Jini™ Discovery and Join Specification

The Jini™ technology is a Java™ platform-centric distributed system designed around the goals of simplicity, flexibility, and federation. The Jini Discovery protocols are used by entities that wish to start participating in a Jini system. This document specifies the interactions between an entity that wishes to perform discovery and entities that are already participating in a Jini system.

Sun microsystems

THE NETWORK IS THE COMPUTER™

901 San Antonio Road
Palo Alto, CA 94303 USA
415 960-1300
fax 415 969-9131

Revision 1.0
January 25, 1999

Contents

Introduction 1

1.1 Overview

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 in order for entities to start communicating usefully with services in a djinn; these steps are described by the *join* protocol.

1.2 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 to 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 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.

1.3 Host Requirements

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

- A functioning JVM, with access to all packages needed in order 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 may apply differently to other networking protocols.

1.3.1 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 (e.g. a simple HTTP server) that facilitates the downloading of Remote Method Invocation (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.

1.4 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 may 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.

1.5 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 Chapter 2.

1.5.1 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 which 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.

1.5.2 The Multicast Request Protocol

The multicast request protocol 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 chapter 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 `net.jini.core.lookup.ServiceRegistrar` implementation.

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

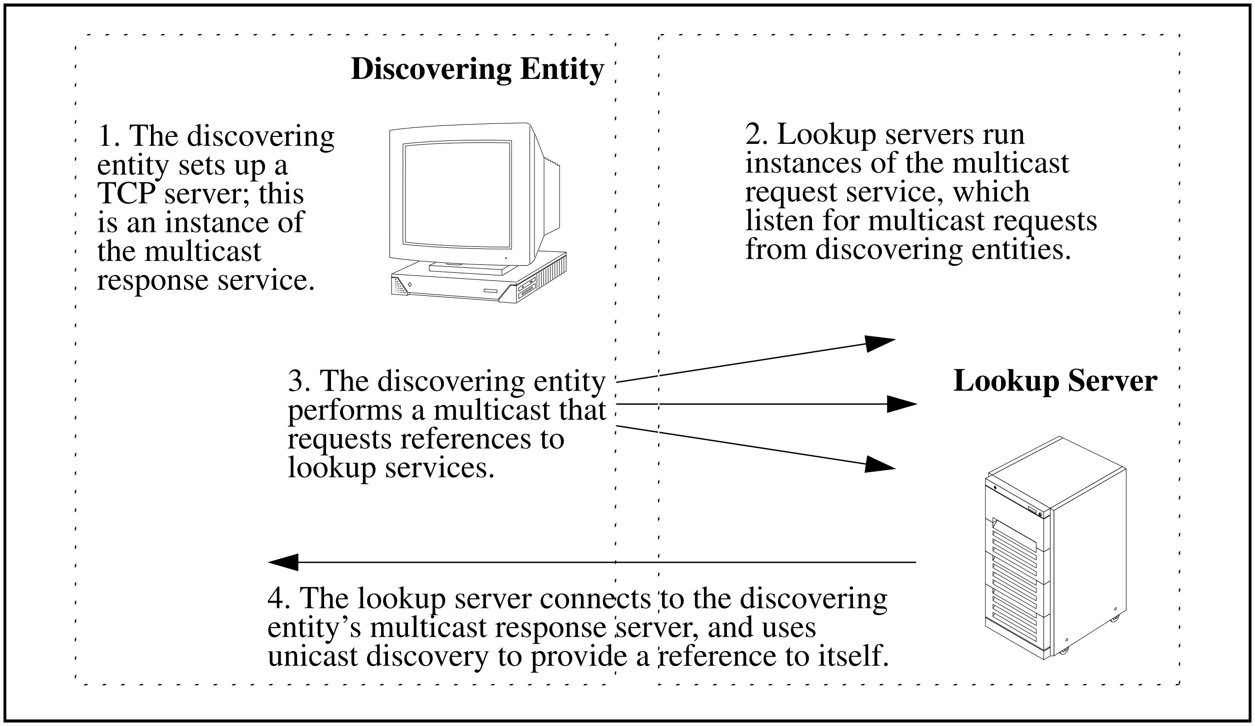


Figure 1-1 The Multicast request protocol

1.5.3 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 multicast announcements of their existence at regular intervals.

1.5.4 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.

1.6 Dependencies

This document relies on the following other specifications:

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

1.7 Comments

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

The Discovery Protocols 2

This chapter describes 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).

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.

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.

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.

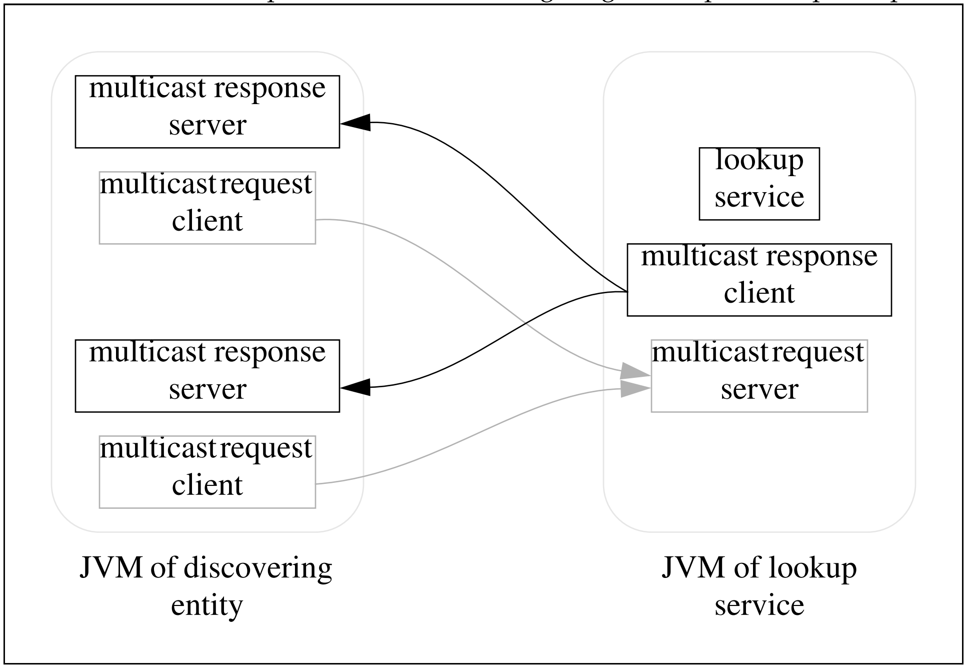
These components are paired; they do not occur separately. Any number of pairs of such components may coexist in a single JVM at any given time.

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 following diagram of protocol participants.



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.

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` writing into a `java.io.ByteArrayOutputStream`. The contents of the packet, in order of appearance, are illustrated by the following fragment of pseudocode which generates the appropriate byte array:

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

```
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]);
}
```

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

To elaborate on the roles of the variables above:

- The `protoVersion` variable contains an integer which indicates the version of the discovery protocol, in order to 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 (organized as an array) of strings 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 (organized as an array) of `net.jini.core.lookup.ServiceID` objects that identify lookup services from which this entity has already heard, and which do not need to respond to this request.
- The `packetBody` variable contains the marshaled 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.

Table 2-1 Fields in 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

In the event that 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.

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, and 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.

2.3 Discovery Using the Multicast Request Protocol

In this section, the discovery sequence is described for local area network (LAN)-based environments that use the multicast request protocol to discover one or more djinns.

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.

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.

2.3.3 Handling Responses from Multiple Djinns

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

If the entity provides a *finder*-style visual interface that allows a user to choose one or more djinns for their system to join, it should loop at step 4 in section 2.3 above, 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 that were selected by the user

On the other hand, if the behavior of the entity is fully automated, it should follow the join protocol described in chapter 3.

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.

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:

Table 2-2 Fields of a multicast announcement packet

count	serialized type	description
1	`int`	protocol version
1	`java.lang.String`	host for unicast discovery
1	`int`	port to connect to

Table 2-2 Fields of a multicast announcement packet

count	serialized type	description
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 `java.io.DataOutput.writeInt` method.
- 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 `java.io.DataOutput.writeUTF` method.
- 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 `java.io.DataOutput.writeInt` method.
- 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 `net.jini.core.lookup.ServiceID.writeBytes` method.
- 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 `java.io.DataOutput.writeInt` method.
- 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 `java.io.DataOutput.writeUTF` method.

In the case where 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.

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 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.

2.5 Unicast Discovery

While workgroup-level devices only need to be able to discover local djinns, a user may need to be able to access services in djinns that may be dispersed more widely (e.g. 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.

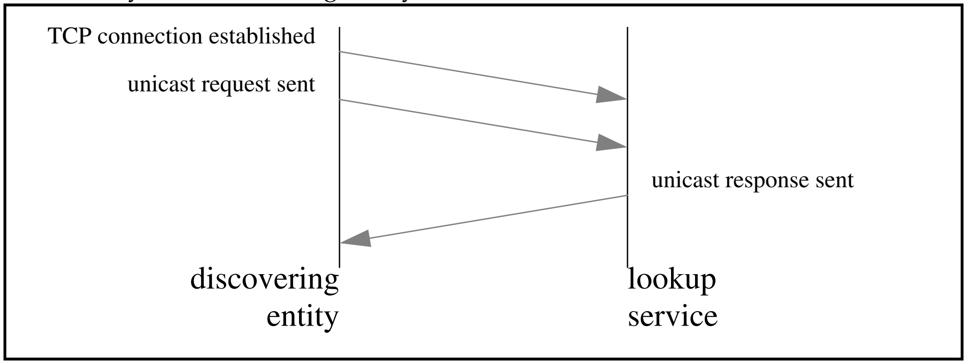
2.5.1 The Protocol

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

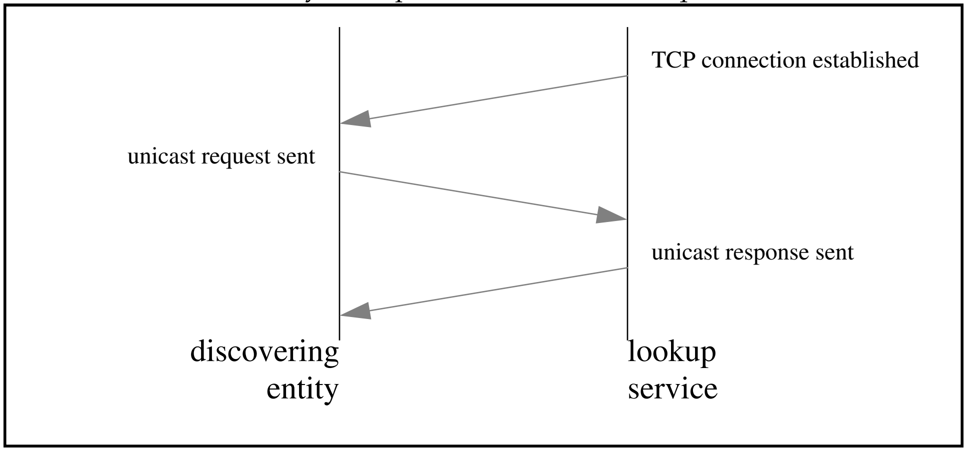
In the case where an entity wishes to obtain a reference to a given djinn, the entity has a lookup locator object for that djinn, and makes a TCP connection to the host and port specified by that lookup locator. It sends a unicast discovery request (see below), to which the remote host responds.

In the case where a lookup service is responding to a multicast request, the request it is responding to 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 below illustrates the flow when unicast discovery is initiated by the discovering entity.



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



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:

protocol version	`int protoVersion;`
	
	`java.io.ByteArrayOutputStream byteStr =`
	`    new java.io.ByteArrayOutputStream();`
	`java.io.DataOutputStream objStr =`
	`    new java.io.DataOutputStream(byteStr);`
	
	`objStr.writeInt(protoVersion);`

the final product	`byte[] requestBody = byteStr.toByteArray();`

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.

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:

registrar to respond with groups joined
 	
the final product
	<pre>net.jini.core.lookup.ServiceRegistrar reg;
	String[] groups;
	
	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();</pre>

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.

The Join Protocol 3

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.

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.

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.

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 registers 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.

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 only cancelled 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.

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.

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.

Network Issues 4

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

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.

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. While 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.

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 may 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.

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.

4.4 Using Multicast IP as the Underlying Transport

In the case where 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 well-known UDP port is used by instances of the discovery request service and joining entities.

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`.

LookupLocator Class 5

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 hostname 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 hostname 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.

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:port/data`
- `protocol://host/data`

The protocol component of a Jini technology URL is, not surprisingly, `jini`. The hostname 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

1. If you speak hexadecimal, you will notice that 4160 is the decimal representation of `CAFE` - `BABE`.

```
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
```

5.2 Serialized Form

The `serialVersionUID` of `LookupLocator` is 1448769379829432795. The serialized fields are:

- `String host` - the host
- `int port` - the port