Acknowledgements

First of all, I would like to thank **Allah Almighty** for His continued kindness, guidance and help, by virtue of which I am able to complete this project. In addition, I would like to thank all of the following individuals:

Dr. Arshad Ali (Project Supervisor) for the support and the guidance he provided me from the start of this project and for providing the professional look to the contents of this project.

Dr Iosif Legrand (Project Supervisor Foreign) for the continued and directed guidance for the implementational details of the project.

Mr. Ali Ahsan who took interest in the project and gave his precious time in areas where I needed help.

I must mention that without my family's support, love and understanding, this project would have remained a virtual commodity.

Abstract

On January 25, 1999, Sun launched JINI™ technology with a global event in San Francisco including a number of partner announcements. Based on this technology, I have been assigned the task of implementing a JINI Service Manager. This project basically has three main players i.e., a client for each service, a service registering service, and core distributed services. These three players are used in a distributed environment using JINI architecture. JINI architecture requires services be registered with a lookup service (provided by JINI). On request, for a particular service, from the remote client, lookup service gets activated and returns a proxy object (stub) of that service. Once the client gets the stub, it can call any of the methods of the service. These methods are executed remotely where they are registered. The client invokes those methods and passes input arguments (if any) to them. In return the remote client can get the data returned by the method.

This architecture helps us start and stop services remotely, which are accessible by the distributed clients. The nitty gritty of the networking, security and transactional services are handled by JINI.

Table of contents

Chapter 1

The Overview of JINI

1. Introduction

1.1 JINI

JINI is the name for a distributed computing environment, which can offer ``network plug and play''. A device or a software service can be connected to a network and announce its presence, and clients that wish to use such a service can then locate it and call it to perform tasks. JINI can be used for mobile computing tasks where a service may only be connected to a network for a short time, but it can more generally be used in any network where there is some degree of change. There are a large number of scenarios where this would be useful:

- A new printer can be connected to the network and announce its presence and capabilities. A client can then use this printer without having to be specially configured to do so.

- A digital camera can be connected to the network and present a user interface that will not only allow pictures to be taken, but it can also be aware of any printers so that the pictures can be printed.

- A configuration file that is copied and modified on individual machines can be made into a network service from a single machine, reducing maintenance costs.

- New capabilities extending existing ones can be added to a running system without disrupting existing services, or without any need to reconfigure clients.

- Services can announce changes of state, such as when a printer runs out of paper. Listeners, typically of an administrative nature, can watch for these changes and flag them for attention.

JINI is not an acronym for anything, and does not have a particular meaning. (Though it gained a *post hoc* it gained an interpretation of ``JINI Is Not Initials''.) A JINI system or *federation* is a collection of clients and services all communicating by the JINI protocols. Often this will consist of applications written in Java, communicating using the Java Remote Method Invocation mechanism. Although JINI is written in pure Java, neither clients nor services are constrained to be in pure Java. They may include native code methods, act as wrappers around non-Java objects, or even be written in some other language altogether. JINI supplies a ``middleware'' layer to link services and clients from a variety of sources.

When you download a copy of ``JINI'', you get a mixture of things. Firstly, JINI is a specification of a set of middleware components. This includes an API (Application Programmer's Interface) so that you as a programmer can write services and components that make use of this middleware. Secondly, it includes an implementation (in pure Java) of the middleware, as a set of Java packages.

In a running JINI system, there are three main players. There is a *service*, such as a printer, a toaster, a marriage agency, etc. There is a *client* which would like to make use of this service. Thirdly, there is a *lookup service* (service locator), which acts as a broker/trader/locator between services and clients. There is an additional component, and that is a *network* connecting all three of these, and this network will generally be running TCP/IP. (The JINI specification is fairly independent of network protocol, but the only current implementation is on TCP/IP.)

1.2 Background

JINI grew from early work in Java to make distributed computing easier. It intends to make ``network devices'' and ``network computing'' into standard components of everyone's computing environment.

1.3 Definition

JINI is a network technology that makes services available on the network transparent to the users.

1.4 Application

When you buy a new piece of office computing equipment such as a desk lamp, or a new home computer appliance such as an alarm clock, it will not only carry out its ``traditional'' functions but will also join into a network of other computer devices and services. The desk lamp will turn itself off when you leave your desk, informed by sensors in your chair; the alarm clock will tell your coffee maker to switch on a few minutes before it wakes you up.

Homes, offices and factories are becoming increasingly networked. Current twisted pair wiring will remain, but will be augmented by wireless networks and networks built on your phone lines and power cables. On top of this will be an infrastructure to allow devices to communicate. TCP/IP will be a part of this, but will not be enough. There will need to be mechanisms for service discovery, for negotiation of properties, and for event signaling (``my alarm has gone off - does anyone want to know?''). JINI supplies this higher level of interaction.

1.5 JINI's Goals

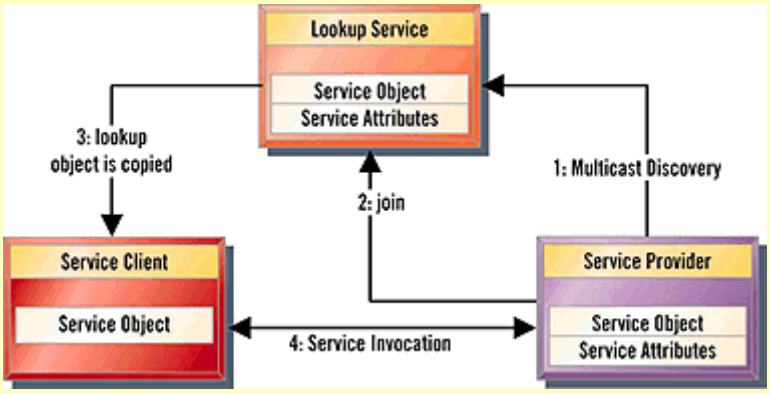
JINI aims to satisfy a diverse constituency, enabling users to share services and resources over the network and providing users easy access to resources anywhere on the network, providing programmers with tools and programming patterns for the development of robust and secure distributed systems, and simplifying the task of building and maintaining network devices, software and users.

1.6 The Programming Model

JINI services are objects written in the Java programming language. Each service has an interface, which defines the operations that can be requested by clients. An example would be a printing service or a disk drive for storage.

When a JINI service is developed, it must announce its presence to other services and users. Users and services, in turn, must discover other services and communicate with them. To enable this, at the heart of JINI are three protocols: discovery, join and lookup. The discovery and join protocols are used when a service is plugged in, the former when a service is looking for a lookup service to register itself and the latter when a service has located a lookup service and wishes to join. The lookup occurs when a service client or a user needs to locate and invoke a service (Figure 1).

Figure 1. The Discovery, Join and Lookup Protocols



When a JINI service is developed, it must announce its presence to other services and users. Users and services, in turn, must discover other services and communicate with them. To enable this, at the heart of JINI are three protocols: discovery, join and lookup. The discovery and join protocols are used when a service is plugged in. The lookup occurs when a service client or user needs to find and invoke a service.

Here, the service provider will have to locate a lookup service by multicasting a request on the network. If a lookup service identifies itself, then the service object is loaded into (that is, it joins) that lookup service. The service is now ready to be looked up and used by other services and clients. Now a client locates a service using the lookup service. Once it is located, the service object is loaded into the client, and finally the client invokes the service.

1.7 Installing JINI

JINI uses some of the Remote Method Invocation (RMI) extensions (for example, activation) available in Java Development Kit 1.2. Therefore, before you start downloading the JINI software kit (available at www.sun.com/JINI), make sure you have installed JDK 1.2, which is

available from Sun Microsystems for both Solaris and Windows 95, 98 and NT. Once you have JDK 1.2 installed, you can download the JINI Starter Kit (JSK), which is made up of the JINI Technology Core Platform (JCP), the JINI Technology Extended Platform (JXP) and the JINI Software Kit (JKS). The JCP includes the specifications and corresponding interfaces and classes for lookup, discovery and join, distributed events, leasing and transactions. This constitutes the core JINI technology. The JXP provides the extended JINI technology infrastructure software and includes the specifications for the discovery utilities and entry utilities. The JSK provides implementations of the lookup services and transaction manager services specified in the JCP.

1.8 Class Hierarchy

- class java.lang.Object
  - class net.jini.entry.**AbstractEntry** (implements net.jini.core.entry.Entry)
    - class net.jini.lookup.entry.**Address**
    - class net.jini.lookup.entry.**Comment**
    - class net.jini.lookup.entry.**Location**
    - class net.jini.lookup.entry.**Name**
    - class net.jini.lookup.entry.**ServiceInfo** (implements net.jini.lookup.entry.ServiceControlled)
    - class net.jini.lookup.entry.**ServiceType** (implements net.jini.lookup.entry.ServiceControlled)
      - class com.sun.jini.lookup.entry.**BasicServiceType**
    - class net.jini.lookup.entry.**Status** (implements net.jini.lookup.entry.ServiceControlled)
  - class com.sun.jini.lease.**AbstractLease** (implements net.jini.core.lease.Lease, java.io.Serializable)
    - class com.sun.jini.lease.landlord.**LandlordLease**

- class com.sun.jini.lease.**AbstractLeaseMap** (implements net.jini.core.lease.LeaseMap, java.io.Serializable)
    - class com.sun.jini.lease.landlord.**LandlordLeaseMap**
- class net.jini.lookup.entry.**AddressBean** (implements net.jini.lookup.entry.EntryBean, java.io.Serializable)
- class com.sun.jini.start.**ClassLoaderUtil**
- class com.sun.jini.lease.landlord.**CodebaseLeaseFactory** (implements com.sun.jini.lease.landlord.LandlordLeaseAccessor, com.sun.jini.lease.landlord.LandlordLeaseFactory)
- class net.jini.lookup.entry.**CommentBean** (implements net.jini.lookup.entry.EntryBean, java.io.Serializable)
- class net.jini.discovery.**Constants**
- class net.jini.lookup.entry.**EntryBeans**
- class java.util.EventObject (implements java.io.Serializable)
    - class net.jini.discovery.**DiscoveryEvent**
    - class net.jini.lease.**LeaseRenewalEvent**
    - class com.sun.jini.lease.**LeaseRenewalEvent**
    - class net.jini.core.event.**RemoteEvent**
        - class net.jini.lease.**ExpirationWarningEvent**
        - class net.jini.discovery.**RemoteDiscoveryEvent**
        - class net.jini.lease.**RenewalFailureEvent**
            - class com.sun.jini.lease.**BasicRenewalFailureEvent**
        - class net.jini.core.lookup.**ServiceEvent**
    - class net.jini.lookup.**ServiceDiscoveryEvent**
- class net.jini.core.event.**EventRegistration** (implements java.io.Serializable)
- class net.jini.discovery.**IncomingMulticastAnnouncement**
- class net.jini.discovery.**IncomingMulticastRequest**
- class net.jini.discovery.**IncomingUnicastRequest**
- class net.jini.discovery.**IncomingUnicastResponse**

- class net.jini.lookup.**JoinManager**
- class com.sun.jini.lookup.**JoinManager**
- class com.sun.jini.lease.landlord.**Landlord.RenewResults** (implements java.io.Serializable)
- class com.sun.jini.lease.landlord.**LandlordLease.Factory** (implements com.sun.jini.lease.landlord.LandlordLeaseAccessor, com.sun.jini.lease.landlord.LandlordLeaseFactory)
- class com.sun.jini.lease.landlord.**LandlordUtil**
- class com.sun.jini.lease.landlord.**LeaseDurationPolicy** (implements com.sun.jini.lease.landlord.LeasePolicy)
- class net.jini.lease.**LeaseRenewalManager**
- class com.sun.jini.lease.**LeaseRenewalManager**
- class net.jini.lookup.entry.**LocationBean** (implements net.jini.lookup.entry.EntryBean, java.io.Serializable)
- class com.sun.jini.lookup.entry.**LookupAttributes**
- class net.jini.discovery.**LookupDiscovery** (implements net.jini.discovery.DiscoveryGroupManagement, net.jini.discovery.DiscoveryManagement)
- class net.jini.discovery.**LookupDiscoveryManager** (implements net.jini.discovery.DiscoveryGroupManagement, net.jini.discovery.DiscoveryLocatorManagement, net.jini.discovery.DiscoveryManagement)
- class net.jini.core.discovery.**LookupLocator** (implements java.io.Serializable)
- class net.jini.discovery.**LookupLocatorDiscovery** (implements net.jini.discovery.DiscoveryLocatorManagement, net.jini.discovery.DiscoveryManagement)
- class com.sun.jini.discovery.**LookupLocatorDiscovery**
- class net.jini.lookup.entry.**NameBean** (implements net.jini.lookup.entry.EntryBean, java.io.Serializable)

- class net.jini.core.transaction.**NestableTransaction.Created** (implements java.io.Serializable)
- class net.jini.discovery.**OutgoingMulticastAnnouncement**
- class net.jini.discovery.**OutgoingMulticastRequest**
- class net.jini.discovery.**OutgoingUnicastRequest**
- class net.jini.discovery.**OutgoingUnicastResponse**
- class com.sun.jini.start.**ParsedArgs**
- class java.security.Permission (implements java.security.Guard, java.io.Serializable)
    - class net.jini.discovery.**DiscoveryPermission** (implements java.io.Serializable)
- class net.jini.core.transaction.server.**ServerTransaction** (implements java.io.Serializable, net.jini.core.transaction.Transaction)
    - class net.jini.core.transaction.server.**NestableServerTransaction** (implements net.jini.core.transaction.NestableTransaction)
- class net.jini.lookup.**ServiceDiscoveryManager**
- class net.jini.core.lookup.**ServiceID** (implements java.io.Serializable)
- class net.jini.lookup.entry.**ServiceInfoBean** (implements net.jini.lookup.entry.EntryBean, java.io.Serializable)
- class net.jini.core.lookup.**ServiceItem** (implements java.io.Serializable)
- class net.jini.core.lookup.**ServiceMatches** (implements java.io.Serializable)
- class com.sun.jini.start.**ServiceStarter**
- class com.sun.jini.start.**ServiceStarter.Created**
- class net.jini.core.lookup.**ServiceTemplate** (implements java.io.Serializable)
- class com.sun.jini.start.**StartUtil**

- class com.sun.jini.start.**StartUtil.ParsedListResult**
- class net.jini.lookup.entry.**StatusBean** (implements net.jini.lookup.entry.EntryBean, java.io.Serializable)
- class net.jini.lookup.entry.**StatusType** (implements java.io.Serializable)
- class java.lang.Throwable (implements java.io.Serializable)
    - class java.lang.Exception
        - class net.jini.core.lease.**LeaseException**
            - class net.jini.core.lease.**LeaseDeniedException**
            - class net.jini.core.lease.**LeaseMapException**
            - class net.jini.core.lease.**UnknownLeaseException**
        - class net.jini.lease.**LeaseUnmarshalException**
        - class net.jini.discovery.**LookupUnmarshalException**
        - class java.lang.RuntimeException
            - class net.jini.space.**InternalSpaceException**
        - class net.jini.core.transaction.**TransactionException**
            - class net.jini.core.transaction.**CannotAbortException**
            - class net.jini.core.transaction.**CannotCommitException**
            - class net.jini.core.transaction.**CannotJoinException**
            - class net.jini.core.transaction.**CannotNestException**
            - class net.jini.core.transaction.server.**CrashCountException**

        - class net.jini.core.transaction.**TimeoutExpiredException**
        - class net.jini.core.transaction.**UnknownTransactionException**
      - class net.jini.core.event.**UnknownEventException**
      - class net.jini.core.entry.**UnusableEntryException**
  - class net.jini.core.transaction.**Transaction.Created** (implements java.io.Serializable)
  - class net.jini.core.transaction.**TransactionFactory**
  - class net.jini.core.transaction.server.**TransactionManager.Created** (implements java.io.Serializable)

1. 9 Interface Hierarchy

- interface net.jini.admin.**Administrable**
- interface com.sun.jini.outrigger.**AdminIterator**
- interface com.sun.jini.admin.**DestroyAdmin**
  - interface com.sun.jini.fiddler.**FiddlerAdmin**(also extends net.jini.admin.JoinAdmin, com.sun.jini.admin.StorageLocationAdmin)
  - interface com.sun.jini.outrigger.**JavaSpaceAdmin**(also extends net.jini.admin.JoinAdmin, com.sun.jini.mahout.RegistryAdmin)
  - interface com.sun.jini.mercury.**MailboxAdmin**(also extends net.jini.admin.JoinAdmin, com.sun.jini.admin.StorageLocationAdmin)
  - interface com.sun.jini.norm.**NormAdmin**(also extends net.jini.admin.JoinAdmin)
  - interface com.sun.jini.reggie.**RegistrarAdmin**(also extends net.jini.lookup.DiscoveryAdmin, net.jini.admin.JoinAdmin, com.sun.jini.admin.StorageLocationAdmin)

- interface net.jini.lookup.**DiscoveryAdmin**
  - interface com.sun.jini.reggie.**RegistrarAdmin**(also extends com.sun.jini.admin.DestroyAdmin, net.jini.admin.JoinAdmin, com.sun.jini.admin.StorageLocationAdmin)
- interface net.jini.discovery.**DiscoveryGroupManagement**
- interface net.jini.discovery.**DiscoveryLocatorManagement**
- interface net.jini.discovery.**DiscoveryManagement**
- interface net.jini.lookup.entry.**EntryBean**
- interface java.util.EventListener
  - interface net.jini.discovery.**DiscoveryListener**
    - interface net.jini.discovery.**DiscoveryChangeListener**
  - interface net.jini.lease.**LeaseListener**
    - interface net.jini.lease.**DesiredExpirationListener**
  - interface com.sun.jini.lease.**LeaseListener**
  - interface net.jini.core.event.**RemoteEventListener**(also extends java.rmi.Remote)
  - interface net.jini.lookup.**ServiceIDListener**
  - interface com.sun.jini.lookup.**ServiceIDListener**
- interface net.jini.event.**EventMailbox**
- interface net.jini.space.**JavaSpace**
- interface net.jini.admin.**JoinAdmin**
  - interface com.sun.jini.fiddler.**FiddlerAdmin**(also extends com.sun.jini.admin.DestroyAdmin, com.sun.jini.admin.StorageLocationAdmin)
  - interface com.sun.jini.outrigger.**JavaSpaceAdmin**(also extends com.sun.jini.admin.DestroyAdmin, com.sun.jini.mahout.RegistryAdmin)
  - interface com.sun.jini.mercury.**MailboxAdmin**(also extends com.sun.jini.admin.DestroyAdmin, com.sun.jini.admin.StorageLocationAdmin)

        - interface com.sun.jini.norm.**NormAdmin**(also extends com.sun.jini.admin.DestroyAdmin)
        - interface com.sun.jini.reggie.**RegistrarAdmin**(also extends com.sun.jini.admin.DestroyAdmin, net.jini.lookup.DiscoveryAdmin, com.sun.jini.admin.StorageLocationAdmin)
    - interface com.sun.jini.lease.landlord.**LandlordLeaseAccessor**
    - interface com.sun.jini.lease.landlord.**LandlordLeaseFactory**
    - interface net.jini.core.lease.**Lease**
    - interface com.sun.jini.lease.landlord.**LeasedResource**
    - interface com.sun.jini.lease.landlord.**LeaseManager**
    - interface com.sun.jini.lease.landlord.**LeasePolicy**
    - interface net.jini.lease.**LeaseRenewalService**
    - interface net.jini.lease.**LeaseRenewalSet**
    - interface com.sun.jini.lease.landlord.**LocalLandlord**
    - interface net.jini.lookup.**LookupCache**
    - interface net.jini.discovery.**LookupDiscoveryRegistration**
    - interface net.jini.discovery.**LookupDiscoveryService**
    - interface net.jini.event.**MailboxRegistration**
    - interface java.util.Map
        - interface net.jini.core.lease.**LeaseMap**
    - interface com.sun.jini.mahout.**RegistryAdmin**
        - interface com.sun.jini.outrigger.**JavaSpaceAdmin**(also extends com.sun.jini.admin.DestroyAdmin, net.jini.admin.JoinAdmin)
    - interface java.rmi.Remote
        - interface com.sun.jini.lease.landlord.**Landlord**
        - interface com.sun.jini.mahout.binder.**RefHolder**
        - interface net.jini.core.event.**RemoteEventListener**(also extends java.util.EventListener)
        - interface net.jini.core.transaction.server.**TransactionManager**(also extends net.jini.core.transaction.server.TransactionConstants)

        - interface net.jini.core.transaction.server.**NestableTransactionManager**
    - interface net.jini.core.transaction.server.**TransactionParticipant**(also extends net.jini.core.transaction.server.TransactionConstants)
- interface java.io.Serializable
    - interface net.jini.core.entry.**Entry**
- interface net.jini.lookup.entry.**ServiceControlled**
- interface net.jini.lookup.**ServiceDiscoveryListener**
- interface net.jini.lookup.**ServiceItemFilter**
- interface net.jini.core.lookup.**ServiceRegistrar**
- interface net.jini.core.lookup.**ServiceRegistration**
- interface com.sun.jini.admin.**StorageLocationAdmin**
    - interface com.sun.jini.fiddler.**FiddlerAdmin**(also extends com.sun.jini.admin.DestroyAdmin, net.jini.admin.JoinAdmin)
    - interface com.sun.jini.mercury.**MailboxAdmin**(also extends com.sun.jini.admin.DestroyAdmin, net.jini.admin.JoinAdmin)
    - interface com.sun.jini.reggie.**RegistrarAdmin**(also extends com.sun.jini.admin.DestroyAdmin, net.jini.lookup.DiscoveryAdmin, net.jini.admin.JoinAdmin)
- interface net.jini.core.transaction.**Transaction**
    - interface net.jini.core.transaction.**NestableTransaction**
- interface net.jini.core.transaction.server.**TransactionConstants**
    - interface net.jini.core.transaction.server.**TransactionManager**(also extends java.rmi.Remote)
        - interface net.jini.core.transaction.server.**NestableTransactionManager**

- interface net.jini.core.transaction.server.**TransactionParticipant**(also extends java.rmi.Remote)

2. Components of a JINI System

In a running JINI system, there are three main players. There is a service, such as a printer, a toaster, a marriage agency, etc. There is a client which would like to make use of this service. Thirdly, there is a lookup service (service locator) which acts as a broker/trader/locator between services and clients. There is an additional component, and that is a network connecting all three of these, and this network will generally be running TCP/IP. (The JINI specification is fairly independent of network protocol, but the only current implementation is on TCP/IP.), as shown in figure:

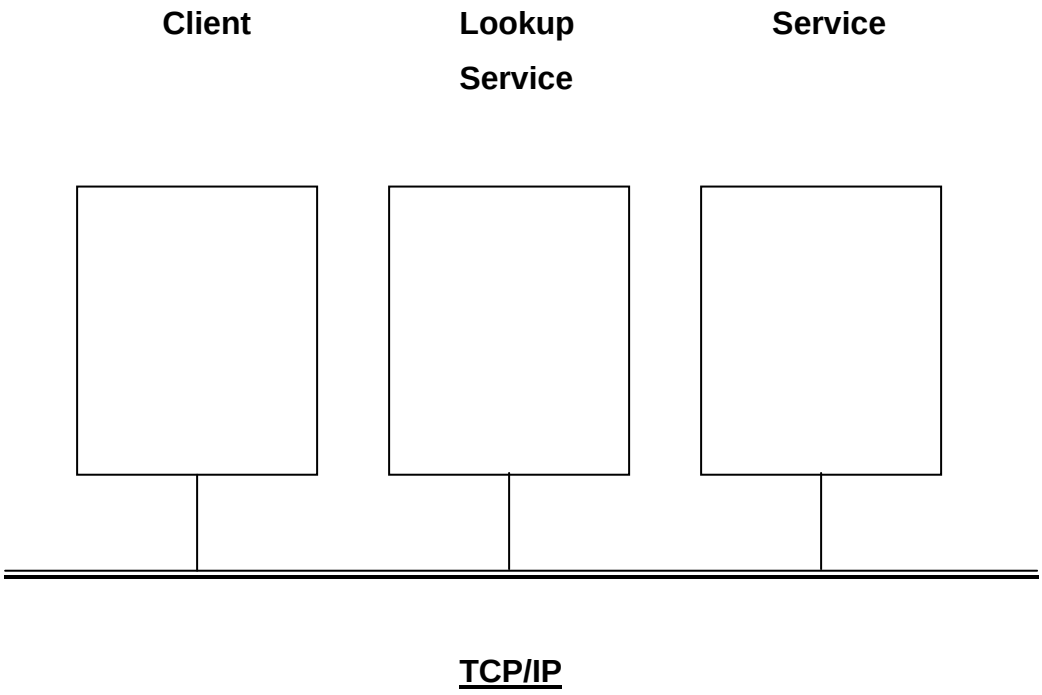


Figure 2.1: Components of a JINI system

2.1 The Lookup Service

A client locates a service by querying a lookup service (service locator). In order to do this, it must first locate such a service. On the other hand, a service must register itself with the lookup service, and in order to do so it must also first locate a service.

The initial phase of both a client and a service is thus discovering a lookup service. Such a service (or set of services) will usually have been started by some independent mechanism. The search for a lookup service can be done either by unicast or by multicast. In fact, the lookup service is just another JINI service, but it is one that is specialized to store services and pass them on to clients looking for them.

2.2 Reggie

Sun supplies a lookup service called Reggie as part of the standard JINI distribution. The specification of a lookup service is public, and in future we may expect to see other implementations of lookup services. There may be any number of these lookup services running in a network. A LAN may run many lookup services to provide redundancy in case one of them crashes. Anybody can start a lookup service (depending on access permissions), but it is first of all not an easy job, and secondly it will usually be started by an administrator, or started at boot time.

Reggie requires support services to work: an HTTP server and an RMI daemon, rmid. If there is already an HTTP server running, this can be used, or a new one can be started. If you don't have access to an HTTP server (such as Apache), then there is a simple one supplied by JINI. This server is incomplete, and is only good for downloading Java class files - it cannot be used as a general-purpose Web server. The JINI HTTP server is in the jar file tools.jar. This can be started by

java -jar tools.jar

This runs on a default port (8080), which means that any user can start it as long as local network policies do not forbid it. It uses the current directory as ``document root'' for locating class files. These can be controlled by parameters:

java -jar tools-jarfile [-port port-number] [-dir document-root-dir] [-trees] [-verbose]

The HTTP server is needed to deliver the stub class files (of the registrar) to clients. These class files are stored in reggie-dl.jar, so this file must be reachable from the document root. For example, on my machine the jar file has full path c:\jini\lib>java –jar c:\jini\lib\.jar.

The other support service needed for is an RMI daemon. This is same machine as:

This command also has major options:

rmid [-port num] [-log dir]

This can control the TCP port used (which defaults to 4160). rmid uses log files to store its state, and they default to being in the sub-directory log. This can be controlled.

is, quite simply, the single most important and most commonly used JINI service. So understanding how it works is crucial to a painless experience of using JINI. Unfortunately, is also a pretty complicated beast. And also uses a number of techniques that may be unfamiliar to many Java programmers (activation, remote code loading, security etc).

The following couple of pages will be helpful in understanding, and particularly solving any particular problem with the service. The sections here cover:

- The Lifecycle

- Command Line Arguments
- Understanding and Customizing the Log Location

2.2.1 The Lifecycle

The implementation of the JINI lookup service is an activatable process. What this means is that the actual lookup service is started by the RMI Activation Daemon (rmid) only when it is first needed.

So if this is the case, if is started automatically by rmid, you may be wondering what all the business is with the complicated command line arguments you need to use to run!

As it turns out, when you run the program, this does *not* start the with the activation daemon, telling it to create an instance when and if needed, and then exits. You may have noticed that the command line returns after a short while, rather than running indefinitely. This is because once it has finished registering with the activation daemon it simply terminates.

The activation daemon, when it detects someone trying to invoke a remote method destined for the lookup service, will launch in its own VM, if needed.

But activation is used for more than simply "lazy" launching of . When registers itself, it passes a special flag to the activation daemon, telling it that should be registered again automatically in the future whenever the activation daemon is restarted. This is quite handy! By using the restart flag, you can launch a whole array of activatable services by simply starting up the activation daemon when your machine boots.

This also means that you typically only need to start once on a given machine, ever. Once has been run once, it will have registered itself with rmid, and rmid will have remembered that it should relaunch if needed in the future. From this point on, whenever you start rmid, will be started also. This "new" instance of will be the same from the standpoint of other JINI services and clients--it will recover the same service ID, registrations, and so on, as it had from its previous run.

This is an important point! If you start a new lookup service every time you sit down to do JINI development, you'll quickly have dozens of them running, since the old ones do not "go away" easily!

Of course, for the activation daemon to remember this information across restarts, it needs its own log directory in which to save such configuration information. By default, the activation daemon saves its configuration in a directory called "log" under the directory from which you run rmid. So if you plan on using rmid to recover activatable processes, you should make sure you run it from the same directory each time.

Since is started under control of rmid, you need to understand how to really shut it down, should you need to. One brute force solution is to run **rmid -stop,** which terminates the activation daemon. Once stopped, you can remove the activation daemon's log directory, causing it to "forget" all of its saved state.

2.2.2 Command Line Arguments

In JINI development, is a complicated program to start with. Let's look in detail at the command line:

java –J-Djava.security.policy=security_policy
java -jar lookup-server-jarfile
lookup-client-codebase

lookup-policy-file

output-log-dir (log_directory)

lookup-service-group

There are six separate arguments here that you can control, so there are six bits of information you have to understand. The first two arguments are parameters to the JVM itself--these control the behavior of the **java** launcher program, and aren't passed to . The last four are parameters to itself. Fortunately, the settings of a number of these parameters are pretty trivial to get right, once you know the location of your JINI installation.

2.2.3 Security Policy File

The first argument sets the location of a security policy file. This policy will control what the application will be allowed to do. JINI ships with two useful policy files, in **jini1_1\examples\lookup**, called **policy** and **policy.all**. These two useful policy files are also shipped in **jini1_1\policy,** called **policy** and **policy.all**. In this project, I have used **policy.all** file, since it eases development headaches.

This policy is not appropriate for general-purpose deployment, though. Check the section below on how uses its log files for some detailed directions of how to customize and use the policy file that comes with JINI.

One thing to note is that you should always pass in a fully-qualified path to the security policy file; **not a relative one**.

2.2.4 JAR File

The lookup-server-jarfile will be .jar or some path to it. The -jar argument specifies that the code to be executed by the **java** launcher lives in an "executable" JAR file. So the next bit of

information you will need is the location of the **.jar** file that comes with JINI. Typically this is in the **lib** directory of the JINI distribution.

Path: c:\jini\lib\.jar

2.2.5 Lookup Client Codebase

Lookup-client-codebase will be the URL for the stub class files, using the HTTP server started earlier. In my case, this is http://myMachineName:8080/-dl.jar. Note that an absolute IP hostname must be used - you cannot use localhost because to the service it means myMachineName, but to the client it would be a different machine altogether! The client would then fail to find -dl.jar on its own machine. Even using an abbreviated address such as myMachine would fail to be resolved if the client is external to the local network. Setting the codebase is a little trickier than the other arguments. The codebase is a URL that tells clients of where the code they will need to use the service can be downloaded from. When clients download the serialized proxy object for the service (via the Discovery mechanisms), they will need to be able to download the actual classfiles that implement this serialized proxy. This is where the codebase comes in--it will be "attached" to the serialized objects, and will tell clients where the code can be downloaded from.

Typically this value is set to a URL that references the **-dl.jar** file on a web server someplace. Many developers will run a web server (typically the small one that comes with JINI), configured so that its "root" directory points to the **lib** directory in the JINI distribution.

2.2.6 Log Directory

The next parameter is the location of a directory for holding logging information for . As runs, it will periodically save its state into this directory, so that it can recover it later if it is restarted.

This will amply be explained in the section "Understanding and Customizing the Log Location," below. But there is one important thing to realize about the log directory: You need to make sure that the directory you specify **doesn't** exist; if it does exist, then will complain and **not start**. If the directory you name does not exist, however, will create it for you, and will run happily.

2.2.7 Lookup Groups

The final argument names a set of "groups" that will join. These are simply the names of communities that the lookup service will support. You should make sure you provide at least one name here. By convention, every network should have a lookup service supporting the "public" JINI community. This community is named in the APIs by the empty string, but to cause to join the public community, you pass the string "public" here.

On my own machine, I run this:

java -jar c:\jini/lib/.jar
http://myMachineName:8080/-dl.jar
c;\jini/policy\policy.all
c:\jini\log
public

After starting, the lookup service will promptly exit ! Don't worry about this - it is actually kept in passive state by rmid, and will be brought back into existence whenever necessary (this is done by the new Activation mechanism of RMI in JDK 1.2).

You only need to start once, even if your machine is switched off or rebooted. The activation daemon rmid restarts it on an as-needed basis, as it keeps information about in its log files.

2.2.8 Understanding and Customizing the Log Location

As noted above, will persistently save information about its state to disk, so that if it crashes, it can recover its state later on. The location of this persistent log is a directory, named on the command line a shown above.

Now, if you kill and try to restart it using the same log directory, it will complain that the log directory already exists. interprets a pre-existing log file as indicating that *another* instance of was (or is) running using that logged information. A new instance of will refuse to start unless you give it its own fresh log directory, which it will create for itself.

So, in general, each instance of that you start should have its own log directory, separate from all other instances.

If you're getting into the business of running multiple instances, that’s really a very tricker issue, but possible, though, you may possibly have to configure all the security policy files used by . You'll note from the command line above that uses a policy file that controls what it is allowed to do. There are a couple of policy files that ship with the JINI release from Sun. The **example\lookup\policy** file is generally used to control the acceses of the lookup service. It's a good idea to take a look at this policy file if you plan to use it:

```
grant codebase "file:${java.class.path}" {
permission java.io.FilePermission "/tmp/_log", "read,write,delete";
permission java.io.FilePermission "/tmp/_log/-", "read,write,delete";
```

```
 // uncomment this one if you need lookup to accept file: codebases
   // permission java.io.FilePermission "<>", "read";
    permission java.lang.RuntimePermission "modifyThreadGroup";
    permission java.lang.RuntimePermission "modifyThread";
   permission java.net.SocketPermission "*:1024-", "connect,accept";
   // for http: codebases
  permission java.net.SocketPermission "*:80", "connect";
   permission java.net.SocketPermission "224.0.1.84", "connect,accept";
    permission java.net.SocketPermission "224.0.1.85", "connect,accept";
    permission   java.util.PropertyPermission   "java.rmi.server.hostname",
"read";
   permission java.util.PropertyPermission "com.sun.jini..*", "read";
   permission java.util.PropertyPermission "net.jini.discovery.*", "read";
               permission net.jini.discovery.DiscoveryPermission "*";
          };
```

Of course, if you use the policy.all policy file, which allows all accesses, then you won't have to customize your policy file (although this is a bad idea in a production environment).

2.3 Unicast discovery

Unicast discovery can be used when you already know the machine on which the lookup service resides, so you can ask for it directly. This is expected to be used for a lookup service that is outside of your local network, which you know the address of anyway (your home network while you are at work, given in some newsgroup or email message, or maybe even advertised on TV!).

2.3.1 LookupLocator

The class LookupLocator in package net.jini.core.discovery is used for this. There are two constructors:

```
LookupLocator(java.lang.String url)
        throws java.net.MalformedURLException;
LookupLocator(java.lang.String host,int port);
```

For the first constructor, the URL must be of the form jini://host/ or jini://host:port/. If no port is given, it defaults to 4160. The host should be a valid DNS name or an IP address (such as 137.92.11.13). No unicast discovery is performed at this stage, though, so any rubbish could be entered. Only a check for syntactic validity of the URL is performed. This syntactic check is not even done for the second constructor.

2.3.2 Information from the LookupLocator

A LookupLocator has methods

```
String getHost();
int getPort();
```

which will return information about the hostname that the locator will use, and the port it will connect on or is already connect on. This is just the information fed into the constructor or left to default values, though. It doesn't give anything new for unicasting. This information *will* be useful in the multicast situation, though, if you need to find out where the lookup service is.

2.3.3 Get Registrar

Search and lookup is performed by the method getRegistrar() of the LookupLocator which returns an object of class ServiceRegistrar.

```
public ServiceRegistrar getRegistrar()
                    throws java.io.IOException,
```

java.lang.ClassNotFoundException

The ServiceRegistrar is discussed in detail later. This performs network lookup on the URL given in the LookupLocator constructor.

UML sequence diagrams are useful for showing the timelines of object existence and the method calls that are made from one object to another. The timeline reads down, and method calls and their returns read across. A UML sequence diagram augmented with a jagged arrow showing the network connection is shown in figure . The UnicastRegister object makes a new() call to create a LookupLocator and this call returns a lookup object. The method call getRegistrar() is then made on the lookup object, and this causes network activity. As a result of this, a ServiceRegistrar object is created in some manner by the lookup object, and this is returned from the method as the registrar.

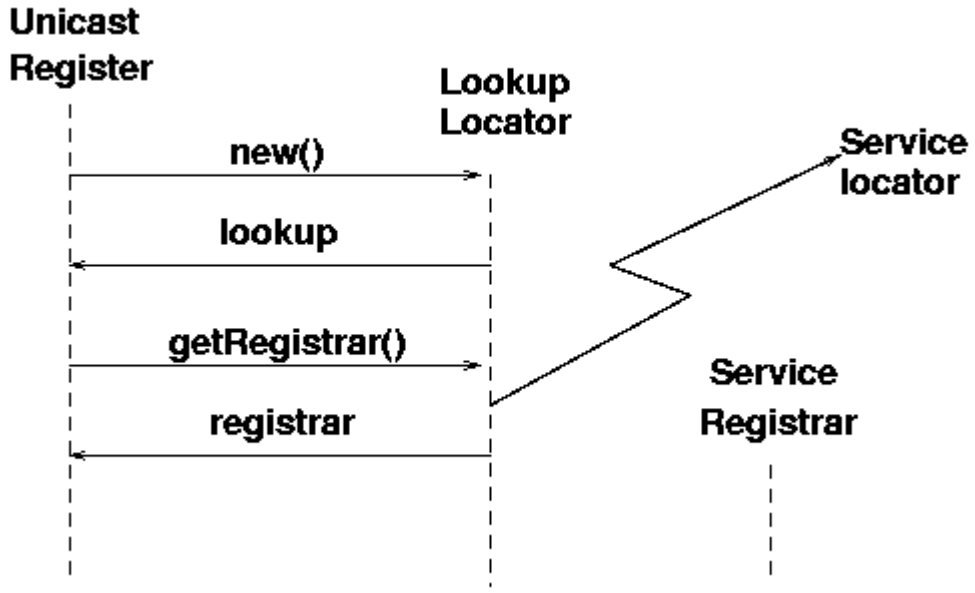


Figure 2.2*:* UML sequence diagram for lookup

The registrar object will be used in different ways for clients and services: the services will use it to register themselves, and the clients will use it to locate services.

2.4 Broadcast discovery

If the location of a lookup service is unknown, it is necessary to make a broadcast search for one. UDP supports a multicast mechanism which the current implementations of JINI use. Because multicast is expensive in terms of network requirements, most routers block multicast packets. This usually restricts broadcast to a local area network, although this depends on the network configuration and the time-to-live (TTL) of the multicast packets.

There may be any number of lookup services running on the network accessible to broadcast search. On a small network, such as a home network, there may be just a single lookup service, but in a large network there may be many - perhaps one or two per department. Each one of these may choose to reply to a broadcast request.

2.4.1 Groups

Some services may be meant for anyone to use, but some may be more restricted in applicability. For example, the Engineering Dept may wish to keep lists of services specific to that department. This may include a departmental diary service, a deparmental inventory, *etc*. The services themselves may be running anywhere in the organisation, but the department would like to be able to store information about them and to locate them from their own lookup service. Of course, this lookup service may be running anywhere too!

So there could be lookup services specifically for a particular group of services such as the Engineering Dept services, and others for the Publicity Dept services. Some lookup services may cater for more than one group - for example a company lookup service may want to hold information about *all* services running for all groups.

When a lookup service is started, it can be given a list of groups to act for as a command line parameter. A service may include such group information by giving a list of groups that it belongs too. This is an array of strings, such as

```
String [] groups = {"Engineering dept"};
```

2.4.2 LookupDiscovery

The class LookupDiscovery in package net.jini.discovery is used for broadcast discovery. There is a single constructor

```
LookupDiscovery(java.lang.String[] groups)
```

The parameter to the LookupDiscovery constructor can take three cases

- null, or LookupDiscovery.ALL_GROUPS, means to attempt to discover all reachable lookup services no matter which group they belong to. This will be the normal case.

- An empty list of strings, or LookupDiscovery.NO_GROUPS, means that the object is created, but no search is performed. In this case, the method setGroups() will need to be called in order to perform a search.

- A non-empty array of strings can be given. This will attempt to discover all lookup services in that set of groups.

2.5 DiscoveryListener

A broadcast is a multicast call across the network, expecting lookup services to reply as they receive it. Doing so may take time, and there will generally be an unknown number of lookup services that can reply. To be notified of lookup services as they are discovered, the application must register a listener with the LookupDiscovery object.

```
public void addDiscoveryListener(DiscoveryListener l)
```

The listener must implement the DiscoveryListener interface:

```
package net.jini.discovery;
public abstract interface DiscoveryListener
{
  public void discovered(DiscoveryEvent e);
  public void discarded(DiscoveryEvent e);
}
```

The discovered() method is invoked whenever a lookup service has been discovered. The API recommends that this method should return quickly, and not make any remote calls. However, for a service it is the natural place to register the service, and for a client it is the natural place to ask if there is a service available and to invoke this service. It may be better to perform these lengthy operations in a separate thread.

There are other timing issues involved: when the DiscoveryListener is created, the broadcast is made. After this, a listener is added to this discovery object. What happens if replies come in very quickly, before the listener is added? The ``JINI Discovery Utilities Specification'' guarantees that these replies will be buffered and

delivered when a listener is added. Conversely, no replies may come in for a long time - what is the application supposed to do in the meantime? It cannot simply exit, because then there would be no object to reply to! it has to be made persistent enough to last till replies come in. One way of handling this is if the application has a GUI interface - then it will stay till the user dismisses it. Another possibility is that the application may be prepared to wait for a while before giving up. In that case the main could sleep for, say, ten seconds and then exit. This will depend on what the application should do if no lookup service is discovered.

The discarded() method is invoked whenever the application discards a lookup service by calling discard() on the registrar object.

2.6 DiscoveryEvent

The parameter to the discovered()method of the DiscoveryListener interface is a DiscoveryEvent object.

```
package net.jini.discovery;

public Class DiscoveryEvent {
    public net.jini.core.lookup.ServiceRegistrar[] getRegistrars();
}
```

This has one public method, getRegistrars() which returns an array of ServiceRegistrar objects. Each one of these implements the ServiceRegistrar interface, just like the object returned from a unicast search for a lookup service. More than one can be returned if a set of replies have come in before the listener was registered - they are collected in an array and returned in a single call to the listener. A UML sequence diagram augmented with jagged arrows showing the network broadcast and replies is shown in figure .

In this figure, creation of a LookupDiscovery object starts the broadcast search, and it returns the discover object. The MulticastRegister adds itself as listener to the discover object. The search continues in a separate thread, and when a new lookup service replies, the discover object invokes the discovered() method in the MulticastRegister, passing it a newly created DiscoveryEvent. The MulticastRegister object can then make calls on the DiscoveryEvent such as getRegistrars(), which will return suitable ServiceRegistrar objects.

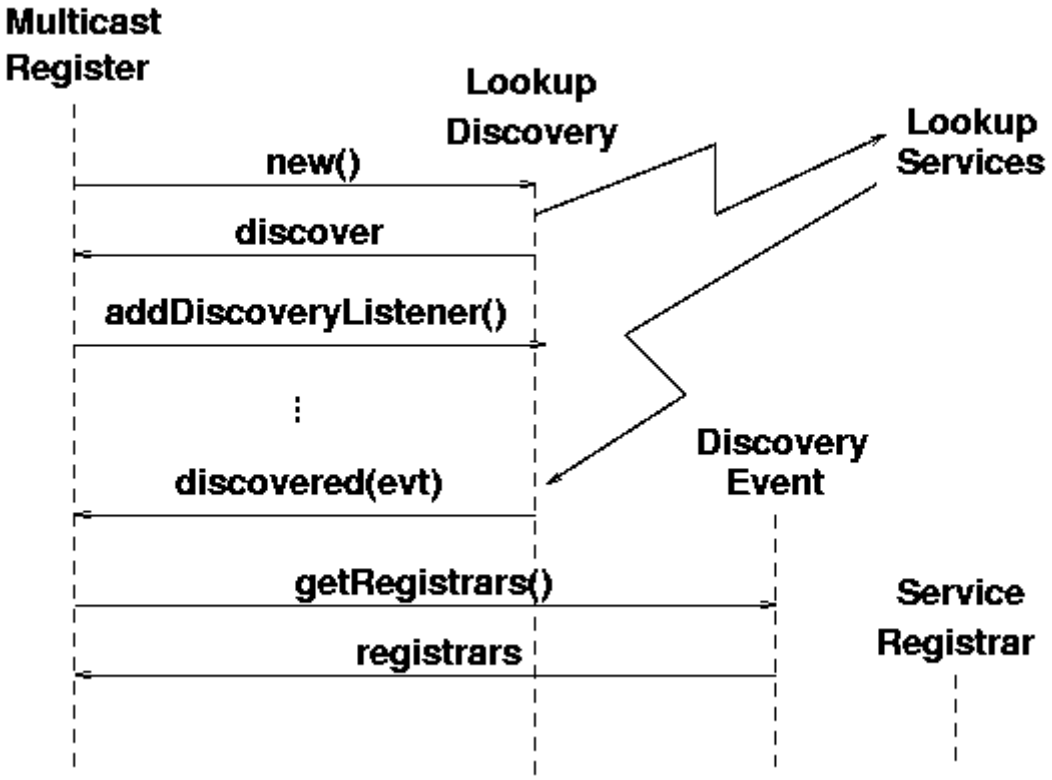


Figure 2.3: UML sequence diagram for discovery

Again, the registrar object will be used in different ways for clients and services: the services will use it to register themselves, and the clients will use it to locate services.

Chapter 3

Entry Objects and Service Registration

3. Entry Objects

Entries are used to pass additional information about services that clients can use to decide if a particular service is what it wants. The primary intention of entries is to provide extra information about services so that clients can decide whether or not they are the services they want to use.

3.1 Entry class

The Entry class allows services to advertise their capabilities in very flexible ways. When a service provider registers a service, it places a copy of the service object (or a service proxy) on the lookup service. This copy is an *instance* of an object, albeit in serialised form. The server can optionally register sets of attributes along with the service object. Each set is given by an instance of a *type* or *class*. So what is stored on each service locator is an instance of a class along with a set of attribute entries.

A service can announce a number of entry attributes when it registers itself with a lookup service. It does so by preparing an array of Entry objects and passing them into the ServiceItem used in the register() method of the registrar. The service can include as much as it wants to in this: in later searches by clients each entry is treated as though it was *or*'ed with the other entries. In other words, the more entries that are given by the service, the greater the chance of matching a client's requirements.

For example, we have a coffee machine on the 7th level of our building, which is known as both ``GP South Building'' and ``General Purpose South Building''. Information such as this, and general stuff about the

coffee machine can be encapsulated in the convenience classes Location and Comment from the net.jini.lookup.entry package. If this was on our network as a service, it would advertise itself as :

```
import net.jini.lookup.entry.Location;
import net.jini.lookup.entry.Comment;

Location loc1 = new Location("7", "728",  "GP South Building");
Location loc2 = new Location("7", "728",  “General Purpose South Building");
Comment comment = new Comment("DSTC coffee machine");

Entry[] entries = new Entry[] {loc1, loc2, comment};

ServiceItem item = new ServiceItem(..., ..., entries);
registrar.register(item, ...);
```

3.2 Service Registration

A **service** is a logical concept such as a blender, a chat service, a disk. It will turn out to be usually defined by a Java interface, and commonly the service itself will be identified by this interface. Each service can be implemented in many ways, by many different vendors. For example, there may be Kami's dating service, Mary's dating service or many others. What makes them the ``same'' service is that they implement the same interface; what distinguishes one from another is that each different implementation uses a different set of objects (or maybe just one object) belonging to different classes.

A service is created by a service provider. A service provider plays a number of roles:

- It creates the objects that implement the service

- It registers one of these - the *service object* with lookup services. The service object is the ``publically visible'' part of the service, and will be downloaded to clients

- It stays alive in a server role, performing various tasks such as keeping the service ``alive''.

In order for the service provider to register the service object with a lookup service, the server must first **find** the lookup service. This can be done in two ways: if the location of the lookup service is known, then the service provider can use unicast TCP to connect directly to it. If the location is not known, the service provider will make UDP multicast requests, and lookup services may respond to these requests. Lookup services will be listening on port 4160 for both the unicast and multicast requests. (4160 is the decimal representation of hexadecimal (CAFE - BABE). Oh well, these numbers have to come from somewhere.) When the lookup service gets a request on this port, it sends an **object** back to the server. This object, known as a **registrar**, acts as a proxy to the lookup service, and runs in the service's JVM (Java Virtual Machine). Any requests that the service provider needs to make of the lookup service are made through this proxy registrar. Any suitable protocol may be used to do this, but in practice the implementations that you get of the lookup service (e.g from Sun) will probably use RMI.

What the service provider does with the registrar is to *register* the service with the lookup service. This involves taking a copy of the service object, and storing it on the lookup service as in figures below:

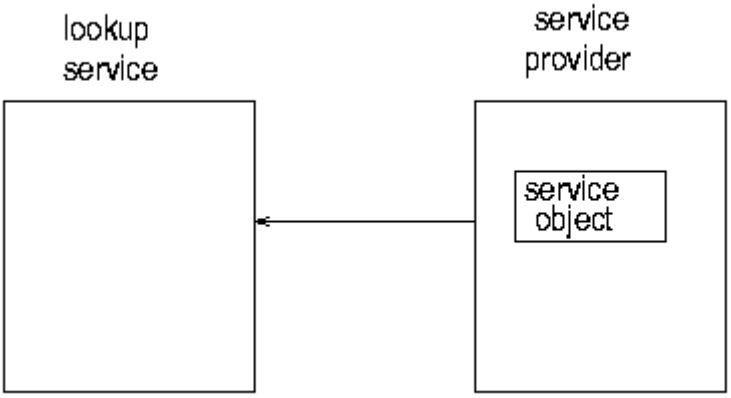


Figure 3.1: Querying for a service locator

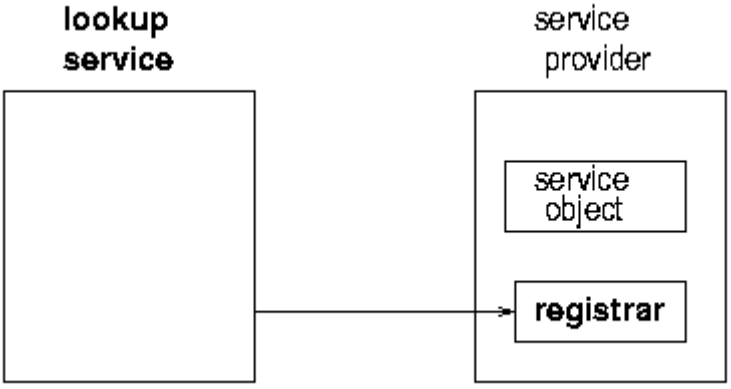


Figure 3.2: Registrar returned

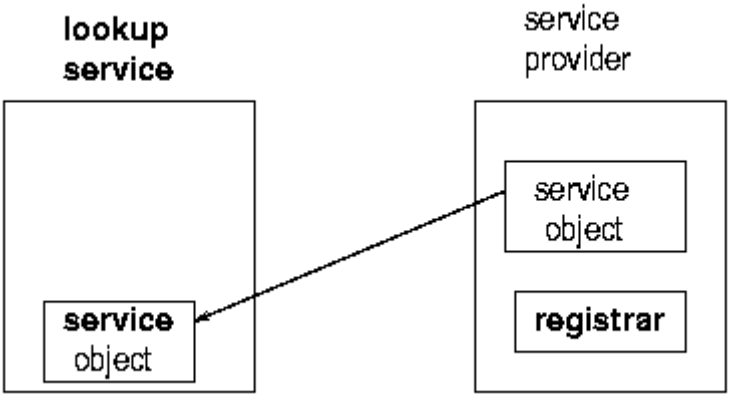


Figure 3.3: Service uploaded

3.3 A word about the ServiceRegistrar

A server for a service finds a service locator using unicast lookup with a LookupLocator or multicast search using LookupDiscovery. In both cases, a ServiceRegistrar object is returned to act as a proxy for the lookup service. The server then registers the service with the service locator using the ServiceRegistrar method register():

```
package net.jini.core.lookup;

public Class ServiceRegistrar
{
public ServiceRegistration register(ServiceItem item,long
leaseDuration)
        throws java.rmi.RemoteException;
}
```

The second parameter here is a request for the length of time (in milliseconds) the lookup service will keep the service registered. This request need not be honored: the lookup service may reject it completely, or only grant a lesser time interval. This is discussed in the section on leases. The first parameter is of type

```
package net.jini.core.lookup;

public Class ServiceItem
{
    public ServiceID serviceID;
    public java.lang.Object service;
    public Entry[] attributeSets;

     public ServiceItem(ServiceID serviceID,  java.lang.Object
     service,
```

```
        Entry[] attrSets);
    }
```

3.4 ServiceItem

The service provider will create a ServiceItem object, using the constructor and pass it into register(). The serviceID is set to null when the service is registered for the first time. The lookup service will set a non-null value as it registers the service. On subsequent registrations or re-registrations, this non-null value should be used. The serviceID is used as a globally unique identifier for the service.

The second parameter is the service object that is being registered. This object will be serialised and sent to the service locator for storage. When a client later requests a service, this is the object it will be given. There are several things to note about the service object:

- The object must be serialisable. Some objects, such as Swing's JTextArea are not serialisable at present and so cannot be used.
- The object is created in the service's JVM. However, when it runs it will do so in the client's JVM. It may need to be a proxy for the actual service. For example, it may be able to show a set of toaster controls, but will have to send messages across the network to the real toaster service, which is connected to the physical toaster.
- If the service object is an RMI proxy, then the object in the ServiceItem is given by the programmer as the UnicastRemoteObject for the proxy stub, not the proxy itself. The Java runtime substitutes the proxy. This subtlety is explored in a later chapter.

The third parameter is a set of entries giving information about the service in addition to the service object/service proxy itself. If there is no additional information, this can be null.

3.5 Registration

The service attempts to register itself by calling register(). This may throw an java.rmi.RemoteException which must be caught. The second parameter is a request to the service locator for the length of time to store the service. The time requested may or may not be honoured. The return value is of type ServiceRegistration

3.5.1 ServiceRegistration

This registration object is created by the lookup service and is returned to run in the service provider. The registration acts as a proxy object to control the state maintained about the exported service object stored on the lookup service. Actually, this can be used to make changes to the entire ServiceItem stored on the lookup service. The registration maintains a field serviceID which is used to identify the ServiceItem on the lookup service. This can be retrieved by getServiceID() for reuse by the server if it needs to do so (which it should). These objects are shown in figure .

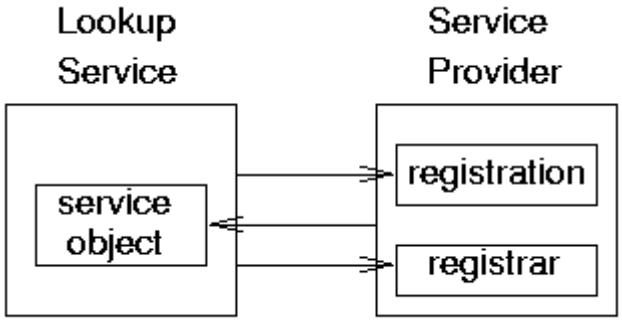


Figure 3.4: Objects in service registration

Other methods such as

```
void addAttributes(Entry[] attrSets);
void modifyAttributes(Entry[] attrSetTemplates, Entry[] attrSets);
```

void setAttributes(Entry[] attrSets);

can be used to change the entry attributes stored on the lookup service.

The final public method for this class is getLease() which returns a Lease object, which allows renewal or cancellation of the lease. This is discussed in more detail in the section of leases.

The major task of the server is then over. It will have successfully exported the service to a number of lookup services. What the server then does depends on how long it needs to keep the service alive or registered. If the exported service can do everything that the service needs to do, and does not need to maintain long-term registration, then the server can simply exit. More commonly, if the exported service object acts as a proxy and needs to communicate back to the service then the server can sleep so that it maintains existence of the service. If the service needs to be re-registered before timeout occurs then the server can also sleep in this situation.

3.6 Running the Unicast Server

If you write a unicast server that just exports its service to the lookup service and does nothing, then you need to compile and run your program/server with jini-core.jar in its CLASSPATH. When run, it will attempt to connect to the service locator, so obviously one needs to be running on the machine specified in order for this to happen. Otherwise, it will throw an exception and terminate.

The instance data for the service object is transferred in serialized form across socket connections. This instance data is kept in this serialized form by the lookup services. Later, when a client asks for the service to be reconstituted, it will use this instance data and also will need the class

files. At this point, the class files will also need to be transferred, probably by an HTTP server. There is no need for additional RMI support services such as rmiregistry or rmid since all registration is done by the method register().

3.6.1 Information from the ServiceRegistration

The ServiceRegistrar object is used to register() the service, and in doing so returns a ServiceRegistration object. This can be used to give information about the registration itself. The relevant methods are

```
ServiceID getServiceID();
Lease getLease();a
```

The service id can be stored by the application if it is going to re-register again later. The lease object can be used to control the lease granted by the lookup locator, and will be discussed in more detail in the chapter on Leases.

3.6.2 Service ID

A service is unique in the world. It runs on a particular machine and performs certain tasks. However, it will probably register itself with many lookup services. It should have the same ``identity'' on all of these. In addition if either the service or one of these locators crashes or restarts, then this identity should be the same as before.

The ServiceID plays the role of unique identifier for a service. It is a 128-bit number generated in a pseudo-random manner, and is effectively unique: the chance that the generator might duplicate this number is vanishingly small. Services do not generate this identifier, as the actual algorithm is not a public method of any class. Instead, a lookup service should be used. When a service

needs a new identifier, it should register with a lookup service using a null service id. The lookup service will then return a value.

The first time a service starts, it should ask for a service id from the first lookup service it registers with. It should reuse this for registration with every other lookup service from then on. If it crashes and restarts, then it should use the same service id again, which implies it should save it in persistent storage and retrieve it on restarting.

Chapter 4

The Client Lookup and Leasing

This chapter looks at what the client has to do once it has found a service locator (Lookup Service) and wishes to find a service.

4. Client Operations

4.1 Client Lookup

The client on the other hand, is trying to get a copy of the service into its own JVM. It goes through the same mechanism to get a registrar from the lookup service. It uses this to search for a service stored on that lookup service using the lookup() method:

```
public Class ServiceRegistrar
{
   public java.lang.Object lookup(ServiceTemplate tmpl)
            throws java.rmi.RemoteException;
public ServiceMatches lookup(ServiceTemplate tmpl, int
maxMatches)
            throws java.rmi.RemoteException;
}
```

The first of these methods just finds a service that matches the request. The second finds a set (upto the maxMatches) requested. If a client wishes to search for more than one match to a service request from a particular lookup service, then it specifies the maximum number of matches it would like returned by the maxMatches parameter of the second for lookup() method. It gets back a ServiceMatches object.

```
package net.jini.core.lookup;
```

```
public Class ServiceMatches
{
    public ServiceItem[] items;
    public int totalMatches ;
}
```

The number of elements in items need not be the same as totalMatches. Suppose there are five matching services stored on the locator. Then totalMatches will be set to five after a lookup. However, if you only specified to search for at most *two* matches, then items will be set to be an array with only two elements.

So, we were talking about that the client goes through the same mechanism(i.e., trying to get a copy of the service into its own JVM) to get a registrar from the lookup service, but this time it does something different with this, which is to request the service object to be copied across to it as shown below:

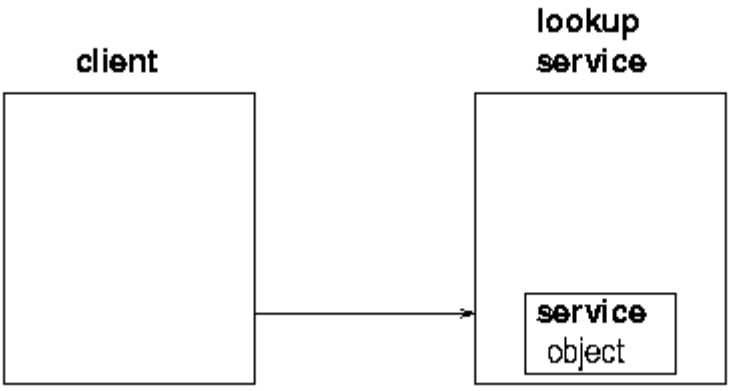


Figure 4.1: Querying for a service locator

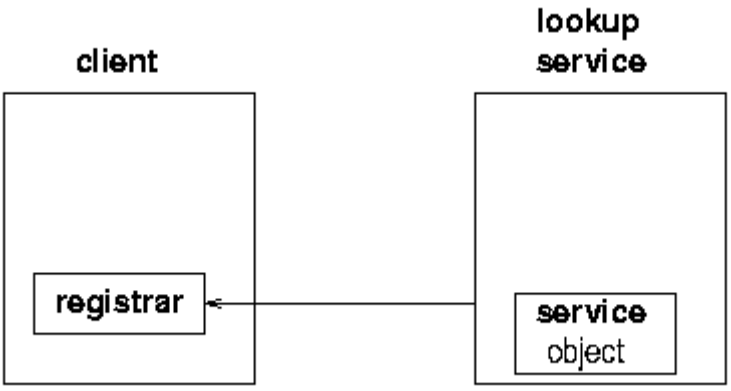


Figure 4.2: Registrar returned

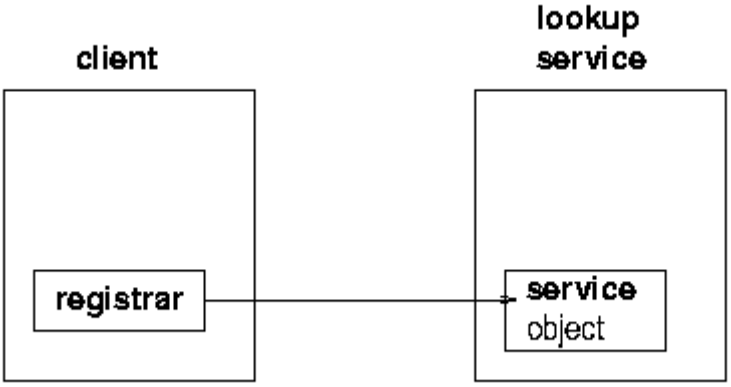


Figure 4.3: Asking for a service

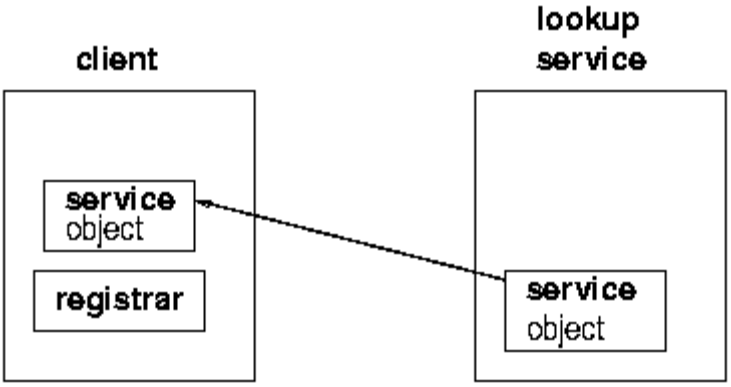


Figure 4.4: Service returned

At this stage there is the original service object running back on its host. There is a copy of the service object stored in the lookup service, and

there is a copy of the service object running in the client's JVM. The client can make requests of the service object running in its own JVM.

4.2 Proxies

Some services can be implemented by a single object, the service object. How does this work if the service is actually a toaster, a printer, or controlling some piece of hardware? By the time the service object runs in the client's JVM, it may be a long way away from its hardware. It cannot control this remote piece of hardware all by itself. In this case, the implementation of the service must be made up of at least *two* objects, one running in the client and another distinct one running in the service provider.

The service object is really a *proxy*, which will communicate back to other objects in the service provider, probably using RMI. The proxy is the part of the service that is visible to clients, but its function will be to pass method calls back to the rest of the objects that form the total implementation of the service. There isn't a standard nomenclature for these server-side implementation objects. I shall refer to them in this book as the ``service backend'' objects.

The motivation for discussing proxies is when a service object needs to control a remote piece of hardware that is not directly accessible to the service object. However, it need not be hardware: there could be files accessible to the service provider that are not available to objects running in clients. There could be applications local to the service provider that are useful in implementing the service. Or it could simply be easier to program the service in ways that involve objects on the service provider, with the service object being just a proxy. The majority of service implementations end up with the service object being just a proxy to service backend objects, and it is quite common to see the service object being referred to

as a *service proxy*. It is sometimes referred to a the service proxy even if the implementation doesn't use a proxy at all!

The proxy needs to communicate with other objects in the service provider. It appears we have a chicken-and-egg situation: how does the proxy find the service backend objects in its service provider? Use a JINI lookup? No, when the proxy is created it is ``primed'' with its own service provider's location so that when run it can find its own ``home''. This will appear as in figure :

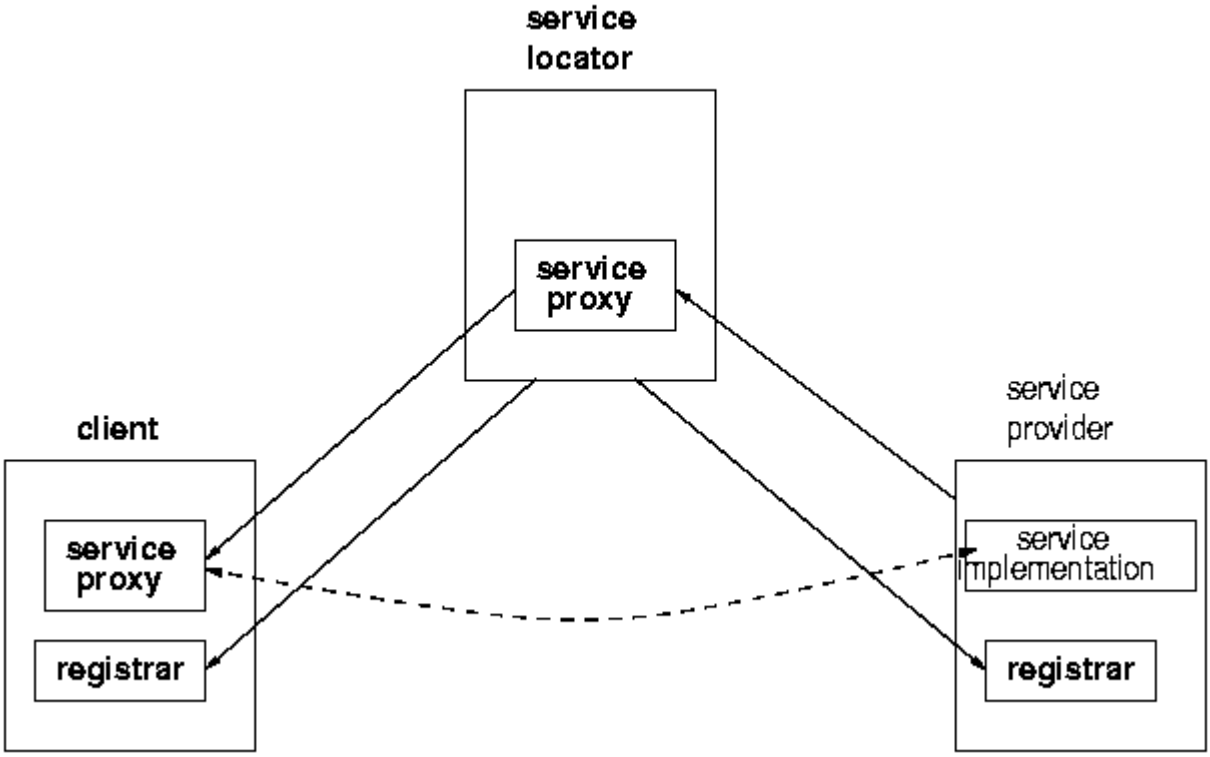


Figure 4.5: A proxy service

4.3 Support Services

The three components of a JINI system are clients, services and service locators, which may run anywhere on the network. These will be

implemented using Java code running in Java Virtual Machines (JVM). The implementation may be in pure Java but it could make use of native code by JNI (Java Native Interface) or make external calls to other applications. Often, each of these applications will run in its own JVM on its own computer, though they could run on the same machine or even share the same JVM. When they run, they will need access to Java class files, just like any other Java application. Each component will use the CLASSPATH environment variable or use the classpath option to the runtime to locate the classes it needs to run.

However, JINI also relies heavily on the ability to move objects across the network, from one JVM to another. In order to do this, particular implementations must make use of support services such as RMI daemons and HTTP (or other) servers. The particular support services required depend on implementation details, and so may vary from one JINI component to another.

4.4 The Concept of Leasing

The network is a dynamic environment. Devices are attached and unattached frequently, and more permanent services often become temporarily unavailable because of crashes or other problems.

JINI™ technology helps clients locate and use the services that are available at the time they are needed, but it doesn't stop there. JINI technology also facilitates in cleaning up after devices are disconnected or services vanish unexpectedly.

One of the central concepts in JINI technology is the concept of a **lease**. Many aspects of the relationships between JINI technology clients, services, and lookup services are lease-based. A lease in the framework of JINI technology is something like the "lease" you get when you drop a quarter into a parking meter. You drop in a quarter, and you get to use the

parking place for 15 minutes. If you return to the parking meter 14 minutes later, you have the option of "renewing the lease" by dropping in another quarter.

Leasing is the mechanism used between applications to give access to resources over a period of time in an agreed manner. Leases are requested for a period of time. In distributed applications, there may be partial failures of the network or of components on this network. Leasing is a way for components to register that they are alive, but for them to be ``timed out'' if they have failed, are unreachable, *etc*. In JINI, one use of leasing is for a service to request that a copy be kept on a lookup service for a certain length of time for delivery to clients on request. The service requests a time in the ServiceRegistrar's method register(). Two special values of the time are :

4.4.1.1 Lease.ANY - the service lets the lookup service decide on the time

4.4.1.2 Lease.FOREVER - the request is for a lease that never expires

The lookup service acts as the granter of the lease, and decides how long it will actually create the lease for. (The lookup service from Sun typically sets the lease time as only five minutes.) Once it has done that, it will attempt to ensure that the request is honoured for that period of time. The lease is returned to the service, and is accessible through the method getLease() of the ServiceRegistration object. These objects are shown in figure

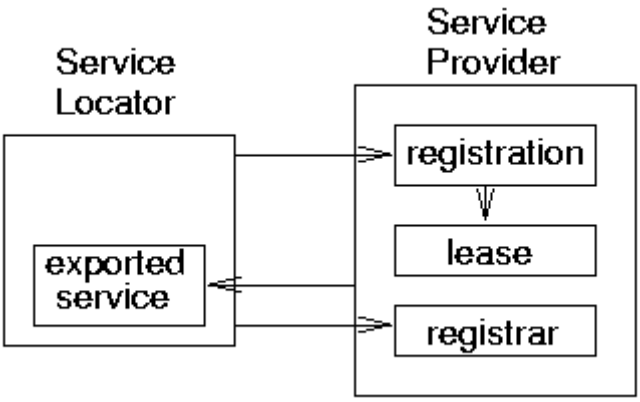


Figure 4.6: Objects in a leased system

```
ServiceRegistration reg = registrar.register();
Lease lease = reg.getLease();
```

The principal methods of the Lease object are

- void cancel() throws
  UnknownLeaseException,
  java.rmi.RemoteException;
- long getExpiration();
- void renew(long duration) throws
  LeaseDeniedException,
  UnknownLeaseException,
  java.rmi.RemoteException;

The expiration value from getExpiration() is the time in milliseconds since the beginning of the epoch (the same as in System.currentTimeMillis()). To find the amount of time still remaining from the present, the current time can be subtracted from this:

```
long duration = lease.getExpiration() - System.currentTimeMillis();
```

4.4.1 Cancellation

A service can cancel its lease by using cancel(). The lease communicates back to the lease management system on the lookup service which cancels storage of the service.

4.4.2 Expiration

When a lease expires, it does so silently. That is, the lease granter (the lookup service) will not inform the lease holder (the service) that it has expired. It is upto the service to call renew() before the lease expires if it wishes the lease to continue the service in milliseconds. Generally leases will be renewed and the manager will function quietly. However, the lookup service may decide not to renew a lease and will cause an exception to be thrown. The method renewUntil() can use Lease.FOREVER with no problems.

4.7 Discovery Management

Both services and clients need to find lookup locators. Services will register with these locators, and clients will query them for suitable services. Finding these lookup locators involves three components:

- A list of LookupLocators for unicast discovery
- A list of groups for lookup locators using multicast discovery
- Listeners whose methods are invoked when a service locator is found

``Discovering a Lookup Service'' considers the cases of a single unicast lookup service, or a set of multicast lookup services. This was all that was available in JINI 1.0. In JINI 1.1, this has been extended to handle a *set* of

unicast lookup services *and* a set of multicast lookup services. JINI 1.1 Helper Utilities defines three interfaces :

- DiscoveryManagement which looks after discovery events
- DiscoveryGroupManagement which looks after groups and multicast search
- DiscoveryLocatorManagement which looks after unicast discovery

Different classes may implement different combinations of these three interfaces.

4.8 Join Manager

A service needs to locate lookup services and register the service with them. Locating services can be done using the utility classes of ``Discovery Management'' as mentioned above. As each lookup service is discovered, it then needs to be registered, and the lease maintained. The class JoinManager performs all of these tasks. There are two constructors:

```
public class JoinManager
{
   public JoinManager(Object obj,
                      Entry[] attrSets,
                      ServiceIDListener callback,
                      DiscoveryManagement discoverMgr,
                      LeaseRenewalManager leaseMgr)
                            throws IOException;

  public JoinManager(Object obj,
                      Entry[] attrSets,
```

```
        ServiceID serviceID,
        DiscoveryManagement discoverMgr,
        LeaseRenewalManager leaseMgr)
        throws IOException;
    }
```

The first of these is when the service is new and does not have a service id. A ServiceIDListener can be added which can note and save the id. The second form is used when the service already has an id.

In short, A JoinManager can be used by a server to simplify many of the aspects of locating lookup services, registering one or more services and renewing leases for them.

Chapter 5

The Security Issue

5. Security

5.1 Security Problem

Security plays an important role in distributed systems. Security for JINI is based on the JDK 1.2 security model. This makes use of a SecurityManager to grant or deny access to resources. A few of the examples may work fine without a security manager. Most will require an appropriate security manager in place or RMI will be unable to download class files. Installing a suitable manager may be done by

```
System.setSecurityManager(new RMISecurityManager());
```

This should be done before any network-related calls.

The security manager will need to make use of a security policy. This is typically given in policy files which are in default locations or are specified to the Java runtime. If policy.all is a policy file in the current directory, then invoking the runtime by

```
java -Djava.security.policy="policy.all" ...
```

will load the contents of the policy file.

5.2 Rmid and JDK 1.3

The security problems of the last section have been partly addressed by a tighter security mechanism introduced in JDK 1.3. These restrict what activatable services can do by using a security mechanism that is under

the control of whoever starts rmid. This means that there has to be cooperation between whoever starts rmid and the person who starts an activatable service that will use rmid.

The simplest mechanism is to just turn the new security system *off*. This means running rmid with an additional argument

```
rmid -J-Dsun.rmi.activation.execPolicy=none
```

All that rmid then checks is that any activatable service that registers with it is started on the same machine as rmid. This is the weak security mechanism in the JDK 1.2 version of rmid, that assumes that users on the same machine are co-operative.

The default new mechanism can also be set explicitly

```
rmid -J-Dsun.rmi.activation.execPolicy=default
```

This requires an additional security policy file that will be used by rmid, and the location of this policy file is also given on the command line for rmid. For example, the following command will start rmid using the new default mechanism with the policy file set to /jini1_1/rmid.policy

```
rmid -J-Djava.security.policy=c:\jini1_1\rmid.policy
```

The policy file used by rmid is a standard JDK 1.2 policy file and grants permissions to do various things. For rmid, the main permission that has to be granted is to use the various options to the activation commands. Granting option permissions is done using the com.sun.rmi.rmid.ExecOptionPermission permission.

For example, is an activatable service. To run this on my system, I used the command

```
java -jar c:\jini\lib\.jar
http://myMachineName:8080\-dl.jar
c:\\jini\policy\policy.all
c:\jini\_log
public
```

To run this with the JDK 1.3 rmid I need to place the following in the security policy file:

```
-Djava.security.policy=c:\jini\policy\policy.all
```

Chapter 6

6. Project Specifications

6.1 Statement

The CMS experiment in 2005 although is based on the discovery of basic science, but resultantly it has pushed technology to its limits. Part of the technical support for this experiment relies heavily on the provision of robust networks that can support a host of services. The most glaring disadvantage of this architecture is that the failure of the network fails the service. As an answer, the java community came up with the JINI concept. This involves distributed services residing at different server stations, providing usage facilities to the clients. This use is not restricted to the geographical location of the service. Regardless of where the service resides, it shall be represented by its proxy in the lookup service (in itself a JINI service).

The project was to write a JINI service manager. Within the service manager, I had to write JINI based services. The service would then be registered with a lookup service. Its proxy will reside in the lookup service and is going to be accessed by the mobile agent (client). The agent/client would then access the lookup service in order to use the services offered by this service. The service in turn fulfils the required task and gives the result back to the agent/client.

For practical purposes, I have written a client as well. This client performs the same functions as the mobile agent had to. This all would be demonstrated on a LAN.

Following are the services, which I have written within the Service Manager.

- **Data Repository Service**. The service accesses the database in order to retrieve any data and gives the result back to the client.

- **Printing Service**. This service prints the given text on the server default printer in the given font and according to the given width and height of the paper (PageFormat object). This service provides only black and white text printing.

6.2 Development Environment

The following version of Kawa was chosen as the development environment:

- KAWA 4.01

6.3 Development Languages

JINI is based on java, and following versions of JINI and JAVA have been used:

- JINI 1.1
- JDK 1.3

6.4 Project Logic

The three main entities i.e., a client, a service provider, which provides service, and a lookup service have to be there in any JINI project. They stay as they are, but I have incorporated a slight change in the design of the system. Instead of one service registered separately with the lookup service, slight design modifications have been introduced which are far more efficient and portable:

- The main, class i.e. MyServer, is the controlling class. After running the HTTP server and rmid with due security policy, once the lookup service () is operational in all respects, MyServer class is run. It detects and returns a list of all available services on the network and tries to register them with the lookup service (LUS). After registering the services, it again returns a list of all those services that have been registered with the lookup-service. This way, the client can come to know which all services it can access and use.

- The system has been made dynamic. By dynamic, I mean that there is no restriction on the number of services that have to be added. The system provides the flexibility for adding any number of services.

The client wishing to access the printing service, accesses the LUS for this. This printing service prints the given text on the server default printer in the given font and according to the given width and height of the paper (PageFormat object).

If a client wishes to make use of the Data Repository Service, it looks up at the available list of services, chooses the desired Data Repository Service, and sends a query to the appropriate database. The service in return, returns the desired information.

6.5 Services Description

6.5.1 My Server class

6.5.1.1 GUI creation

This is an essential portion of the whole JINI project. This is just to make it visible to every one the number of available services on the network and to make an attempt to get those services registered with the lookup service. The client can thereafter have a clear view of all the registered services it can make use of.

6.5.1.2 Listing services

The method listServices() does a lot of work. First of all, it checks to see the number of services residing at a particular place, it registers those with the lookup service. This is the technique that has been implemented throughout this project. For adding more services dynamically to the registering services, a standard, of naming the services in a manner that their names end with “Service”, has to be followed. Like “DataRepositoryService” and “PritingService”.

When the server starts, the client/user first sees all the services that have been found and then registered. At the end, the result is published indicating the total number of services that were registered with the lookup service by the method registerService().

6.5.1.3 Registering Services

The method registerService() takes an object and a string as input parameters. This is the main portion, which basically registers the services with the lookup service. First of all, we set the security manager because as it has already been mentioned earlier that setting the security manager is the foremost important thing. So, we set the security manager to allow the RMI class loader to go to the codebase for classes that are not available

Locally. We do so as:

System.setSecurityManager (new RMISecurityManager ());

Then we create the attributes (an array of entry objects) that gives the additional information about this server and use it to register this server with the lookup service. It is done with the help of JoinManager. As mentioned earlier, a service needs to locate services and register them with it. Locating services can be done using the utility classes of “Discovery Management''. As each lookup service is discovered, it then needs to be registered, and the lease maintained. The class JoinManager performs all of these tasks.

JoinManager() finds and registers with the lookup service.

```
aeAttributes = new Entry[1];
aeAttributes[0] = new Name(name);
JoinManager joinmanager;
```

```
joinmanager = new JoinManager(o,aeAttributes,
(ServiceIDListener)o, new LeaseRenewalManager());
System.out.println("MyServer: JoinManager = " +
joinmanager);
```

Then the process of finding the JINI lookup service () is carried out.

```
int iPort;
String sHost;
lookup = new LookupLocator ("jini://localhost");
sHost = lookup.getHost ();
iPort = lookup.getPort ();
```

The location of the host and the port can also be printed as well, but as such there is no need for that.

Thereafter, we get the lookup service's ServiceRegistrar (the class by which interaction with the lookup service is possible).

```
ServiceID id;
ServiceRegistrar registrar;
registrar = lookup.getRegistrar();
id = registrar.getServiceID ();
```

In the main, we also set the properties using method setProperties().

This is how a service is listed and registered.

6.5.2 Data Repository Service

The DataRepositoryService extends UnicastRemoteObject implements DataRepositoryInterface, Serializable, ServiceIDListener. It has a method execQuery() of class Vector that is accessed by the client. The client performs a search on the lookup server to find the service that has the name attribute of "MyServer". The lookup service returns an interface object to the service. On this interface

object the client accesses the method execQuery() to retrieve the desired data from the appropriate data base. This method contains the Java Data Base Connectivity part.

6.5.2.1 Java Data Base Connectivity Part

Introduction

The JDBC 2.0 API includes two packages:

- Java.sql package--the JDBC 2.0 core API
    - JDBC API included in the JDK™ 1.1 release (previously called JDBC 1.2). This API is compatible with any driver that uses JDBC technology.
    - JDBC API included in the Java 2 SDK, Standard Edition, version 1.2 (called the JDBC 2.0 core API). This API includes the JDBC 1.2 API and adds many new features.
- java.sql package--the JDBC 2.0 Optional Package API. This package extends the functionality of the JDBC API from a client-side API to a server-side API, and it is an essential part of Java™ 2 SDK, Enterprise Edition technology.

Being an Optional Package, it is not included in the Java 2 Platform SDK, Standard Edition, version 1.2, but it is readily available from various sources.

SQL Package:

The java.sql package contains API for the following:

- Making a connection with a data source

    - DriverManager class
    - Driver interface
    - DriverPropertyInfo class
    - Connection interface
- Sending SQL statements to a database
    - Statement interface for sending basic SQL statements
    - PreparedStatement interface for sending prepared statements or basic SQL statements (derived from Statement)
    - CallableStatement interface for calling database stored procedures (derived from PreparedStatement)
- Retrieving and updating the results of a query
    - ResultSet interface
- Mapping an SQL value to the standard mapping in the Java programming language
    - Array interface
    - Blob interface
    - Clob interface
    - Date class
    - Ref interface
    - Struct interface
    - Time class
    - Timestamp class
    - Types class
- Custom mapping an SQL user-defined type to a class in the Java programming language
    - SQLData interface
    - SQLInput interface
    - SQLOutput interface

- Providing information about the database and the columns of a ResultSet object
    - DatabaseMetaData interface
    - ResultSetMetaData interface
- Throwing exceptions
    - SQLException thrown by most methods when there is a problem accessing data and by some methods for other reasons
    - SQLWarning thrown to indicate a warning
    - DataTruncation thrown to indicate that data may have been truncated
    - BatchUpdateException thrown to indicate that not all commands in a batch update executed successfully
- Providing security
    - SQLPermission interface

JDBC Requirements

Your driver should include instructions for installing it. For JDBC drivers written for specific DBMSs, installation consists of just copying the driver onto your machine; there is no special configuration needed. The JDBC-ODBC Bridge driver is not quite as easy to set up. If you download either the Solaris or Windows versions of JDK1.1, you will automatically get the JDBC-ODBC Bridge driver, which does not itself require any special configuration. ODBC, however, does. If you do not already have ODBC on your machine, you will need to see your ODBC driver vendor for information on installation and configuration.

If you do not already have a DBMS installed, you will need to follow the vendor's instructions for installation. Most users will have a DBMS installed and will be working with an established database.

Establishing a Connection

The first thing you need to do is establish a connection with the DBMS you want to use. This involves two steps: (1) loading the driver and (2) making the connection.

Loading Drivers

Loading the driver or drivers you want to use is very simple and involves just one line of code. If, for example, you want to use the JDBC-ODBC Bridge driver, the following code will load it:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Your driver documentation will give you the class name to use. For instance, if the class name is jdbc.DriverXYZ , you would load the driver with the following line of code:

```
Class.forName("jdbc.DriverXYZ");
```

You do not need to create an instance of a driver and register it with the DriverManager because calling Class.forName will do that for you automatically. If you were to create your own instance, you would be creating an unnecessary duplicate, but it would do no harm.

When you have loaded a driver, it is available for making a connection with a DBMS.

Making the Connection

The second step in establishing a connection is to have the appropriate driver connect to the DBMS. The following line of code illustrates the general idea:

```
Connection con = DriverManager.getConnection(url,
    "myLogin", "myPassword");
```

This step is also simple, with the hardest thing being what to supply for url . If you are using the JDBC-ODBC Bridge driver, the JDBC URL will start with jdbc:odbc: . The rest of the URL is generally your data source name or database system. So, if you are using ODBC to access an ODBC data source called " Testdb, " for example, your JDBC URL could be jdbc:odbc:Testdb . In place of " myLogin " you put the name you use to log in to the DBMS; in place of " myPassword " you put your password for the DBMS. So if you log in to your DBMS with a login name of " Fernanda " and a password of " J8, " just these two lines of code will establish a connection:

```
String url = "jdbc:odbc: Testdb ";
Connection con = DriverManager.getConnection(url,
"Fernanda", "J8");
```

If you are using a JDBC driver developed by a third party, the documentation will tell you what subprotocol to use, that is, what to put after jdbc: in the JDBC URL. For example, if the driver developer has registered the name acme as the subprotocol, the first and second parts of the JDBC URL will be jdbc:acme: . The driver documentation will also give you guidelines for the rest of the JDBC URL. This last part of the

JDBC URL supplies information for identifying the data source.

If one of the drivers you loaded recognizes the JDBC URL supplied to the method DriverManager.getConnection , that driver will establish a connection to the DBMS specified in the JDBC URL. The DriverManager class, true to its name, manages all of the details of establishing the connection for you behind the scenes. Unless you are writing a driver, you will probably never use any of the methods in the interface Driver , and the only DriverManager method you really need to know is DriverManager.getConnection .

The connection returned by the method DriverManager.getConnection is an open connection you can use to create JDBC statements that pass your SQL statements to the DBMS.

Interface Connection

A connection (session) with a specific database. Within the context of a Connection, SQL statements are executed and results are returned.

A Connection's database is able to provide information describing its tables, its supported SQL grammar, its stored procedures, the capabilities of this connection, and so on. This information is obtained with the getMetaData method.

Interface Statement

The object used for executing a static SQL statement and obtaining the results produced by it.

Only one ResultSet object per Statement object can be open at any point in time. Therefore, if the reading of one ResultSet object is interleaved with the reading of another, each must have been generated by different Statement objects. All statement execute methods implicitly close a statment's current ResultSet object if an open one exists

Interface ResultSet

A table of data representing a database result set, which is usually generated by executing a statement that queries the database.

A ResultSet object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The next method moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set.

A default ResultSet object is not updatable and has a cursor that moves forward only. Thus, it is possible to iterate through it only once and only from the first row to the last row. New methods in the JDBC 2.0 API make it possible to produce ResultSet objects that are scrollable and/or updatable. The following code fragment, in which con is a valid Connection object, illustrates how to make a result set that is scrollable and insensitive to updates by others, and that is updatable. See ResultSet fields for other options.

```
Statement stmt = con.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

```
ResultSet rs = stmt.executeQuery("SELECT a, b
FROM TABLE2");
 // rs will be scrollable, will not show changes made by
others,
  // and will be updatable
```

Interface ResultSetMetaData

An object that can be used to get information about the types and properties of the columns in a ResultSet object. The following code fragment creates the ResultSet object rs, creates the ResultSetMetaData object rsmd, and uses rsmd to find out how many columns rs has and whether the first column in rs can be used in a WHERE clause.

```
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM
TABLE2");
ResultSetMetaData rsmd = rs.getMetaData();
int numberOfColumns = rsmd.getColumnCount();
boolean b = rsmd.isSearchable(1);
```

6.5.3 My Client class (for accessing data repository service)

The class MyClient is basically meant for the Data Reposiroty Service. In this, we, first of all set the security manager. So, we set the security manager to allow the RMI class loader to go to the codebase for classes that are not available
Locally. We do so as:

System.setSecurityManager (new RMISecurityManager ());

6.5.3.1 Finding the JINI Lookup service ()

Next, we find the JINI lookup service () and print its location.

```
lookup = new LookupLocator ("jini://localhost");
```

```
sHost = lookup.getHost ();
iPort = lookup.getPort ();
System.out.println ();
System.out.println ("client: LookupLocator = " + lookup);
System.out.println ("client: LookupLocator.host = " + sHost);
System.out.println ("client: LookupLocator.port = " + iPort);
```

6.5.3.2 Getting the lookup service's Service Registrar

Then, we get the lookup service's ServiceRegistrar (the class by which interaction with the lookup service is possible) and also print it.

```
registrar = lookup.getRegistrar ();
id = registrar.getServiceID ();
System.out.println ("client: ServiceRegistrar = " + registrar);
System.out.println ("client: ServiceID = " + id);
```

6.5.3.3 Accessing the data repository service

Therefafter, we perform a search on the lookup server to find the service that has the name attribute of "MyServer". The lookup service returns an interface object to the service.

```
aeAttributes = new Entry[1];
aeAttributes[0] = new Name ("DataRepositoryService");
template = new ServiceTemplate (null,null, eAttributes);
myInterface = (DataRepositoryInterface) registrar.lookup (template);
```

Here, myInterface is the Service Object. Using this, the client accesses the method execQuery() in the DataRepositortService. Before querying it prompts the client to input a query. Then it informs the client that the query entered is about to be sent to the Data Repository Service. Thereafter, you can see the desired results.

6.5.4 Printing Service

This service prints the given text on the server default printer in the given font and according to the given width and height of the paper (PageFormat object). This service provides only black and white text printing.

6.5.5 Printer Service Client (for accessing printing service)

The class Printer Service Client is basically meant for the Printing Service. In this, we, first of all set the security manager as:

System.setSecurityManager (new RMISecurityManager ());

Next, we find the JINI lookup service () and print its location.

```
lookup = new LookupLocator ("jini://localhost");
sHost = lookup.getHost ();
iPort = lookup.getPort ();
```

Then, we get the lookup service's ServiceRegistrar (the class by which interaction with the lookup service is possible).

```
registrar = lookup.getRegistrar ();
id = registrar.getServiceID ();
```

Therefafter, we perform a search on the lookup server to find the service that has the name attribute of "MyServer". The lookup service returns an interface object to the service.

```
aeAttributes = new Entry[1];
aeAttributes[0] = new Name (name);
template = new ServiceTemplate (null,null, aeAttributes);
proxy = (PrintingServiceInterface) registrar.lookup (template);
```

Now if this proxy is an instance of PrintingServiceInterface, then the proxy is returned.

6.5.6 Format class

Format class takes text (as a String) as an input and breaks the text into lines according to the width and FontMetrics object passed to it. It then generates a vector of those lines. The main method of the

class is process(), which has two loops to put words in a line(vector) and to move to the next line. As a result we get a vector containing the lines of the passed string. These lines are then get by subsequent calls to hasNext() and next() methods. This class can also return the subset of the object according to the specified lines as a Format class object. The subset is used by the Page class.

6.5.7 Page class

Page class takes the Page Format for width and height of actual page, Format class object for lines to be printed. It obtains x and y coordinates of the Page Format object to start formatting the page. After getting all the information it then reset the pointer to the start of the data and print each line according to line spacing and height by adding these values to y coordinate until end of data is reached. In this way it formats the page according to our need by setting its height and width, which is given by user.

7. Recommendations and Future Enhancements

Based on this project, many enhancements are possible:

- If this technology can somehow be incorporated in a word processor to make it JINI enabled, then it can use the print service over the network without having information or drivers on the machine.
- And once we have this JINI enabled word processor, then other services can be incorporated into it, like fax, scanning service etc.
- Similarly, we can have JINI enabled web browsers, media players etc to access the remote resources over the network.

8. Glossary and Bibliography

The following glossary is no way complete. However, it is an attempt to list a few number of sources and references for further reading.

- JAVA An object oriented programming language
- JDK Java Developer Kit
- JINI Not acronym for anything, a network technology
- JSK JINI Software Kit
- JCP JINI Technology Core Platform
- JVM Java Virtual Machine
- KAWA An integrated development environment for JAVA
- LUS Lookup Service
- RMID Remote Method Invocation Daemon
- A Lookup Service supplied by SUN
- UDP User Datagram Protocol

Coming over to the bibliography side, the widest range of references on JINI Network Technology generally is to be found on the various Internet newsgroups that deal with this subject. They include:

- www.jini.org
- www.java.sun.com
- www.artima.com/jini (a source for JAVA and JINI developers)
- http://developer.java.sun.com/developer/products/jini
- http://www.enete.com/download/#_nuggets_
- http://www.artima.com/javaseminars/modules/Jini/CodeExamples.html
- http://www.jinivision.com/
- http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml
- http://www.eli.sdsu.edu/courses/spring99/cs696/notes/index.html
- http://developer.jini.org/exchange/users/jmcclain/index.html
- http://sourceforge.net/projects/jini-tools

and many subsidiary newsgroups under these.

Jini.org includes a frequently asked questions (FAQs) file that can be downloaded, which includes a large number of questions relating JINI network technology and a very long list of references including surveys, history and the analytical work. Besides this various tutorials have also been deeply studied in order to complete this project.