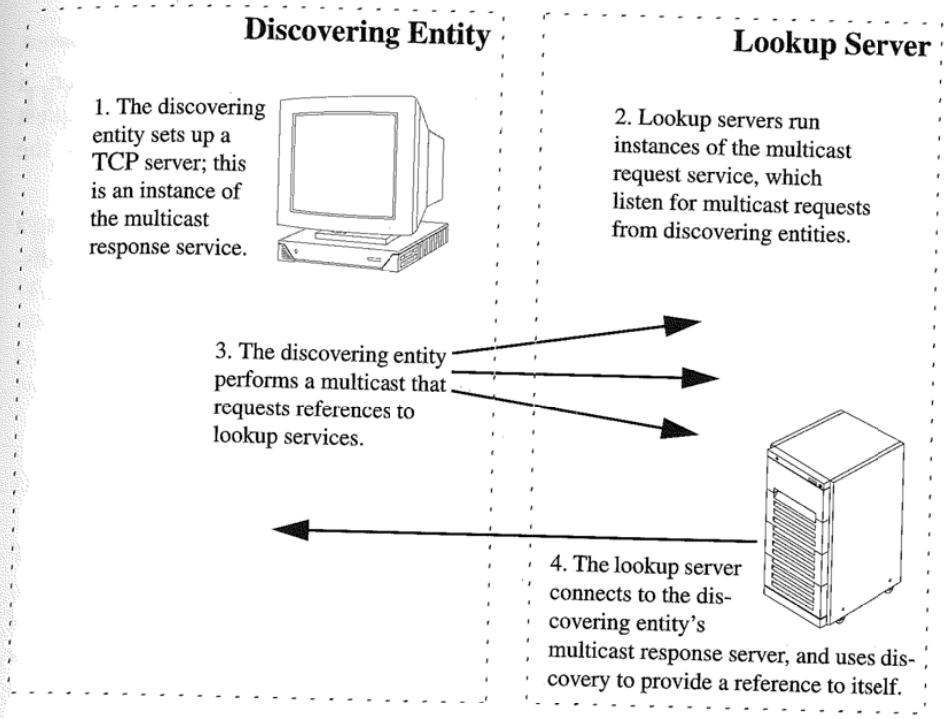

FIGURE DJ.1.1: ***The Multicast Request Protocol***

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.1.5 Dependencies

This document relies on the following other specifications:

- *Java Remote Method Invocation Specification*
- *Jini Lookup Service Specification*

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 DJ.2.1.

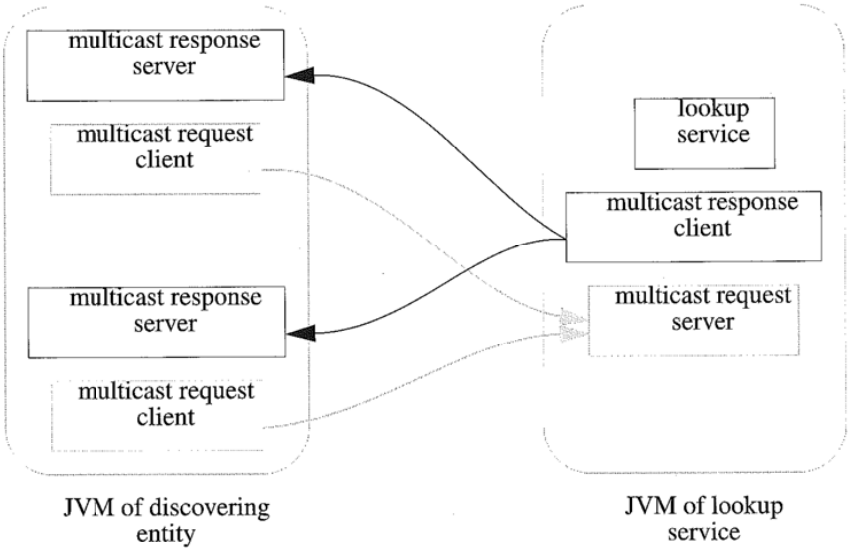

FIGURE DJ.2.1: ***Multicast Request Protocol Participants***

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 DJ.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.

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 instance of this field 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 Jini Discovery unicast 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 DJ.2.2 illustrates the flow when unicast discovery is initiated by a discovering entity.

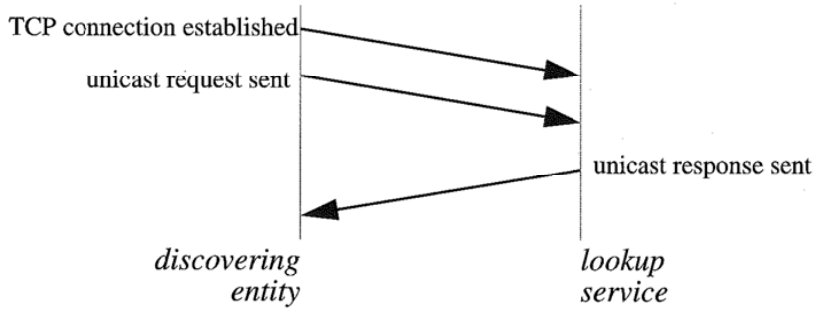

FIGURE DJ.2.2: ***Unicast Discovery Initiated by a Discovering Entity***

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

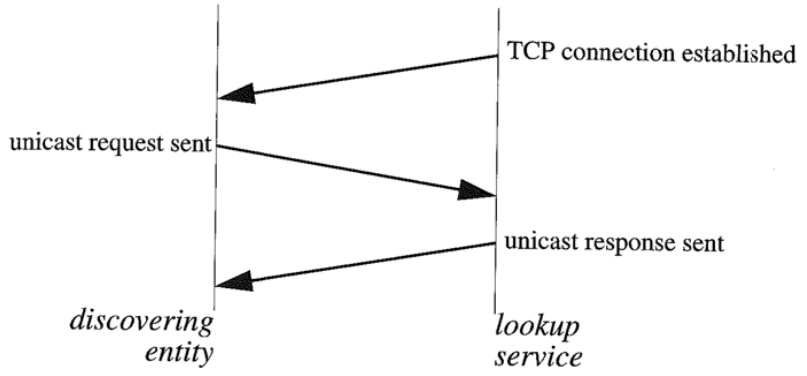

FIGURE DJ.2.3: ***Unicast Discovery Initiated by a Lookup Service***

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.[1]

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
```

[1] 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

THE JINI DISCOVERY UTILITIES SPECIFICATION describes a set of utility classes and interfaces that will help users discover lookup services. They implement mechanisms that drive the discovery protocols and that invoke your code at relevant moments, turning the network protocol into useful Java language abstractions.

DU

The Jini Discovery Utilities Specification

DU.1 Introduction

EACH individual party in a Java Virtual Machine (JVM) on a given host is independently responsible for obtaining references to lookup services. In this specification we first coves utility classes that such parties can use to simplify multicast discovery tasks. We then present lower-level utility classes that are useful in building these kinds of utilities.

DU.1.1 Dependencies

This specification relies on the following other specifications:

- *Java Object Serialization Specification*
- *Jini Lookup Service Specification*
- *Jini Discovery and Join Specification*

DU.2 Multicast Discovery Utility

PARTIES can obtain references to lookup services via the multicast discovery protocols by making use of the LookupDiscovery class.

```
package net.jini.discovery;

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

public final class LookupDiscovery {
    public static final String[] ALL_GROUPS = null;
    public static final String[] NO_GROUPS = new String[0];

    public LookupDiscovery(String[] groups)
        throws IOException {…}
    public void addDiscoveryListener(DiscoveryListener l) {…}
    public void removeDiscoveryListener(DiscoveryListener l)
        {…}
    public void discard(ServiceRegistrar reg) {…}
    public String[] getGroups() {…}
    public void setGroups(String[] groups)
        throws IOException {…}
    public void addGroups(String[] groups)
        throws IOException {…}
    public void removeGroups(String[] groups) {…}
    public void terminate() {…}
}
```

The LookupDiscovery class relies upon the DiscoveryEvent class:

```
package net.jini.discovery;

import net.jini.core.lookup.ServiceRegistrar;
import java.util.EventListener;
```

```
import java.util.EventObject;

public class DiscoveryEvent extends EventObject {
    public DiscoveryEvent(Object source,
                          ServiceRegistrar[] regs) {…}
    public ServiceRegistrar[] getRegistrars() {…}
}
```

The `LookupDiscovery` class also relies upon the `DiscoveryListener` interface:

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

These classes and interfaces hide the details of the underlying protocol implementation, but provide enough information to the programmer to be flexible and useful.

DU.2.1 The `LookupDiscovery` Class

The `net.jini.discovery.LookupDiscovery` class encapsulates the operation of the multicast discovery protocols, including the automatic switch from use of the multicast request protocol to the multicast announcement protocol. Each instance of the `LookupDiscovery` class must behave as if it operated independently of all other instances. The semantics of the methods on this class are:

- The constructor takes a set of groups in which the caller is interested as parameter. This set is represented as an array, none of whose elements may be `null`. The empty set is represented by an empty array, and no set (indicating that all lookup services should be discovered) is indicated by a `null` reference. The constructor may throw a `java.io.IOException` if a problem occurs in starting discovery.
- The `addDiscoveryListener` method adds a listener to the set of objects listening for discovery events. Once a listener is registered, it is notified of all lookup services that have been discovered to date, and is then notified as new lookup services are discovered or existing lookup services are discarded.
- The `removeDiscoveryListener` method removes a listener from the set of objects that are listening for discovery events.

- The `discard` method removes a particular lookup service from the set that is considered to already have been discovered. This allows the lookup service to be discovered again; it is intended as a mechanism for programmers to remove stale entries from the set so that they do not have to keep trying to contact lookup services that no longer exist.
- The `getGroups` method returns the set of groups that this `LookupDiscovery` object is attempting to discover. If the set is empty, this method returns the empty array, and if there is no set, it returns the `null` reference.
- The `terminate` method ends discovery. After this method has been called, no new lookup services will be discovered.

Discovery usually starts as soon as an instance of this class is created and ends either when the instance is finalized prior to garbage collection, or when the `terminate` method is called. However, if the empty set is passed to the constructor, discovery will not be started until the `setGroups` method is called with either no set or a non-empty set.

DU.2.2 Useful Constants

The `ALL_GROUPS` constant can be passed to the `LookupDiscovery` constructor and to the `setGroups` method to indicate that all lookup services within range should be discovered. The `NO_GROUPS` constant indicates that no groups should be discovered (implying that discovery should be postponed until another call to `setGroups`).

If the `getGroups` method returns the empty array, that array is guaranteed to be referentially equal to the `NO_GROUPS` constant (that is, it can be tested for equality using the == operator).

DU.2.3 Changing the Set of Groups to Discover

Programmers may modify the set of groups to be discovered on the fly, using the methods described below. In each case, a set of groups is represented as an array of strings, none of whose elements may be `null`. The empty set is denoted by the empty array, and no set (indicating that all lookup services should be discovered) is indicated by `null`. Duplicated group names are ignored.

- The `setGroups` method changes the set of groups to be discovered to the given set (or to no set, if indicated).

- The addGroups method augments the set of groups to be discovered. This method throws a java.lang.UnsupportedOperationException if there is no set to be augmented.
- The removeGroups method removes members from the set of groups to be discovered. No exception is thrown if an attempt is made to remove a group that is not currently in the set to be discovered. This method throws a java.lang.UnsupportedOperationException if there is no set to remove members from.

When groups are removed from the set to be discovered, any already discovered lookup services that are no longer members of any of the groups to be discovered are removed from the set maintained by the particular LookupDiscovery object in use, and all listeners are notified that they have been discarded.

If groups are added to the set to be discovered, the multicast request protocol is used to discover lookup services for those groups. If there are no responses to multicast requests, the LookupDiscovery object switches over to listening for multicast announcements for those groups.

Since calling either the setGroups or addGroups method may result in the multicast request protocol being started afresh, either method may throw a java.io.IOException if a problem occurs in starting the protocol.

If any of the setGroups, addGroups, or removeGroups methods is called after the terminate method has been called, the invocation will throw a java.lang.IllegalStateException.

DU.2.4 The DiscoveryEvent Class

The net.jini.discovery.DiscoveryEvent class encapsulates the information made available by the multicast discovery protocols. The sole new method of the DiscoveryEvent class is getRegistrars, which returns an array of lookup service registrars. The getSource method returns the LookupDiscovery object that originated the given event.

DU.2.5 The DiscoveryListener Interface

Objects that wish to register for notifications of multicast discovery events must implement the net.jini.discovery.DiscoveryListener interface. Its discovered method is called whenever new lookup services are discovered, with an event containing a set of discovered lookup services represented as an array.

The discarded method is called whenever previously discovered lookup services have been discarded by the originating LookupDiscovery object; the event contains a set of discarded lookup services represented as an array. An event is delivered to listeners whenever the discard method is called on a LookupDiscovery object, and also if a call to either its removeGroups or setGroups method results in lookup services being discarded.

DU.2.6 Security and Multicast Discovery

When a LookupDiscovery object is created, the creator must have permission either to attempt discovery of each group specified in the set to discover, or to attempt discovery of all groups if the set is null. This is also true for the addGroups and setGroups methods on the LookupDiscovery class. If appropriate permissions have not been granted, the constructor and these methods will throw a java.lang.SecurityException.

Discovery permissions are controlled in security policy files using the net.jini.discovery.DiscoveryPermission permission.

```
package net.jini.discovery;

import java.security.Permission;
import java.io.Serializable;

public final class DiscoveryPermission extends Permission
    implements Serializable
{
    public DiscoveryPermission(String group) {…}
    public DiscoveryPermission(String group, String actions)
        {…}
}
```

The actions parameter is ignored. The following examples illustrate the use of this permission:

```
permission net.jini.discovery.DiscoveryPermission "*";
```
All groups

```
permission net.jini.discovery.DiscoveryPermission "";
```
Only the "public" group

```
permission net.jini.discovery.DiscoveryPermission "foo";
```
The group "foo"

```
permission net.jini.discovery.DiscoveryPermission "*.sun.com";
```
Groups ending in ".sun.com"

Each declaration grants permission to attempt discovery of one name. A name does not necessarily correspond to a single group:

- The name `*` grants permission to attempt discovery of *all* groups.
- A name beginning with `*.` grants permission to attempt discovery of all groups that match the *remainder* of that name; for example, the name `"*.example.org"` would match a group named `"foonly.example.org"` and also a group named `"sf.ca.example.org"`.
- The empty name `""` denotes the *public* group.
- All other names are treated as individual groups and must match exactly.

A restriction of the Java Development Kit (JDK) 1.2 security model requires that appropriate `net.jini.discovery.DiscoveryPermission` be granted to the Jini software codebase itself, in addition to any codebases that may use Jini software classes.

DU.2.7 Serialized Forms

Class	serialVersionUID	Serialized Fields
`DiscoveryEvent`	5280303374696501479L	`ServiceRegistrar[] regs`
`DiscoveryPermission`	–3036978025008149170L	*none*

DU.3 Protocol Utilities

THE utilities we will now present are intended for use by implementors of multicast discovery utilities, and for others who might need to exercise more control over their usage of the Jini Discovery protocols.

DU.3.1 Marshalling Multicast Requests

The `OutgoingMulticastRequest` class provides facilities for marshalling multicast discovery requests into a form suitable for transmission over a network. This class is useful for programmers who are implementing the component of one of the discovery protocols that sits on a device that wishes to join a djinn.

```
package net.jini.discovery;

import net.jini.core.lookup.ServiceID;
import java.io.IOException;
import java.net.DatagramPacket;

public class OutgoingMulticastRequest {
    public static DatagramPacket[]
        marshal(int port, String[] groups, ServiceID[] heard)
            throws IOException {…}
}
```

This class cannot be instantiated, and its sole method, `marshal`, is static. This method takes as parameter the port of the multicast response service to advertise, along with a set of groups to look for and a set of service IDs from which this system has already heard. The latter two arguments are represented as arrays. No parameter may be `null`, and the arrays must have no members that are `null`, and none should be duplicated (implementations are not required to check for duplicated members).

This method returns an array of `DatagramPacket` objects; this array contains at least one member, and will contain more if the request is not small enough to fit

in a single packet. Each such object has been fully initialized; it contains a multicast request as payload and is ready to send over the network.

In the event of error, this method may throw a `java.io.IOException` if marshalling fails. In some instances the exception thrown may be a more specific subclass of this exception.

DU.3.2 Unmarshalling Multicast Requests

The `IncomingMulticastRequest` class provides facilities for unmarshalling multicast discovery requests into a form in which the individual parameters of the request may be easily accessed. This class is useful for programmers who are implementing the component of one of the discovery protocols that works with a lookup service implementation within a djinn.

```
package net.jini.discovery;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import net.jini.core.lookup.ServiceID;

public class IncomingMulticastRequest {
    public IncomingMulticastRequest(DatagramPacket dgram)
        throws IOException {…}
    public InetAddress getAddress() {…}
    public int getPort() {…}
    public String[] getGroups() {…}
    public ServiceID[] getServiceIDs() {…}
}
```

This class may be instantiated using a `java.net.DatagramPacket`. The payload of the `DatagramPacket` is assumed to contain nothing but the marshalled discovery request. If the marshalled request is corrupt, a `java.io.IOException` or a `java.lang.ClassNotFoundException` will be thrown. In some such instances a more specific subclass of either exception may be thrown that will give more detailed information.

The methods of this class are mostly self-explanatory.

- The `getAddress` method returns the IP address of the host to which the caller should respond.

- The `getPort` method returns the TCP port number on that host to which the caller should connect.
- The `getGroups` method returns the groups in which the originator of this request is interested. The array returned by this method may be of zero length; none of its fields will be `null`; and items may or may not be duplicated.
- The `getServiceIDs` method returns the set of service IDs of lookup services from which the originator has already heard. The array returned by this method may have length equal to zero, but none of its fields will be `null`, and items may or may not be duplicated.
- The `equals` method returns true if both instances have the same address, port, groups, and service IDs.

DU.3.3 Marshalling Multicast Announcements

The `OutgoingMulticastAnnouncement` class encapsulates details of announcing a lookup service.

```
package net.jini.discovery;

import java.io.IOException;
import java.net.DatagramPacket;
import net.jini.core.lookup.ServiceID;
import net.jini.core.discovery.LookupLocator;

public class OutgoingMulticastAnnouncement {
    public static DatagramPacket[]
        marshal(ServiceID id, LookupLocator loc,
                String[] groups)
        throws IOException {…}
}
```

The sole method of this class, `marshal`, is static. It takes as parameters the service ID of the lookup service being advertised, the locator via which unicast discovery of that lookup service may be performed, and the names of the groups of which that service is a member. If a problem occurs with marshalling the request, a `java.net.IOException` will be thrown.

This method returns an array of `DatagramPacket` objects, each of which has been initialized such that it is ready to be multicast.

DU.3.4 Unmarshalling Multicast Announcements

The `IncomingMulticastAnnouncement` class permits access to the fields of a multicast announcement datagram that has been received.

```
package net.jini.discovery;

import java.io.IOException;
import java.net.DatagramPacket;
import net.jini.core.lookup.ServiceID;
import net.jini.core.discovery.LookupLocator;

public class IncomingMulticastAnnouncement {
    public IncomingMulticastAnnouncement(DatagramPacket p)
        throws IOException {…}
    public ServiceID getServiceID() {…}
    public LookupLocator getLocator() {…}
    public String[] getGroups() {…}
}
```

The constructor takes a datagram packet as argument. If it cannot decode the contents of the datagram packet, it throws a `java.lang.ClassNotFoundException` or a `java.io.IOException`. The `getServiceID` method returns the service ID of the originator. The `getLocator` method returns the locator via which unicast discovery of the originator may be performed. The `getGroups` method returns the groups represented by the originator; the array returned by this method may be `null`, will not be empty, and will contain no `null` elements. Elements may or may not be duplicated. The `equals` method returns true if both instances have the same service ID.

DU.3.5 Easy Access to Constants

The `Constants` class provides easy access to some constants used during the lookup discovery process.

```
package net.jini.discovery;

import java.net.InetAddress;
import java.net.UnknownHostException;

public class Constants {
```

```
    public static final short discoveryPort = 4160;
    public static final InetAddress getRequestAddress()
        throws UnknownHostException {…}
    public static final InetAddress getAnnouncementAddress()
        throws UnknownHostException {…}
}
```

The value of the `discoveryPort` variable is the UDP port number over which the multicast request and announcement protocols operate, and also the TCP port number over which the unicast discovery protocol operates by default.

The `getRequestAddress` and `getAnnouncementAddress` methods return the addresses of the multicast groups over which multicast request and multicast announcement take place, respectively. These methods may throw a `java.net.UnknownHostException` if called in a circumstance under which multicast address resolution is not permitted.

DU.3.6 Marshalling Unicast Discovery Requests

The `OutgoingUnicastRequest` class provides facilities for marshalling unicast discovery requests into a form suitable for transmission over a network.

```
package net.jini.discovery;

import java.io.IOException;
import java.io.OutputStream;

public class OutgoingUnicastRequest {
    public static void marshal(OutputStream str)
        throws IOException {…}
}
```

This class cannot be instantiated, and its only public method is static.

DU.3.7 Unmarshalling Unicast Discovery Requests

The `IncomingUnicastRequest` class provides facilities for unmarshalling unicast discovery requests.

```
package net.jini.discovery;

import java.io.InputStream;
```

```
import java.io.IOException;

public class IncomingUnicastRequest {
    public IncomingUnicastRequest(InputStream str)
        throws IOException {…}
}
```

Since, under the current version of the unicast discovery protocol, no useful information is transmitted in a request, this class has no public methods.

DU.3.8 Marshalling Unicast Discovery Responses

The `OutgoingUnicastResponse` class provides marshalling facilities for unicast discovery responses.

```
package net.jini.discovery;

import java.io.IOException;
import java.io.OutputStream;
import net.jini.core.lookup.ServiceRegistrar;

public class OutgoingUnicastResponse {
    public static void marshal(OutputStream s,
                               ServiceRegistrar reg
                               String[] groups)
        throws IOException {…}
}
```

This class may not be instantiated. The sole static method, `marshal`, writes the given registrar proxy to the given output stream, and indicates that it is a member of the given set of groups (which is represented as an array which should have no `null` members, but may contain duplicates). If a problem occurs during marshalling or writing, it throws a `java.io.IOException`.

DU.3.9 Unmarshalling Unicast Discovery Responses

The `IncomingUnicastResponse` class allows a caller to unmarshal a unicast discovery response.

```
package net.jini.discovery;

import java.io.IOException;
import java.io.InputStream;
import net.jini.core.lookup.ServiceRegistrar;

public class IncomingUnicastResponse {
    public IncomingUnicastResponse(InputStream s)
        throws IOException, ClassNotFoundException {…}
    public ServiceRegistrar getRegistrar() {…}
    public String[] getGroups() {…}
}
```

The constructor unmarshals a response from an input stream, and throws an exception if the reading or the unmarshalling fails. The `getRegistrar` method returns the unmarshalled registrar proxy. The `getGroups` method returns the set of groups of which the given lookup service is a member. This set is represented as an array of strings, with no `null` members (duplicate members may appear, however). The `equals` method returns true if both instances have the same registrar.

THE JINI ENTRY SPECIFICATION defines the notion of an entry, which is a typed collection of objects that can be stored and matched against with simple, exact-match rules. As you will see, the lookup service uses entries as attributes, so the matching rules for entries are the rules for matching a single lookup attribute.

EN

The Jini Entry Specification

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 1.2 Java Development Kit (JDK) software, 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.

EN.1.6 Serialized Form

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

THE JINI ENTRY UTILITIES SPECIFICATION defines exactly one utility: the `AbstractEntry` *class, which is a useful—but not required—superclass for* `Entry` *classes. This class uses the standard properties for* `Entry` *classes to provide default implementations of common methods, such as* `equals` *and* `hashCode`.

EU

The Jini Entry Utilities Specification

EU.1 Entry Utilities

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. The details of entries and their semantics are discussed in the *Jini Entry Specification.*

When designing entries, certain tasks are commonly done in similar ways. This specification defines a utility class for such common tasks.

EU.1.1 `AbstractEntry`

The class `net.jini.entry.AbstractEntry` is a specific implementation of `Entry` that provides useful implementations of `equals`, `hashCode`, and `toString`:

```
package net.jini.entry;

public abstract class AbstractEntry implements Entry {
    public boolean equals(Object o) {…}
    public int hashCode() {…}
    public String toString() {…}
    public static boolean equals(Entry e1, Entry e2) {…}
    public static int hashCode(Entry entry) {…}
    public static String toString(Entry entry) {…}
}
```

The static method `AbstractEntry.equals` returns `true` if and only if the two entries are of the same class and for each field *F*, the two objects' values for *F* are either both `null` or the invocation of `equals` on one object's value for *F* with the other object's value for *F* as its parameter returns `true`. The static method `hashCode` returns zero XOR the `hashCode` invoked on each non-`null` field of the entry. The static method `toString` returns a string that contains each field's name and value. The non-static methods `equals`, `hashCode`, and `toString` return a result equivalent to invoking the corresponding static method with `this` as the first argument.

EU.1.2 Serialized Form

Class	serialVersionUID	Serialized Fields
AbstractEntry	5071868345060424804L	*none*

Entry Utilities (EU)

THE JINI DISTRIBUTED LEASING SPECIFICATION defines the leasing programming model used throughout the Jini architecture to prevent the leakage of resources. Creating a lease is a one-bid negotiation in which the grantor of the lease decides the final answer. Leases allow a grantor of a resource to give an upper bound on how long it is willing to hold onto resources that may have no interested users. As you will see, the lookup service uses leases to ensure the timeliness of each registered service.

LE

The Jini Distributed Leasing Specification

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

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 document are:

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

LE.1.3 Dependencies

This document relies on the following specifications:

- *Java Remote Method Invocation Specification*

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. 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;
```