Jini™ Lookup Service Specification

The Jini™ system is a Java™ platform-centric distributed system designed around the goals of simplicity, flexibility, and federation. The Jini Lookup service provides a central registry of services available within a djinn. The lookup service is a primary means for programs to find services within a djinn, and is the foundation for providing user interfaces through which users and administrators can discover and interact with services in the djinn.

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

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

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

1.2 The Lookup Service Model

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

When a new service is created (for example, when a new device is added to the djinn), the service registers itself with the djinn's lookup service, providing an initial collection of attributes. For example, a printer might include attributes

indicating speed (in pages per minute), resolution (in dots per inch) and whether duplex printing is supported. Among the attributes might be an indicator that the service is new and needs to be configured.

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

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

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

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

1.3 Attributes

The attributes of a service item are represented as a set of attribute sets. An individual *attribute set* is represented as an instance of some class for the Java platform, each attribute being a public field of that class. The class provides strong typing of both the set and the individual attributes. A service item can contain multiple instances of the same class with different attribute values, as

well as multiple instances of different classes. For example, an item might have multiple instances of a `Name` class, each giving the common name of the service in a different language, plus an instance of a `Location` class, an `Owner` class, and various service-specific classes. The schema used for attributes is not constrained by this specification, but a standard foundation schema for Jini systems is defined in the *Jini Lookup Attribute Schema Specification*.

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

1.4 Dependencies

This specification relies on the following other specifications:

- Java Remote Method Invocation Specification
- Java Object Serialization Specification
- Jini™ Entry Specification
- Jini™ Distributed Event Specification
- Jini™ Distributed Leasing Specification
- Jini™ Discovery and Join Specification

1.5 Comments

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

The ServiceRegistrar 2

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

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

2.1 ServiceID

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

```
public final class ServiceID implements Serializable {
    public ServiceID(long mostSig, long leastSig);
    public ServiceID(DataInput in) throws IOException;

    public void writeBytes(DataOutput out) throws IOException;
    public long getMostSignificantBits();
    public long getLeastSignificantBits();
}
```

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

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

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

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

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

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

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

2.2 *ServiceItem*

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

```
public class ServiceItem implements Serializable {
    public ServiceItem(ServiceID serviceID,
                       Object service,
                       Entry[] attributeSets);
    public ServiceID serviceID;
    public Object service;
    public Entry[] attributeSets;
}
```

The constructor simply assigns each parameter to the corresponding field.

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

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

2.3 ServiceTemplate and Item Matching

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

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

The constructor simply assigns each parameter to the corresponding field.

A service item (`item`) matches a service template (`tmpl`) if:

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

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

2.4 Other Supporting Types

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

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

The constructor simply assigns each parameter to the corresponding field.

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

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

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

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

2.5 *ServiceRegistrar*

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

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

```
public interface ServiceRegistrar {

    ServiceRegistration register(ServiceItem item,
                                 long leaseDuration)
           throws RemoteException;

    Object lookup(ServiceTemplate tmpl)
           throws RemoteException;

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

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

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

    Class[] getEntryClasses(ServiceTemplate tmpl)
           throws RemoteException;

    Object[] getFieldValues(ServiceTemplate tmpl,
                            int setIndex,
                            String field)
           throws NoSuchFieldException, RemoteException;

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

    ServiceID getServiceID();

    LookupLocator getLocator() throws RemoteException;

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

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

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

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

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

Any duplicate attribute sets included in a service item are eliminated in the stored representation of the item. The lease duration request (specified in milliseconds) is not exact; the returned lease is allowed to have a shorter (but not longer) duration than what was requested. The registration is persistent across restarts (crashes) of the lookup service until the lease expires or is cancelled.

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

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

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

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

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

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

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

Sequence numbers for a given event id are strictly increasing. If there is no gap between two sequence numbers, no events have been missed; if there is a gap, events might (but might not) have been missed. For example, a gap might occur if the lookup service crashes, even if no events are lost due to the crash.

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

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

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

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

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

The `getLocator` method returns a `LookupLocator` that can be used if necessary for unchaste discovery of the lookup service. The definition of this class is given in the *Jini™ Technology Discovery and Join Specification*.

The `getGroups` method returns the set of groups that this lookup service is currently a member of. The semantics of these groups is defined in the *Jini Technology Discovery and Join Specification*.

2.6 ServiceRegistration

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

```
public interface ServiceRegistration {

    ServiceID getServiceID();

    Lease getLease();

    void addAttributes(Entry[] attrSets)
        throws UnknownLeaseException, RemoteException;

    void modifyAttributes(Entry[] attrSetTemplates,
                          Entry[] attrSets)
        throws UnknownLeaseException, RemoteException;

    void setAttributes(Entry[] attrSets)
        throws UnknownLeaseException, RemoteException;
}
```

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

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

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

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

The `modifyAttributes` method is used to modify existing attribute sets. The lengths of `attrSetTemplates` and `attrSets` must be equal, or `IllegalArgumentException` is thrown. The service item's attribute sets are modified as follows. For each array index *i*: if `attrSets[`*i*`]` is null, then every entry that matches `attrSetTemplates[`*i*`]` is deleted; otherwise, for every non-null field in `attrSets[`*i*`]`, the value of that field is stored into the corresponding field of every entry that matches `attrSetTemplates[`*i*`]`. The class of `attrSets[`*i*`]` must be the same as, or a superclass of, the class of `attrSetTemplates[`*i*`]`, or `IllegalArgumentException` is thrown. If the modifications result in duplicate entries within the service item, the duplicates are eliminated. `UnknownLeaseException` is thrown if the registration lease has expired or been cancelled.

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

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

2.7 Serialized Forms

The `serialVersionUID` of `ServiceID` is -7803375959559762239. The serialized fields are:

- `long mostSig` — the most significant bits
- `long leastSig` — the least significant bits

The `serialVersionUID` of `ServiceItem` is 717395451032330758. The serialized fields are the declared public fields.

The `serialVersionUID` of `ServiceTemplate` is 7854483807886483216. The serialized fields are the declared public fields.

The `serialVersionUID` of `ServiceMatches` is -5518280843537399398. The serialized fields are the declared public fields.

The `serialVersionUID` of `ServiceEvent` is 1304997274096842701. The serialized fields are:

- `ServiceID serviceID` — the service ID
- `int transition` — the transition