Jan Newmarch's Guide to JINI Technologies

Jan Newmarch

Version 2.01

30 December 1999

http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml

1. Contents

This version uses Jini 1.1 (alpha). For a version using Jini 1.0, see
http://pandonia.canberra.edu.au/java/jini/tutorial.1.03/Jini.xml

1.1 Chapters

1. Overview
    1. Jini
    2. Components
    3. Service Registration
    4. Client Lookup
    5. Proxies
    6. Client Structure
    7. Server Structure
    8. Partitioning an Application
    9. Support Services
    10. Summary
2. TroubleShooting Jini
    1. The Chapter that Shouldn't Exist
    2. Java Packages
    3. Jini Packages
    4. Lookup Service
    5. RMI Stubs
3. Discovering a Lookup Service
    1. Lookup Service
    2. Unicast discovery
    3. Broadcast discovery
    4. ServiceRegistrar
    5. Summary
4. Entry Objects
    1. Entry class
    2. Restrictions on entries
    3. Convenience Classes
    4. Further Uses of Entries
    5. Summary
5. Service Registration
    1. ServiceRegistrar
    2. ServiceItem
    3. Registration
    4. ServiceRegistration
    5. Entries
    6. Summary
6. Client Search
    1. ServiceRegistrar
    2. ServiceMatches
    3. Matching
    4. Summary
7. Leasing
    1. Leases
    2. LeaseRenewalManager
    3. Granting and handling leases
    4. Summary
8. Simple Example
    1. Problem Description
    2. Service Specification
    3. Common Classes
    4. Client
    5. Service Proxy
    6. Uploading a Complete Service
    7. Summary
9. Choices for Service Architecture
    1. Proxy Choices
    2. RMI Proxy for FileClassifier
    3. Non-RMI Proxy for FileClassifier
    4. RMI and non-RMI Proxies for FileClassifier
    5. Using the Naming service
    6. Using Other Services
10. Discovery Management
    1. Management Interfaces
    2. LookupLocatorDiscovery
    3. LookupDiscoveryManager
    4. Conclusion
11. Join Manager
    1. Join Manager
    2. Jini 1.0 JoinManager
    3. Summary
12. Security
    1. Quick Fix
    2. Why AllPermission is Bad
    3. Removing AllPermission
    4. Jini with Protection
    5. Service Requirements
    6. Client Requirements
    7. Transaction Manager and other Activatable Services
13. More Complex Examples
    1. Where is the Code
    2. Running threads from discovery
    3. Inexact Service Matching
    4. Matching using Local Services

5. Finding a Service Once Only
6. Leased Changes to a Service

14. Remote Events
    1. Event Models
    2. Remote Events
    3. Event Registration
    4. Listener List
    5. Listener Source
    6. File Classifier with Events
    7. Monitoring Changes in Services
    8. Summary
15. ClientLookupManager
    1. Introduction
    2. ServiceItemFilter
    3. Finding a Service Immediately
    4. Using a Filter
    5. Building a Cache of Services
    6. Monitoring Changes to the Cache
16. Transactions
    1. Introduction
    2. Transactions
    3. TransactionManager
    4. TransactionParticipant
    5. Mahalo
    6. Example
    7. Summary
17. Lego MindStorms
    1. Making Hardware into Jini Services
    2. MindStorms
    3. MindStorms as a Jini Service
    4. RCXPort
    5. RCX Programs
    6. Jini classes
    7. Getting it running
    8. Entry objects for a robot
    9. Adding entries to a server
    10. A client-side RCX class
    11. Higher level mechanisms: Not Quite C
    12. Summary
18. CORBA and Jini
    1. CORBA
    2. CORBA to Java Mapping
    3. Jini Proxies
    4. Simple Example
    5. Room Booking Example
    6. Migrating a CORBA Client to Jini
    7. Jini Service as a CORBA Object
19. User Interfaces for Jini Services
    1. User Interfaces as Entries
    2. User Interfaces from Factory Objects

3. Objects Returned from getUI()
4. Support Classes
5. Images
6. MindStorms

20. Activation
    1. Introduction
    2. A Service using Activation
    3. LeaseRenewalService
    4. LookupDiscoveryService
21. Architecture
    1. Threads
22. etc (to come :-)

1.2 Appendices

1. Properties used in Jini

2. Files

The source to the programs is available as a zip file programs.zip. The compiled classes are available as a zip file classes.zip.

The documentation files are also available as a zip file docs.zip. You might not like what you get from this doc file - I am using XML as document format, using a mishmash of the DocBook DTD with extra tags and HTML thrown in. A Perl program, `xml_handler.pl`, is a handler called by my Apache Web server to deliver XML files as HTML. Another perl program, `docbook2html.pl`, will convert them standalone, but it will leave the hypertext links pointing to the XML files.

3. Update Information

This tutorial will be updated on a regular basis, maybe several times a week. If you want to receive an email announcement of each version release, send email to jan@ise.canberra.edu.au with subject ''Jini tutorial update''

4. Feedback

I would appreciate any feedback on this tutorial, to ensure that it is correct and meets the needs of readers. The feedback that I have already received has been very valuable. Please send comments/corrections/requests, etc, to jan@ise.canberra.edu.au

5. Other Resources

1. The Jini FAQ at http://www.artima.com/jini/faq.html
2. Jini Community Web Site http://www.jini.org/
3. Noel Enete's ''Nuggets'' tutorial at www.enete.com/download/#_nuggets_
4. Eran Davidov's `timeservice` example at www.artima.com/jini/resources/timeservice.html
5. Roger Whitney's lecture notes on ''Java Distributed Computing'' at http://www.eli.sdsu.edu/courses/spring99/cs696/notes/
6. Benot Marchal's articles for Digital Cat's Java Resource Center at www.javacats.com/US/articles/Ben/Jini1.html (and Jini2.html, Jini3.html, etc)
7. Jini Home page http://www.sun.com/jini/index.html

8. RMI Home page http://java.sun.com/products/jdk/rmi/index.html
9. Jini mailing list jini-users@java.sun.com
10. Jini mailing list archives http://archives.java.sun.com/archives/jini-users.html
11. RMI mailing list rmi-users@java.sun.com
12. RMI mailing list archives http://archives.java.sun.com/archives/rmi-users.html
13. Jini Interface Repository for standardised service interfaces (empty at present) http://www.artima.com/jini/interrepo/

6. Changes

See the file Changes for the list of version releases and changes made between versions.

7. Copyright Information

This tutorial (and the pages it contains) are covered by the OpenContent license. This basically gives you freedom to do what you want with the document in a similar way to the GNU license for software. Any changes or additions that you make to this set of pages as a Web document must be made publically available, and may be folded into later versions of this tutorial. Any such inclusions will of course be credited with *your* copyright, under the OpenContent license.

On the other hand, the *programs* that this tutorial references are also available unbundled from the tutorial, and do not have the restriction that source derived from them must be made publically available. So you can derive your own source code from them, without having to make it public. I'm not sure what suitable license will do this...

And, of course, there is no warranty...

8. Acknowledgements

The author is grateful for comments on this tutorial from

1. Brian Jeltema, jeltema@richdist.east.sun.com
2. Roger Whitney, whitney@cs.sdsu.edu
3. Robbert van den Beld, rbe@mms-dresden.de
4. Chitrarasu Muthaiyan, chitrarasu@cswl.com
5. Stuart Remphrey, Stuart.Remphrey@Aus.Sun.COM
6. JJ Larrea, jlarrea@redtop.com
7. John McClain, John.McClain@East.Sun.COM

The author is currently on a sabbatical program at the CRC for Distributed Systems Technology, http://www.dstc.edu.au and the work reported in this tutorial has been funded in part by the Co-operative Research Centre Program through the Department of Industry, Science and Tourism of the Commonwealth Government of Australia.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Overview of Jini

Contents

1. Jini
2. Components
3. Service Registration
4. Client Lookup
5. Proxies
6. Client Structure
7. Server Structure
8. Partitioning an Application
9. Support Services
    1. HTTP server
    2. RMI daemon
10. Summary

This chapter gives a brief overview of the components of a Jini system and the relationships between them.

1. Jini

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

1. A new printer connected to the network can announce its presence and capabilities. A client can then use this printer without having to be specially configured to do so.
2. A digital camera can be connected to the network. It can present a user interface that will not only allow pictures to be taken, but is also aware of any printers so that the pictures can be printed if possible.
3. 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.
4. New capabilities extending existing ones can be added to a running system without disruption of existing services, or any need to reconfigure possible clients
5. Services can announce changes of state, such as a printer running out of paper. Listeners, typically of an administrative nature, can watch for these and flag them for attention

Jini is not an acronym for anything, and does not have a particular meaning. A Jini system or *federation* is a collection of clients and services all communicating by the Jini protocols. Normally this will consist of applications written in Java, communicating using the Java Remote Method Invocation mechanism. While 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. They just need to talk the Jini protocols. This gives a Jini federation a flexibility beyond Java, in which the framework is supplied using simple Java mechanisms, but can link services and clients from a variety of sources.

2. Components

Jini is just one of a large number of distributed systems architectures, including industry-pervasive systems such as CORBA and DCOM. It is distinguished by being based on Java, and deriving many features purely from this Java basis. There are other Java frameworks from Sun which would appear to overlap Jini, such as Enterprise Java Beans. This book does not delve much into the relationships between all these various systems, but the reader should be aware that Jini is only one competitor in a non-empty market. What will condition the success or failure of Jini is partly the politics of the market, but also (hopefully!) the technical capabilities of Jini, and this book will deal with some of the technical issues involved in using Jini.

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.)

Code will be moved around between these three pieces, and this is done by serializing the objects (and *marshalling* them to allow storage without reconstitution), and using Java’s socket support to send and receive objects. In addition, objects in one JVM (Java Virtual Machine) may need to invoke methods on an object in another JVM and this will typically be done using RMI (Remote Method Invocation), although the Jini specification does not require this (it may require RMI *semantics*, but not necessarily implementation).

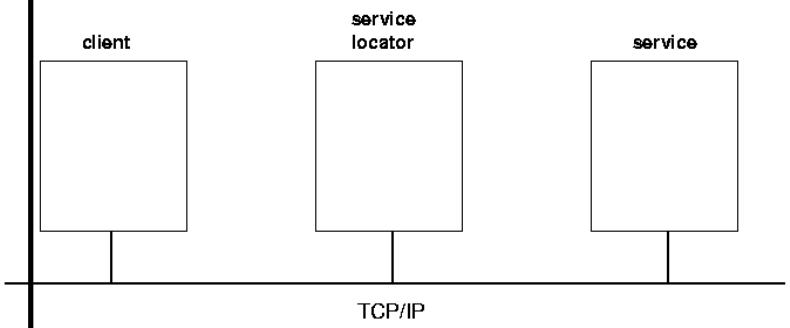


Figure 1: Components of a Jini system

3. Service Registration

The first stage is for the service to register itself with the lookup service/service locator. In order to peform this registration, it must first find a lookup service. This can be done in two ways: if the location of the lookup service is known, then the service can use unicast TCP to connect directly to it. If the location is not known, the service will make UDP multicast requests, and lookup services will respond to these requests. Lookup services will be listening on port 4160 for both the unicast and multicast requests. When the lookup service gets a request on this port, it sends an *object* back to the service. 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 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 does with the registrar is to *register* the service with the lookup service. This involves taking a copy of the service, and storing it on the lookup:

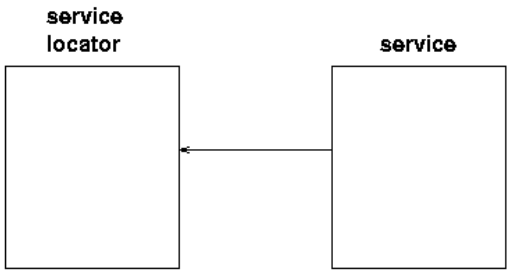


Figure 2: Querying for a service locator

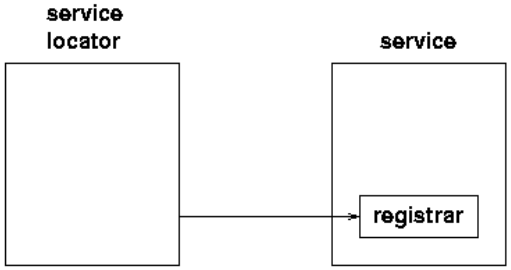


Figure 3: Registrar returned

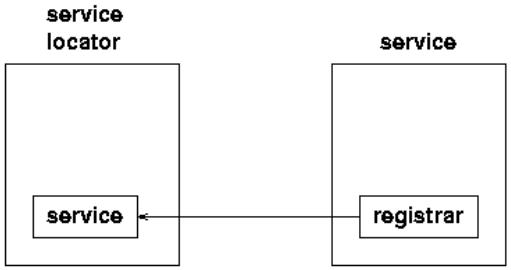


Figure 4: Service uploaded

4. Client Lookup

The client on the other hand, goes through the same mechanism to get a registrar from the lookup service. But this time it does something different with this, which is to request the service to be copied across to it.

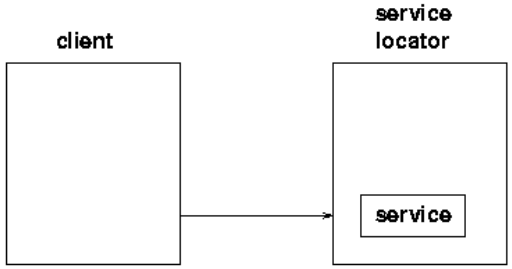


Figure 5: Querying for a service locator

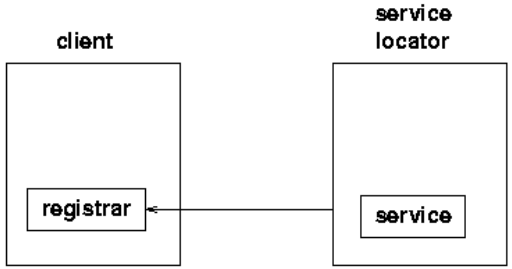


Figure 6: Registrar returned

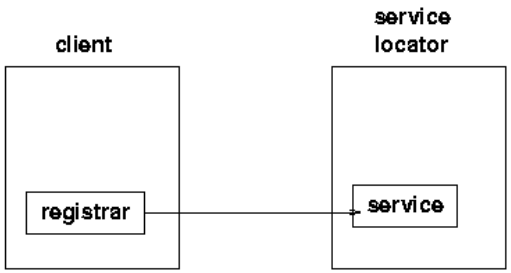


Figure 7: Asking for a service

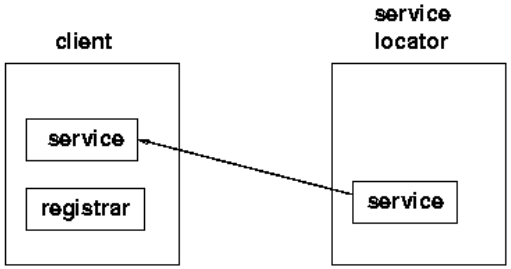
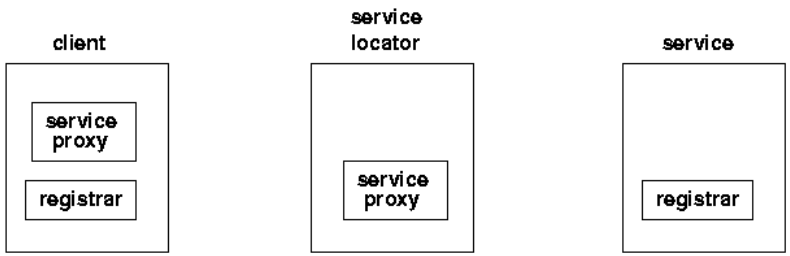


Figure 8: Service returned

At this stage there is the original service running back on its host. There is a copy of the service stored in the lookup service, and there is a copy of the service running in the client's JVM. The client can make requests of the service object running in its own JVM.

5. Proxies

How does this work if the service is actually a toaster, a printer, or controlling some piece of hardware? By the time it runs in the client's JVM, it may be a long way away from its hardware. In this case, rather than sending out a copy of the service itself, the service will send out a *proxy*, which will communicate back to the service, probably using RMI. It appears we have a chicken-and-egg situation: how does the proxy find its service? Use a Jini lookup? No, when the proxy is created it is ''primed'' with its own service's location so that when run it can find its own ''home''. This will look like



Proxy Service

How is the proxy primed? This isn't specified by Jini, and can be done in a large number of ways. For example, an RMI naming service can be used such as `rmiregistry`, where the proxy is given the name of the service. This isn't very common, as RMI proxies can be passed more directly as returned objects from method calls, and these can refer to ordinary RMI server objects or to RMI activatable objects. Or the proxy can be implemented *without* any direct use of RMI, and can then use an RMI exported service or some other protocol altogether such as ftp, http or a home-grown protocol. These various possibilities are all illustrated in later chapters.

6. Client Structure

Internally a client will look like

Pseudocode	Where discussed
`prepare for discovery`	Discovering a lookup service
`discover a lookup service`	Discovering a lookup service
`prepare a template for lookup search`	Entry objects and Client search
`lookup a service`	Client search
`call the service`	

The following code is simplified from the real case, by omitting various checks on exceptions and other conditions. It attempts to find a `FileClassifier` service, calling the method `getMIMEType()` on this service. The full version is given in a later chapter. We don't give detailed explanations right now, this is just to show how the above schema translates into actual code.

```
public class TestUnicastFileClassifier {

    public static void main(String argv[]) {
        new TestUnicastFileClassifier();
    }

    public TestUnicastFileClassifier() {
```

```
        LookupLocator lookup = null;
        ServiceRegistrar registrar = null;
        FileClassifier classifier = null;

        // Prepare for discovery
        lookup = new LookupLocator("jini://www.all_about_files.com");

        // Discover a lookup service
        // This uses the synchronous unicast protocol
        registrar = lookup.getRegistrar();

        // Prepare a template for lookup search
        Class[] classes = new Class[] {FileClassifier.class};
        ServiceTemplate template = new ServiceTemplate(null, classes, null);

        // Lookup a service
        classifier = (FileClassifier) registrar.lookup(template);

        // Call the service
        MIMEType type;
        type = classifier.getMIMEType("file1.txt");
        System.out.println("Type is " + type.toString());
    }
} // TestUnicastFileClassifier
```

7. Server Structure

A server application will internally look like

Pseudocode	Where discussed
`prepare for discovery`	Discovering a lookup service
`discover a lookup service`	Discovering a lookup service
`create information about a service`	Entry objects
`export a service`	Service registration
`renew leasing periodically`	Leasing

The following code is simplified from the real case, by omitting various checks on exceptions and other conditions. It exports an implementation of a file classifier service, as a `FileClassifierImpl` object. The full version is given in a later chapter. We don't give detailed explanations right now, this is just to show how the above schema translates into actual code.

```
public class FileClassifierServer implements DiscoveryListener {

    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

    public static void main(String argv[]) {
        new FileClassifierServer();

        // keep server running forever to
```

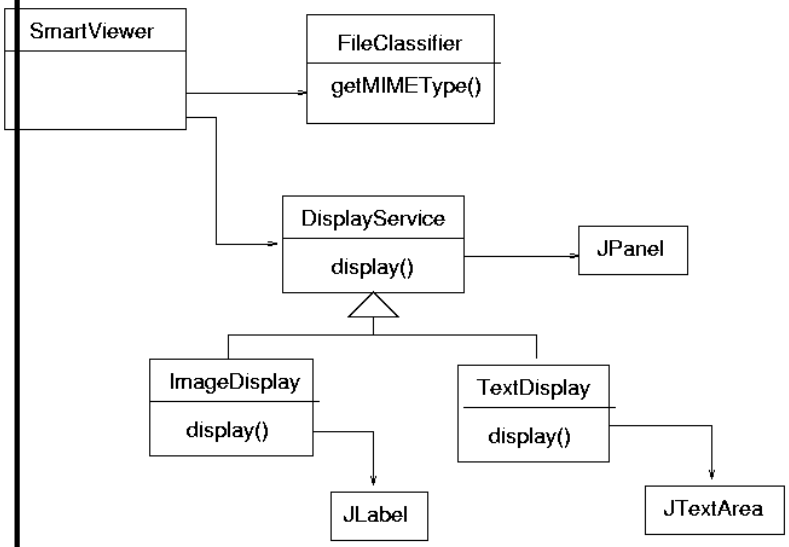


```
        // - allow time for locator discovery and
        // - keep re-registering the lease
        Thread.currentThread().sleep(Lease.FOREVER);

    }

    public FileClassifierServer() {
        LookupDiscovery discover = null;

        // Prepare for discovery - empty here

        // Discover a lookup service
        // This uses the asynchronous multicast protocol,
        // which calls back into the discovered() method
        discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {
        ServiceRegistrar registrar = evt.getRegistrars()[0];
        // At this point we have discovered a lookup service

        // Create information about a service
        ServiceItem item = new ServiceItem(null,
                                           new FileClassifierImpl(),
                                           null);

        // Export a service
        ServiceRegistration reg = registrar.register(item, Lease.FOREVER);

        // Renew leasing
        leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
    }
} // FileClassifierServer
```

8. Partitioning an Application

Jini uses a "service" view of applications. This is in contrast to the simple object-oriented view of an application. Of course, a Jini "application" will be made up of objects, but these will be distributed out into individual services, which will communicate via their proxy objects. The Jini specification claims that in many monolithic applications there are one or more services waiting to be released, and making them into services increases their possible uses.

To see this, we can look at a "smart file viewer". This is an application that will be given a filename, and based on the structure of the name will decide what type of file it is (`.rtf` is Rich Text Format, `.gif` is a GIF file, and so on). Using this classification it will then call up an appropriate viewer for that type of file, such as an image viewer or document viewer. A UML class diagram for this might look like

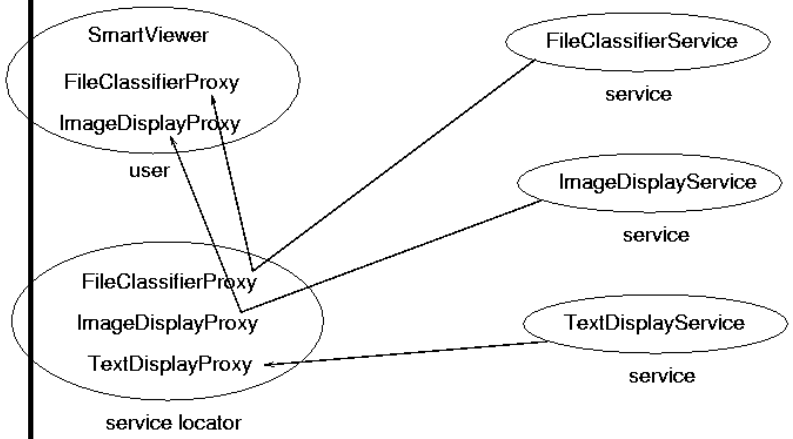


There are a number of services in this. Classifying a file into types is one (which will be used heavily in the sequel - because it is simple). This service can be used in lots of different situations, not just viewing contents. Each of the different viewer classes is another. This is not to say that every class should become a service! That would be overkill. What makes these qualify as services is that they

1. have a simple interface
2. are useful in more than one situation
3. can be replaced or varied

They are *reusable*, and this is makes them good candidates for services. They do not require high-bandwidth communication (such as a stack would), and are not completely trivial.

If the application is re-organised as a collection of services, then it could look like



Each service may be running on a different machine on the network (or on the same machine - it doesn't matter). Each exports a proxy to whatever service locators are running. The `SmartViewer` application finds and downloads whatever services it needs, as it needs them.

9. 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 standard Java code, and will run in Java Virtual Machines (JVM). Typically, each of these applications will run in a single JVM, although there are techniques available to run multiple Java applications in the same JVM. When they run, they will need access to Java class files, just like any other Java application. An application can use the `CLASSPATH` environment variable or use the `classpath` option to the runtime to locate these classes.

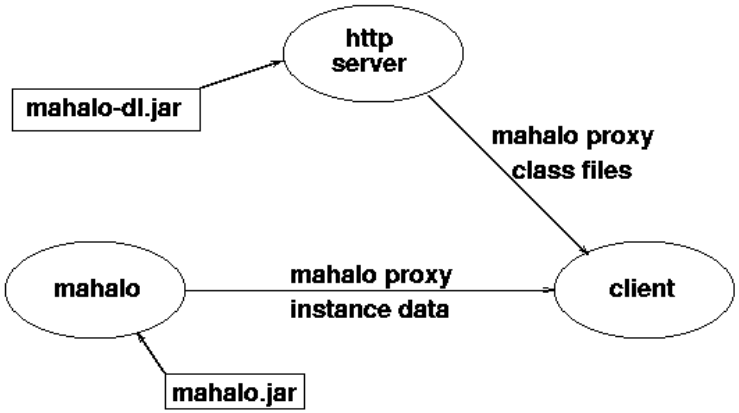
However, Jini relies heavily on the ability to move objects across the network, from one JVM to another. In order to do this, particular implementations must make use of support services such as RMI daemons and HTTP (or other) servers. The particular support services required depend on implementation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

9.1 HTTP server

A Java object running as a service has a proxy component exported to the service locators and then onto a client. It passes through one JVM in ''passive'' form and is activated (brought to life) in the client's JVM. Essentially, a ''snapshot'' of the object's state is taken using serialization, and this snaphot is moved around. An object consists of both code and data, and it cannot be reconstituted from just its data - the code is also required. Where is the code? This is where a distributed Jini application differs from an ordinary one: the code is not likely to be on the client side. If it was *required* to be on the client side, then Jini would lose almost all of its flexibility because it wouldn't be possible to just add new devices and their code to a network. The class definitions are most likely on the server, or perhaps on the vendor's home Web site.

So class definitions for service proxy objects must also be downloaded, usually from where the service came from. This could be done using a variety of methods, but most commonly an HTTP or FTP protocol is used. The service specifies the protocol used and also the location of the class files using the `java.rmi.server.codebase` property. The object's serialized data contains this codebase, which is used by the client to access the class files.

If the codebase specifies an HTTP url, then there must be an HTTP server running at that url, and the class files must be there also. This often means one HTTP server per service, but this isn't required: a set of services could make their class files available from a single HTTP server, and this server could be running on a different machine to the services. This gives two sets of class files: the set needed to run the service (specified by `CLASSPATH`) and the set needed to reconstitute objects at the client (specified by the `codebase` property). For example, the `mahalo` service supplied by Sun as a transaction manager uses the class files in `mahalo.jar` to run the service, and the class files in `mahalo-dl.jar` to reconstitute the transaction manager proxy at the client.



To run `mahalo`, the `CLASSPATH` must include `mahalo.jar`, and to reconstitute its proxy on a client, the `codebase` property must be set to `mahalo-dl.jar`.

9.2 RMI daemon

A proxy service gets exported to the client. In most cases this will need to communicate back to its host service. There are many ways to do this. One mechanism is the Java Remote Method Invocation (RMI) system. This comes in two flavours in JDK 1.2: the original `UnicastRemoteObject` and the newer `Activable` class. Whereas `UnicastRemoteObject` requires a process to remain alive and running, `Activable` objects can be stored in a passive state and the Activation system will create a new JVM if needed when a method call is made on the object. While passive, an activatable object will need to be stored on some server, and this server must be one which can accept method calls and activate the objects. Sun supply such a server, called `rmid`.

This is real, real obscure and deep stuff if you are new to RMI or even to the changes it is going through. So why is it needed? Sun supply a service locator called `reggie`, and this is really just another Jini service that plays a special role. In particular, it exports proxy objects - the `registrar` objects. What makes this complex is that `reggie` uses `Activable` in its implementation. In order to run `reggie`, you first have to start an `rmid`

server on the same machine, and then `reggie` will register with it.

This isn't all downhill: running `rmid` has beneficial side-effects. It maintains log files of its own state, which includes the activable objects it is storing. So `reggie` can crash or terminate, and `rmid` will restore it as needed. Indeed, even `rmid` can crash or be terminated, and it will use its log files to restore state so that it can still accept calls for `reggie` objects.

10. Summary

A Jini system is made up of three parts

1. Service
2. Client
3. Service locator

Code is moved between these applications. A `registrar` acts as a proxy to the lookup locator, and runs on both the client and service.

A service and a client both possess a certain structure, which is detailed in the following chapters.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

TroubleShooting Jini

Contents

1. The Chapter that Shouldn't Exist
2. Java Packages
3. Jini Packages

This chapter looks at some of the problems that can arise in a Jini system. Most of them are configuration problems of some kind. This chapter is woefully incomplete :-(((Contributions welcome!!!

1. The Chapter that Shouldn't Exist

Jini is advertised as ''network plug and play''. This carries the idea of ''zero administration'' where you buy a device, switch it on and *voila* it is there and available. Well, this may happen in the future, but right now there are a number of back-room games that you have to succeed at. Once you have won at these, ''network plug and play'' *does* work, but if you lose at any stage then it can be all uphill!

What is hard is issues of getting the right files in the right places with the right permissions. About 50% of the messages in the Jini mailing list are all about these configuration problems. These shouldn't occur, and that is why this is ''The Chapter that Shouldn't Exist''.

This is the second chapter in this book, so right now you shouldn't have managed to fail at anything! Each of the early sections contains instructions on what to do to get the example programs working. So skip on to the next chapters, but come back here when things go wrong. Your luck may vary: I got quite a reasonable way on my first attempts without problems, and some are even luckier. Some aren't...

2. Java Packages

Typical error:

```
Exception in thread "main" java.lang.NoClassDefFoundError:
        basic/InvalidLookupLocator
```

Most of the code in this tutorial is organised into packages. To run the examples the classes must be accessible from your class path. For example, one of the programs in the `basic` directory is `InvalidLookupLocator.java`. This defines the class `InvalidLookupLocator` in the package `basic`. The program must be run using the fully qualified path name, as in

```
java basic.InvalidLookupLocator
```

(Note the '.', not a '/'.)

In order to find this class, the classpath must be set correctly for the Java runtime. If you have copied the zip file `classes.zip` then the class files for this tutorial are in there. You only need to reference this

```
CLASSPATH=classes.zip:...
```

If you have downloaded the source files, then they are all in subdirectories such as `basic`, `complex`, etc. After compilation the class files should also be in these subdirectories, such as `basic/InvalidLookupLocator.class`. An alternative to using `classes.zip` is to set the classpath to include the directory containing those subdirectories. For example, if the full path is say `/home/jan/classes/basic/InvalidLookupLocator.class`, then set classpath to


```
CLASSPATH=/home/jan/classes:...
```

An alternative to setting the `CLASSPATH` environment variable is to use the `-classpath` option to the Java runtime engine

```
java -classpath /home/jan/classes basic.InvalidLookupLocator
```

3. Jini Packages

Typical error

```
Exception in thread "main" java.lang.NoClassDefFoundError:
            net/jini/discovery/DiscoveryListener
```

The Jini class files are all in `jar` files. The Jini distribution has them in a subdirectory `lib` when it is unpacked. There are a whole bunch of these jar files:

```
jini-core.jar         mahalo-dl.jar         sun-util.jar
jini-examples-dl.jar  mahalo.jar            tools.jar
jini-examples.jar     reggie-dl.jar
jini-ext.jar          reggie.jar
```

The jar file `jini-core.jar` contains the major packages of Jini

```
net.jini.core
net.jini.core.discovery
net.jini.core.entry
net.jini.core.event
net.jini.core.lease
net.jini.core.lookup
net.jini.core.transaction
```

If the Java compiler or runtime can't find a class in one of these packages then you need to make sure that the `jini-core.jar` is in your classpath.

The jar file `jini-ext.jar` contains a set of packages which are not in the core, but are still heavily used

```
net.jini.admin
net.jini.discovery
net.jini.entry
net.jini.lookup
net.jini.lookup.entry
net.jini.space
```

If the Java compiler or runtime can't find a class in one of these packages then you need to make sure that the `jini-ext.jar` is in your classpath.

The jar file `sun-util.jar` contains the packages from the `com.sun.jini` hierarchy. These contain a number of ''convenience'' classes that are not essential but can be useful. These are less frequently used.

A compile or run of a Jini application will typically have an environment set something like

```
JINI_HOME=wherever_Jini_home_is
CLASSPATH=.:$JINI_HOME/lib/jini-core.jar:$JINI_HOME/lib/jini-ext.jar
```

4. Lookup Service

Typical error

```
java.rmi.activation.ActivationException: ActivationSystem not running;
nested exception is:
        java.rmi.NotBoundException: java.rmi.activation.ActivationSystem
java.rmi.NotBoundException: java.rmi.activation.ActivationSystem
```

The command `rmid` starts the ‘‘activation system’’ running. If this cannot start properly or dies just after starting, you will get this message. Usually it is caused by incorrect file permissions.

5. RMI Stubs

Typical error

```
java.rmi.StubNotFoundException:
         Stub class not found: option3.FileClassifierImpl_Stub;
nested exception is:
        java.lang.ClassNotFoundException: option3.FileClassifierImpl_Stub
```

Many of the examples export services as remote RMI objects. These objects are subclasses of `UnicastRemoteObject`. What gets exported is not the object itself, but a *stub* that will act as proxy for the object (which continues to run back in the server). The stub has to be created using the `rmic` compiler as in

```
rmic -v1.2 -d . option3.FileClassifierImpl
```

This will create a `FileClassifierImpl_Stub.class` in the `option3` subdirectory. The stub class file needs to be accessible to the Java runtime in the same way as the original class file.

Another typical error is

```
java.rmi.ServerException: RemoteException occurred in server thread; nested exception is:
        java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:
        java.lang.ClassNotFoundException: option3.FileClassifierImpl_Stub
```

This arises when an object is trying to get a remote reference to the `FileClassifierImpl`, and it is trying to load the class file for the stub from an HTTP server. What makes this one particularly annoying is that it may not be referring to the `FileClassifierImpl_Stub` at all! The class will often implement a remote interface such as `RemoteFileClassifier`. This in turn implements the common class `FileClassifier`. *Class files for these also have to be available!* The HTTP server must carry not only the class files for the stubs, but the class files for all superclasses and interfaces that are not available to all.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Discovering a Lookup Service

Contents

1. Lookup Service
    1. Reggie
2. Unicast discovery
    1. LookupLocator
    2. Running the InvalidLookupLocator
    3. Information from the LookupLocator
    4. Get Registrar
    5. Running the UnicastRegister
3. Broadcast discovery
    1. Groups
    2. LookupDiscovery
    3. DiscoveryListener
    4. DiscoveryEvent
    5. Staying alive
    6. Running the MulticastRegister
    7. Broadcast range
4. ServiceRegistrar
    1. Information from the ServiceRegistrar
5. Summary

This chapter looks at what is involved in discovering a lookup service/service locator. This is common to both services and clients.

1. 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 specialised to store services and pass them on to clients looking for them.

1.1 Reggie

Sun supply a lookup service called `reggie` as part of Jini. There may be any number of these lookup services running in a network, so that if one goes down it doesn’t matter. It is even possible for services and clients to function without any lookup services running, but this is not the usual case. Anybody can start a lookup service (depending on access permissions), but it will usually be started by an administrator, or can be 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 `/home/jan/tmpdir/jini1_0/lib/reggie-dl.jar`. I set document root to `/home/jan/tmpdir/jini1_0/lib`, so the relative URL from this server is just `/reggie-dl.jar`.

The other support service needed for `reggie` is an RMI daemon. This is `rmid`, and is also part of the Jini distribution. This must be run on the same machine as `reggie`:

```
rmid &
```

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.

There is a security issue with `rmid` on multi-user systems such as Unix. The activation system that it supports allows anyone to run programs using the user id of `rmid`. So you should *never* run it using a sensitive user id such as `root`, but instead should run it as the least privileged user, `nobody`.

Once these two are running, `reggie` can be started:

```
java -jar reggie.jar ...
```

with a number of compulsory parameters

```
java -jar lookup-server-jarfile lookup-client-codebase lookup-policy-file \
        output-log-dir lookup-service-group
```

These parameters are

1. The `lookup-server-jarfile` will be `reggie.jar` or some path to it.
2. `lookup-client-codebase` will be the url for the reggie stub class files, using the HTTP server started earlier. In my case, this is `http://jannote.dstc.edu.au:8080/reggie-dl.jar`. Note that an absolute IP hostname must be used - you cannot use `localhost` because to the *reggie service* it means `jannote.dstc.edu.au`, but to the *client* it would be a different machine altogether! The client would then fail to find `reggie-dl.jar` on its own machine. Even using an abbreviated address such as `jannote` would fail to be resolved if the client is external to the local network.
3. The `lookup-policy-file` controls security accesses. To get going, set this to the path to `policy.all` in the Jini distribution, but for deployment, use a less dangerous policy file.
4. The `output-log-dir` can be set to any path to store the log files.
5. The `lookup-service-group` can be set to the public group `public`.

On my own machine, I run this by

```
java -jar /home/jan/tmpdir/jini1_0/lib/reggie.jar \
     http://jannote.dstc.edu.au:8080/reggie-dl.jar \
     /home/jan/tmpdir/jini1_0/example/lookup/policy \
     /tmp/reggie_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 `reggie` 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 `reggie` in its log files.

2. 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!).

2.1 LookupLocator

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

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

public Class LookupLocator {

    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 `localhost` or some other valid DNS name. No unicast discovery is performed at this stage, though, so any rubbish could be entered. Only a check for syntactic validity is performed. This is not even done for the second constructor.

The following program creates some objects with valid and invalid host/URLs. They are only checked for syntactic validity rather than existence as URLs:

```
package basic;

import net.jini.core.discovery.LookupLocator;

/**
 * InvalidLookupLocator.java
 *
 *
 * Created: Tue Mar  9 14:14:06 1999
 *
 * @author Jan Newmarch
```

```
 * @version 1.1
 *    moved to package basic
 */

public class InvalidLookupLocator  {

    static public void main(String argv[]) {
        new InvalidLookupLocator();
    }

    public InvalidLookupLocator() {
        LookupLocator lookup;

        // this is valid
        try {
            lookup = new LookupLocator("jini://localhost");
            System.out.println("First lookup creation succeeded");
        } catch(java.net.MalformedURLException e) {
            System.err.println("First lookup failed: " + e.toString());
        }

        // this is probably an invalid URL,
        // but the URL is syntactically okay
        try {
            lookup = new LookupLocator("jini://ABCDEFG.org");
            System.out.println("Second lookup creation succeeded");
        } catch(java.net.MalformedURLException e) {
            System.err.println("Second lookup failed: " + e.toString());
        }

        // this IS a malformed URL
        try {
            lookup = new LookupLocator("A:B:C://ABCDEFG.org");
            System.out.println("Third lookup creation succeeded");
        } catch(java.net.MalformedURLException e) {
            System.err.println("Third lookup failed: " + e.toString());
        }

        // this is valid
        lookup = new LookupLocator("localhost", 80);
        System.out.println("Fourth lookup creation succeeded");
    }

} // InvalidLookupLocator
```

2.2 Running the InvalidLookupLocator

All programs will need to be compiled using the JDK 1.2 compiler. Jini programs will not compile or run under JDK 1.1 (any versions). This program defines the class `InvalidLookupLocator` in package `basic`. The source code will in the file `InvalidLookupLocator.java` in the `basic` subdirectory. From the parent directory, this can be compiled by a command such as


```
javac basic/InvalidLookupLocator.java
```

to leave the class file also in the `basic` subdirectory.

The `CLASSPATH` will need to include the Jini file `jini-core.jar` for compilation of the source code. When a service is run, this Jini files will need to be in its `CLASSPATH`. Similarly, when a client runs, it will also need this file in its `CLASSPATH`. And finally, when a lookup service is run it too will need this file (and also `reggie.jar`) in its `CLASSPATH`. The reason for this repetition is that the service, the client and lookup service all three separate applications, running in three separate Java Virtual Machines, and quite likely will be on three separate computers. They will each have separate requirements, but at least this is common to all three!

The `InvalidLookupLocator` has no additional requirements. It does not perform any network calls, and does not require any additional service to be running. So it can be run simply by

```
java -classpath ... basic.InvalidLookupLocator
```

2.3 Information from the LookupLocator

A `LookupLocator` has methods

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

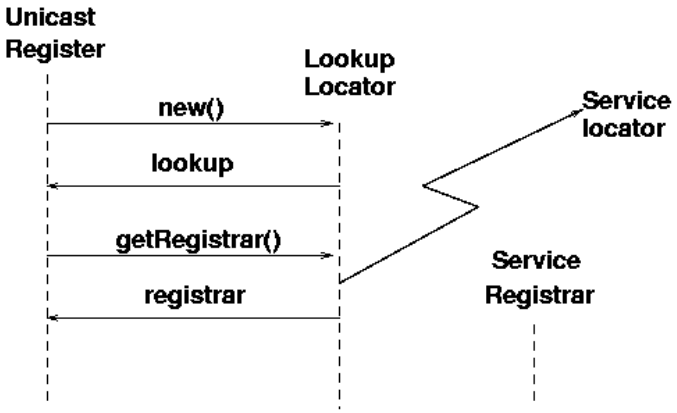
which will return information about the hostname that the locator will use, and the port it will ask on. This is just the information fed into the constructor or left to default values, though. It doesn't give anything new for unicasting. This information *will* be useful in the multicast situation, though.

2.4 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. A UML sequence diagram augmented with a jagged arrow showing the network connection is



By this stage, the program looks like

```
package basic;

import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;

/**
 * UnicastRegistrar.java
 *
 *
 * Created: Fri Mar 12 22:34:53 1999
 *
 * @author Jan Newmarch
 * @version 1.2
 *    renamed as UnicastRegister
 *    moved to package basic
 */

public class UnicastRegister  {

    static public void main(String argv[]) {
        new UnicastRegister();
    }

    public UnicastRegister() {
        LookupLocator lookup = null;
        ServiceRegistrar registrar = null;

        try {
```



```
            lookup = new LookupLocator("jini://localhost");
        } catch(java.net.MalformedURLException e) {
            System.err.println("Lookup failed: " + e.toString());
            System.exit(1);
        }

        try {
            registrar = lookup.getRegistrar();
        } catch (java.io.IOException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        } catch (java.lang.ClassNotFoundException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        }
        System.out.println("Registrar found");

        // the code takes separate routes from here for client or service
    }

} // UnicastRegister
```

The `registrar` object will be used in different ways for clients and services.

This program might not run as is, due to security issues. If that is the case, see the chapter on Security.

2.5 Running the UnicastRegister

The program needs to be compiled and run with `jini-code.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, the program will throw an exception and terminate. Here the host specified is `localhost`. It could, however, be any machine accessible on the local or remote network (as long as it is running a service locator). For example, to connect to the service locator running on my current workstation, the parameter to `LookupLocator` would be `jini://sunshade.dstc.edu.au`.

This program will receive a `ServiceRegistrar` from the service locator. However, it does so by a simple `readObject()` on a socket connected to the service locator, and so does not need any additional support services such as `rmiregistry` or `rmid`. It can be run by

```
java -classpath ... basic.UnicastRegister
```

3. Broadcast discovery

If the location of a lookup service is unknown, it is neccessary to make a broadcast search for one. The search may be across the Internet, but is usually restricted to the local LAN or business network. 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.

3.1 Groups

Some services may be meant for anyone to use, but some may be more restricted in applicability. For example, there may be a departmental diary containing deadlines for that department. A service may state this by giving a list of *groups* that it belongs too. This is a set of strings, such as

```
String [] groups = {"Engineering dept", "Jan's services"};
```

3.2 LookupDiscovery

The class `LookupDiscovery` in package `net.jini.discovery` is used for this. 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 search for all services in that set of groups.

3.3 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 handle this indeterminacy, the `LookupDiscovery` object can have a listener registered with it that is invoked as each reply comes in.

```
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 is to go to sleep for, say, a thousand seconds.

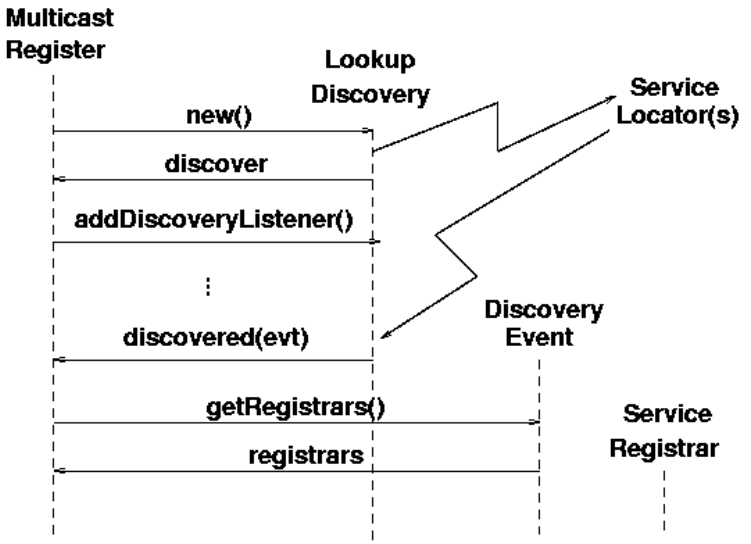
The `discarded()` method is invoked whenever the application discards a lookup service by calling `discard()` on the registrar object.

3.4 DiscoveryEvent

The parameter to the `discover()` method 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 is the same class as 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



By this stage the program looks like

```
package basic;

import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
```

```
/**
 * MulticastRegister.java
 *
 *
 * Created: Fri Mar 12 22:49:33 1999
 *
 * @author Jan Newmarch
 * @version 1.2
 *    renamed as MulticastRegister
 *    moved sleep() to main() from constructor
 *    moved to package basic
 */

public class MulticastRegister implements DiscoveryListener {

    static public void main(String argv[]) {
        new MulticastRegister();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(10000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public MulticastRegister() {
        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            e.printStackTrace();
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();

        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];

            // the code takes separate routes from here for client or service
            System.out.println("found a service locator");
        }
    }
```



```
    public void discarded(DiscoveryEvent evt) {

    }
} // MulticastRegister
```

3.5 Staying alive

In the constructor we create a `LookupDiscovery` object, add a `DiscoveryListener` and then the constructor terminates. The `main()` method, having called this constructor, promptly goes to sleep. What is going on here? The constructor to `LookupDiscovery` actually starts up a number of threads, to broadcast the service and to listen for replies. (See the eventual chapter on Architecture). When replies come in, the listener thread will call the `discovered()` method of the `MulticastRegister`. However, these threads are *daemon* threads, and are not enough to keep the application alive - there must be at least one *user* thread running.

The `sleep()` ensures that a user thread continues to run, even though it apparently does nothing! This will keep the application alive so that the daemon threads (running in the "background") can discover some lookup locators. Ten seconds (10,000 milliseconds) is long enough for that. To stay alive *after* this ten seconds expires will depend on either increasing the sleep time or creating another user thread in the `discovered()` method.

I have placed the `sleep()` in the `main()` method. It is perfectly reasonable to place it in the application constructor, and some examples do this. However, it looks a bit strange there, so I prefer this placement. Note that although the constructor will have terminated without us assigning its object reference, a live reference has been passed into the `discover` object as a `DiscoveryListener` and it will keep the reference alive in its own daemon threads. This means that the application object will still exist for its `discovered()` method to be called.

Any other method that results in a user thread continuing to exist will do just as well. For example, a client that has an AWT or Swing user interface will stay alive because there are many user threads created by any of these GUI objects. For services which typically will not a GUI interface running, another simple way is to create an object and then wait on it

```
Object keepAlive = new Object();
try {
    keepAlive.wait();
} catch(InterruptedException e) {
}
```

3.6 Running the MulticastRegister

The program needs to be compiled and run with `jini-core.jar` and `jini-ext.jar` in its `CLASSPATH`. The extra jar file is needed as it contains the class files from the `net.jini.discovery` package. When run, the program will attempt to find all service locators that it can. If there are none, it will find none - pretty boring. So one or more should be set running in the near network or on the local machine.

This program will receive `ServiceRegistrar`'s from the service locators. However, it does so by a simple `readObject()` on a socket connected to a service locator, and so does not need any additional support services such as `rmiregistry`.

3.7 Broadcast range

Services and clients search for lookup locators using the multicast protocol, by sending out packets as UDP datagrams. It makes announcements on UDP 224.0.1.84 on port 4160. How far do these announcements reach? This is controlled by the "time to live" field on the packets, and by default the current implementation of `LookupDiscovery` sets this to be 15. This normally restricts the packets to the local network. However, this may be changed by giving the system property `net.jini.discovery.ttl` a different value. But be careful about setting this: many people will get irate if you flood the networks with multicast packets.

4. ServiceRegistrar

The `ServiceRegistrar` is an abstract class which is implemented by each lookup service. The actual details of this implementation are not relevant here. The *role* of a `ServiceRegistrar` is to act as a *proxy* for the lookup service. This proxy runs in the application, which may be a service or a client.

This is the first object that is moved from one Java process to another in Jini. It is shipped from the lookup service to the application looking for the lookup service, using a socket connection. From then it runs as an object in the application's address space, and the application makes normal method calls to it. When needed, it communicates back to its lookup service, but does so without explicit RMI calls being needed by the application. This proxy object should not cache any information on the application side, but get "live" information from the lookup service as needed. The implementation of the lookup service supplied by Sun does exactly this.

This object has two major methods, one used by a service attempting to register

```
public ServiceRegistration register(ServiceItem item,
                                    long leaseDuration)
                             throws java.rmi.RemoteException
```

and the other(s) by a client trying to locate a particular service

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

The details of these are given in Service Registration and in Client Search . For now, an overview will suffice.

A service will register an object (that is, an instance of a class), and a set of attributes for that object. For example, a printer may specify that it can handle Postscript documents, or a toaster that it can deal with frozen slices of bread. The service may register itself, or it may register a proxy that can act on its behalf. Note carefully: *the registered object will be shipped around the network. When it finally gets to run, it may be a long way away from where it was originally created.* It will have been created in the service's JVM, transferred to the lookup locator by `register()` and then to the client's JVM by `lookup()`.

A client is trying to find a service, using some properties of the service that it knows about. Whereas the service can export a live object, the client cannot use a service object as a property - because then it would already have the thing, and wouldn't need to try to find one! What it can do is to use a *class* object, and try to find if there are any *instances* of this class lying around in service locators. As discussed later in Client Search, it is best if the client asks for an `interface` class object. In addition to this class specification, the client may specify a set of attribute values that it requires from the service.

The next step is to look at the possible forms of attribute values, and how matching will be performed. This is done using Jini Entry objects. The simplest services, and the least demanding clients, will not require any attributes: the `Entry[]` array will be `null`. You may wish to skip ahead to service registration or to client search and come back to entries later.

4.1 Information from the ServiceRegistrar

The `ServiceRegistrar` is returned after a successful discovery has been made. This object has a number of methods that will return useful information about the lookup service itself. So in addition to using this object to register a service or lookup a service, you can use it to find out about the lookup locator. The major methods are

```
String[] getGroups();;
LookupLocator getLocator();
ServiceID getServiceID();
```

The method `getGroups()` will return a list of the groups that the locator is a member of.

The second method, `getLocator()` is more interesting. This is exactly the same object as is used in the unicast lookup, but now its fields are filled in by the discovery process. So you can find which host the locator is running on, and its hostname by

```
registrar.getLocator().getHost();
```

This can be used to find a list of which locators are running. From there information such as host can be found - this will be information filled in by the discovery process, rather than being preset as in unicast lookup.

```
public void discovered(DiscoveryEvent evt) {

    ServiceRegistrar[] registrars = evt.getRegistrars();

    for (int n = 0; n < registrars.length; n++) {
        ServiceRegistrar registrar = registrars[n];
        System.out.println("Service locator at " +
                           registrar.getLocator().getHost());
    }
}
```

You could use this to find where a service locator is so that the *next* time this program runs it could connect directly by unicast.

The third method `getServiceID()` is unlikely to be of much use to you. In general, service ID's are used to give a globally identifier for the service (no different services should have the same ID), and a service should have the same ID with all service locators. However, this is the service ID of the lookup service, not of any services registered with it.

5. Summary

Both services and clients need to find lookup services. Discovering a lookup service may be done using unicast or multicast protocols. Unicast discovery is a synchronous mechanism. Multicast discovery is an asynchronous mechanism that requires use of a listener to respond when a new service locator is discovered.

When a service locator is discovered, it sends a `ServiceRegistrar` object to run in the client or service. This acts as a proxy for the locator. This object may be queried for information, such as the host the service locator is on.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Entry Objects

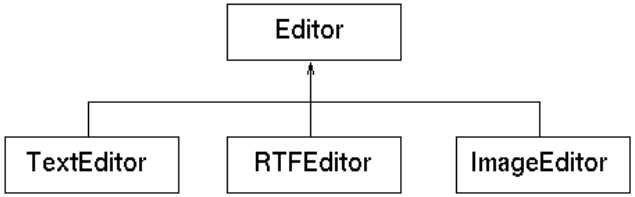
Contents

1. Entry class
    1. Simplistic
2. Restrictions on entries
3. Convenience Classes
4. Further Uses of Entries
5. Summary

Entries are used to pass additional information about services that clients can use to decide if a particular service is what it wants.

1. Entry class

When a service registers it places a copy of itself (or a suitable proxy) on the lookup service. This copy is an *instance* of an object, albeit in serialised form. It can optionally register sets of attributes along with its intance. 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. For example, a set of file editors may be available as services. Each editor is capable of editing different types of file as in



(Note: these would probably be interfaces, rather than instantiable classes.)

A client can search for a suitable editor in two ways

1. By asking for an instance of a specific class such as `ImageEditor`
2. By asking for an instance of the general class `Editor` with the *additional* information that it can handle a certain type of file

The type of search performed depends on the problem domain and the amount of information that clients have. Jini can handle either case. It handles the first case by only specifying a class object, such as `ImageEditor.class`. The Jini `Entry` class is designed to help with the second situation by specifying a superclass object such as `Editor.class` and allowing the additional information to be given in the request by adding extra objects.

The `Entry` class also allows services to advertise their capabilities in ways that would otherwise require use of multiple inheritance, which is not available in Java. Suppose an editor belonged to class `SuperEditor` which can handle plain text, RTF files *and* image files. In a multiple inheritance system it would do so by subclassing from all of the classes. In Jini, it just advertises itself as being an `Editor`, but has entries attached for each of the types that it can handle.

To manage this, we would have a class `FileType` which gives information about the types of file handled:

```
public Class FileType implements Entry {
    public String type; // this is a MIME type

    public FileType(String type) {
       this.type = type;
    }
}
```

For a text editor, the attribute set would be `FileType("plain/text")`. For a RTF editor, the attribute set would be `FileType("application/rtf")`. (Of course, there may be more than one field in the class!)

For the `SuperEditor`, its capabilities would be given by using an array of entries:

```
Entry[] entries = new Entry[] {new FileType("plain/text"),
                               new FileType("application/rtf")
                               new FileType("image/gif")
                              };
```

On the other side, a client wishes to find services that can handle the attributes that it requires. The client uses the same `Entry` class for this. For any particular `Entry`, the client specifies

1. Which fields must match *exactly* (a non-null value)
2. Which fields it does not care about - a `null` value)

1.1 Simplistic

This is pretty crude. A printer typically has the capacity to print a certain number of pages per minute. If it specifies this using an `Entry`, then it actually makes it rather hard to find! A client can request a printer service in which it does not care about speed, or for a particular speed. It cannot ask for printers with a speed *greater than* some value. It cannot ask for a printer *without* a capability, such as anything *except* a color printer. An attribute must either match exactly or be ignored. The relational operators such as '<' and '!=' are not supported.

If you want to search for a printer at a particular speed, then printer speed capabilities may need to be given simpler descriptive values such as ''fast'', ''average'' or ''slow''. Then, once you have a ''fast'' printer service returned to the client, it can perform a query on the service itself for its actual speed. This would be done outside of the Jini mechanisms, using whatever interface has been agreed for the description of printers. A similar problem, that of finding a physically ''close'' service is taken up in the chapter on ''More Complex Examples''.

The mechanism chosen, of exact matches with wildcards, is comparatively easy to implement. It is a pity from the programmer's view that a more flexible mechanism was not used. One suggestion often made in the Jini mailing list is that there should be a `boolean matches()` method on the service object. However, it would involve un-marshalling the service on the locator in order to run the `matches()` method and this suffers from a variety of problems

1. What security permissions should the filter run with?
2. Slows the lookup service down
3. What happens if the filter modifies its arguments - deep copying to avoid this will cause further slow-downs

2. Restrictions on entries

Entries are shipped around in marshalled form. Exported service objects are serialized, moved around and reconstituted as objects at some remote client. Entries are similarly serialized and moved around. However, when it comes to comparing them, this is usually done on the lookup service and they are not reconstituted on the lookup service. So when it comes to comparing an entry from a service and an entry from a client request, it is the serialized forms that are compared.

An entry cannot have as field one of the primitive types such as `int` or `char`. If one of these fields is required, then it must be wrapped up in a class such as `Integer` or `Character`. This to make it easier to perform ''wildcarding'' for matching (see next chapter for details). A wildcard for any object can be the ''pattern'' `null` which will work for any class, including wrapper classes such as `Boolean` (on the other hand, what is the wildcard for `boolean`: `true` or `false`?

The only fields of interest are public, non-static, non-transient and non-final.

An entry class must have a no-args constructor.

3. Convenience Classes

The class `AbstractEntry` is a subclass of `Entry`, and is designed as a convenience class. It implements methods such as `equals()` and `toString()`. Hoever, it doesn't really seem to add much that would justify using it, in my opinion.

In addtion, Sun's implementation of Jini contains a further set of convenience classes, all subclassed out of `AbstractEntry`. These require the `jini-ext.jar` file and are

- `Address` - the address of the physical component of a service.
- `Comment` - a free-form comment about a service.
- `Location` - the location of the physical component of a service. This is distinct from the `Address` class in that it can be used alone in a small, local organization.
- `Name` - the name of a service as used by users. A service may have multiple names.
- `ServiceInfo` - generic information about a service. This includes the name of the manufacturer, the product, and the vendor.
- `ServiceType` - human-oriented information about the "type" of a service. This is not related to its data or class types, and is more oriented towards allowing someone to determine what a service (for example, a printer) does and that it is similar to another, without needing to know anything about data or class types for the Java platform.
- `Status` - the base class from which other status-related entry classes may be derived.

For example, the `Address` class contains

```
String country;
String locality;           // City or locality name.
String organization;       // Name of the company or organization that provides this service.
String organizationalUnit; // The unit within the organization that provides this service.
String postalCode;         // Postal code.
String stateOrProvince;    // Full name or standard postal abbreviation of a state or province.
String street;             // Street address.
```

You may find these classes useful; on the other hand, what services would like to advertise, and what clients would like to match on is pretty much unknown as yet. These classes are not part of the formal Jini specification.

4. Further Uses of Entries

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. An expectation in this is that the information in an entry is primarily *static*. However, entries are objects, and could also implement *behaviour* as well as *state*. This could be used to extend the behaviour of a service, beyond that of the service itself.

The main use of this at present is to define the user interface for a service. This topic is looked at in detail in a much later chapter, but in brief: to allow for differing display devices, there may be a need for many user interfaces, or none. If all the possibilities have to be included in the service itself then it will become cumbersome and inflexible. It is better to put this elsewhere, such as in entry objects.

5. Summary

An entry is additional information about a service. A service may have any number of entries. Clients requests services by class and by entries, using a simple matching system. There are a number of convenience classes that subclass `Entry`.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Service Registration

Contents

1. ServiceRegistrar
2. ServiceItem
3. Registration
4. ServiceRegistration
    1. Running the SimpleService
    2. Information from the ServiceRegistration
    3. Service ID
5. Entries
6. Summary

This chapter looks at how services register themelves with lookup locators so that they can later be found by clients.

1. ServiceRegistrar

A service find 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 service then registers itself 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 chapter 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);
}
```

2. ServiceItem

The service will make an object of this type, 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. It only has meaning to the particular service locator that sets it.

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. To avoid confusion between the original service and the object that is sent to the lookup service, I shall use the term *exported service object*, although this terminology is not used in the formal Jini documentation. There are several things to note about the exported 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.

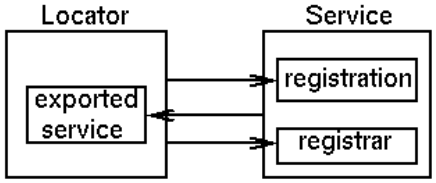
The third parameter is a set of entries giving information about the service in addition to the service item itself. If there is no additional information, this can be `null`.

3. 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`

4. ServiceRegistration

This object is created by the service locator and is returned to run in the service. This acts as a proxy object to control the state maintained about the exported service object stored on the lookup service. This object maintains a field `serviceID` which is used to identify the exported service object on the lookup service. This can be retrieved by `getServiceID()` for reuse by the service if it needs to do so.



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 Leases

The service then has nothing else to do. If the exported service object can do the full task of the service, then this instance can exit. If the exported service object acts as a proxy and needs to communicate back to the service then the service can sleep so that it continues existence. If the service needs to stay around so that it can re-register itself when timeout occurs then it can sleep as well.

A unicast service that exports itself and has nothing else to do is in the following program:

```
package basic;

import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import java.io.Serializable;

/**
 * SimpleService.java
 *
 *
 * Created: Fri Mar 12 22:34:53 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    moved to package basic
 */

public class SimpleService implements Serializable {

    static public void main(String argv[]) {
        new SimpleService();
    }

    public SimpleService() {
        LookupLocator lookup = null;
        ServiceRegistrar registrar = null;

        try {
            lookup = new LookupLocator("jini://localhost");
        } catch(java.net.MalformedURLException e) {
            System.err.println("Lookup failed: " + e.toString());
            System.exit(1);
        }

        try {
            registrar = lookup.getRegistrar();
```

```
        } catch (java.io.IOException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        } catch (java.lang.ClassNotFoundException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        }

        // register ourselves as service, with no serviceID
        // or set of attributes
        ServiceItem item = new ServiceItem(null, this, null);
        ServiceRegistration reg = null;
        try {
            // ask to register for 10,000,000 milliseconds
            reg = registrar.register(item, 10000000L);
        } catch(java.rmi.RemoteException e) {
            System.err.println("Register exception: " + e.toString());
        }
        System.out.println("Service registered");

        // we can exit here if the exported service object can do
        // everything, or we can sleep if it needs to communicate
        // to us or we need to renew a lease later
        //
        // Typically, we will need to renew a lease later
    }

} // SimpleService
```

4.1 Running the SimpleService

The program needs to be compiled and run 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.

This does not need additional support services such as `rmiregistry` or `rmid` since all registration is done by the method `register()`. Objects are transferred in serialized form across socket connections.

4.2 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();
```

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. For now, we can just use it to find out how long the lease has been granted for by its method `getExpiration()`:


```
long duration = reg.getLease().getExpiration() -
                System.currentTimeMillis();
System.out.println("Lease expires at: " +
                   duration +
                   " milliseconds from now");
```

4.3 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 are 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. The code above is not well-behaved in this respect.

5. 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, ...);
```

6. Summary

A service uses the `ServiceRegistrar` object returned as a proxy from a locator to register itself with that locator. The service prepares a `ServiceItem` that contains a service object and a set of entries. The service object may be a proxy for the real service. It registers this using the `register` method of the `ServiceRegistrar` object.

Information about a registration is returned as a `ServiceRegistration` object. This may be queried for information such as the lease and its duration.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Client Search

Contents

1. ServiceRegistrar
2. ServiceMatches
3. Matching
4. Summary

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

1. ServiceRegistrar

A client gets a `ServiceRegistrar` object 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.

The lookup methods use a class of type `ServiceTemplate` to specify the service looked for:

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

public Class ServiceTemplate {
    public ServiceID serviceID;
    public java.lang.Class[] serviceTypes;
    public Entry[] attributeSetTemplates;

    ServiceTemplate(ServiceID serviceID,
                    java.lang.Class[] serviceTypes,
                    Entry[] attrSetTemplates);
}
```

The `serviceID` is `null` if this is the first time a search is being made. If it is a repeat, then the non-null value from previous searches should be used. The `attributeSetTemplates` is a set of `Entry` elements to perform matching of attributes, as discussed later in this chapter.

The major parameter to the `lookup()` method is a list of `serviceTypes`. We know that services export instances of a class. How does the client ask so that it gets a suitable instance delivered from the lookup locator? The possibilities are

1. The client could ask for it by using an instance in the search request. But this would be pretty stupid, as the client would then already have what it is searching for!
2. The client could ask for a `Class` object, and this object could be the class of the instance it wants to find.

This is also not very good. Firstly, the client may be able to do a `new` operation on this class, so that again it would not need to search for an instance on the network. The other possibility is that it doesn't have all the classes needed to create all the fields and needs to fetch them from somewhere. This has the problems discussed in the next point.

3. The client could ask for a `Class` object which is a *superclass* of the instance on the lookup locator. This is better, as it does not require detailed knowledge of the properties of the instance, or even which subclass it is. However, on the down side, classes have implementation code, and code has version problems. Since the client and the server will probably be on different machines there is a real possibility of a *version mismatch*.
4. The client could ask for a `Class` object which is a *interface*. Interfaces do not have implementation code that can change. Ideally, an interface should never change at all. A client can ask for an instance of an interface and not care what class it gets or what version of the class. As long as it implements the interface, that is enough. This is the preferred mechanism.

So the client will usually request a *interface* object.

To be more concrete, a toaster may be defined by an interface

```
public interface Toaster extends java.io.Serializable {
    public void setDarkness(int dark);
    public void startToasting();
}
```

A Breville "Extra Lift" toaster will implement this in one particular way, as will other toasters

```
public class BrevilleExtraLiftToaster implements Toaster {
    public void setDarkness(int dark) {
        ...
    }
    public void startToasting() {
        ...
    }
}
```

When the toaster service starts, it exports an object of class `BrevilleExtraLiftToaster` to the lookup service. However, the client does not know what type of toaster is out there, so will make a request such as

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

    // specify the interface object
    Class[] toasterClasses = new Class[1];
    toasterClasses[0] = Toaster.class;

    // prepare a search template of serviceID, classes and entries
    ServiceTemplate template = new ServiceTemplate(null,
                                                   toasterClasses,
                                                   null);

    // now find a toaster
    Toaster toaster = null;
    try {
        toaster = (Toaster) registrar.lookup(template);
    } catch(java.rmi.RemoteException e) {
        System.exit(2);
    }
```


Notice that the `lookup()` can throw an exception. This can occur, if say, the service requested is not serialisable.

At the end of this, an object has been transported across to the client that is an instance of a class implementing the `Toaster` interface, and has been coerced to be of this type. This object has two methods `setDarkness()` and `startToasting()`. No other information is available about the toaster capabilities, because the interface does not specify any more and in this case the set of attribute values was `null`. So the client can call

```
    toaster.setDarkness(1);
    toaster.startToasting();
```

Before leaving this section, what is the role of `System.setSecurityManager(new RMISecurityManager())`? A serialized object has been transported acrosss the network, and is reconstituted and coerced to an object implementing `Toaster`. We know that here it will in fact be an object of class `BrevilleExtraLiftToaster`, but the client doesn't need to know that. Or does it? Certainly the client will not have a class definition for this class on its side. But when the toaster object begins to run, then it must run using its `BrevilleExtraLiftToaster` code! Where does it get it from? From the server, most likely by an HTTP request on the server. This means that it is *loading a class definition* across the network, and this requires security access. So a security manager capable of granting this access must be installed before the load request is made.

Note the difference between loading a *serialized instance* and loading a *class definition*: the first does not require access rights, only the second one does. So if the client had the class definitions of *all possible* toasters then it would never need to load a class, and a new security manager would not be needed. This is not likely, but may perhaps be needed in a high security environment.

2. ServiceMatches

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 constructor for `lookup()`. 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.

In addition, not all elements of this array need be non-null! Note that in `lookup(tmpl)` when asking for only one match, an exception can be returned such as when the service is not serialisable. No exception is thrown here, because although one match might be bad, the others may still be okay. So a value of `null` as the array element value is used to signify this

```
    ServiceMatches matches = registrar.lookup(template, 10);
    // NB: matches.totalMatches may be greater than matches.items.length
    for (int n = 0; n < matches.items.length; n++) {
        Toaster toaster = (Toaster) matches.items[n].service;
        if (toaster != null) {
            toaster.setDarkness(1);
            toaster.startToasting();
        }
    }
```

(which will start upto 10 toasters cooking at once!)

3. Matching

A client is attempting to find one or more services that satisfy its requirements. It does so by creating a `ServiceTemplate` object and using this in a registrar's `lookup()` call. A `ServiceTemplate` object has three fields

```
ServiceID         serviceID;
java.lang.Class[] serviceTypes;
Entry[]           attributeSetTemplates;
```

An overview: firstly, if the client is repeating a request, then it may have recorded the `serviceID` from an earlier request. The `serviceID` is a ''universally unique identifier'' so can be used to identify a service unambiguously. This can be used by the service locator as a filter to quickly discard other services. Secondly, a client may want to find a service satisfying a number of interface requirements at once. For example, a client may look for a service that implements both `Toaster` and `FireAlarm` (so that it can properly handle burnt toast). Finally, the client will specify a set of attributes that must be satisfied by each service. Each attribute required by the client is taken in turn and matched against the set offered by the service. For example, in addition to requesting a `Toaster` with a `FireAlarm`, a client entry may specify a location in ''GP South Building''.This will be tried against all the variations of location offered by the service. A single match is good enough. An additional client requirement of, say, manufacturer would also have to be matched by the service.

More formally from the `ServiceTemplate` API documentation:

1. A service item (`item`) matches a service template (`tmpl`) if: `item.serviceID` equals `tmpl.serviceID` (or if `tmpl.serviceID` is `null`); and `item.service` is an instance of every type in `tmpl.serviceTypes`; and `item.attributeSets` contains at least one matching entry for each entry template in `tmpl.attributeSetTemplates`.
2. An entry matches an entry template if the class of the template is the same as, or a superclass of, the class of the entry, and every non-null field in the template equals the corresponding field of the entry. Every entry can be used to match more than one template. Note that in a service template, for `serviceTypes` and `attributeSetTemplates`, a `null` field is equivalent to an empty array; both represent a wildcard.

4. Summary

A client prepares a `ServiceTemplate` which is a list of class objects and a list of entries. For each service locator that is found, the client can query this using the `ServiceRegistrar` object's `lookup()` method, to see if the locator has a service matching the template. If the match is successful, an object is returned that can be cast into the class required. Service methods can then be invoked on this object.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication

License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Leasing

Contents

1. Leases
    1. Cancellation
    2. Expiration
2. LeaseRenewalManager
3. Granting and handling leases
    1. Abstract Lease
    2. Landlord Package
    3. Landlord Lease
    4. Leased Resource
    5. Lease Policy
    6. Lease Manager
    7. Landlord
4. Summary

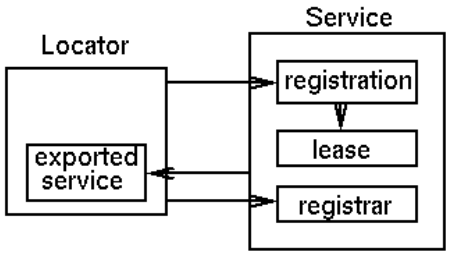
Leasing is the mechanism used between applications to give access to resources in an agreed manner

1. Leases

Leases are requested for a period of time. In Jini, the major use 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

1. `Lease.ANY` - the service lets the lookup service decide on the time
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 seconds.) 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.




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


The principal methods of the `Lease` object are

```
package net.jini.core;

public interface Lease {
    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()`). The time from now is then

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

1.1 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.

1.2 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. This is a bit of a nuisance for the programmer, because you will need to call `renew()` sufficiently in advance of expiration to allow for network and scheduling delays. (You can be informed - too late - if the lease on a lookup service has just expired, by asking for change events on the lookup service. This is covered later.)

2. LeaseRenewalManager

Jini supplies a class `LeaseRenewalManager` that looks after the process of calling `renew()` at suitable times. *Warning:* in Jini 1.0 this was in package `com.sun.jini.lease`; in Jini 1.1 it is now in package `net.jini.lease`

```
package net.jini.lease;

public Class LeaseRenewalManager {
    public LeaseRenewalManager();
    public LeaseRenewalManager(Lease lease,
                               long expiration,
                               LeaseListener listener);
    public void renewFor(Lease lease, long duration,
                         LeaseListener listener);
    public void renewUntil(Lease lease,
                           long expiration,
                           LeaseListener listener);
    // etc
}
```

This manages a set of leases, which may be set by the constructor or added later by `renewFor()` or `renewUntil()`. The time requested in these is in milliseconds. The `expiration` time is since the epoch, whereas the `duration` time is from now. 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. This will be caught by the renewal manager and will cause the listener's `notify()` method to be called with a `LeaseRenewalEvent` as parameter. This will allow the application to take corrective action if its lease is denied. The listener may be `null`.

You have to be careful about setting the `duration` in `renewFor()` (in Jini v1.0). If you want the service to be registered forever, it is tempting to use `Lease.FOREVER`. However, the current implementation just adds this to `System.currentTimeMillis()` which overflows to a negative value which is not checked. So it never does any renewals. Strictly, you need to check that

```
duration + System.currentTimeMillis() > 0
```

before calling `renewFor()`. This will be fixed in later versions. The method `renewUntil()` can use `Lease.FOREVER` with no problems.

3. Granting and handling leases

All of the above discussion was from the side of a client that *receives* a lease, and has to manage it. The converse of this is the agent that *grants* leases and has to manage things from *its* side. This is more advanced material which can be skipped for now: it is not needed until the chapter on ''Remote Events''. An example of use is given in the chapter on ''More Complex Examples''.

A lease can be granted for almost any networked service, any one which ones wants to use the idea of being able to register services and have them time-out if no renewal is made. Because of the possible variety of services, some parts of the Jini lease mechanisms cannot be given totally, and must be left as interfaces for applications to fill in. A lease is given as an interface and any agent that wishes to grant leases must implement this interface. Jini gives two implementations, an `AbstractLease` and a subclass of this, a `LandlordLease`.

A main issue in implementing a particular lease class lies in setting a policy for handling the initial request for a lease period, and in deciding what to do when a renewal request comes in. Some simple possibilities are

1. Always grant the requested time
2. Ignore the requested time and always grant a fixed time

Of course, there are many more possibilities, based on expected time to live of the lessor, system load, *etc*.

There are other issues, though. Any particular lease will need a time-out mechanism. A group of leases can be managed together, and this can reduce the amount of overhead of managing individual leases.

3.1 Abstract Lease

An abstract lease gives a basic implementation of a lease, that can *almost* be used for simple leases. *Warning:* this class, and those that depend on it are still not fully specified, and may change in future versions of Jini.


```
package com.sun.jini.lease;

public abstract class AbstractLease implements Lease, java.io.Serializable {
    protected AbstractLease(long expiration);
    public long getExpiration();
    public int getSerialFormat();
    public void setSerialFormat(int format);
    public void renew(long duration);
    protected abstract long doRenew(long duration);
}
```

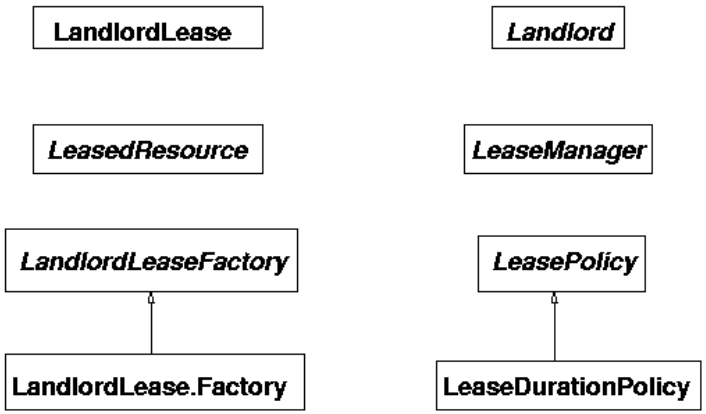
This class supplies straightforward implementations of much of the `Lease` interface, with three provisos: firstly, the constructor is `protected`, so that constructing a lease with a specified duration is devolved to a subclass. This means that lease duration policy must be set by this subclass. Secondly, the `renew()` method calls into the abstract `doRenew()` method, again to force a subclass to implement a renewal policy. Thirdly, it does not implement the method `cancel()`, so that this must also be left to a subclass. So this class implements the easy things, and leaves all matters of policy to concrete subclasses.

3.2 Landlord Package

The ''landlord'' is a package to allow more complex leasing systems to be built. It is not part of the Jini specification, but is supplied as a set of classes and interfaces. (*Warning: this is part of the `com.sun.jini` package, which may change for Jini 1.1.*) The set is not complete in itself - some parts are left as interfaces, and need to have class implementations. These will be supplied by a particular application.

A landlord looks after a set of leases. The leases in turn are managed by their landlord. Leases are indentified to the landlord by a ''cookie'', which is just some object that uniquely labels each lease to its landlord. It could be an `Integer`, for example, with a new value for each lease. A landlord does not need to create leases itelf, as it can use a *landlord lease factory* to do this. (But of course it *can* create them, depending on how it an implementation is done.) When a lease wishes to cancel or renew itself, it asks its landlord to do it. A client is unlikely to ask the landlord directly, as it will only have been given a lease, not a landlord.

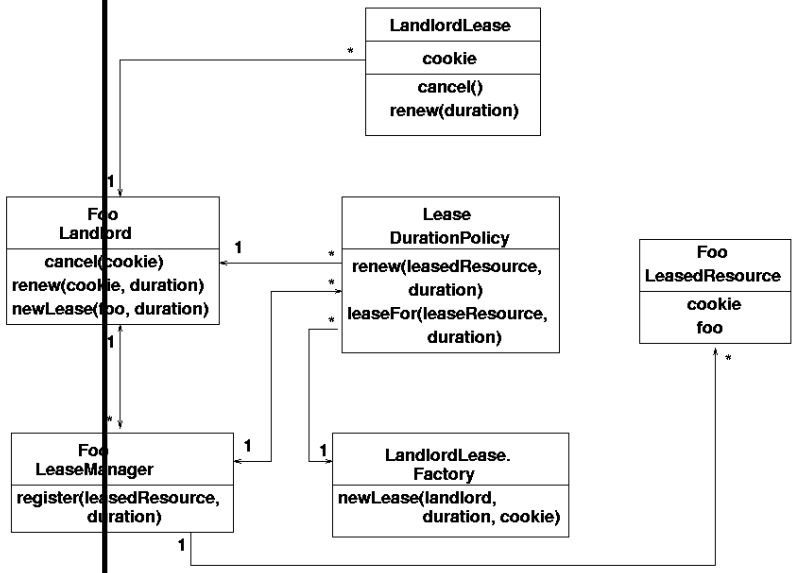
The principal classes and interfaces in the `landlord` package are



where the interfaces are shown in italicised font and the classes in normal font. This fairly complex set of classes and interfaces is driven by a number of factors

1. A simple lease implementation was needed for a variety of situations. If a particular implementation is chosen, then a factory is needed for this, and a corresponding factory interface to generate other particular implementations. A lease (on the client) also requires the existence of some handler for its methods on the lease granting side, which is the landlord
2. To handle all policy issues such as initial granting of lease times, and requests for lease renewal, a policy object is used. This needs something to work on, and to store results. The lease is over on the client side, and this needs to be done on the leasing side. The required information for this policy management is wrapped up in a leased resource
3. For each lease on the client side there will be a leased resource on the lessor side. These must be stored and managed somehow. There may be a few leases, but there could be many thousands. There could be relationships between them (such as linear order), or none at all. So to avoid decisions about storage structures which will be wrong half of the time, lease management is just left as an interface

The interfaces assume that they will be implemented in certain ways, in that each implementation class will contain a reference to certain other interfaces. This doesn't show in the interface specifications! If we have a landlord for, say a `Foo` resource, then we could end up with a class structure like

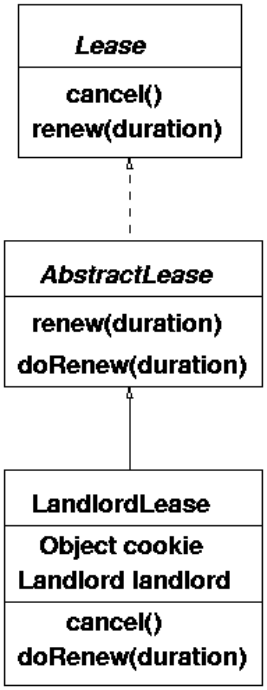


Some comments are appropriate about the directions and multiplicities:

1. A landlord can be managing many leases. But it doesn't know what the leases are - they know their landlord, and call its methods using the lease cookie. So many `LandlordLease` objects contain a reference to a `FooLandlord`
2. Certain requests need to get forwarded through the system. For example, a `renew()` request from a lease will get passed to a landlord. This cannot handle it directly, since this is a matter requiring policy decisions. It must be passed to a lease policy object. One way of doing this (as shown in the figure) is for the landlord to have a reference to a lease manager which has a reference to a lease policy. Similarly, a `newLease()` request from the landlord will need to invoke a `newLease()` method on the factory, and this can be done by ensuring that the lease policy also has a reference to the factory
3. A factory may be used by many lease policies; a policy may be used by many lease managers; a lease manager may be used by many landlords

3.3 Landlord Lease

The class `LandlordLease` extends `AbstractLease`. This has private fields `cookie` and `landlord`.



Implementation of the methods `cancel()` and `doRenew()` is deferred to its landlord.

```
public void cancel() {
    landlord.cancel(cookie);
}

protected long doRenew(long renewDuration) {
    return landlord.renew(cookie, renewDuration);
}
```

The class can be used as is, with no subclassing needed. Note that the landlord system *produces* these leases, but does not actually keep them anywhere - they are passed on to clients, which then use the lease to call the landlord and hence interact with the landlord lease system. Within this system the cookie is used as an identifier for the lease.

3.4 Leased Resource

A `LeasedResource` is a convenience wrapper around a resource that includes extra information and methods for use by landlords. It defines an interface

```
public interface LeasedResource {
    public void setExpiration(long newExpiration);
    public long getExpiration();
    public Object getCookie();
}
```

This includes the ‘‘cookie ’’, a unique identifier for a lease within a landlord system as well as expiration information for the lease. This is all the information maintained about the lease, which has been given out to a client.

An implementation will typically include the resource that is leased, plus a method of setting the cookie. For example,

```
/**
 * FooLeasedResource.java
 *
 *
 * Created: Mon Jun 14 18:04:33 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */
package foolandlord;

import com.sun.jini.lease.landlord.LeasedResource;

public class FooLeasedResource implements LeasedResource  {

    static protected int cookie = 0;
    protected int thisCookie;
    protected Foo foo;
    protected long expiration = 0;

    public FooLeasedResource(Foo foo) {
        this.foo = foo;
        thisCookie = cookie++;
    }

    public void setExpiration(long newExpiration) {
        this.expiration = newExpiration;
    }

    public long getExpiration() {
        return expiration;
    }
    public Object getCookie() {
        return new Integer(thisCookie);
    }
```

```
    public Foo getFoo() {
        return foo;
    }
} // FooLeasedResource
```

3.5 Lease Policy

A lease policy is used when a lease is first granted, and when it tries to renew itself. The time requested may be granted, modified or denied. A lease policy is specified by the `LeasePolicy` interface.

```
package com.sun.jini.lease.landlord;

public interface LeasePolicy {

    public Lease leaseFor(LeasedResource resource, long requestedDuration)
        throws LeaseDeniedException;

    public long renew(LeasedResource resource, long requestedDuration)
        throws LeaseDeniedException, UnknownLeaseException;

    public boolean ensureCurrent(LeasedResource resource);
}
```

This includes a ‘‘factory’’ method `leaseFor()` to return a lease based on the policy and request.

An implementation of this policy is given by the `LeaseDurationPolicy`. This grants and renews leases based on constant values for maximum and default lease durations.

```
package com.sun.jini.lease.landlord;

public class LeaseDurationPolicy implements LeasePolicy {

    public LeaseDurationPolicy(long maximum, long defaultLength,
        Landlord landlord, LeaseManager mgr, LandlordLeaseFactory factory);

    public Lease leaseFor(LeasedResource resource, long requestedDuration)
        throws LeaseDeniedException;
    public long renew(LeasedResource resource, long requestedDuration);
    public boolean ensureCurrent(LeasedResource resource);
}
```

In adddition to implementing the interface methods, the constructor also passes in the factory to be used (which will probably be a `LandlordLease.Factory`) and maximum and default lengths for leases. The maximum is to set a hard upper limit (which could be, say, `Lease.FOREVER`), while the default is if the client asks for a duration of `Lease.ANY`.

3.6 Lease Manager

The operations that can be carried out on a lease are creation, renewal and cancellation. The first two are subject to the lease policy, and must be handled by the `leaseFor()` and `renewed()` methods of the policy. This sets or alters the properties of a single lease. There may be many leases for a resource, or even many resources with one or more leases. Some level of management for a group of leases may be needed, and this is done by a `LeaseManager`. This defines the interface

```
package com.sun.jini.lease.landlord;

public interface LeaseManager {
    public void register(LeasedResource resource, long duration);
    public void renewed(LeasedResource resource, long duration,
                        long oldExpiration);
}
```

This doesn’t actually manage the leases, since they have been given to the client. Rather, it handles the lease *resource* which has the cookie identifier and the expiration time for the lease.

An implementation of this will look after a set of leases (really, their resources) in some manner, by adding a new lease resource to its set and by updating information about renewals. The interface does not include a method for informing the manager of cancelled leases, though - that is done to the `Landlord` instead, from the lease itself when *its* `cancel()` method is called.

This split responsibility between `LeaseManager` and `Landlord` is a little awkward and can possibly lead to memory leaks with the manager holding a reference to a lease (resource) that the landlord has cancelled. Either the list of lease resources must be shared between the two, or the landlord must ensure that it passes on cancellations to the manager.

There is also an issue of how the lease manager is informed of changes to individual leases by the lease policy.The `LeaseDurationPolicy` will pass on this information in its `leaseFor()` and `renew()` methods, but other implementations of `LeasePolicy` need not. As we only use this policy implementation, we are okay here.

A third issue is about who looks after leases expiring, and how this can be done. No part of the landlord specifications talk about this or give a suitable method. This suggests that it too is subject to some sort of policy, but it is not one with landlord support. It is left to implementations of one of the landlord interfaces or to a subclass. A convenient place to locate this checking is in the lease manager since it has knowledge of all the leases and their duration. Possible ways include

1. A thread per lease, which will sleep and timeout when the lease should expire. This will need to sleep again if the lease has been renewed in the meantime
2. A single sleeper thread sleeping for the minimum period of all leases. This may need to be interrupted if a new lease is created with a shorter expiration period
3. A “polling’’ mechanism in which a thread sleeps for a fixed time and then cleans up all leases that have expired in the meantime
4. A lazy method, in which no active thread looks for lease expiries, but just cleans them up if it comes across expired leases while doing something else (which is done by Outrigger with Entries)

The `FooLeaseManager` implements this third method

```
/**
 * FooLeaseManager.java
 *
```

```
 *
 * Created: Mon Jun 14 19:14:45 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */
package foolandlord;

import java.util.*;
import net.jini.core.lease.Lease;
import com.sun.jini.lease.landlord.LeaseManager;
import com.sun.jini.lease.landlord.LeasedResource;
import com.sun.jini.lease.landlord.LeaseDurationPolicy;
import com.sun.jini.lease.landlord.Landlord;
import com.sun.jini.lease.landlord.LandlordLease;
import com.sun.jini.lease.landlord.LeasePolicy;

public class FooLeaseManager implements LeaseManager {

    protected static long DEFAULT_TIME = 30*1000L;

    protected Vector fooResources = new Vector();
    protected LeaseDurationPolicy policy;

    public FooLeaseManager(Landlord landlord) {
        policy = new LeaseDurationPolicy(Lease.FOREVER,
                                         DEFAULT_TIME,
                                         landlord,
                                         this,
                                         new LandlordLease.Factory());
        new LeaseReaper().run();
    }

    public void register(LeasedResource r,long duration) {
        fooResources.add(r);
    }

    public void renewed(LeasedResource r, long duration, long olddur) {
        // no smarts in the scheduling, so do nothing
    }

    public void cancelAll(Object[] cookies) {
        for (int n = cookies.length; --n >= 0; ) {
            cancel(cookies[n]);
        }
    }

    public void cancel(Object cookie) {
        for (int n = fooResources.size(); --n >= 0; ) {
            FooLeasedResource r = (FooLeasedResource) fooResources.elementAt(n);
            if (r.getCookie().equals(cookie)) {
                fooResources.removeElementAt(n);
            }
        }
```



```
    }

    public LeasePolicy getPolicy() {
        return policy;
    }

    public LeasedResource getResource(Object cookie) {
        for (int n = fooResources.size(); --n >= 0; ) {
            FooLeasedResource r = (FooLeasedResource) fooResources.elementAt(n);
            if (r.getCookie().equals(cookie)) {
                return r;
            }
        }
        return null;
    }

    class LeaseReaper extends Thread {
        public void run() {
            while (true) {
                try {
                    Thread.sleep(DEFAULT_TIME) ;
                }
                catch (InterruptedException e) {
                }
                for (int n = fooResources.size()-1; n >= 0; n--) {
                    FooLeasedResource r = (FooLeasedResource)
                                          fooResources.elementAt(n)
;
                    if (!policy.ensureCurrent(r)) {
                        System.out.println("Lease expired for cookie = " +
                                           r.getCookie()) ;
                        fooResources.removeElementAt(n);
                        // replace this landlord.cancel(r.getCookie()) ;

                    }
                }
            }
        }
    }

} // FooLeaseManager
```

3.7 Landlord

The `Landlord` is the final interface in the package that we need for a basic landlord system. Other classes and interfaces, such as `LeaseMap` are for handling of leases in ‘‘batches’’, and will not be dealt with here. The `Landlord` interface is

```
package com.sun.jini.lease.landlord;

public interface Landlord extends Remote {

    public long renew(Object cookie, long extension)
```

```
        throws LeaseDeniedException, UnknownLeaseException, RemoteException;

    public void cancel(Object cookie)
        throws UnknownLeaseException, RemoteException;

    public RenewResults renewAll(Object[] cookie, long[] extension)
        throws RemoteException;

    public void cancelAll(Object[] cookie)
        throws LeaseMapException, RemoteException;
}
```

The `renew()` and `cancel()` methods are usually called from the `renew()` and `cancel()` methods of a particular lease. The `renew()` method needs to use a policy object to ask for renewal. In the `FooManager` implementation it gets this policy from the `FooLeaseManager`. The `cancel()` method needs to modify the list of leases. In the `FooManager` implementation it passes this onto the `FooLeaseManager` since that is the only object that maintains a list of resources.

There must be a method to ask for a new lease for a resource. This is not specified by the landlord package. This request will probably be made on the lease granting side and this should have access to the landlord object, which forms a central point for lease management. So an implementation of this interface will quite likely have a method such as

```
    public Lease newFooLease(Foo foo, long duration);
```

which will give a lease for a resource.

The lease used in the landlord package is a `LandlordLease`. This contains a private field, which is a reference to the landlord itself. The lease is given to a client as a result of `newFooLease()`, and this client will usually be a remote object. This will involve serialising the lease and sending it to this remote client. While serialising it, the landlord field will also be serialised and sent to the client. When the client methods such as `renew()` are called, the implementation of the `LandlordLease` will make a call to the landlord, which by then will be remote from its origin. So the landlord object invoked by the lease will need to be a remote object making a remote call. The `Landlord` interface already extends `Remote`, but if it is to run as a remote object, then the easiest way is for `FooLandlord` to extend the `UnicastRemoteObject` class.

Putting all this together for the `FooLandlord` class gives

```
/**
 * FooLandlord.java
 *
 *
 * Created: Mon Jun 14 22:48:11 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package foolandlord;

import com.sun.jini.lease.landlord.*;
import net.jini.core.lease.LeaseDeniedException;
import net.jini.core.lease.Lease;
import java.rmi.server.UnicastRemoteObject;
```



```
import java.rmi.Remote;

public class FooLandlord extends  UnicastRemoteObject
                          implements Landlord {

    FooLeaseManager manager;

    public FooLandlord() throws java.rmi.RemoteException {
        manager = new FooLeaseManager(this);
    }

    public void cancel(Object cookie) {
        manager.cancel(cookie);
    }

    public void cancelAll(Object[] cookies) {
        manager.cancelAll(cookies);
    }

    public long renew(java.lang.Object cookie,
                      long extension)
        throws net.jini.core.lease.LeaseDeniedException,
               net.jini.core.lease.UnknownLeaseException {
        LeasedResource resource = manager.getResource(cookie);
        if (resource != null) {
            return manager.getPolicy().renew(resource, extension);
        }
        return -1;
    }

    public Lease newFooLease(Foo foo, long duration)
        throws LeaseDeniedException {
        FooLeasedResource r = new FooLeasedResource(foo);
        return manager.getPolicy().leaseFor(r, duration);
    }

    public Landlord.RenewResults renewAll(java.lang.Object[] cookies,
                                          long[] extensions) {
        long[] granted = new long[cookies.length];
        Exception[] denied = new Exception[cookies.length];

        for (int n = cookies.length; --n >= 0; ) {
            try {
                granted[n] = renew(cookies[n], extensions[n]);
                denied[n] = null;
            } catch(Exception e) {
                granted[n] = -1;
                denied[n] = e;
            }
        }
        return new Landlord.RenewResults(granted, denied);
    }
} // FooLandlord
```

4. Summary

When a service registers with a lookup service, the lookup service returns a `Lease` object. This can be examined for lease duration, and requests for renewal may be made with it. Timing is a bit of a nuisance, so a `LeaseRenewalManager` can be used to handle renewal of leases.

Management of leases has also been dealt with on the lease granting side. This may be done directly, or by using the landlord management package.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

A Simple Example

Contents

1. Problem Description
2. Service Specification
3. Common Classes
    1. MIMEType
    2. FileClassifier interface
4. Client
    1. Unicast Client
    2. Multicast Client
5. Service Proxy
6. Uploading a Complete Service
    1. FileClassifier implementation
    2. Server
    3. Client
    4. What classes need to be where?
    5. Running the FileClassifier
7. Summary

This chapter looks at a simple problem, implementing it in a number of different ways

1. Problem Description

Applications often need to work out the type of a file, to see if it is a text file, an HTML document, an executable, etc. This can be done in two ways:

1. By examining the file's *name*
2. By examing the file's *contents*

Utilities such as the Unix `file` command use the second method, and have a complex description file (such as `/etc/magic` or `/usr/share/magic`) to aid in this. Many other applications such as Web browsers, mail readers (and even some operating systems!) use the first method and work out a file's type based on its name.

A common file classification is into MIME types such as `text/plain` and `image/gif`. There are tables of "official" MIME types (unofficial ones can be added on an *adhoc* basis), and there are also tables of mappings from filename endings to corresponding MIME types. These tables have entries such as

```
application/postscript          ai eps ps
application/rtf                 rtf
application/zip                 zip
image/gif                       gif
image/jpeg                      jpeg jpg jpe
text/html                       html htm
text/plain                      txt
```

and are stored in files for applications to access.

This storage of tables separate from the applications that would use them is rated as bad from the O/O point of view, since each application would need to have code to interpret the tables. The multiplicity of these tables and the ability of users to modify them makes this a maintenance problem. It would be better to encapsulate at least the filename to MIME type mapping table in an object. We define a MIME class

```
package standalone;

/**
 * MIMEType.java
 *
 *
 * Created: Mon Mar 15 22:09:13 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    moved to package standalone
 */

public class MIMEType  {

    /**
     * A MIME type is made up of 2 parts
     * contentType/subtype
     */
    protected String contentType;
    protected String subtype;

    public MIMEType(String type) {
        int slash = type.indexOf('/');
        contentType = type.substring(0, slash-1);
        subtype = type.substring(slash+1, type.length());
    }

    public MIMEType(String contentType, String subtype) {
        this.contentType = contentType;
        this.subtype = subtype;
    }

    public String toString() {
        return contentType + "/" + subtype;
    }
} // MIMEType
```

and a mapping class

```
package standalone;

/**
 * FileClassifier.java
 *
 *
 * Created: Mon Mar 15 22:18:59 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    moved to package standalone
 */

public class FileClassifier  {

    static MIMEType getMIMEType(String fileName) {
        if (fileName.endsWith(".gif")) {
            return new MIMEType("image", "gif");
        } else if (fileName.endsWith(".jpeg")) {
            return new MIMEType("image", "jpeg");
        } else if (fileName.endsWith(".mpg")) {
            return new MIMEType("video", "mpeg");
        } else if (fileName.endsWith(".txt")) {
            return new MIMEType("text", "plain");
        } else if (fileName.endsWith(".html")) {
            return new MIMEType("text", "html");
        } else
            // fill in lots of other types,
            // but eventually give up and
            return null;
    }
} // FileClassifier
```

This mapping class has no constructors, as it justs acts as a lookup table via its static method `getMIMEType()`.

Applications may make use these classes as they stand, by simply compiling with them and having the class files available at runtime. This will still result in duplication throughout JVMs, possible multiple copies of the class files, and potentially severe maintenance problems if applications need to be re-compiled. It may be better to have the `FileClassifier` as a network service. What will be involved in this?

2. Service Specification

If we wish to make a version of `FileClassifier` available across the network, there are a number of possibilities. The client will be asking for an instance of a class, and generally will not care too much about the details of this instance. For example, it will want an instance of a `DiskDrive` or a `Calendar`. Usually it will not care which drive it gets, or which calendar. If it requires further specification it can either ask for a subclass instance (such as a `SeagateDiskDrive`, or use an `Entry` object for this additional information.

Services will have particular implementations, and will upload these to the service locators. The uploaded service will be of a quite specific class, and may have associated entries.

There are several options that the client could use in trying to locate a suitable service.

Option 1
: The silly one: push the entire implementation upto the lookup service and make the client ask for it by its class. Then the client might just as well create the classifier as a local object as it has all the information needed! This doesn't lend itself to flexibility with new unknown services coming along if the client already has to know the details. So this option is not feasible.

Option 2
: Let the client ask for a superclass of the service. This is better, as it allows new implementations of a service to just be implemented as new subclasses. It is not ideal, as classes have implementation code, and if ever this changes over time then there is a maintenance issue with the possibility of version "skew" over time. This *can* be used for Jini: it just isn't the best way.

Option 3
: Separate the interface completely from the implementation. Make the interface available to the client, and upload the implementation to the lookup service. Then when the client asks for an instance object that implements the interface, it will get *any* object for this interface. This will reduce maintenance: if the client is coded just in terms of the interface then it will not need recompilation even if the implementation changes. Note that these words will translate straight into Java terms: the client knows about a Java `interface`, whereas the server deals in terms of a Java class that `implement`'s the `interface`.

The ideal mechanism in the Jini world is to specify services by interfaces, and have all clients know this interface. Then each service can be an implementation of this interface. This is simple in Java terms, simple in specification terms, and simple for maintenance. This is not the complete set of choices for the service, but is enough to allow a service to be *specified* and get on with building the client. The additional possibilities for service *implementation* are looked at later in this chapter.

3. Common Classes

The client and any implementations of a service must share some common classes. For a file classification service the common classes are the classifier itself (which can be implemented as many different services) and the return value, the MIME type. These have to change very slightly from their standalone form.

3.1 MIMEType

The class `MIMEType` is known to the client and to any file classifier service. The class files can be expected to be known to the JVMs of all clients and services. The method `getMIMEType()` will return an object from the service. There are implementation possibilities that can affect this object:

1. If the service runs in the client JVM, then nothing special needs to be done
2. If the service runs in a separate JVM, then the `MIMEType` object must be serialized for transport to the client JVM. For this to be possible, it must implement the `Serializable` interface

There is a difference in the object depending on possible implementations. If it implements `Serializable` then it can be used in both the remote and local cases, but if it doesn't then it can only be used in the local case.

Making decisions about interfaces based on future implementation concerns is traditionally rated as poor design. However, a document from Sun "A Note on Distributed Computing" by Jim Waldo *et al* argues that in the case of distributed objects it is important to determine if the objects may be running remotely and adjust interfaces and classes accordingly at the design stage. This is to take into account possible extra failure modes of methods, and in this case, an extra requirement on the object. The paper is reprinted in the Jini specification book from Sun and is also at http://www.sun.com/research/techrep/1994/abstract_29.html

This leads to an interface which adds the `Serializable` interface to the previous version.


```
package common;

import java.io.Serializable;

/**
 * MIMEType.java
 *
 *
 * Created: Wed Mar 17 14:17:32 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    moved to package common
 */

public class MIMEType implements Serializable {

    /**
     * A MIME type is made up of 2 parts
     * contentType/subtype
     */
    protected String contentType;
    protected String subtype;

    public MIMEType(String type) {
        int slash = type.indexOf('/');
        contentType = type.substring(0, slash-1);
        subtype = type.substring(slash+1, type.length());
    }

    public MIMEType(String contentType, String subtype) {
        this.contentType = contentType;
        this.subtype = subtype;
    }

    public String toString() {
        return contentType + "/" + subtype;
    }
} // MIMEType
```

3.2 FileClassifier interface

Changes have to be made to the file classifier interface as well. Firstly, interfaces cannot have `static` methods, so we shall have to turn the method `getMIMEType()` into a public instance method. In addition, instances implementing this may need to be shipped around as proxy objects, so it should extend `Serializable`. This is the same situation as with `MIMEType`.

In addition, all methods are defined to throw a `java.rmi.RemoteException`. This type of exception is used by Java (not just the RMI component) to mean "a network error has occurred". This could be a lost connection, a missing server, a class not downloadable, *etc.* There is a little subtlety here, related to the `java.rmi.Remote` class: the methods of `Remote` must all throw a `RemoteException`, but the converse is not true. If all the methods throw `RemoteException`, it does not mean the class implements/extends `Remote`. It only means that an implementation *may* be implemented as a remote (distributed) object, and this implementation might also use the RMI `Remote` interface.

This gives the following interface

```
package common;

import java.io.Serializable;

/**
 * FileClassifier.java
 *
 *
 * Created: Wed Mar 17 14:14:36 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    moved to package common
 */

public interface FileClassifier extends Serializable {

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException;

} // FileClasssifier
```

Why does this interface throw a `java.rmi.RemoteException` in the `getMIMEType()` method? Well, an interface is supposed to be above all possible implementations, and should never change. Some of the implementations discussed in later sections do not throw such an exception. However, other implementations in other sections use a `Remote` implementation, and this will require that the method throws an `java.rmi.RemoteException`. Since it is not possible to just add a new exception in a subclass or interface implementation, the possibility must be added in the interface specification.

4. Client

The client is the same for all of the possible server implementations discussed in later sections and in later chapters. The client does not care how the server-side implementation is done, just as long as it gets a service that it wants, and it specifies this by asking for a `FileClassifier` interface.

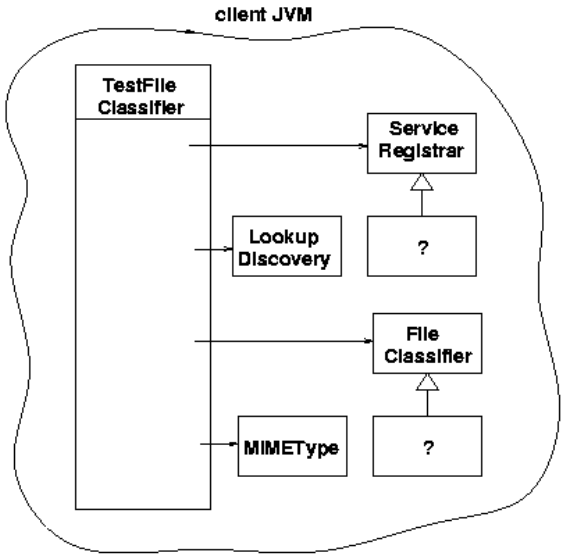
4.1 Unicast Client

If there is a known service locator which will know about the service, then there is no need to search. This doesn't mean that the location of the service is known, only of the locator. For example, there may be a (fictitious) organisation "All About Files" at `www.all_about_files.com` that would know about various file services, keeping track of them as they come on line, move, disappear, *etc.* A client would ask the service locator running on this site for the service, wherever it is. This uses the unicast lookup techniques.

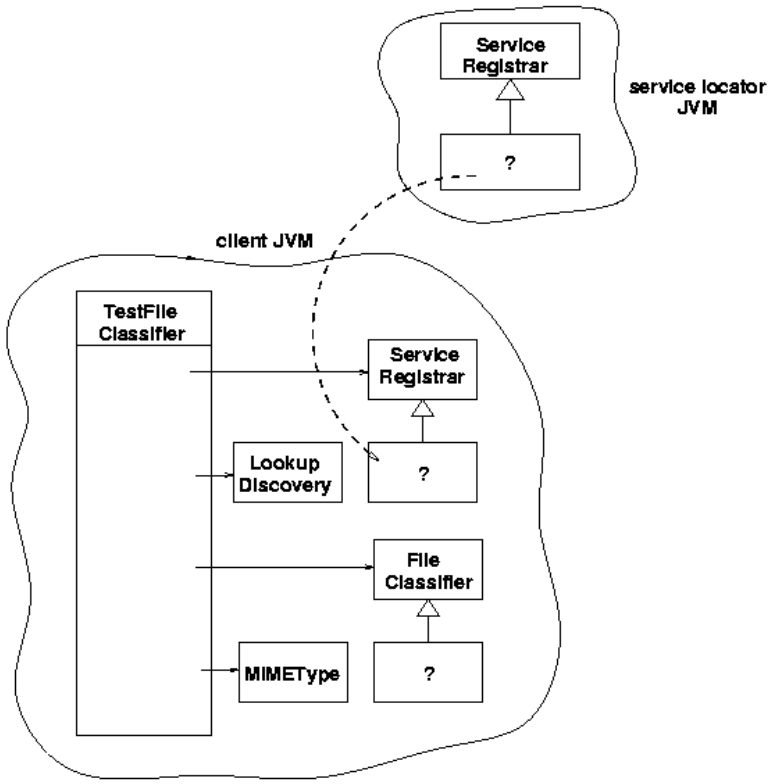

```
package client;

import common.FileClassifier;
import common.MIMEType;

import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import java.rmi.RMISecurityManager;
import net.jini.core.lookup.ServiceTemplate;

/**
 * TestUnicastFileClassifier.java
 *
 *
 * Created: Wed Aug 04
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class TestUnicastFileClassifier {

    public static void main(String argv[]) {
        new TestUnicastFileClassifier();
    }

    public TestUnicastFileClassifier() {
        LookupLocator lookup = null;
        ServiceRegistrar registrar = null;
        FileClassifier classifier = null;

        try {
            lookup = new LookupLocator("jini://www.all_about_files.com");
        } catch(java.net.MalformedURLException e) {
            System.err.println("Lookup failed: " + e.toString());
            System.exit(1);
        }

        System.setSecurityManager(new RMISecurityManager());

        try {
            registrar = lookup.getRegistrar();
        } catch (java.io.IOException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        } catch (java.lang.ClassNotFoundException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        }

        Class[] classes = new Class[] {FileClassifier.class};
```

```
        ServiceTemplate template = new ServiceTemplate(null, classes, null);
        try {
            classifier = (FileClassifier) registrar.lookup(template);
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        }

        if (classifier == null) {
            System.out.println("Classifier null");
            System.exit(2);
        }
        MIMEType type;
        try {
            type = classifier.getMIMEType("file1.txt");
            System.out.println("Type is " + type.toString());
        } catch(java.rmi.RemoteException e) {
            System.err.println(e.toString());
        }
        // System.exit(0);
    }
} // TestUnicastFileClassifier
```

The client's JVM looks like



It has objects that implement the interfaces `ServiceRegistrar` and `FileClassifier`, but it doesn't know - or need to know - what classes they are. When the service locator's JVM is added in, it looks like



This shows that the client gets its `registrar` from the JVM of the service locator. This object is not specified in detail. Sun supply a service locator known as `reggie`. This locator implements the `ServiceRegistrar` using an implementation that neither clients nor services are expected to know. The classes which implement this object are contained in the file `reggie-dl.jar` and are downloaded to the clients and services using (typically) an HTTP server.

4.2 Multicast Client

More likely, client will need to search through all of the service locators till it finds one holding a service it is looking for. It would need to use a multicast search for this. If it only needs one occurrence of the service, then it can do something like exit after using the service. More complex behaviour will be illustrated in later examples. The client does not need to have long-term persistence. But it does need a user thread to remain in

existence for long enough to find service locators and find a suitable service. So in `main()` a user thread again sleeps for a short period (ten seconds).

```
package client;

import common.FileClassifier;
import common.MIMEType;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;

/**
 * TestFileClassifier.java
 *
 *
 * Created: Wed Mar 17 14:29:15 1999
 *
 * @author Jan Newmarch
 * @version 1.3
 *    moved sleep() from constructor to main()
 *    moved to package client
 *    simplified Class.forName to Class.class
 */

public class TestFileClassifier implements DiscoveryListener {

    public static void main(String argv[]) {
        new TestFileClassifier();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(100000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestFileClassifier() {
        System.setSecurityManager(new RMISecurityManager());

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
```

```
}

public void discovered(DiscoveryEvent evt) {

    ServiceRegistrar[] registrars = evt.getRegistrars();
    Class [] classes = new Class[] {FileClassifier.class};
    FileClassifier classifier = null;
    ServiceTemplate template = new ServiceTemplate(null, classes,
                                                   null);

    for (int n = 0; n < registrars.length; n++) {
        System.out.println("Service found");
        ServiceRegistrar registrar = registrars[n];
        try {
            classifier = (FileClassifier) registrar.lookup(template);
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
            System.exit(2);
        }
        if (classifier == null) {
            System.out.println("Classifier null");
            continue;
        }
        MIMEType type;
        try {
            String fileName;

            fileName = "file1.txt";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);

            fileName = "file2.rtf";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);

            fileName = "file3.abc";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);
        } catch(java.rmi.RemoteException e) {
            System.err.println(e.toString());
        }
        // System.exit(0);
    }
}

private void printType(String fileName, MIMEType type) {
    System.out.print("Type of " + fileName + " is ");
    if (type == null) {
        System.out.println("null");
    } else {
        System.out.println(type.toString());
    }
}
```



```
    public void discarded(DiscoveryEvent evt) {
        // empty
    }
} // TestFileClassifier
```

5. Service Proxy

A *service* will be delivered from out of a *server*. That is, a server will be started, create one or more service objects, register these with service locators, and then wait for network requests to come in for the service. What the server will actually export is usually a *proxy* for the service. The proxy is an object that will eventually run in a client, and will make calls back across the network to the service object within the server.

The service and its proxy are tightly integrated: they must communicate using a protocol known to them both, and must exchange information in an agreed manner. However, the relative *size* of each is up to the designer of a service and its proxy. For example, the proxy may be ''fat'', which means it does a lot of processing on the client side. The service itself is then typically ''thin'', not doing much at all. Alternatively, the proxy may be thin, doing little more (or nothing more) than passing requests between the client and the fat service.

As well as this choice of size, there is also a choice of communication mechanisms between the two. Client-server systems often have the choice of *message-based* or *remote procedure call*. These choices are also available between a Jini proxy and its service. Since they are both in Java, there is a standard RPC-like mechanism called RMI (Remote Method Invocation), and this can be used if wanted. There is *no* necessity to use this, but many implementations of Jini proxies will do so since it is easy. It does force a particular choice of thin proxy to a fat client, though, and this may not be ideal for all situations.

This chapter will look at one possibility only, where the proxy is fat and is the whole of the service. The next chapter will look in more detail at the other possibilities.

6. Uploading a Complete Service

The file classifier service does not rely on any particular properties of its host - it is not hardware or operating system dependant, for example. In this case it is possible to upload the entire service to the client and let it run there. The proxy is the service, and no processing elements need to be left on the server.

6.1 FileClassifier implementation

The implementation of this is straightforward

```
package option2;

import common.MIMEType;
import common.FileClassifier;

/**
 * FileClassifierImpl.java
 *
 *
 * Created: Wed Mar 17 14:22:13 1999
 *
 * @author Jan Newmarch
 * @version 1.0
```

```
 */

public class FileClassifierImpl implements FileClassifier {

    public MIMEType getMIMEType(String fileName) {
        if (fileName.endsWith(".gif")) {
            return new MIMEType("image", "gif");
        } else if (fileName.endsWith(".jpeg")) {
            return new MIMEType("image", "jpeg");
        } else if (fileName.endsWith(".mpg")) {
            return new MIMEType("video", "mpeg");
        } else if (fileName.endsWith(".txt")) {
            return new MIMEType("text", "plain");
        } else if (fileName.endsWith(".html")) {
            return new MIMEType("text", "html");
        } else
            // fill in lots of other types,
            // but eventually give up and
            return null;
    }


    public FileClassifierImpl() {
        // empty
    }

} // FileClassifierImpl
```

6.2 Server

The server for this needs to create an instance of the exportable service, register this and keep the lease alive. In the `discovered()` method it not only registers the service but also adds it to a `LeaseRenewalManager`, to keep the lease alive "forever". This manager runs its own threads to keep re-registering the leases, but these are daemon threads. So in the `main()` method the user thread goes to sleep for as long as we want the server to stay around. Note that if the server does terminate, then the lease will fail to be renewed and the exported service will be discarded from lookup locators even though the server is not required for delivery of the service. The `serviceID` is initially set to `null` and then extracted from the first lookup service that relies and re-used. Ideally it should also be stored (say on disk) and retrieved the next time the service starts. Note that we get the service id from the *registration* not the *registrar*.

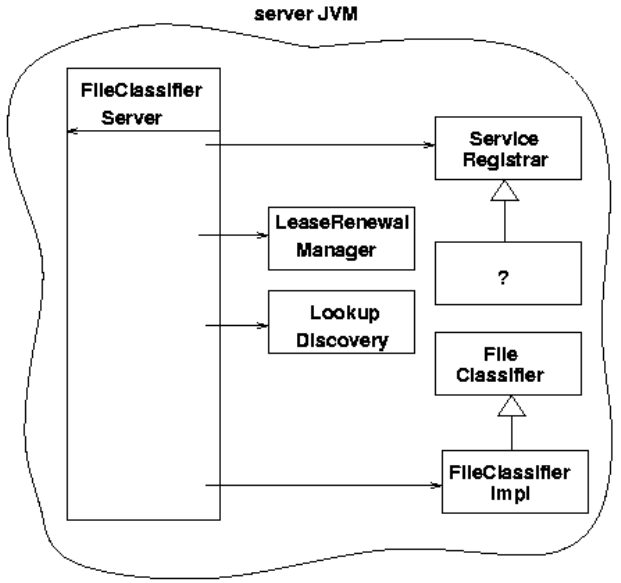
```
package option2;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
import net.jini.core.lookup.ServiceID ;
// import com.sun.jini.lease.LeaseRenewalManager; // Jini 1.0
// import com.sun.jini.lease.LeaseListener;       // Jini 1.0
```



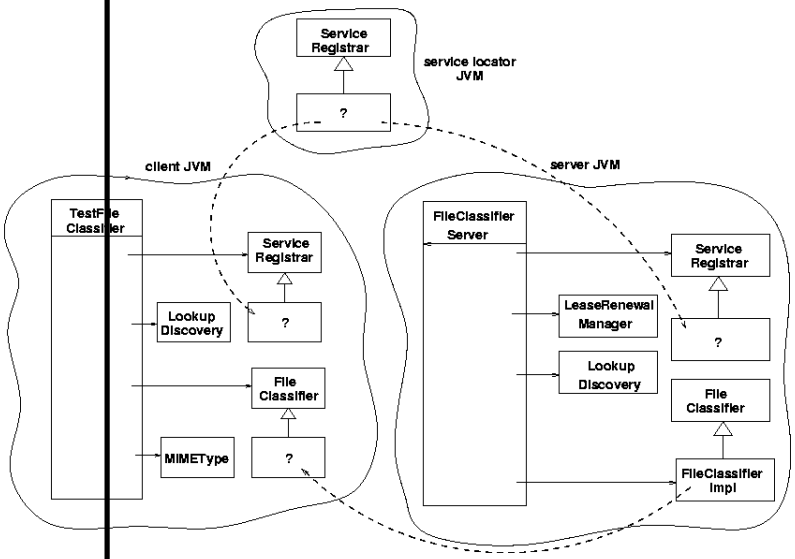
```
// import com.sun.jini.lease.LeaseRenewalEvent;   // Jini 1.0
import net.jini.lease.LeaseListener;              // Jini 1.1
import net.jini.lease.LeaseRenewalEvent;          // Jini 1.1
import net.jini.lease.LeaseRenewalManager;        // Jini 1.1

/**
 * FileClassifierServer.java
 *
 *
 * Created: Wed Mar 17 14:23:44 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    added LeaseRenewalManager
 *    moved sleep() from constructor to main()
 */

public class FileClassifierServer implements DiscoveryListener,
                                             LeaseListener {

    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();
    protected ServiceID serviceID = null;

    public static void main(String argv[]) {
        new FileClassifierServer();

        // keep server running forever to
        // - allow time for locator discovery and
        // - keep re-registering the lease
                try {
            Thread.currentThread().sleep(Lease.FOREVER);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public FileClassifierServer() {

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();

        for (int n = 0; n < registrars.length; n++) {
```

```
            ServiceRegistrar registrar = registrars[n];

            ServiceItem item = new ServiceItem(serviceID,
                                               new FileClassifierImpl(),
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.println("Register exception: " + e.toString());
            }
            System.out.println("service registered");

            // set lease renewal in place
            leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);

            // set the serviceID if necessary
            if (serviceID == null) {
                serviceID = reg.getServiceID();
            }
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }

    public void notify(LeaseRenewalEvent evt) {
        System.out.println("Lease expired " + evt.toString());
    }

} // FileClassifierServer
```

The server by itself running in its JVM looks like



This receives an object implementing `ServiceRegistrar` from the service locator (such as `reggie`). If we add in the service locator and the client in their JVM's, it becomes



The unknown `FileClassifier` object in the client is here supplied by the service object `FileClassifierImpl`.

6.3 Client

The client for this service was discussed earlier. The client does not need any special information about this implementation of the service and so can remain quite generic.

6.4 What classes need to be where?

We have the classes

1. `common.MIMEType`
2. `common.FileClassifier`
3. `option2.FileClassifierImpl`
4. `option2.FileClassifierServer`
5. `client.TestFileClassifier`

These could be running on upto four different machines

1. The server machine for `FileClassifier`
2. The machine for the lookup service
3. The machine running the client `TestFileClassifier`
4. An HTTP server will need to run somewhere to deliver the class file definition of `FileClassifierImpl` to clients

What classes need to be known to which machines?

The server running `FileClassifierServer` needs to know the following classes and interfaces

1. The `common.FileClassifier` interface
2. The class `common.MIMEType`
3. The class `option2.FileClassifierServer`
4. The class `option2.FileClassifierImpl`

These classes all need to be in the `CLASSPATH` of the server.

The class `option2.FileClassifierImpl` will need to be accessible to an HTTP server, as discussed in the next section.

The lookup service does not need to know any of these classes. It just deals with them in the form of a `java.rmi.MarshalledObject`

The client needs to know

1. The `common.FileClassifier` interface
2. The class `common.MIMEType`
3. The class `client.TestFileClassifier`

6.5 Running the FileClassifier

We now have a service `FileClassifierServer` and a client `TestFleClassifier` to run. There should also be at least one lookup locator already running. The `CLASSPATH` should be set for each to include the classes discussed in the last section, in addition to the standard ones.

A serialized instance of `option2.FileClassifierImpl` will be passed from the server to the locator and then to the client. Once on the client, it will need to be able to run the class file for this object, so will need to load its class file from an HTTP server. The location of this class file relative to the server's `DocumentRoot` will need to be specified by the service invocation. For example, if it is stored in `/DocumentRoot/classes/option2/FileClassifierImp.class` then the service will be started by:

```
java -Djava.rmi.codebase=http://hostname/classes \
     option2.FileClassifierServer
```

In this, `hostname` is the name of the host the server is running on. Note that this host name *cannot* be `localhost`, because the local host for the server will not be the local host for the client!

The client will be loading a class definition across the network. It will need to allow this in a security policy file by

```
java -Djava.security.policy=policy.all client.TestFileClassifier
```

7. Summary

The material of the previous chapters is put together in a couple of examples. The requirements of class structures for a Jini system are discussed. Two options, of uploading a complete service and of uploading a proxy only are given. A discussion is also given of what classes need to be available to each component of a Jini system.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Choices for Service Architecture

Contents

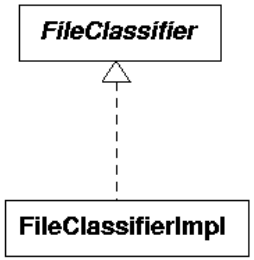
1. Proxy Choices
    1. Proxy is the Service
    2. RMI Proxy
    3. Non-RMI Proxy
    4. RMI and Non-RMI Proxies
2. RMI Proxy for FileClassifier
    1. What Doesn't Change
    2. RemoteFileClassifier
    3. FileClassifierImpl
    4. FileClassifierServer
    5. What classes need to be where
    6. Running the RMI Proxy FileClassifier
3. Non-RMI Proxy for FileClassifier
    1. FileClassifierProxy
    2. FileServerImpl
    3. Server
    4. What classes need to be where2
    5. Running the RMI Proxy FileClassifier
4. RMI and non-RMI Proxies for FileClassifier
    1. FileClassifierProxy
    2. ExtendedFileClassifier
    3. ExtendedFileClassifierImpl
    4. FileClassifierServer
    5. What classes need to be where?
5. Using the Naming service
    1. Server using Naming Service
    2. Proxy using Naming Service
6. Using Other Services
    1. Heart interface
    2. Heart implementation
    3. Heart server
    4. Heart client

Most applications will make some use of RMI (Remote Method Invocation). There are several ways in which this may be done. In addition, it is possible to avoid use of RMI proxies completely. This chapter looks at such RMI-related issues.

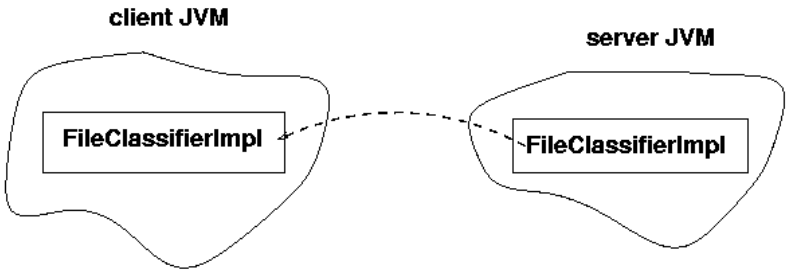
1. Proxy Choices

1.1 Proxy is the Service

One extreme is where the proxy is so fat, that there is nothing left to do on the server side. The role of the server is to register the proxy with service locators, and just to stay alive (renewing leases on the service locators). The service itself runs entirely within the client. A class diagram for the file classifier problem using this method is



We have already seen the full object diagram for the JVMs, but just concentrating on these classes looks like

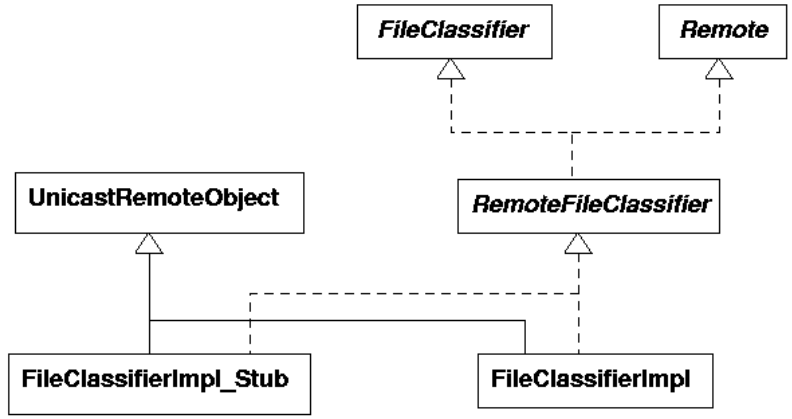


The client asks for a `FileClassifier`. What is uploaded to the service locators, and so what the client gets, is a `FileClassifierImpl`. The `FileClassifierImpl` runs entirely within the client, and does not communicate back to its server at all. This can also be done for any service, when the service is purely a software one which does not need any link back to the server. It could be something like a calendar which is independent of location, or a diary which uses files on the client side rather than the server side.

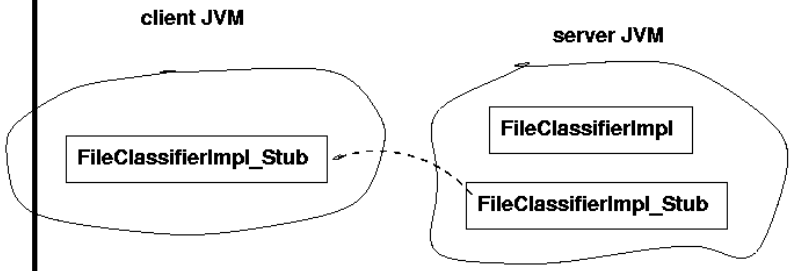
1.2 RMI Proxy

The opposite extreme to this is where *all* of the processing is done on the server side. The proxy just exists on the client to take calls from the client, invoke the method in the service on the server, and return the result to the client. Java's RMI does this in a fairly transparent way (once all the correct files and additional servers are set up!)

A class diagram for an implementation of the file classifier using this mechanism is



The objects in the JVM's are



The full code for this is given in the section ''RMI Proxy for FileClassifier''.

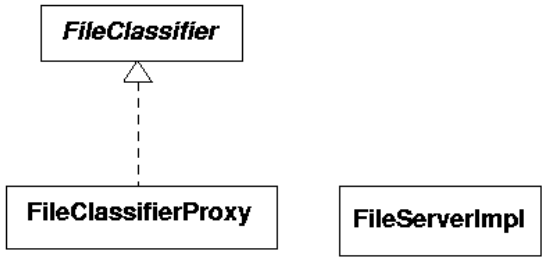
The class structure for this is much more complex in order to get RMI to work. The interface `RemoteFileClassifier` has to be defined, and the implementation class has to implement (or call suitable methods from) the `UnicastRemoteObject` class. The `FileClassifierImpl_Stub` is generated from `FileClassifierImpl` by use of the `rmic` compiler. Implementing the `Remote` interface allows the methods of the `FileClassifierImpl` to be called remotely. Inheriting from `UnicastRemoteObject` allows RMI to export the stub rather than the service, which remains on the server.

Apart from creating the stub class by using `rmic`, the stub is essentially invisible to the programmer: the server code is written to export the implementation, but the RMI runtime component of Java recognises this and actually exports the stub instead. This can cause a little confusion: the programmer writes code to export an object of one class, but an object of a different one appears in the service locator and in the client.

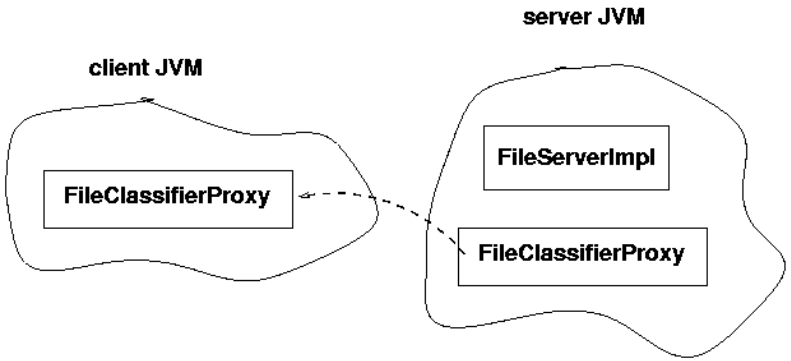
This structure is useful when the service needs to do no processing on the client side, but does need to do a lot on the server side. For example, a diary that stores all information communally on the server rather than individually on each client. Services that are tightly linked to a piece of hardware on the server are other cases.

1.3 Non-RMI Proxy

If RMI is not used and the proxy and service want to share processing, then both the service and the proxy must be created explicitly on the server side. The proxy is explicitly exported by the server and must implement the interface. But on the server side this requirement does not hold, since the proxy and service are not tightly linked by a class structure any more. The class diagram for the file classifier with this organisation is



and the JVM's at runtime are



This doesn't specify how the proxy and the server communicate. They could open up a socket connection, for example and exchange messages using a message structure only they understand. Or they could communicate using a well-known protocol, such as HTTP for example: the proxy could make HTTP requests, and the service could act as an HTTP server handling these requests and returning documents. A version of the file classifier using sockets to communicate is given in the section ''Non-RMI Proxy for FileClassifier''.

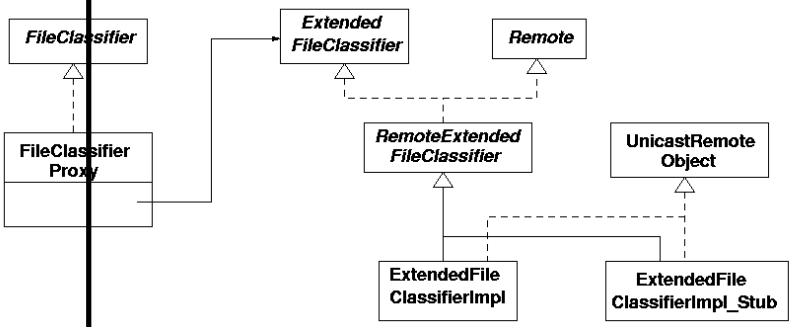
This model is good for bringing ''legacy'' client-server applications into the Jini world. Client-server applications often communicate using a specialised protocol between the client and server. Copies of the client have to be distributed out to all machines. If there is a bug in the client, they all have to be updated which is often impossible. Worse, if there is a change to the procotol, then the server must be rebuilt to handle old and new versions while attempts are made to update all the clients. This is a tremendous problem with Web browsers for example, with varying degrees of support for HTML 3.2 and HTML 4.0 features, let alone new protocol extensions such as stylesheets and XML. CGI scripts that attempt to deliver the ''right'' version of documents to various browsers are clumsy but necessary hacks.

What can be done instead is to distribute a ''shell'' client, that just contacts the server and uploads a proxy. The proxy is the real ''heart'' of the client, whereas the service is the server part of the original client-server system. When changes occur, the service and its proxy can be updated together, and there is no need to make changes to the ''shell' out on all the various machines.

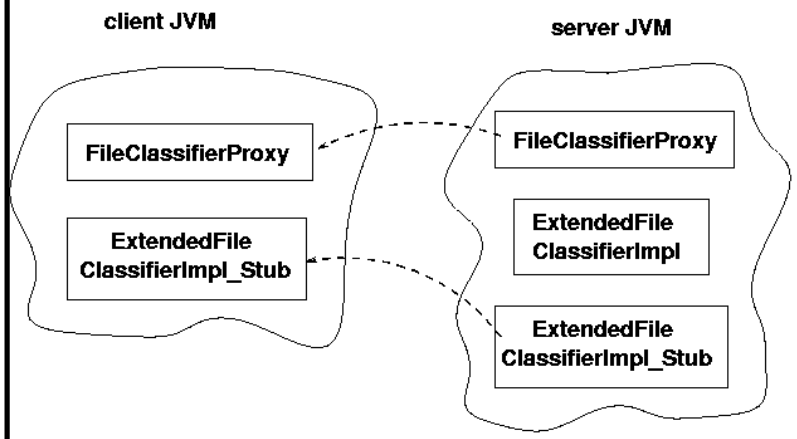
1.4 RMI and Non-RMI Proxies

The last variation is to have a service, an explicit proxy and an RMI proxy. Both of the proxies are exported: the explicit proxy has to be exported by registering it with lookup services, while the RMI proxy is exported by the RMI runtime mechanisms. The RMI proxy can be used for RPC-like communication mechanism between the explicit proxy and the service. This is just like the last case, but instead of requiring the proxy and service to implement their own communication protocol, it uses RMI instead. The proxy and service can be of any relative size, just like in the last case. What this does is to simplify the task of the programmer.

In the later section ''RMI and non-RMI Proxies for FileClassifier'' there is a non-RMI proxy `FileClassifierProxy` implementing the `FileClassifier` interface. This communicates with an object which implements the `ExtendedFileClassifier` interface. There is an object on the server of type `ExtendedFileClassifierImpl` and an RMI proxy for this on the client side of type `ExtendedFileClassifierImpl_Stub` The class diagram is



While this looks complex, it is really just a combination of the last two cases. The proxy makes local calls on the RMI stub which makes remote calls on the service. The JVM's look like



2. RMI Proxy for FileClassifier

An RMI proxy can be used when all of the work done by the service is done on the server side. Then the server exports a thin proxy that simply channels method calls from the client across the network to the "real" service in the server, and returns the result back to the client. The programming for this is relatively simple. The service has to do two major things in its class structure:

1. Implement `Remote`. This is because methods will be called on the service from the proxy, and these will be remote calls on the service
2. Inherit from `UnicastRemoteObject` (or `Activatable`). This means that its constructor will create and export a proxy or stub object, without the programmer having to do anything more. (An alternative to inheritance is for the object to call the `UnicastRemoteObject.exportObject()` method.)

2.1 What doesn't change

The client is not concerned about the implementation of the service at all, and doesn't change. The `FileClassifier` interface doesn't change either, since this is fixed and used by any client and any service implementation. We have already declared its methods to throw `RemoteException` so a proxy is able to call its methods remotely. And the `MIMEType` doesn't change since we have already declared it to implement `Serializable` - it is passed back across the network from the service to its proxy

2.2 RemoteFileClassifier

The implementation will need to implement both the `FileClassifier` and the `Remote` interface. It is convenient to define another interface `RemoteFileClassifier` just to do this.

```
package option3;

import common.FileClassifier;
```



```
import java.rmi.Remote;

/**
 * RemoteFileClassifier.java
 *
 *
 * Created: Wed Mar 17 14:14:36 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public interface RemoteFileClassifier extends FileClassifier, Remote {

} // RemoteFileClasssifier
```

2.3 FileClassifierImpl

The service itself will look like

```
package option3;

import java.rmi.server.UnicastRemoteObject;
import common.MIMEType;
import common.FileClassifier;

/**
 * FileClassifierImpl.java
 *
 *
 * Created: Wed Mar 17 14:22:13 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class FileClassifierImpl extends UnicastRemoteObject
                                implements RemoteFileClassifier {

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        System.out.println("Called with " + fileName);
        if (fileName.endsWith(".gif")) {
            return new MIMEType("image", "gif");
        } else if (fileName.endsWith(".jpeg")) {
            return new MIMEType("image", "jpeg");
        } else if (fileName.endsWith(".mpg")) {
            return new MIMEType("video", "mpeg");
        } else if (fileName.endsWith(".txt")) {
            return new MIMEType("text", "plain");
        } else if (fileName.endsWith(".html")) {
            return new MIMEType("text", "html");
```

```
        } else
            // fill in lots of other types,
            // but eventually give up and
            return new MIMEType(null, null);
    }


    public FileClassifierImpl()  throws java.rmi.RemoteException {
        // empty
    }

} // FileClassifierImpl
```

2.4 FileClassifierServer

A server which creates and manages this service is

```
package rmi;

import option3.FileClassifierImpl;
import option3.RemoteFileClassifier;

import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
// import com.sun.jini.lease.LeaseRenewalManager;
// import com.sun.jini.lease.LeaseListener;
// import com.sun.jini.lease.LeaseRenewalEvent;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lease.LeaseListener;
import net.jini.lease.LeaseRenewalEvent;
import java.rmi.RMISecurityManager;

/**
 * FileClassifierServer.java
 *
 *
 * Created: Wed Mar 17 14:23:44 1999
 *
 * @author Jan Newmarch
 * @version 1.2
 *    added LeaseRenewalManager
 *    moved sleep() from constructor to main()
 *    uses Jini 1.1 LeaseRenewalManager
 */

public class FileClassifierServerRMI implements DiscoveryListener, LeaseListener {
```



```
protected FileClassifierImpl impl;
protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

public static void main(String argv[]) {
    new FileClassifierServerRMI();

    // no need to keep server alive, RMI will do that
}

public FileClassifierServerRMI() {
    try {
        impl = new FileClassifierImpl();
    } catch(Exception e) {
        System.err.println("New impl: " + e.toString());
        System.exit(1);
    }

    // install suitable security manager
    System.setSecurityManager(new RMISecurityManager());

    LookupDiscovery discover = null;
    try {
        discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
    } catch(Exception e) {
        System.err.println(e.toString());
        System.exit(1);
    }

    discover.addDiscoveryListener(this);
}

public void discovered(DiscoveryEvent evt) {

    ServiceRegistrar[] registrars = evt.getRegistrars();
    RemoteFileClassifier service;

    for (int n = 0; n < registrars.length; n++) {
        ServiceRegistrar registrar = registrars[n];

        // export the proxy service
        ServiceItem item = new ServiceItem(null,
                                           impl,
                                           null);
        ServiceRegistration reg = null;
        try {
            reg = registrar.register(item, Lease.FOREVER);
        } catch(java.rmi.RemoteException e) {
            System.err.print("Register exception: ");
            e.printStackTrace();
            // System.exit(2);
            continue;
        }
        try {
            System.out.println("service registered at " +
```

```
                                registrar.getLocator().getHost());
            } catch(Exception e) {
            }
            leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }

    public void notify(LeaseRenewalEvent evt) {
        System.out.println("Lease expired " + evt.toString());
    }

} // FileClassifierServerRMI
```

2.5 What classes need to be where?

We have the classes

1. `common.MIMEType`
2. `common.FileClassifier`
3. `option3.RemoteFileClassifier`
4. `option3.FileClassifierImpl`
5. `option3.FileClassifierImpl_Stub`
6. `option3.FileClassifierServer`
7. `client.TestFileClassifier`

(The class `FileClassifierImpl_Stub` is added to our classes by `rmic` as discussed in the next section.) These could be running on upto four different machines

1. The server machine for `FileClassifierServer`
2. The HTTP server, which may be on a different machine
3. The machine for the lookup service
4. The machine running the client `TestFileClassifier`

What classes need to be known to which machines?

The server running `FileClassifierServer` needs to know the following classes and interfaces

1. The `common.FileClassifier` interface
2. The `option3.RemoteFileClassifier` interface
3. The class `common.MIMEType`
4. The class `option3.FileClassifierServer`
5. The class `option3.FileClassifierImpl`

The lookup service does not need to know any of these classes. It just deals with them in the form of a `java.rmi.MarshalledObject`

The client needs to know

1. The `common.FileClassifier` interface
2. The class `common.MIMEType`

In addition, the HTTP server needs to be able to load and store classes. It needs to be able to access

1. The `option3.FileClassifierImpl_Stub` interface
2. The `option3.RemoteFileClassifier` interface
3. The `common.FileClassifier` interface
4. The class `common.MIMEType`

The reason for all of these is slightly complex. In the `FileClassifierProxy` constructor, the class `FileClassifierImpl` is passed in. The RMI runtime converts this to `FileClassifierImpl_Stub`. This class implements the same interfaces as `FileClassifierImpl`: that is, `RemoteFileClassifier` and hence `FileClassifier`, so these also need to be available. In the implementation, `FileClassifierImpl` references the class `MIMEType`, so this must also be available.

What does the phrase "available" mean in the last paragraph? The HTTP server will look for files based on the `java.rmi.server.codebase` property of the application server. The value of this property is a URL. Often, URLs can be *file* or *http* references. But for this case, the URL will be used by clients running anywhere, so it cannot be a file reference specific to a particular machine. For the same reason, it cannot be just `localhost` - unless you are running every part of a Jini federation on a single computer!

If `java.rmi.server.codebase` is an http reference, then the above class files must be accessible from that reference. For example, suppose the property is set to

```
java.rmi.server.codebase=http://myWebHost/classes
```

(where `myWebHost` is the name of the HTTP server's host) and this Web server has its `DocumentRoot` set to `/home/webdocs` then these files must exist

```
/home/webdocs/classes/option3/FileClassifierImpl_Stub.class
/home/webdocs/classes/option3/RemoteFileClassifier.class
/home/webdocs/classes/common/FileClassifier.class
/home/webdocs/classes/common/MIMEType.class
```

2.6 Running the RMI Proxy FileClassifier

Again we have a server and a client to run. Calling the client is unchanged from option2, with a security policy required.

```
java -Djava.security.policy=policy.all client.TestFileClassifier
```

The server is more complex, because the RMI runtime is manipulating RMI stubs, and these have additional requirements. Firstly, RMI stubs must be generated during compilation. Secondly, security rights must be set since an `RMISecurityManager` is used.

Although the `FileClassifierImpl` is created explicitly by the server, it is not this class file that is moved around. It continues to exist on the server machine. Rather, a *stub* file is moved around, and will run on the client machine. This stub is responsible for sending the method requests back to the implementation class on the server. This stub has to be generated from the implementation class by the stub compiler `rmic`:

```
rmic -v1.2 -d /home/webdocs/classes option3.FileClassifierImpl
```

where the `-v1.2` option says to generate JDK 1.2 stubs only, and the `-d` option says where to place the resultant stub class files so that they can be located by the HTTP server (in this case, in the local file system). Note that the pathnames for directories here and later do not include the package name of the class files. The

class files (here `FileClassifierImpl_Stub.class`) will be placed/looked for in the appropriate subdirectories.

The value of `java.rmi.server.codebase` must specify the protocol used by the HTTP server to find the class files. This could be the `file` protocol or the `http` protocol. If the class files are stored on my Web server's pages under `classes/option3/FileClassifierImpl_Stub.class`. the codebase would be specified as

```
java.rmi.server.codebase=http://myWebHost/classes/
```

(where `myWebHost` is the name of the HTTP's server host).

The server also sets a security manager. This is a restrictive one, so it needs to be told to allow access. This can be done by setting the property `java.security.policy` to point to a security policy file such as `policy.all`.

Combining all these points leads to startups such as

```
java -Djava.rmi.server.codebase=http://myWebHost/classes/ \
     -Djava.security.policy=policy.all \
     option3.FileClassifierServer
```

3. Non-RMI Proxy for FileClassifier

Many client-server programs communicate by message passing, often using a TCP socket. The two sides need to have an agreed protocol, that is they must have a standard set of message formats, and know what messages to receive and what replies to send at any time. Jini can be used in this sort of case by providing a wrapper around the client and server, and making them available as a Jini service. The original client then becomes a proxy agent for the server, and is distributed to Jini clients for execution. The original server runs within the Jini server and performs the real work of the service, just as in the thin proxy model. What differs is the class structure and how the components communicate.

The proxy and the service do not need to belong to the same class, or even share common superclasses. Unlike the RMI case, the proxy is not derived from the service so does not have a shared class structure. Both are written independently, using their own appropriate class hierarchies. The proxy still has to implement the `FileClassifier` interface, since that is what the client is asking for and the proxy is delivering.

If RMI is not used, then any other distributed communication mechanism can be employed. Typically client-server systems will use something like reliable TCP ports, but this is not the only choice. It is the one used in this example, though. So the service listens on an agreed port, the client connects to this port, and they exchange messages.

The message format adopted for this problem is really simple:

1. The proxy sends a message giving the file extension that it wants classified. This can be sent as a newline terminated string
2. The service will either succeed or fail in the classification. If it fails, it sends a single line of the string `"null"` followed by a newline. If it succeeds, it sends two lines, the first being the content type, the second the subtype.

The proxy will then use this reply to either return `null` or a new `MIMEType` object.

3.1 FileClassifierProxy

The proxy object will be exported completely to a Jini client such as `TestFileClassifier`. When this client calls the `getMIMEType()` method, the proxy opens up a connection on an agreed TCP port to the service and exchanges messages on this port. It then returns a suitable result. The code is

```
package socket;

import common.FileClassifier;
import common.MIMEType;
import java.net.Socket;

import java.io.Serializable;
import java.io.IOException;
import java.rmi.Naming;

import java.io.*;

/**
 * FileClassifierProxy
 *
 *
 * Created: Thu Oct 25 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class FileClassifierProxy implements FileClassifier, Serializable {

    static public final int PORT = 2981;
    protected String host;

    public FileClassifierProxy(String host) {
        this.host = host;
    }

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        // open a connection to the service on port XXX
        int dotIndex = fileName.lastIndexOf('.');
        if (dotIndex == -1 || dotIndex + 1 == fileName.length()) {
            // can't find suitable index
            return null;
        }
        String fileExtension = fileName.substring(dotIndex + 1);

        // open a client socket connection
        Socket socket = null;
        try {
             socket = new Socket(host, PORT);
        } catch(Exception e) {
            // nothing yet
```

```
        }

        String type = null;
        String subType = null;

        /*
         * protocol:
         * Write: file extension
         * Read: "null" + '\n'
         *       type + '\n' + subtype + '\n'
         */
        try {
            InputStreamReader inputReader =
                new InputStreamReader(socket.getInputStream());
            BufferedReader reader = new BufferedReader(inputReader);
            OutputStreamWriter outputWriter =
                new OutputStreamWriter(socket.getOutputStream());
            BufferedWriter writer = new BufferedWriter(outputWriter);

            writer.write(fileExtension);
            writer.newLine();
            writer.flush();

            type = reader.readLine();
            if (type.equals("null")) {
                return null;
            }
            subType = reader.readLine();
        } catch(IOException e) {
            return null;
        }
        // and finally
        return new MIMEType(type, subType);
    }
} // FileClassifierProxy
```

3.2 FileServerImpl

On the server side will be running the service. This will run in its own thread (inheriting from `Thread`) and will listen for connections. When one is received, it will create a new `Connection` object also in its own thread to handle the message exchange. (This creation of another thread is probably overkill here where the entire message exchange is very short, but is better practice for more complex situations.)

```
/**
 * FileServerImpl.java
 *
 *
 * Created: Tue Oct 26 10:23:03 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */
```



```
package socket;

import java.net.*;
import java.io.*;

public class FileServerImpl extends Thread {

    protected ServerSocket listenSocket;

    public FileServerImpl() {
        try {
            listenSocket = new ServerSocket(FileClassifierProxy.PORT);
        } catch(IOException e) {
            e.printStackTrace();
        }
    }

    public void run() {
        try {
            while(true) {
                Socket clientSocket = listenSocket.accept();
                new Connection(clientSocket).start();
            }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
} // FileServerImpl

class Connection extends Thread {

    protected Socket client;

    public Connection(Socket clientSocket) {
        client = clientSocket;
    }

    public void run() {
        String contentType = null;
        String subType = null;

        try {
            InputStreamReader inputReader =
                new InputStreamReader(client.getInputStream());
            BufferedReader reader = new BufferedReader(inputReader);
            OutputStreamWriter outputWriter =
                new OutputStreamWriter(client.getOutputStream());
            BufferedWriter writer = new BufferedWriter(outputWriter);

            String fileExtension = reader.readLine();

            if (fileExtension.equals("gif")) {
                contentType = "image";
```

```
                subType = "gif";
            } else if (fileExtension.equals("txt")) {
                contentType = "text";
                subType = "plain";
            } // etc

            if (contentType == null) {
                writer.write("null");
            } else {
                writer.write(contentType);
                writer.newLine();
                writer.write(subType);
            }
            writer.newLine();
            writer.close();
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

3.3 Server

The Jini server must start a `FileServerImpl` to listen for later connections. Then it can register a proxy object `FileClassifierProxy` with each lookup service which will send it on to interested clients. The proxy object must know where the service is listening in order to attempt a connection to it, and this is given by first making a query for the local host and then passing the hostname to the proxy in its constructor.

```
package socket;

import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
// import com.sun.jini.lease.LeaseRenewalManager;
// import com.sun.jini.lease.LeaseListener;
// import com.sun.jini.lease.LeaseRenewalEvent;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lease.LeaseListener;
import net.jini.lease.LeaseRenewalEvent;
import java.rmi.RMISecurityManager;
import java.net.*;

/**
 * FileClassifierServer.java
 *
 *
 * Created: Oct 25 1999
 *
```



```
 * @author Jan Newmarch
 * @version 1.1
 *    uses Jini 1.1 LeaseRenewalManager
 */

public class FileClassifierServer implements DiscoveryListener, LeaseListener {

    protected FileClassifierProxy proxy;
    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

    public static void main(String argv[]) {
        new FileClassifierServer();
        try {
            Thread.sleep(1000000L);
        } catch(Exception e) {
        }
    }

    public FileClassifierServer() {
        try {
            new FileServerImpl().start();
        } catch(Exception e) {
            System.err.println("New impl: " + e.toString());
            System.exit(1);
        }

        // set RMI scurity manager
        System.setSecurityManager(new RMISecurityManager());

        // proxy primed with address
        String host = null;
        try {
            host = InetAddress.getLocalHost().getHostName();
        } catch(UnknownHostException e) {
            e.printStackTrace();
        }

        proxy = new FileClassifierProxy(host);
        // now continue as before
        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();
```

```
        for (int n = 0; n < registrars.length; n++) {
System.out.println("found registrars");
            ServiceRegistrar registrar = registrars[n];

            // export the proxy service
            ServiceItem item = new ServiceItem(null,
                                               proxy,
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.print("Register exception: ");
                e.printStackTrace();
                // System.exit(2);
                continue;
            }
            try {
                System.out.println("service registered at " +
                                   registrar.getLocator().getHost());
            } catch(Exception e) {
            }
            leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }

    public void notify(LeaseRenewalEvent evt) {
        System.out.println("Lease expired " + evt.toString());
    }

} // FileClassifierServer
```

3.4 What classes need to be where?

We have the classes

1. `common.MIMEType`
2. `common.FileClassifier`
3. `socket.FileClassifierProxy`
4. `socket.FileServerImpl`
5. `socket.FileClassifierServer`
6. `client.TestFileClassifier`

These could be running on upto four different machines

1. The server machine for `FileClassifierServer`
2. The HTTP server, which may be on a different machine
3. The machine for the lookup service
4. The machine running the client `TestFileClassifier`

What classes need to be known to which machines?

The server running `FileClassifierServer` needs to know the following classes and interfaces

1. The `common.FileClassifier` interface
2. The class `common.MIMEType`
3. The class `socket.FileClassifierServer`
4. The class `socket.FileClassifierProxy`
5. The class `socket.FileServerImpl`

The lookup service does not need to know any of these classes. It just deals with them in the form of a `java.rmi.MarshalledObject`

The client needs to know

1. The `common.FileClassifier` interface
2. The class `common.MIMEType`

In addition, the HTTP server needs to be able to load and store classes. It needs to be able to access

1. The `socket.FileClassifierProxy` interface
2. The `common.FileClassifier` interface
3. The class `common.MIMEType`

3.5 Running the RMI Proxy FileClassifier

Again we have a server and a client to run. Calling the client is unchanged from option2, with a security policy required.

```
java -Djava.security.policy=policy.all client.TestFileClassifier
```

The value of `java.rmi.server.codebase` must specify the protocol used by the HTTP server to find the class files. This could be the `file` protocol or the `http` protocol. If the class files are stored on my Web server's pages under `classes/socket/FileClassifierProxy.class`. the codebase would be specified as

```
java.rmi.server.codebase=http://myWebHost/classes/
```

(where `myWebHost` is the name of the HTTP's server host).

The server also sets a security manager. This is a restrictive one, so it needs to be told to allow access. This can be done by setting the property `java.security.policy` to point to a security policy file such as `policy.all`.

Combining all these points leads to startups such as

```
java -Djava.rmi.server.codebase=http://myWebHost/classes/ \
     -Djava.security.policy=policy.all \
     FileClassifierServer
```

4. RMI and non-RMI Proxies for FileClassifier

An alternative that is often used for client-server systems instead of message passing is remote procedure calls (RPC). This involves a client that does some local processing and makes some RPC calls to the server. We can also bring this into the Jini world by using a proxy that does some processing on the client side, and makes

use of an RMI proxy/stub when it needs to make calls back to the service.

Some file types are more common than others: gifs, doc files, html files, *etc* abound. But there are many more types, ranging from less common ones such as FrameMaker MIF files through to downright obscure ones such as PDP11 overlay files. An implementation of a file classifier might place the common types in a proxy object which makes them quickly available to clients, and the less common ones back on the server, accessible through a (slower) RMI call.

4.1 FileClassifierProxy

The proxy object will implement `FileClassifier` in order that clients can find it. The implementation will handle some file types locally, but others it will pass on to another object, that implements the `ExtendedFileClassifier` interface. The `ExtendedFileClassifier` has one method, `getExtraMIMEType()`. The proxy is told about this other object at constructor time. This class is

```
/**
 * FileClassifierProxy.java
 *
 *
 * Created: Tue Oct 26 21:36:36 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package extended;

import common.FileClassifier;
import common.ExtendedFileClassifier;
import common.MIMEType;
import java.rmi.RemoteException;

public class FileClassifierProxy implements FileClassifier {

    /**
     * The service object that knows lots more MIME types
     */
    protected ExtendedFileClassifier extension;

    public FileClassifierProxy(ExtendedFileClassifier ext) {
        this.extension = ext;
    }

    public MIMEType getMIMEType(String fileName)
        throws RemoteException {
            if (fileName.endsWith(".gif")) {
            return new MIMEType("image", "gif");
        } else if (fileName.endsWith(".jpeg")) {
            return new MIMEType("image", "jpeg");
        } else if (fileName.endsWith(".mpg")) {
            return new MIMEType("video", "mpeg");
        } else if (fileName.endsWith(".txt")) {
            return new MIMEType("text", "plain");
        } else if (fileName.endsWith(".html")) {
            return new MIMEType("text", "html");
        } else {
            // we don't know it, pass it on to the service
            return extension.getExtraMIMEType(fileName);
        }
    }
} // FileClassifierProxy
```

4.2 ExtendedFileClassifier

The `ExtendedFileClassifier` interface will be the top level interface for the service and an RMI proxy for the service. It will be publically available for all clients to use. An immediate sub-interface, `RemoteExtendedFileClassifier` will add the `Remote` interface

```
/**
 * ExtendedFileClassifier.java
 *
 *
 * Created: Tue Oct 26 21:11:49 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package common;

import java.io.Serializable;
import java.rmi.RemoteException;

public interface ExtendedFileClassifier extends Serializable {

    public MIMEType getExtraMIMEType(String fileName)
        throws RemoteException;

} // ExtendedFileClassifier
```

and

```
/**
 * RemoteExtendedFileClassifier.java
 *
 *
 * Created: Tue Oct 26 21:14:19 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package extended;
```

```
import java.rmi.Remote;

interface RemoteExtendedFileClassifier extends common.ExtendedFileClassifier, Remote {

} // RemoteExtendedFileClassifier
```

4.3 ExtendedFileClassifierImpl

The implementation of the `ExtendedFileClassifier` interface is done by an `ExtendedFileClassifierImpl` object. This will also need to extend `UnicastRemoteObject` in order that the RMI runtime can create an RMI proxy for it. Since this object may handle requests from many proxies, an alternative implementation of searching for MIME types that is more efficient for repeated searches is used

```
/**
 * ExtendedFileClassifierImpl.java
 *
 *
 * Created: Tue Oct 26 21:16:23 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package extended;

import java.rmi.server.UnicastRemoteObject;
import common.MIMEType;
import java.util.HashMap;
import java.util.Map;

public class ExtendedFileClassifierImpl extends UnicastRemoteObject
    implements RemoteExtendedFileClassifier {

    /**
     * Map of String extensions to MIME types
     */
    protected Map map = new HashMap();

    public ExtendedFileClassifierImpl() throws java.rmi.RemoteException {
        /* This object will handle all classification attempts
         * that fail in client-side classifiers. It will be around
         * a long time, and may be called frequently. So it is worth
         * optimising the implementation by using a hash map
         */
        map.put("rtf", new MIMEType("application", "rtf"));
        map.put("dvi", new MIMEType("application", "x-dvi"));
        map.put("png", new MIMEType("image", "png"));
        // etc
    }

    public MIMEType getExtraMIMEType(String fileName)
        throws java.rmi.RemoteException {
        MIMEType type;
```



```
        String fileExtension;
        int dotIndex = fileName.lastIndexOf('.');

        if (dotIndex == -1 || dotIndex + 1 == fileName.length()) {
            // can't find suitable suffix
            return null;
        }

        fileExtension= fileName.substring(dotIndex + 1);
        type = (MIMEType) map.get(fileExtension);
        return type;
    }
} // ExtendedFileClassifierImpl
```

4.4 FileClassifierServer

The final piece in this jigsaw is the server that creates the service (and implicitly the RMI proxy for the service) and also the proxy primed with knowledge of the service.

```
package extended;

import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
// import com.sun.jini.lease.LeaseRenewalManager;
// import com.sun.jini.lease.LeaseListener;
// import com.sun.jini.lease.LeaseRenewalEvent;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lease.LeaseListener;
import net.jini.lease.LeaseRenewalEvent;
import java.rmi.RMISecurityManager;

/**
 * FileClassifierServer.java
 *
 *
 * Created: Oct 25 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    uses Jini 1.1 LeaseRenewalManager
 */

public class FileClassifierServer implements DiscoveryListener, LeaseListener {

    protected FileClassifierProxy proxy;
    protected ExtendedFileClassifierImpl impl;
    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();
```

```
    public static void main(String argv[]) {
        new FileClassifierServer();
        try {
            Thread.sleep(1000000L);
        } catch(Exception e) {
        }
    }

    public FileClassifierServer() {
        try {
            impl = new ExtendedFileClassifierImpl();
        } catch(Exception e) {
            System.err.println("New impl: " + e.toString());
            System.exit(1);
        }

        // set RMI scurity manager
        System.setSecurityManager(new RMISecurityManager());

        // proxy primed with impl
        proxy = new FileClassifierProxy(impl);

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();

        for (int n = 0; n < registrars.length; n++) {
System.out.println("found registrars");
            ServiceRegistrar registrar = registrars[n];

            // export the proxy service
            ServiceItem item = new ServiceItem(null,
                                               proxy,
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.print("Register exception: ");
                e.printStackTrace();
                // System.exit(2);
                continue;
```



```
            }
            try {
                System.out.println("service registered at " +
                                   registrar.getLocator().getHost());
            } catch(Exception e) {
            }
            leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }

    public void notify(LeaseRenewalEvent evt) {
        System.out.println("Lease expired " + evt.toString());
    }

} // FileClassifierServer
```

4.5 What classes need to be where?

We have the classes

1. `common.MIMEType`
2. `common.FileClassifier`
3. `common.ExtendedFileClassifier`
4. `extended.FileClassifierProxy`
5. `extended.RemoteExtendedFileClassifier`
6. `extended.ExtendedFileServerImpl`
7. `extended.FileClassifierServer`
8. `client.TestFileClassifier`

The server running `FileClassifierServer` needs to know the following classes and interfaces

1. The `common.FileClassifier` interface
2. The class `common.MIMEType`
3. `common.ExtendedFileClassifier`
4. The class `extended.FileClassifierServer`
5. The class `extended.FileClassifierProxy`
6. `extended.RemoteExtendedFileClassifier`
7. The class `extended.ExtendedFileServerImpl`

The lookup service does not need to know any of these classes. It just deals with them in the form of a `java.rmi.MarshalledObject`

The client needs to know

1. The `common.FileClassifier` interface
2. The class `common.MIMEType`

In addition, the HTTP server needs to be able to load and store classes. It needs to be able to access

1. The `extended.FileClassifierProxy` interface
2. `extended.RemoteExtendedFileClassifier`
3. `extended.ExtendedFileServerImpl_Stub`
4. The `common.FileClassifier` interface
5. The class `common.MIMEType`

5. Using the Naming service

This section is probably not worth keeping any more, and hasn't been brought up to date for this version. I shall either discard it or rework for a later version.

The option 3 service and the last section exported an RMI proxy object directly. They were able to do this because the Jini protocol allows objects to be moved around to their destination, where the "lookup" to find the object has been done by the Jini discovery protocol. In more "traditional' uses of RMI, a client wishing to use an RMI service has to first locate it using RMI lookup mechanisms. This is possible with Jini also, but it is a little more complex and probably not so useful.

An RMI exportable object can advertise itself in various ways, and Jini registration is only one of these. RMI comes with a simple lookup service, the `Naming` service. This uses methods such as `Naming.bind()` and `Naming.rebind()` to register a name for the RMI object in an RMI registry, and `Naming.lookup()` to retrieve it from there.

An RMI registry is started by the command `rmiregistry`. This registry must be run on the same machine as a service trying to bind to it. So if there are many services running on many machines, then there will be many RMI registries running, one per machine. A server will create an RMI object and then register it with its local RMI naming service. It will *also* still need to export a service registration to the Jini lookup service. So this method will export *two* services: an RMI service to an RMI naming registry, and a Jini service to a Jini lookup service.

On the other side, a client wishing to locate an object using this method will still need to find the Jini service, and will follow the methods of option 3. When it gets its Jini proxy, it will then need to locate its RMI proxy to make calls on it. But this time, it has not carried the RMI proxy along, and this must be found from the RMI registry. In order to use this, it must know the internet address of the registry's machine. By the time it has reached the client it is too late to have this information, so the Jini proxy must be "primed" with the address of the registry and the name of the RMI service while it is still in the service and before it is exported to the Jini lookup service.

5.1 Server using Naming Service

The implementation of the service remains unaltered as the `option3.FileClassifierImpl`. The server must bind this implementation into the rmi naming registry using `bind()` or `rebind()`. It must also prime the proxy with the address of this rmi registry, and the name the implementation is bound to.

```
package rmi;

import java.rmi.Naming;
import java.net.InetAddress;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
```



```
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
// import com.sun.jini.lease.LeaseRenewalManager;
// import com.sun.jini.lease.LeaseListener;
// import com.sun.jini.lease.LeaseRenewalEvent;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lease.LeaseListener;
import net.jini.lease.LeaseRenewalEvent;
import java.rmi.RMISecurityManager;

/**
 * FileClassifierServerNaming.java
 *
 *
 * Created: Wed Mar 17 14:23:44 1999
 *
 * @author Jan Newmarch
 * @version 1.2
 *    added LeaseRenewalManager
 *    moved sleep() from constructor to main()
 *    uses Jini 1.1 LeaseRenewalManager
 */

public class FileClassifierServerNaming implements DiscoveryListener, LeaseListener {

    // This is just a name for the service
    // It can be anything, just needs to shared by both
    // ends of the Naming service
    static final String serviceName = "FileClassifier";

    protected option3.FileClassifierImpl impl;
    protected FileClassifierNamingProxy proxy;
    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

    public static void main(String argv[]) {
        new FileClassifierServerNaming();

        // no need to keep server alive, RMI will do that
    }

    public FileClassifierServerNaming() {
        try {
            impl = new option3.FileClassifierImpl();
        } catch(Exception e) {
            System.err.println("New impl: " + e.toString());
            System.exit(1);
        }

        // register this with RMI registry
        System.setSecurityManager(new RMISecurityManager());
        try {
            Naming.rebind("rmi://localhost/" + serviceName, impl);
        } catch(java.net.MalformedURLException e) {
            System.err.println("Binding: " + e.toString());
            System.exit(1);
        } catch(java.rmi.RemoteException e) {
```

```
            System.err.println("Binding: " + e.toString());
            System.exit(1);
        }

        System.out.println("bound");
        // find where we are running
        String address = null;
        try {
            address = InetAddress.getLocalHost().getHostName();
        } catch(java.net.UnknownHostException e) {
            System.err.println("Address: " + e.toString());
            System.exit(1);
        }

        String registeredName = "//" + address + "/" + serviceName;

        // make a proxy that knows the service address
        proxy = new FileClassifierNamingProxy(registeredName);

        // now continue as before
        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();

        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];

            // export the proxy service
            ServiceItem item = new ServiceItem(null,
                                               proxy,
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.print("Register exception: ");
                e.printStackTrace();
                // System.exit(2);
                continue;
            }
            try {
                System.out.println("service registered at " +
                                   registrar.getLocator().getHost());
            } catch(Exception e) {
            }
```



```
                leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
            }
        }

        public void discarded(DiscoveryEvent evt) {

        }

        public void notify(LeaseRenewalEvent evt) {
            System.out.println("Lease expired " + evt.toString());
        }

    } // FileClassifierServerNaming
```

5.2 Proxy using Naming Service

The proxy which uses the naming service can still continue to use the "option 3" implementation. The way it finds this is different, as it uses the naming service `lookup()` method.

```
package rmi;

import common.FileClassifier;
import common.MIMEType;

import option3.RemoteFileClassifier;

import java.io.Serializable;
import java.io.IOException;
import java.rmi.Naming;

/**
 * FileClassifierNamingProxy
 *
 *
 * Created: Thu Mar 18 14:32:32 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class FileClassifierNamingProxy implements FileClassifier, Serializable {

    protected String serviceLocation;
    transient RemoteFileClassifier server = null;

    public FileClassifierNamingProxy(String serviceLocation) {
        this.serviceLocation = serviceLocation;
    }

    private void readObject(java.io.ObjectInputStream stream)
        throws java.io.IOException, ClassNotFoundException {
        stream.defaultReadObject();
```

```
        try {
            Object obj = Naming.lookup(serviceLocation);
            server = (RemoteFileClassifier) obj;
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }
    }

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        return server.getMIMEType(fileName);
    }
} // FileClassifierNamingProxy
```

6. Using Other Services

In all the examples os far, a proxy has been created in a server and registered with a lookup service. Meanwhile, a service object has usually been left behind in the server to handle calls from the proxy. However, there is really no need for the service to exist on the server, and the proxy could make use of other services elsewhere.

In this section we shall give an example of this. Recently an Australian, Pat Farmer, attempted to set a world record for jogging the longest distance. While he was running around, I became involved in a project to broadcast his heart beart live to the Web: a heart monitor was attached to him, which talked on an RS232 link to a mobile he was carrying. This did a data transfer to a program running at `www.micromed.com.au` which forwarded the data to a machine at DSTC. This ran a Web server delivering an applet, and the applet talked back to a server on this DSTC machine which farmed out the data to each applet as it was received from the heart monitor.

Now that the experiment is over, the broadcast data is sitting as a file at http://www.micromed.com.au/patfarmer/v2/patfhr.ecg, and it can be viewed on the applet from http://www.micromed.com.au/patfarmer/v2/heart.html. We can make it into a Jini service as follows

1. Have a service which we can locate using the service type (“display a heart monitor trace”) and information about it such as whose heart trace it is showing
2. The service can connect to an http address encoded into the service by its constructor (or other means), read from this and display the contents read assuming it is heart cardiograph data
3. The information about whose trace it is can be given by a `Name` entry

The client shows as

The heart monitor service can be regarded in a number of ways:

1. It is a full-blown service uploaded to the client, that just happens to use an external data source supplied from an HTTP server
2. It is a “fat” proxy to the HTTP service, which acts as a client to this service by displaying the data

Many other non-RMI services can be built that act in this “fat proxy” style.

6.1 Heart interface

The `Heart` interface only has one method, and that is to `show()` the heart trace in some manner.

```
/**
 * Heart.java
 *
 *
 * Created: Thu Jul 15 16:11:56 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package heart;

public interface Heart extends java.io.Serializable {

    public void show();
} // Heart
```

6.2 Heart server

The `HeartServer` is similar to the “option 2” method, where it exports a complete service. This service, of type `HeartImpl` is primed with a URL of where the heart data is stored. This data will later be delivered by an HTTP server. This implementation is enough to locate the service. However, rather than just anyone’s heart data, a client may wish to search for a particular persons data. This can be done by adding a `Name` entry as additional information to the service.

```
package heart;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
// import com.sun.jini.lease.LeaseRenewalManager;
// import com.sun.jini.lease.LeaseListener;
// import com.sun.jini.lease.LeaseRenewalEvent;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lease.LeaseListener;
import net.jini.lease.LeaseRenewalEvent;
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.Name;

/**
 * HeartServer.java
 *
 *
 * Created: Wed Mar 17 14:23:44 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    uses Jini 1.1 LeaseRenewalManager
 */

public class HeartServer implements DiscoveryListener,
                                    LeaseListener {

    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

    public static void main(String argv[]) {
        new HeartServer();

        // keep server running forever to
        // - allow time for locator discovery and
        // - keep re-registering the lease
                try {
            Thread.currentThread().sleep(Lease.FOREVER);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public HeartServer() {

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();

        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];

            ServiceItem item = new ServiceItem(null,
                                    // new HeartImpl("file:/home/jan/projects/jini/doc/heart/TECG3.ecg"),
                                    new HeartImpl("http://www.micromed.com.au/patfarmer/v2/patfhr.ecg"),
                                    new Entry[] {new Name("Pat Farmer")});
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
```



```
                System.err.println("Register exception: " + e.toString());
            }
            System.out.println("service registered");

            // set lease renewal in place
            leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }

    public void notify(LeaseRenewalEvent evt) {
        System.out.println("Lease expired " + evt.toString());
    }

} // HeartServer
```

6.3 Heart client

The client searches for a service implementing the `Heart` interface, with the additional requirement that it be for a particular person. Once it has this, it just calls the method `show()` to display this in some manner.

```
package heart;

import heart.Heart;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;

import net.jini.core.entry.Entry;
import net.jini.lookup.entry.Name;

/**
 * HeartClient.java
 *
 *
 * Created: Wed Mar 17 14:29:15 1999
 *
 * @author Jan Newmarch
 * @version 1.3
 *    moved sleep() from constructor to main()
 *    moved to package client
 *    simplified Class.forName to Class.class
 */

public class HeartClient implements DiscoveryListener {

    public static void main(String argv[]) {
        new HeartClient();

        // stay around long enough to receive replies
        try {
```

```
            Thread.currentThread().sleep(1000000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public HeartClient() {
        System.setSecurityManager(new RMISecurityManager());

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);

    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();
        Class [] classes = new Class[] {Heart.class};
        Entry [] entries = new Entry[] {new Name("Pat Farmer")};
        Heart heart = null;
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       entries);

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Service found");
            ServiceRegistrar registrar = registrars[n];
            try {
                heart = (Heart) registrar.lookup(template);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(2);
            }
            if (heart == null) {
                System.out.println("Heart null");
                continue;
            }
            heart.show();
        }
    }

    public void discarded(DiscoveryEvent evt) {
        // empty
    }
} // HeartClient
```


6.4 Heart implementation

The `HeartImpl` class opens a connection to an HTTP server and requests delivery of a file. For heart data it needs to display this at a reasonable rate, so it reads, draws, sleeps, in a loop. It acts as a fat client to the HTTP server, displaying the data in a suitable format (in this case, it uses HTTP as a transport mechanism for data delivery). As a ''client-aware'' service it customises this delivery to the characteristics of the client platform, just occupying a ''reasonable'' amount of screen space and using local colors and fonts.

```
/**
 * HeartImpl.java
 *
 *
 * Created: Thu Jul 15 15:57:47 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package heart;

import java.io.*;
import java.net.*;
import java.awt.*;

public class HeartImpl implements Heart {

    protected String url;

    /*
     * If we want to run it standalone we can use this
     */
    public static void main(String argv[]) {

        HeartImpl impl =
            new HeartImpl("file:/home/jan/projects/jini/doc/heart/TECG3.ecg");
        impl.show();
    }

    public HeartImpl(String u) {
        url = u;
    }

    double[] points = null;
    Painter painter = null;

    String heartRate = "--";


    public void setHeartRate(int rate) {
        if (rate > 20 && rate <= 250) {
            heartRate = "Heart Rate: " + rate;
        } else {
            heartRate = "Heart Rate: --";
```

```
        }
        // ? ask for repaint?
    }

    public void quit(Exception e, String s) {
        System.err.println(s);
        e.printStackTrace();
        System.exit(1);
    }

    public void show() {
        int SAMPLE_SIZE = 300 / Toolkit.getDefaultToolkit().
                                                getScreenResolution();
        Dimension size = Toolkit.getDefaultToolkit().
                                getScreenSize();
        int width = (int) size.getWidth();
        // capture points in an array, for redrawing in app if needed
        points = new double[width * SAMPLE_SIZE];
        for (int n = 0; n < width; n++) {
            points[n] = -1;
        }


        URL dataUrl = null;
        InputStream in = null;

        try {
            dataUrl = new URL(url);
            in = dataUrl.openStream();
        } catch (Exception ex) {
            quit(ex, "connecting to ECG server");
            return;
        }

        Frame frame = new Frame("Heart monitor");
        frame.setSize((int) size.getWidth()/2, (int) size.getHeight()/2);
        try {
            painter = new Painter(this, frame, in);
            painter.start();
        } catch (Exception ex) {
            quit(ex, "fetching data from ECG server");
            return;
        }
        frame.setVisible(true);
    }

    /*
    public void paint(Graphics g) {

        if (points == null)
            return;
        int SAMPLE_SIZE = 300 / Toolkit.getDefaultToolkit().
                                                getScreenResolution();
        g.setColor(Color.red);
```



```
        int min = 127;
        int max = -128;

        for (int n = 0; n < points.length; n++) {
            int x = n + 1;
            int magnitude = points[n];
            if (magnitude == -1) {
                return;
            }

            if (x % SAMPLE_SIZE == 0) {
                // draw only on multiples of sample size
                // Data is in the range -128 .. 127. Need to
                // draw so -128 is at the bottom, 127 at the top
                int x0 = x / SAMPLE_SIZE;
                g.drawLine(x0, min, x0, max);
                min = 127;
                max = -128;
            } else {
                if (magnitude > max) max = magnitude;
                if (magnitude < min) min = magnitude;
            }
        }
    }
    */
} // HeartImpl

class Painter extends Thread {

    static final int DEFAULT_SLEEP_TIME = 25; // milliseconds
    static final int CLEAR_AHEAD = 15;
    static final int MAX = 255;
    static final int MIN = 0;
    final int READ_SIZE = 10;

    protected HeartImpl app;
    protected Frame frame;

    protected InputStream in;
    protected final int RESOLUTION = Toolkit.getDefaultToolkit().
                                                getScreenResolution();
    protected final int UNITS_PER_INCH = 125;
    protected final int SAMPLE_SIZE = 300 / RESOLUTION;
    protected int sleepTime = DEFAULT_SLEEP_TIME;

    public Painter(HeartImpl app, Frame frame, InputStream in) throws Exception {
        this.app = app;
        this.frame = frame;
        this.in = in;
    }

    public void run() {

        while (!frame.isVisible()) {
```

```
   try {
       Thread.sleep(1000);
   } catch(Exception e) {
       // ignore
   }
}

int height = frame.getSize().height;
int width = frame.getSize().width;
int x = 1; // start at 1 rather than 0 to avoid drawing initial line
           // from -128 .. 127
int magnitude;
int nread;
int max = MIN; // top bound of magnitude
int min = MAX;  // bottom bound of magnitude
int oldMax = MAX + 1;
byte[] data = new byte[READ_SIZE];
Graphics g = frame.getGraphics();
g.setColor(Color.red);
try {
    Font f = new Font("Serif", Font.BOLD, 20);
    g.setFont(f);
} catch (Exception ex) {
    // ....
}



try {
    boolean expectHR = false;   // true ==> next byte is heartrate

    while ((nread = in.read(data)) != -1) {
        for (int n = 0; n < nread; n++) {
            int thisByte = data[n] & 0xFF;
            if (expectHR) {
                expectHR = false;
                app.setHeartRate(thisByte);
                continue;
            } else if (thisByte == 255) {
                expectHR = true;
                continue;
            }

            // we are reading bytes, from -127..128
            // conver to unsigned
            magnitude = thisByte;

            // then convert to correct scale
            magnitude -= 128;
            // scale and convert to window coord from the top downwards
            int y = ((128 - magnitude) * RESOLUTION) / UNITS_PER_INCH;
            app.points[x] = y;

            // draw only on multiples of sample size
```



```
                if (x % SAMPLE_SIZE == 0) {
                    // delay to draw at a reasonable rate
                    Thread.sleep(sleepTime);

                    int x0 = x / SAMPLE_SIZE;
                    g.clearRect(x0, 0, CLEAR_AHEAD, height);
                    if (oldMax != MAX + 1) {
                        g.drawLine(x0-1, oldMax, x0, min);
                    }
                    g.drawLine(x0, min, x0, max);
                    oldMax = max;
                    min = 1000;
                    max = -1000;
                    if (app.heartRate != null) {
                        g.setColor(Color.black);
                        g.clearRect(0, 180, 200, 100);
                        g.drawString(app.heartRate, 0, 220);
                        g.setColor(Color.red);
                    }
                } else {
                    if (y > max) max = y;
                    if (y < min) min = y;
                }
                if (++x >= width * SAMPLE_SIZE) {
                    x = 0;
                }
            }
        }
    } catch(Exception ex) {
        if (! (ex instanceof SocketException)) {
            System.out.println("Applet quit -- got " + ex);
        }
    } finally {
        try {
            if (in != null) {
                in.close();
                in = null;
            }
        } catch (Exception ex) {
            // hide it
        }
    }
  }
}
```

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Discovery Management

Contents

1. Management Interfaces
2. LookupLocatorDiscovery
3. LookupDiscoveryManager
4. Conclusion

This chapter discusses some utility classes that make it easier to deal with lookup services. This chapter only applies to Jini 1.1.

1.

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:

1. A list of `LookupLocators` for unicast discovery
2. A list of groups for lookup locators using multicast discovery
3. Listeners whose methods are invoked when a service locator is found

The chapter ‘‘Discovering a Lookup Service’’ considered 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

1. `DiscoveryManagement` which looks after discovery events
2. `DiscoveryGroupManagement` which looks after groups and multicast search
3. `DiscoveryLocatorManagement` which looks after unicast discovery

Different classes may implement different combinations of these three interfaces. The class `LookupDiscovery` was changed in Jini 1.1 to use `DiscoveryGroupManagement` and `DiscoveryManagement`. This performs multicast search, informing its listeners when lookup services are discovered. The class `LookupLocatorDiscovery` is new in Jini 1.1 and is discussed later in this chapter. It performs a similar task for unicast discovery. It implements the two interfaces `DiscoveryLocatorManagement` and `DiscoveryManagement`. Another class discussed later is `LookupDiscoveryManager` handles both unicast and broadcast discovery, and so implements all three interfaces. With these three cases covered, it is unlikely that you will need to implement these interfaces yourself.

The `DiscoveryManagement` interface is package net.jini.discovery; public interface DiscoveryManagement { public void addDiscoveryListener(DiscoveryListener l); public void removeDiscoveryListener(DiscoveryListener l); public ServiceRegistrar[] getRegistrars(); public void discard(ServiceRegistrar proxy); public void terminate(); } The method `addDiscoveryListener()` is the most important of these, as it allows a listener object to be informed whenever a new lookup service is discovered.

The `DiscoveryGroupManagement` interface is

```
package net.jini.discovery;

public interface DiscoveryGroupManagement {

    public static final String[] ALL_GROUPS = null;
```



```
    public static final String[] NO_GROUPS = new String[0];

    public String[] getGroups();
    public void addGroups(String[] groups) throws IOException;
    public void setGroups(String[] groups) throws IOException;
    public void removeGroups(String[] groups);
}
```

The most important of these is `setGroups()`. If the groups have initially been set to `NO_GROUPS`, no multicast search is performed. If it is later changed by `setGroups()` then this initiates search. Similarly, `addGroups()` will also initiate search. (This is why they may throw remote exceptions.)

The third interface is `DiscoveryLocatorManagement`

```
package net.jini.discovery;

public interface DiscoveryLocatorManagement {
    public LookupLocator[] getLocators();
    public void addLocators(LookupLocator[] locators);
    public void setLocators(LookupLocator[] locators);
    public void removeLocators(LookupLocator[] locators);
}
```

An implementation will generally set the locators in its own constructor, so these methods will probably only be useful if you need to change the set of unicast addresses for the lookup services.

2. LookupLocatorDiscovery

In the chapter on ‘‘Discovering a Lookup Service’’, the section on finding a lookup service at a known address only looked at a single address. If lookup services at multiple addresses are required, then a naive solution would be to put the code from that chapter into a loop. The `LookupLocatorDiscovery` class provides a more satisfctory solution by providing the same event handling method as in the multicast case: that is, you supply a list of addresses, and when a lookup service is found at one of these address, a listener object is informed.

The class is specified by

```
package net.jini.discovery;

public class LookupLocatorDiscovery implements DiscoveryManagement,
                                                DiscoveryLocatorManagement {
    public LookupLocatorDiscovery(LookupLocator[] locators);
    public LookupLocator[] getDiscoveredLocators();
    public LookupLocator[] get UndiscoveredLocators();
}
```

Rewriting the unicast example from the chapter ‘‘Discovering a Lookup Service’’ using this utility class makes it look much like the example on multicast discovery from the same chapter. The similarities are that it now uses the same event model for lookup service discovery, the differences are that it uses a set of `LookupLocator` objects rather than a set of groups.

```
package discoverymgt;

import net.jini.discovery.LookupLocatorDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
```

```
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;
import java.net.MalformedURLException;

/**
 * UniicastRegister.java
 *
 *
 * Created: Sunday 19 Dec 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class UnicastRegister implements DiscoveryListener {

    static public void main(String argv[]) {
        new UnicastRegister();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(10000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public UnicastRegister() {
        LookupLocatorDiscovery discover = null;
        LookupLocator[] locators = null;
        try {
            locators = new LookupLocator[] {new LookupLocator("jini://localhost")};
        } catch(MalformedURLException e) {
            e.printStackTrace();
        }
        try {
            discover = new LookupLocatorDiscovery(locators);
        } catch(Exception e) {
            System.err.println(e.toString());
            e.printStackTrace();
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();

        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];

            // the code takes separate routes from here for client or service
```



```
            System.out.println("found a service locator");
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }
} // MulticastRegister
```

3. LookupDiscoveryManager

An application (client or service) that wants to use a set of lookup services at fixed, known addresses and also to use whatever lookup services it can find by multicast can use the utility class `LookupDiscoveryManager`. Most of the methods of this class come from its interfaces:

```
package net.jini.discovery;

public class LookupDiscoveryManager implements DiscoveryManagement,
                                                DiscoveryGroupManagement,
                                                DiscoveryLocatorManagement {
    public LookupDiscoveryManager(String[] groups,
                                  LookupLocator[] locators,
                                  DiscoveryListener listener)
                                      throws IOException;
}
```

This differs from `LookupDiscovery` and `LookupLocatorDiscovery` in that it insists on a `DiscoveryListener` in its constructor. Programs using this can follow the same event model as the last example:

```
package discoverymgt;

import net.jini.discovery.LookupDiscoveryManager;
import net.jini.discovery.DiscoveryGroupManagement;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;
import java.net.MalformedURLException;
import java.io.IOException;
import java.rmi.RemoteException;

/**
 * AllcastRegister.java
 *
 *
 * Created: Sunday 19 Dec 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class AllcastRegister implements DiscoveryListener {
```

```
    static public void main(String argv[]) {
        new AllcastRegister();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(10000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public AllcastRegister() {
        LookupDiscoveryManager discover = null;
        LookupLocator[] locators = null;
        try {
            locators = new LookupLocator[] {new LookupLocator("jini://localhost")};
        } catch(MalformedURLException e) {
            e.printStackTrace();
        }
        try {
            discover = new LookupDiscoveryManager(DiscoveryGroupManagement.ALL_GROUPS,
                                                  locators,
                                                  this);
        } catch(IOException e) {
            System.err.println(e.toString());
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();

        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];

            try {
                System.out.println("found a service locator at " +
                                   registrar.getLocator().getHost());
            } catch(RemoteException e) {
                e.printStackTrace();
            }
            // the code takes separate routes from here for client or service
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }
} // MulticastRegister
```


4. Conclusion

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Join Manager

Contents

1. Join Manager
2. Jini 1.0 JoinManager
    1. Getting information from JoinManager
3. Summary

Finding a lookup service involves a common series of steps. Subequent interaction with them also involves common steps. A join manager encapulates these into one convenience class.

1. 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 each class 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. The other parameters are for the serice and its entry attributes, a `DiscoveryManagement` object to set groups and unicast locators (typically this will be done using a `LookupDiscoveryManager()` and a lease renewal manager. The following example uses this to register a `FileClassifierImpl`. There is no need for a `DiscoveryListener`, since the join manager adds itself as a listener and handles the registration with the lookup service.

```
package joinmgr;

import option2.FileClassifierImpl;

import net.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceRegistrar;
import java.rmi.RemoteException;
```



```
import net.jini.lookup.ServiceIDListener;
import net.jini.lease.LeaseRenewalManager;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.DiscoveryListener;


/**
 * FileClassifierServer.java
 *
 *
 * Created: Wed Mar 17 14:23:44 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    converted to Jini 1.1
 */

public class FileClassifierServer
    implements ServiceIDListener {

    public static void main(String argv[]) {
        new FileClassifierServer();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(1000000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }

    }

    public FileClassifierServer() {

        JoinManager joinMgr = null;
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            joinMgr = new JoinManager(new FileClassifierImpl(), /* service */
                                      null /* attr sets */,
                                      this /* ServiceIDListener*/,
                                      mgr /* DiscoveryManagement */,
                                      new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void serviceIDNotify(ServiceID serviceID) {
        // called as a ServiceIDListener
```

```
        // Should save the id to permanent storage
        System.out.println("got service ID " + serviceID.toString());
    }

} // FileClassifierServer
```

There are a number of other methods in `JoinManager`, which allow one to modify the state of a service registration.

2. Jini 1.0 JoinManager

A version of `JoinManager` was present in Jini 1.0. At that time no formal specification was given, and it was liable to change. It has, to the version given above. This section describes the old version for those still using Jini 1.0.

There are a number of possible constructors for `JoinManager`. The simplest is

```
JoinManager(java.lang.Object obj,
            Entry[] attrSets,
            ServiceIDListener callback,
            LeaseRenewalManager leaseMgr)
```

This specifies the service to be managed and its entry attributes. The `callback` will have its `serviceIDNotify()` method called when a new locator is discovered. This argument can be `null` if the application has no interest in this. The `leaseMgr` can also be set to `null` and will then be created as needed.

Using this constructor will initiate a search for service locators belonging to the group "public", which is defined by a group value of the empty string `""`. There is no constant for this, and the locators from Sun do not appear to belong to this group, so most applications will need to follow this up immediately with a call to search for locators belonging to any group:

```
JoinManager joinMgr = new JoinManager(obj, null, null, null);
joinMgr.setGroups(LookupDiscovery.ALL_GROUPS);
```

The second constructor

```
JoinManager(java.lang.Object obj,
            Entry[] attrSets,
            java.lang.String[] groups,
            LookupLocator[] locators,
            ServiceIDListener callback,
            LeaseRenewalManager leaseMgr)
```

adds `groups` and `locators`. This allows multicast search for locators belonging to certain groups, and also unicast lookup for known locators. A multicast only search for any groups would use

```
JoinManager joinMgr = new JoinManager(obj, null,
                                      LookupDiscovery.ALL_GROUPS,
                                      null, null, null);
```

where both additional parameters are set to `null`. On the other hand, a unicast lookup for a single known site would be done by


```
LookupLocator[] locators = new LookupLocator[1];
locators[0] new LookupLocator("http://www.all_about_files.com");
JoinManager joinMgr = new JoinManager(obj, null,
                                      LookupDiscovery.NO_GROUPS,
                                      locators, null, null);
```

(This code ignored exception handling.)

For example, the option2 service can become

```
package joinmgr;

import option2.FileClassifierImpl;

import com.sun.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import com.sun.jini.lookup.ServiceIDListener;
import com.sun.jini.lease.LeaseRenewalManager;
import net.jini.discovery.LookupDiscovery;

/**
 * FileClassifierServer1_0.java
 *
 *
 * Created: Wed Mar 17 14:23:44 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    renamed since Jini 1.1 has new JoinManager
 */

public class FileClassifierServer1_0 implements ServiceIDListener {

    public static void main(String argv[]) {
        new FileClassifierServer1_0();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(1000000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }

    }

    public FileClassifierServer1_0() {

        JoinManager joinMgr = null;
        try {
            /* this is one way of doing it
            joinMgr = new JoinManager(new FileClassifierImpl(),
                                      null,
                                      this,
                                      new LeaseRenewalManager());
```

```
            joinMgr.setGroups(null);
            */
            /* here is another */
            joinMgr = new JoinManager(new FileClassifierImpl(),
                                      null,
                                      LookupDiscovery.ALL_GROUPS,
                                      null,
                                      this,
                                      new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void serviceIDNotify(ServiceID serviceID) {
        System.out.println("got service ID " + serviceID.toString());
    }

} // FileClassifierServer1_0
```

2.1 Getting information from JoinManager

The `JoinManager` would appear to be unhelpful in supplying information about the lookup locators it finds. It is available, but by a slightly circuitous route. A service can register a `ServiceIDListener` to the `JoinManager`. This will be invoked whenever a *new* locator is found, by its `serviceIDNotify()` method. A `ServiceID` is not particularly useful, so we just ignore it. However, within this method we do know that a new service locator has been found.

The *complete set* of service locators may be found from the `JoinManager` method `getJoinSet()`, whuch returns an array of `ServiceRegistrar`'s. We have met this class before: its method `getLocator()` will return a `LookupLocator`, which has information such as host in `getHost()`. These can be put together as

```
protected JoinManager joinmgr;
...

joinmgr = new JoinManager(service, null,
                                        this, new LeaseManager());
...

public  void serviceIDNotify(ServiceID serviceID) {
    ServiceRegistrar registrars = joinmgr.getJoinSet();
    for (int n = 0; n < registrar.length; n++) {
        LookupLocator locator = registrars[n].getLocator();
        String hostName = locator.getHost();
        ...
    }
}
```

If you want to find out which is the latest locator to be found, it looks like you will have to cache the previous set, and find which is new in the array returned. Each call to `getJoinSet()` will return a new array.

3. Summary

A `JoinManager` can be used by a service to simplify many of the aspects of locating, registering and renewing leases for a service.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Security

Contents

1. Quick Fix
2. Why AllPermission is Bad
3. Removing AllPermission
4. Jini with Protection
5. Service Requirements
6. Client Requirements
7. Transaction Manager and other Activatable Services

Security plays an important role in distributed systems. The Jini security model is based on the JDK 1.2 security system.

1. Quick Fix

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. Some of the examples may work fine without a security manager Others may require an appropriate security manager in place. 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.

A totally permissive policy file can contain

```
grant {
    permission java.security.AllPermission "", "";
};
```

This will allow all permissions, and should *never* be used outside of a test and development environment - and moreover, one that is insulated from other potentially untrusted machines. (Standalone is good here!) The big advantage of this is that it gets you going on the *rest* of Jini, without worrying about security issues while you are grappling with other problems!

2. Why AllPermission is Bad

Granting all permissions to everyone is a very trusting act, in the potentially hostile world of the Internet. Not everyone is ‘‘mister nice guy’’. The client is vulnerable to attack because it is downloading code that satisfies a request for a service, and then executing that code. The checks that it is a *genuine* service are really non-existent: it has to implement the requested interface, and maybe satisfy conditions on associated Entry objects. If it passes these, then it can do anything.

As an example of a deliberately hostile service, a client asking for a simple file classifier could end up getting this object:

```
package hostile;

import common.MIMEType;
import common.FileClassifier;

/**
 * HostileFileClassifier1.java
 *
 *
 * Created: Mon Jul 12 14:22:13 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class HostileFileClassifier1 implements FileClassifier {

    public MIMEType getMIMEType(String fileName) {
        if (java.io.File.pathSeparator.equals("/")) {
            // Unix - don't uncomment the next line!
            // Runtime.getRuntime().exec("/bin/rm -rf /");
        } else {
            // DOS - don't uncomment the next line!
            // Runtime.getRuntime().exec("format c: /u");
        }
        return null;
    }


    public HostileFileClassifier1() {
        // empty
    }

} // HostileFileClassifier1
```

This object would be exported from a hostile service to run completely in any client unfortunate enough to download it.

It is not necessary to actually call a method on the downloaded object - the mere act of downloading can do the damage, if the object overrides the deserialization method:

```
package hostile;

import common.MIMEType;
import common.FileClassifier;

/**
 * HostileFileClassifier2.java
 *
```

```
 *
 * Created: Mon Jul 12 14:22:13 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class HostileFileClassifier2 implements FileClassifier,
    java.io.Externalizable {

    public MIMEType getMIMEType(String fileName) {
        return null;
    }

    public void readExternal(java.io.ObjectInput in) {
        if (java.io.File.pathSeparator.equals("/")) {
            // Unix - don't uncomment the next line!
            // Runtime.getRuntime().exec("/bin/rm -rf /");
        } else {
            // DOS - don't uncomment the next line!
            // Runtime.getRuntime().exec("format c: /u");
        }
    }

    public void writeExternal(java.io.ObjectOutput out)
        throws java.io.IOException{
        out.writeObject(this);
    }

    public HostileFileClassifier2() {
        // empty
    }

} // HostileFileClassifier2
```

The two classes above assume that clients will make requests for the implementation of a particular interface, and this means that the attacker would need to know this interface. It would require some knowledge of the clients it is attacking (that they will ask for this interface). At the moment, there are no standard interfaces, so this may not be a feasible way of attacking many clients. As interfaces such as those for a printer become specified and widely used, attacks based on hostile implementations of services may become more common.

3. Removing AllPermission

Setting the security access to `AllPermission` is easy and removes all possible security issues that may hinder development of a Jini application. But it leaves your system open, so that you must start using a more rigorous security policy at some stage - hopefully before others have damaged your system. The problem with moving away from this policy is that permissions are *additive* rather than *subtractive*. That is, you can't take permissions away from `AllPermission`, but have to start with an empty permission set and add to that.

Not giving enough permission can result in at least three cases:

1. A security-related exception can be thrown. This is comparatively easy to deal with, because the exception will tell you what permission is being denied. You can then decide if you should be granting this permission or not
2. A security-related exception can be thrown *but caught by some library object and ignored*. This happens within the multicast lookup methods, which make multicast requests. If this permission is denied it will be retried several times before giving up. This leads to a cumulative time delay before anything else can happen. The application may be able to continue, and will just suffer this time delay
3. A security-related exception can be thrown *but caught by some library object and ignored*. The application may be unable to continue in any rational way after this, and may just appear to hang. This may happen if network access is requested but denied, and then a thread waits for messages which can never arrive. Or it may just get stuck in a loop...

The first two cases will occur if permissions are turned off for the service providers such as `option3.FileClassifierServer`. The third occurs for the client `client.TestFileClassifier`.

There is a system property `java.security.debug` that can be set to print information about various types of access to the security mechanisms. This can be used with a slack security policy to find out exactly what permissions are being granted. Then, with the screws tightened, you can see where permission is being denied. An appropriate value for this property is `access`, as in

```
java -Djava.security.debug=access ...
```

For example, running `client.TestFileClassifier` with few permissions granted may result in a trace including

```
...
access: access allowed (java.util.PropertyPermission socksProxyHost read)
access: access allowed (java.net.SocketPermission 127.0.0.1:1174 accept,resolve)
access: access denied (java.net.SocketPermission 130.102.176.249:1024 accept,resolve)
access: access denied (java.net.SocketPermission 130.102.176.249:1025 accept,resolve)
access: access denied (java.net.SocketPermission 130.102.176.249:1027 accept,resolve)
...
```

The denied access is an attempt to make a socket accept or resolve request on my laptop (IP address 130.102.176.249), probably for RMI-related sockets. Since the client just sits there indefinitely making this request on one random port after another, this permission needs to be opened up as the client otherwise appears to just hang.

4. Jini with Protection

The safest way for a Jini client or service to be part of a Jini federation is through abstinence: that is, refuse to take part. This doesn't get you very far in populating a federation. The JDK 1.2 security model allows a number of ways in which more permissive activity may take place:

1. Grant permission only for certain activities, such as socket access at various levels on particular ports, or access to certain files for reading, writing or execution

   ```
   grant {
       permission java.net.SocketPermission "224.0.1.85", "connect,accept";
       permission java.net.SocketPermission "*.edu.au:80", "connect";
   }
   ```

2. Grant access only to particular hosts, subdomains or domains

```
grant codebase "http://sunshade.dstc.edu.au/classes/" {
    permission java.security.AllPermission "", "";
}
```

3. Require digital signatures attached to code

```
grant signedBy "sysadmin" {
    permission java.security.AllPermission "", "";
}
```

For any particular security access, you will need to decide which of these is appropriate. This will depend on the overall security policy for your organisation - and if your organisation doesn't have such a policy that you can refer to then you certainly shouldn't be exposing your systems to the internet (or to anyone *within* the organisation, either)!

5. Service Requirements

In order to partake in a Jini federation, a service must become ''sufficiently'' visible. A service needs to find a service locator before it can advertise its services. This can be by unicast to particular locations or by multicast.

Unicast discovery does not need any particular permissions to be set. The discovery can be done without any policy file needed.

For the multicast case, the service must have `DiscoveryPermission` for each group that it is trying to join. For all groups, the wildcard "*" can be used. So to join all groups, the permission granted should be

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

To join, say, the groups `printers` and `toasters`, the permission would be (I'm not totally sure about this???)

```
permission net.jini.discovery.DiscoveryPermission,
           "printers, toasters";
```

Once this permission is given, the service will make a multicast broadcast on 224.0.1.84. Socket permission for these requests and announcements must be given by

```
permission java.net.SocketPermission "224.0.1.84", "connect,accept";
permission java.net.SocketPermission "224.0.1.85", "connect,accept";
```

The service may export a `UnicastRemoteObject`. This will require listening on a port for requests. The default constructor will assign a random port (above 1024?) for this. If desired, this port may be specified by other constructors. This will require further socket permissions, such as

```
permission java.net.SocketPermission "localhost:1024-", "connect,accept";
permission java.net.SocketPermission "*.dstc.edu.au:1024-", "connect,accept";
```

to accept connections on any port above 1024 from the `localhost` or any computer in the `dstc.edu.au` domain.

A number of parameters may be set by preferences, such as `net.jini.discovery.ttl`. It does no harm to allow the Jini system to look for these parameters, and this may be allowed by

```
permission java.util.PropertyPermission "net.jini.discovery.*", "read";
```


A fairly minimal policy file suitable for a service exporting an RMI object could then be

```
grant {
    permission net.jini.discovery.DiscoveryPermission "*";
    // multicast request address
    permission java.net.SocketPermission "224.0.1.85", "connect,accept";
    // multicast announcement address
    permission java.net.SocketPermission "224.0.1.84", "connect,accept";

    // RMI connections
    permission java.net.SocketPermission "*.canberra.edu.au:1024-", "connect,accept";
    permission java.net.SocketPermission "130.102.176.249:1024-", "connect,accept";
    permission java.net.SocketPermission "127.0.0.1:1024-", "connect,accept";

    // reading parameters
    // like net.jini.discovery.debug!
    permission java.util.PropertyPermission "net.jini.discovery.*", "read";
};
```

6. Client Requirements

The client is most at risk in the Jini environment. The service exports objects; the lookup locator stores objects, but does not "bring them to life" and execute any of their methods; but the client brings an external object into its address space and runs it using all of the permissions that it has as a process running in an operating system. So it will run under the permissions of a particular user, in a particular directory, with user accesses to the local file system and network. It could destroy files, make network connections to undesirable sites (or desirable, depending on your tastes!) and download images from them, start processes to send obnoxious mail to anyone in your address book, and generally make of a mess of your electronic identity!

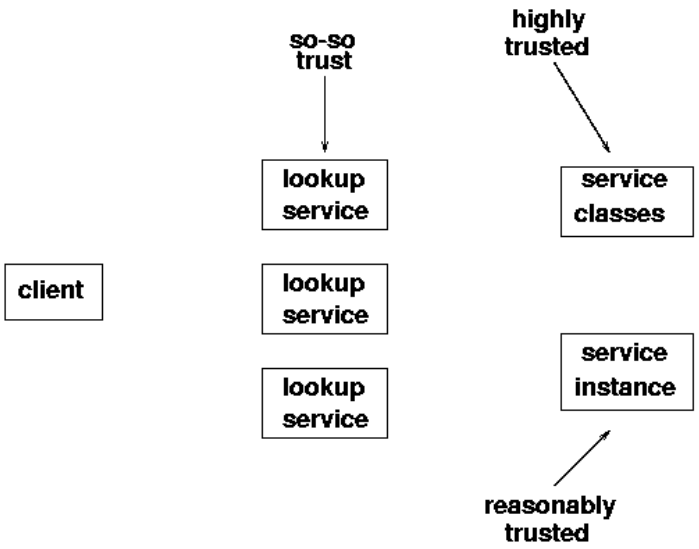
A client using multicast search to find service locators will need to grant discovery permission and multicast announcement permission, just like the service

```
permission net.jini.discovery.DiscoveryPermission "*";
permission java.net.SocketPermission "224.0.1.84", "connect,accept";
permission java.net.SocketPermission "224.0.1.85", "connect,accept";
```

RMI connections on random ports may also be needed

```
permission java.net.SocketPermission "*.dstc.edu.au:1024-", "connect,accept"
```

In addition to these, class definitions will probably need to be uploaded so that services can actually run in the client. This is the most serious risk area for the client, as the code contained in these class definitions will be run in the client, and any errors or malicious code will have their effect because of this. The client view of the different levels of trust can be given as



This is the most likely candidate to need signed trust certificates, perhaps. The permission that has to be granted is, for example

```
permission java.net.SocketPermission "127.0.0.1:80", "connect,accept";
permission java.net.SocketPermission "*.dstc.edu.au:80", "connect,accept";
```

A fairly minimal policy file suitable for a client could then be

```
grant {
    permission net.jini.discovery.DiscoveryPermission "*";

    // multicast request address
    permission java.net.SocketPermission "224.0.1.85", "connect,accept";
    // multicast announcement address
    permission java.net.SocketPermission "224.0.1.84", "connect,accept";

    // RMI connections
    permission java.net.SocketPermission "127.0.0.1:1024-", "connect,accept";
    permission java.net.SocketPermission "*.canberra.edu.au:1024-", "connect,accept";
    permission java.net.SocketPermission "130.102.176.249:1024-", "connect,accept";

    // DANGER
    // HTTP connections - this is where external code may come in - careful!!!
    permission java.net.SocketPermission "127.0.0.1:80", "connect,accept";
    permission java.net.SocketPermission "*.dstc.edu.au:80", "connect,accept";
```



```
    // reading parameters
    // like net.jini.discovery.debug!
    permission java.util.PropertyPermission "net.jini.discovery.*", "read";

};
```

7. Transaction Manager and other Activatable Services

The Jini distribution includes a transaction manager called "mahalo". This uses the new activation methods of RMI in JDK 1.2. Without worrying about any other arguments, the call to run this transaction manager is

```
java -jar mahalo.jar
```

(assuming the jar file is in the classpath). The transaction manager is a Jini service, and will need class definitions to be uploaded to clients. The class files are in `mahalo-dl.jar`, and will come from an HTTP server as specified in the first command-line argument:

```
java -jar mahalo.jar http://localhost/mahalo-dl.jar
```

(or appropriate HTTP address).

In order to register the service with a service locator and to set internal values for the transaction manager *running as a service*. This is done using the normal `java.security.policy` property

```
java -Djava.security.policy=policy.txn \
     -jar mahalo.jar http://localhost/mahalo-dl.jar
```

This will allow the service to be registered and uploaded. A suitable policy could be

```
grant {
    // rmid wants this
    permission java.net.SocketPermission "127.0.0.1:1098", "connect,resolve";

    // other RMI calls want these, too
    permission java.net.SocketPermission "127.0.0.1:1024-", "connect,resolve";
    permission java.net.SocketPermission "130.102.176.153:1024-", "connect,resolve";

    // access t transaction manager log files
    permission java.io.FilePermission "/tmp/mahalo_log", "read,write";
    permission java.io.FilePermission "/tmp/mahalo_log/-", "read,write,delete";

    // properties used by transaction manager
    permission java.util.PropertyPermission "com.sun.jini.mahalo.managerName", "read";
    permission java.util.PropertyPermission "com.sun.jini.use.registry", "read";
};
```

When the service is run, the service has to be *activated* in some JVM - the one running `rmid`. This in turn may need its own security policy, which is specified in the second command-line argument

```
java -Djava.security.policy=policy.txn -jar mahalo.jar \
      http://localhost/mahalo-dl.jar \
      policy.actvn
```

A suitable activation policy could be

```
grant {
    // rmid wants this
    permission java.net.SocketPermission "127.0.0.1:1098", "connect,resolve";

    // other RMI calls want these, too
    permission java.net.SocketPermission "127.0.0.1:1024-", "connect,resolve";
    permission java.net.SocketPermission "130.102.176.153:1024-", "connect,resolve";

    // access t transaction manager log files
    permission java.io.FilePermission "/tmp/mahalo_log", "read,write";
    permission java.io.FilePermission "/tmp/mahalo_log/-", "read,write,delete";

    // properties used by transaction manager
    permission java.util.PropertyPermission "com.sun.jini.mahalo.managerName", "read";
    permission java.util.PropertyPermission "com.sun.jini.use.registry", "read";

    // needed for activation
    permission java.net.SocketPermission "224.0.1.84", "connect,accept,resolve";
    permission java.io.FilePermission "/tmp/mahalo_log/-", "read";
    permission java.util.PropertyPermission "com.sun.jini.thread.debug", "read";
    permission java.lang.RuntimePermission "modifyThreadGroup";
    permission java.lang.RuntimePermission "modifyThread";

    // for dowloading mahalo-dl.jar from HTTP server
    permission java.net.SocketPermission "*:8080", "connect,accept,resolve";
    permission net.jini.discovery.DiscoveryPermission "*";
};
```

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

More Complex Examples

Contents

1. Where is the Code?
    1. Problem domain
    2. NameEntry interface
    3. Implementation one
    4. What files need to be where?
    5. Factories
    6. What files need to be where?
    7. Implementation three
    8. What files need to be where?
2. Running threads from discovery
    1. Server threads
    2. Join Manager threads
    3. Client threads
3. Inexact Service Matching
4. Matching using Local Services
5. Finding a Service Once Only
6. Leased Changes to a Service
    1. Leased FileClassifier
    2. FileClassifier Leased Resource
    3. FileClassifier Lease Manager
    4. FileClassifier Landlord

This chapter delves into some of the more complex things that can happen with Jini applications.

1. Where is the Code?

Clients, servers and service locators can use code from a variety of sources. Which source it uses can depend on the structure of a client and a server. This section looks at some of the variations that can occur.

1.1 Problem domain

A service may require information about a client before it can (or will) proceed. For example, a banking service may require a user id and a PIN number. Using the techniques already discussed, this could be done by the client collecting the information and calling suitable methods such as `void setName(String name)` in the service (or more likely, in the service's proxy) running in the client.

```
public class Client {

    String getName();

}

class Service {

    void setName(String name);

}
```

A service may wish to have more control over the setting of names and passwords than this. For example, it may wish to run verification routines based on the pattern of keystroke entries. More mundanely, it may wish to set time limits on the period between entering the name and the password. Or it may wish to enforce some particular user interface to collect this information. Whatever, the service proxy may wish to perform some sort of input processing on the client side before communicating with the real service. This section explores what happens when the service proxy needs to find extra classes in order to perform this processing.

A *standalone application* to get a user name might use a GUI interface such as

with implementation

```
package standalone;

import java.awt.*;
import java.awt.event.*;

/**
 * NameEntry.java
 *
 *
 * Created: Sun Mar 28 23:47:02 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class NameEntry extends Frame {

    public NameEntry() {
        super("Name Entry");
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {System.exit(0);}
            });

        Label label = new Label("Name");
        TextField name = new TextField(20);
        add(label, BorderLayout.WEST);
        add(name, BorderLayout.CENTER);
        name.addActionListener(new NameHandler());

        pack();
    }

    public static void main(String[] args) {

        NameEntry f = new NameEntry();
        f.setVisible(true);
    }
} // NameEntry
```



```
class NameHandler implements ActionListener {
    public void actionPerformed(ActionEvent evt) {
        System.out.println("Name was: " + evt.getActionCommand());
    }
}
```

The classes used in this are

1. A set of standard classes, `Frame, Label, TextField, ActionListener, ActionEvent, BorderLayout, WindowEvent, System`.
2. A couple of new classes `NameEntry, NameHandler`.

At compile time and at runtime these will need to be accessible.

1.2 NameEntry interface

In moving to a Jini system, we have already seen that different components may only need access to a subset of the total set of classes. The same will apply here, but it will critically depend on how the application is changed into a Jini system.

We don't want to be overly concerned about program logic of what is done with the user name once it has been entered, as the interesting part in this section is the location of classes. All versions will need an interface definition, which we can make simply as

```
package common;

/**
 * NameEntry.java
 *
 *
 * Created: Mon Mar 29 11:36:25 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public interface NameEntry  {

    public void show();

} // NameEntry
```

Then the client can call upon an implementation to simply `show()` itself and collect information in whatever way it chooses. (Note: we don't want to get involved here in the ongoing discussion about the most appropriate interface definition for GUI classes - this is taken up in a later chapter.)

1.3 Naive implementation

A simple implementation is

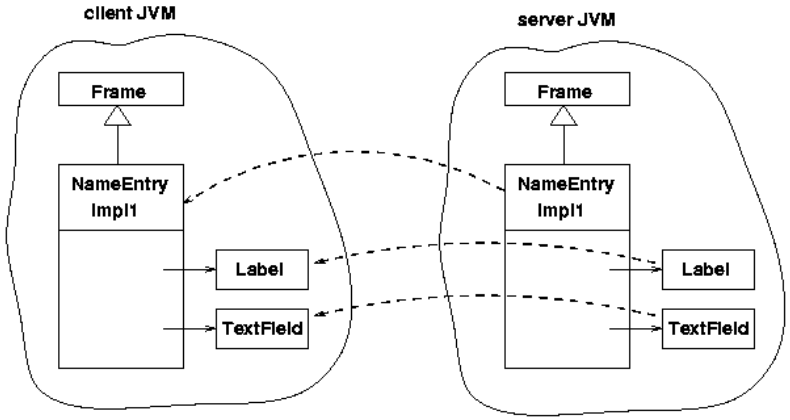
```
package complex;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import com.sun.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import com.sun.jini.lookup.ServiceIDListener;
import com.sun.jini.lease.LeaseRenewalManager;

/**
 * NameEntryImpl1.java
 *
 *
 * Created: Mon Mar 29 11:38:33 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class NameEntryImpl1 extends Frame implements common.NameEntry,
                                      ActionListener, java.io.Serializable {

    public NameEntryImpl1() {
        super("Name Entry");
        /*
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {System.exit(0);}
            public void windowOpened(WindowEvent e) {}});
        */
        setLayout(new BorderLayout());
        Label label = new Label("Name");
        add(label, BorderLayout.WEST);
        TextField name = new TextField(20);
        add(name, BorderLayout.CENTER);
        name.addActionListener(this);

        // don't do this here!
        // pack();
    }

    /**
     * method invoked on pressing <return> in the TextField
     */
    public void actionPerformed(ActionEvent evt) {
        System.out.println("Name was: " + evt.getActionCommand());
    }
```



```
    public void show() {
        pack();
        super.show();
    }


} // NameEntryImpl1
```

This creates the GUI elements in the constructor. When exported, this *entire* user interface will be serialized and exported. The instance data isn't too big in this case (about 2,100 bytes), but that is because the example is small. A GUI with several hundred objects will be much larger. This is overhead, which could be avoided by deferring creation to the client side.

A figure showing which instances are running in which JVM is



Another problem with this code is that it firstly creates an object on the server machine that has heavy reliance on environmental factors on the server. It then removes itself from that environment and has to re-establish itself on the target client environment. On my current system, this shows by `TextField` complaining that it cannot find a whole bunch of fonts on my server machine. Well it doesn't matter here: it gets moved to the client machine. (As it happens, the fonts aren't available there either, so I end with two batches of complaint messages, from the server and from the client. I should only get the client complaints.) It *could* matter if the service died because of missing stuff on the server side, which would exist on the client.

1.4 What files need to be where?

The client needs to know the interface class `NameEntry`

The server needs to know the class files for

1. NameEntry
2. Server1
3. NameEntryImpl1

The HTTP server needs to know the class files for

1. NameEntryImpl1

1.5 Factories

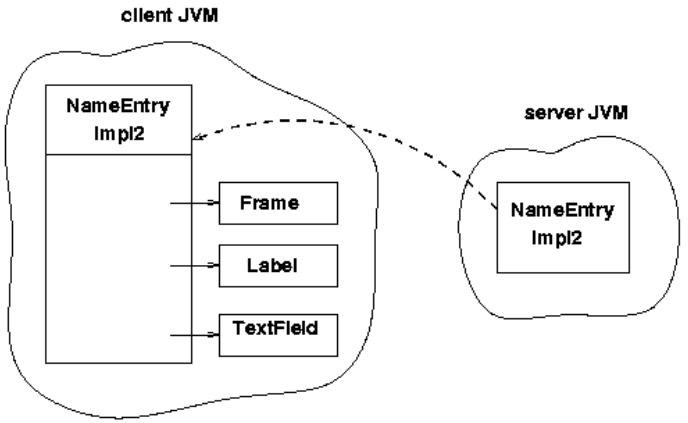
The second implementation minimises the amount of serialised code that must be shipped around, by creating as much as possible on the client side. We don't even need to declare the class as a subclass of Frame as that class also exists on the client side. The client calls the interface method show(), and all the GUI creation is moved to there. Essentially, what is created on the server side is a factory object, and this object is moved to the client. The client than makes calls on this factory to create the user interface.

```
package complex;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import com.sun.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import com.sun.jini.lookup.ServiceIDListener;
import com.sun.jini.lease.LeaseRenewalManager;

/**
 * NameEntryImpl2.java
 *
 *
 * Created: Mon Mar 29 11:38:33 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class NameEntryImpl2 implements common.NameEntry,
                                       ActionListener, java.io.Serializable {

    public NameEntryImpl2() {
    }

    /**
     * method invoked on pressing <return> in the TextField
     */
    public void actionPerformed(ActionEvent evt) {
        System.out.println("Name was: " + evt.getActionCommand());
    }

    public void show() {
        Frame fr = new Frame("Name Entry");
```



```
        fr.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {System.exit(0);}
            public void windowOpened(WindowEvent e) {}});

        fr.setLayout(new BorderLayout());
        Label label = new Label("Name");
        fr.add(label, BorderLayout.WEST);
        TextField name = new TextField(20);
        fr.add(name, BorderLayout.CENTER);
        name.addActionListener(this);

        fr.pack();
        fr.show();
    }


} // NameEntryImpl2
```

A figure showing which instances are running in which JVM is



There are some standard classes that cannot be serialised: one example is the Swing JTextArea class (as of Swing 1.1). This has been frequently logged as a bug against Swing. Until this is fixed, the *only* way one of these objects can be used by a service is to create it on the client.

1.6 What files need to be where?

The client needs to know the interface class NameEntry

The server needs to know the class files for

1. `NameEntry`
2. `Server2`
3. `NameEntryImpl2`
4. `NameEntryImpl2$1`

The last class in the list is an `anonymous class`, that acts as the `WindowListener`. The class file is produced by the compiler. In version 1, this part of the code was commented out for simplicity.

The HTTP server needs to know the class files for

1. `NameEntryImpl2`
2. `NameEntryImpl2$1`

1.7 Implementation three

Apart from the standard classes and a common interface, the previous implementations just used a single class that was uploaded to the lookup service and then passed on to the client. A more realistic situation might require the uploaded service to access a number of other classes that could not be expected to be on the client machine. It is simple to modify the examples to use a server-side specific class for the action listener, instead of the class itself. This looks like

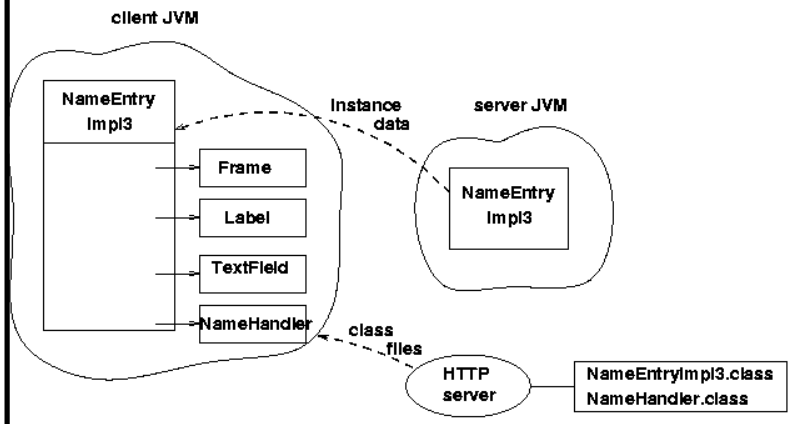
```
package complex;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import com.sun.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import com.sun.jini.lookup.ServiceIDListener;
import com.sun.jini.lease.LeaseRenewalManager;

/**
 * NameEntryImpl3.java
 *
 *
 * Created: Mon Mar 29 11:38:33 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class NameEntryImpl3 implements common.NameEntry,
                                       java.io.Serializable {

    public NameEntryImpl3() {
    }

    public void show() {
        Frame fr = new Frame("Name Entry");

        fr.addWindowListener(new WindowAdapter() {
```



```
            public void windowClosing(WindowEvent e) {System.exit(0);}
            public void windowOpened(WindowEvent e) {}});

        fr.setLayout(new BorderLayout());
        Label label = new Label("Name");
        fr.add(label, BorderLayout.WEST);
        TextField name = new TextField(20);
        fr.add(name, BorderLayout.CENTER);
        name.addActionListener(new NameHandler());

        fr.pack();
        fr.show();
    }


} // NameEntryImpl3

class NameHandler implements ActionListener {
    /**
     * method invoked on pressing <return> in the TextField
     */
    public void actionPerformed(ActionEvent evt) {
        System.out.println("Name was: " + evt.getActionCommand());
    }
} // NameHandler
```

This version uses a class `NameHandler` that only exists on the server machine. When the client attempts to deserialise the `NameEntryImpl3` instance it will fail to find this class, and be unable to complete deserialisation. How is this resolved? Well, in the same way as before, by making it available through the HTTP server.

A figure showing which instances are running in which JVM is



1.8 What files need to be where?

The client needs to know the interface class `NameEntry`

The server needs to know the class files for

1. `NameEntry`
2. `Server3`
3. `NameEntryImpl3`
4. `NameEntryImpl3$1`
5. `NameHandler`

The `NameHandler` class file is another one produced by the compiler.

The HTTP server needs to know the class files for

1. `NameEntryImpl3`
2. `NameEntryImpl3$1`
3. `NameHandler`

2. Running threads from discovery

In all of the simple examples using explicit registration, a single thread was used. That is, as a service locator was discovered, the registration process commenced in the same thread. Now this registration may take some time, so it would be preferable to do this in a separate thread. This is also recommended in the specification.

2.1 Server threads

Running another thread is not a difficult procedure. Basically we have to define a new class that extends `Thread` and move most of the registration into its `run` method. This is done in the following version of the "option 2" file classifier server, using an inner class

```
package complex;

import option2.FileClassifierImpl;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
// import com.sun.jini.lease.LeaseRenewalManager;
// import com.sun.jini.lease.LeaseListener;
// import com.sun.jini.lease.LeaseRenewalEvent;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lease.LeaseListener;
import net.jini.lease.LeaseRenewalEvent;

/**
 * FileClassifierServer.java
 *
 *
 * Created: Wed Mar 17 14:23:44 1999
 *
 * @author Jan Newmarch
 * @version 1.2
 *    added LeaseRenewalManager
 *    moved sleep() from constructor to main()
 *    uses Jini 1.1 LeaseRenewalManager
 */

public class FileClassifierServer implements DiscoveryListener,
                                              LeaseListener {

    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

    public static void main(String argv[]) {
        new FileClassifierServer();

        // keep server running forever to
        // - allow time for locator discovery and
        // - keep re-registering the lease
        try {
            Thread.currentThread().sleep(Lease.FOREVER);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
```

```
}

public FileClassifierServer() {

    LookupDiscovery discover = null;
    try {
        discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
    } catch(Exception e) {
        System.err.println(e.toString());
        System.exit(1);
    }

    discover.addDiscoveryListener(this);
}

public void discovered(DiscoveryEvent evt) {

    ServiceRegistrar[] registrars = evt.getRegistrars();

    for (int n = 0; n < registrars.length; n++) {
        ServiceRegistrar registrar = registrars[n];

        new RegisterThread(registrar).start();
    }
}

public void discarded(DiscoveryEvent evt) {

}

public void notify(LeaseRenewalEvent evt) {
    System.out.println("Lease expired " + evt.toString());
}

/**
 * an inner class to register the service in its own thread
 */
class RegisterThread extends Thread {

    ServiceRegistrar registrar;

    RegisterThread(ServiceRegistrar registrar) {
        this.registrar = registrar;
    }

    public void run() {
        ServiceItem item = new ServiceItem(null,
                                           new FileClassifierImpl(),
                                           null);
        ServiceRegistration reg = null;
        try {
            reg = registrar.register(item, Lease.FOREVER);
        } catch(java.rmi.RemoteException e) {
            System.err.println("Register exception: " + e.toString());
```



```
            }
            System.out.println("service registered");

            // set lease renewal in place
            leaseManager.renewUntil(reg.getLease(), Lease.FOREVER,
                                    FileClassifierServer.this);
        }
    }
} // FileClassifierServer
```

2.2 Join Manager threads

If you use a `JoinManager` to handle lookup and registration, then it essentially does this for you: creates a new thread to handle registration. Thus the examples in the chapter on "JoinManager" do not need any modification, as the `JoinManager` already uses the concepts of this section.

2.3 Client threads

It is probably more important to create threads in the client than in the server, since the client will actually perform some computation (which may be lengthy) based on the service it discovers. Again this is a simple matter, moving code into a new class that implements `Thread`. Doing this to the `TestFileClassifier` results in

```
package client;

import common.FileClassifier;
import common.MIMEType;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;


/**
 * TestFileClassifierThread.java
 *
 *
 * Created: Tue Apr 17 14:29:15 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    simplified Class.forName to Class.class
 */

public class TestFileClassifierThread implements DiscoveryListener {

    public static void main(String argv[]) {
        new TestFileClassifierThread();
```

```
        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(10000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestFileClassifierThread() {
        System.setSecurityManager(new RMISecurityManager());

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);

    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Service found");
            ServiceRegistrar registrar = registrars[n];

            new LookupThread(registrar).start();
        }

            // System.exit(0);
    }

    public void discarded(DiscoveryEvent evt) {
        // empty
    }

    class LookupThread extends Thread {

        ServiceRegistrar registrar;

        LookupThread(ServiceRegistrar registrar) {
            this.registrar = registrar;
        }


        public void run() {

            Class[] classes = new Class[] {FileClassifier.class};
            FileClassifier classifier = null;
```



```
            ServiceTemplate template = new ServiceTemplate(null, classes,
                                                           null);

            try {
                classifier = (FileClassifier) registrar.lookup(template);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(2);
            }
            if (classifier == null) {
                System.out.println("Classifier null");
                return;
            }
            MIMEType type;
            try {
                type = classifier.getMIMEType("file1.txt");
                System.out.println("Type is " + type.toString());
            } catch(java.rmi.RemoteException e) {
                System.err.println(e.toString());
            }
        }
    }

} // TestFileClassifier
```

3. Inexact Service Matching

Suppose you have a printer service that prints at 30 pages per minute. A client wishes to find a printer that will print *at least* 24 pages per minute. How will this client find the service? The standard Jini pattern matching will either be for an exact match on an attribute or an ignored match on an attribute. So the only way a client can find this printer is to *ignore* the speed attribute and perform a later selection among all the printers that it sees.

We can define a printer interface that will allow us to access printer speed (plus other capabilities) as

```
package common;

import java.io.Serializable;

/**
 * Printer.java
 *
 *
 * Created: Tue Apr 20 21:16:22 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public interface Printer extends Serializable {
```

```
    public void print(String str);
    public int getSpeed();

} // Printer
```

Given this, a client can choose a suitably fast printer in a two-step process

1. Find a service using the lookup exact/ignore match algorithm
2. Query the service to see if it satifies other types of boolean condition

The following program illustrates how to find a printer that is “fast enough”:

```
package client;

import common.Printer;

import java.rmi.RMISecurityManager;

import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceMatches;

/**
 * TestPrinterSpeed.java
 *
 *
 * Created: Mon Mar 29 12:03:51 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    simplified Class.forName to Class.class
 */

public class TestPrinterSpeed implements DiscoveryListener {

    public TestPrinterSpeed() {

        System.setSecurityManager(new RMISecurityManager());

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);

    }
```



```
public void discovered(DiscoveryEvent evt) {

    ServiceRegistrar[] registrars = evt.getRegistrars();
    Class[] classes = new Class[] {Printer.class};

    ServiceTemplate template = new ServiceTemplate(null, classes,
                                                   null);

    for (int n = 0; n < registrars.length; n++) {
        ServiceRegistrar registrar = registrars[n];
        ServiceMatches matches;

        try {
            matches = registrar.lookup(template, 10);
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
            continue;
        }
        // NB: matches.totalMatches may be greater than matches.items.length
        for (int m = 0; m < matches.items.length; m++) {
            Printer printer = (Printer) matches.items[m].service;

            // Inexact matching is not performed by lookup()
            // we have to do it ourselves on each printer
            // we get
            int speed = printer.getSpeed();
            if (speed >= 24) {
                // this one is okay, use its print() method
                printer.print("fast enough printer");
            } else {
                // we can't use this printer, so just say so
                System.out.println("Printer too slow at " + speed);
            }
        }

    }
}

public void discarded(DiscoveryEvent evt) {
    // empty
}

public static void main(String[] args) {

    TestPrinterSpeed f = new TestPrinterSpeed();

    // stay around long enough to receive replies
    try {
        Thread.currentThread().sleep(10000L);
    } catch(java.lang.InterruptedException e) {
        // do nothing
```

```
        }
    }

} // TestPrinterSpeed
```

4. Matching using Local Services

When a user connects their laptop into a brand-new network, they will probably know little about the environment they have joined. If they want to use services in this network, they will probably want to use general terms and have them translated into specific terms for this new environment. For example, the user may want to print a file on a *nearby* printer. In this situation, there is little likelihood that the new user knows how to work out the distance between themselves and the printers. However, a local service could be running which *does* know how to calculate physical distances between objects on the network.

Finding a ‘‘close enough’’ printer then becomes a matter of querying service locators both for printers *and* for a distance service. As each printer is found, the distance service can be asked to calculate the distance between itself and the laptop (or camera, or any other device that wants to print).

The complexity of the task to be done by clients is growing: a client has to find two sets of services, and when it finds one (a printer) invoke the other (the distance service). This calls for lookup processing to be handled in separate threads. In addition, as each locator is found, it may know about printers, it may know about distance services, both, or none! When the client starts up, it will be discovering these services in arbitrary order, and the code must be structured to deal with this.

Some of the cases that may arise are

1. A printer may be discovered before any distance service has been found. In this case, the printer must be stored for later distance checking.
2. A printer may be discovered after a distance service has been found. It can be checked immediately.
3. A distance service is found, after some printers have been found. This saved set should be checked at this point.

In this problem, we only need to find one distance service, but possibly many printers. The client code given shortly will save printers in a `Vector`, and a distance service in a single variable.

In searching for printers, we only want to find those that have location information. However, we do not want to match on any particular values. The client will have to use wildcard patterns in a location object. The location information of a printer will need to be retrieved along with the printer, so it can be used. So instead of just storing printers, we need to store `ServiceItem`’s, which carry the attribute information as well as the objects.

Of course, for this to work, the client also needs to know where it is!

A client satisfying these requirements is

```
package client;

import common.Printer;
import common.Distance;

import java.util.Vector;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
```



```
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.lookup.entry.Location;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.entry.Entry;

/**
 * TestPrinterDistance.java
 *
 *
 * Created: Tue Apr 17 14:29:15 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    simplified Class.forName to Class.class
 */

public class TestPrinterDistance implements DiscoveryListener {

    protected Distance distance = null;
    protected Object distanceLock = new Object();
    protected Vector printers = new Vector();

    public static void main(String argv[]) {
        new TestPrinterDistance();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(10000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestPrinterDistance() {
        System.setSecurityManager(new RMISecurityManager());

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);

    }

    public void discovered(DiscoveryEvent evt) {
```

```
        ServiceRegistrar[] registrars = evt.getRegistrars();

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Service found");
            ServiceRegistrar registrar = registrars[n];

            new LookupThread(registrar).start();
        }

            // System.exit(0);
    }

    public void discarded(DiscoveryEvent evt) {
        // empty
    }

    class LookupThread extends Thread {

        ServiceRegistrar registrar;

        LookupThread(ServiceRegistrar registrar) {
            this.registrar = registrar;
        }


        public void run() {

            synchronized(distanceLock) {
                // only look for one distance service
                if (distance == null) {
                    lookupDistance();
                }
                if (distance != null) {
                    // found a new distance service
                    // process any previously found printers
                    synchronized(printers) {
                        for (int n = 0; n < printers.size(); n++) {
                            ServiceItem item = (ServiceItem) printers.elementAt(n);
                            reportDistance(item);
                        }
                    }
                }
            }

            ServiceMatches matches = lookupPrinters();
            for (int n = 0; n < matches.items.length; n++) {
                if (matches.items[n] != null) {
                    synchronized(distanceLock) {
                        if (distance != null) {
                            reportDistance(matches.items[n]);
                        } else {
                            synchronized(printers) {
                                printers.addElement(matches.items[n]);
                            }
```



```
                    }
                }
            }
        }
    }

    /*
     * We must be protected by the lock on distanceLock here
     */
    void lookupDistance() {
        // If we don't have a distance service, see if this
        // locator knows of one
        Class[] classes = new Class[] {Distance.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        try {
            distance = (Distance) registrar.lookup(template);
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
        }

    }

    ServiceMatches lookupPrinters() {
        // look for printers with
        // wildcard matching on all fields of Location
        Entry[] entries = new Entry[] {new Location(null, null, null)};

        Class[] classes = new Class[1];
        try {
            classes[0] = Class.forName("common.Printer");
        } catch(ClassNotFoundException e) {
            System.err.println("Class not found");
            System.exit(1);
        }
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       entries);
        ServiceMatches matches = null;
        try {
            matches =  registrar.lookup(template, 10);
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
        }
        return matches;
    }

    /**
     * report on the distance of the printer from
     * this client
     */
    void reportDistance(ServiceItem item) {
        Location whereAmI = getMyLocation();
        Location whereIsPrinter = getPrinterLocation(item);
```

```
            if (whereIsPrinter != null) {
                int dist = distance.getDistance(whereAmI, whereIsPrinter);
                System.out.println("Found a printer at " + dist +
                                   " units of length away");
            }
        }

        Location getMyLocation() {
            return new Location("1", "1", "Building 1");
        }

        Location getPrinterLocation(ServiceItem item) {
            Entry[] entries = item.attributeSets;
            for (int n = 0; n < entries.length; n++) {
                if (entries[n] instanceof Location) {
                    return (Location) entries[n];
                }
            }
            return null;
        }
    }

} // TestFileClassifier
```

A number of services will need to be running. At least one distance service will be needed, implementing the interface `Distance`

```
package common;

import net.jini.lookup.entry.Location;

/**
 * Distance.java
 *
 *
 * Created: Wed Apr 21 13:12:35 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public interface Distance extends java.io.Serializable {

    int getDistance(Location loc1, Location loc2);

} // Distance
```

An example implementation is


```
package complex;

import net.jini.lookup.entry.Location;

/**
 * DistanceImpl.java
 *
 *
 * Created: Wed Apr 21 15:24:15 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class DistanceImpl implements common.Distance {

    public DistanceImpl() {

    }

    /**
     * A very naive distance metric
     */
    public int getDistance(Location loc1, Location loc2) {
        int room1, room2;
        try {
            room1 = Integer.parseInt(loc1.room);
            room2 = Integer.parseInt(loc2.room);
        } catch(Exception e) {
            return -1;
        }
        int value = room1 - room2;
        return (value > 0 ? value : -value);
    }

} // DistanceImpl
```

We have already covered some printers, we can just reuse them. A simple program to start up a distance service and two printers is

```
package complex;

import printer.Printer30;
import printer.Printer20;
import complex.DistanceImpl;

// import com.sun.jini.lookup.JoinManager;
import net.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
// import com.sun.jini.lookup.ServiceIDListener;
// import com.sun.jini.lease.LeaseRenewalManager;
import net.jini.lookup.ServiceIDListener;
import net.jini.lease.LeaseRenewalManager;
```

```
import net.jini.discovery.LookupDiscovery;
import net.jini.lookup.entry.Location;
import net.jini.core.entry.Entry;
import net.jini.discovery.LookupDiscoveryManager;

/**
 * PrinterServerLocation.java
 *
 *
 * Created: Wed Mar 17 14:23:44 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *   uses JoinManager 1.1
 */

public class PrinterServerLocation implements ServiceIDListener {

    public static void main(String argv[]) {
        new PrinterServerLocation();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(1000000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }

    }

    public PrinterServerLocation() {

        JoinManager joinMgr = null;
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            // distance service
            joinMgr = new JoinManager(new DistanceImpl(),
                                      null,
                                      this,
                                      mgr,
                                      new LeaseRenewalManager());


            // slow printer in room 20
            joinMgr = new JoinManager(new Printer20(),
                                      new Entry[] {new Location("1", "20",
                                                                "Building 1")},
                                      this,
                                      mgr,
                                      new LeaseRenewalManager());
```



```
            // fast printer in room 30
            joinMgr = new JoinManager(new Printer30(),
                                      new Entry[] {new Location("1", "30",
                                                                "Building 1")},
                                      this,
                                      mgr,
                                      new LeaseRenewalManager());


        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void serviceIDNotify(ServiceID serviceID) {
        System.out.println("got service ID " + serviceID.toString());
    }

} // PrinterServerLocation
```

5. Finding a Service Once Only

There may be many service locators on the local network, perhaps specialising in certain groups of services. There will certainly be many service locators running on the Internet, which could act as global repositories of information. Any service will probably register with as many service locators as it can, to improve its chances of being found. On the other hand, clients looking for a service may be content to find just a single suitable implementation, or may wish to find all service implementations. This second case can cause some uniqueness problems: if a client finds every service which has been registered with multiple locators, then it will probably find the same service more than once!

Why is this an issue? Well, suppose the client wants to find all power drills in the factory, and ask them to drill exactly one hole. Or suppose it finds all backup services for the system, and wants each one to perform a single backup. Then it needs to know the *identity* of each service, so that it can tell when it is getting a duplicate copy from another locator source. Otherwise, it might get each drill to make six holes as it gets a copy from six service locators, or might perform six backups of the same data. Whenever a service can perform a *non-idempotent* service (i.e. one in which repeating the action has a different effect each time), then duplicate copies on the client side must be avoided.

There is nothing explicit in Jini that will allow one to identify when a duplicate copy has been received from another locator. However, if all parties in a Jini federation follow certain procedures then it becomes possible. The procedure is actually contained in the Jini ‘‘Lookup Service’’ specification, and hinges on the use of the `ServiceID`.

A `ServiceID` can be specified when registering a service with a service locator. If this is the first time this service has ever been registered, then this should be `null`. The service locator will then *generate* a non-null `ServiceID` that can be used in future to identify this service. This object is specified to be unique, so that a service locator cannot generate the same one for two different services and two different locators cannot generate the same one. This provides a *unique identifier* that can be used to identify duplicates.

The procedure to follow is

1. The very, very first time a service is registered, use `null` as the `serviceID` value of the `ServiceItem` used in `ServiceRegistrar.register()`.
2. The returned `ServiceRegistration` has a method `getServiceID()` to give the `ServiceID`. This should then be used in any future registrations both with this service locator and *with any other one*. It should be noted that `JoinManager` already does this, although this is not stated in its documentation. We have done this too in examples.
3. The client has a choice of two `lookup()` methods to use with its `ServiceRegistrar` object: the first just returns a single object, the second returns an array of `ServiceMatches` objects. This second one is more useful here, as it can give the array of `ServiceItem` objects, and the `ServiceID` can be extracted from there.
4. The client should maintain a list of service id's that it has seen, and compare any new ones against it - then it can check if a service is a new one, or a previously seen one.

A modified client doing this is:

```
package unique;

import common.FileClassifier;
import common.MIMEType;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceID;

import java.util.Vector;

/**
 * TestFileClassifier.java
 *
 *
 * Created: Wed Mar 17 14:29:15 1999
 *
 * @author Jan Newmarch
 * @version 1.3
 *    moved sleep() from constructor to main()
 *    moved to package client
 *    simplified Class.forName to Class.class
 */

public class TestFileClassifier implements DiscoveryListener {

    protected Vector serviceIDs = new Vector();

    public static void main(String argv[]) {
        new TestFileClassifier();

        // stay around long enough to receive replies
```



```
        try {
            Thread.currentThread().sleep(10000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestFileClassifier() {
        System.setSecurityManager(new RMISecurityManager());

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);

    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();
        Class [] classes = new Class[] {FileClassifier.class};
        FileClassifier classifier = null;
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Service found");
            ServiceRegistrar registrar = registrars[n];
            ServiceMatches matches = null;
            try {
                 matches = registrar.lookup(template, 10);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(2);
            }

            ServiceItem[] items = matches.items;
            for (int m = 0; m < items.length; m++) {
                ServiceID id = items[m].serviceID;
                if (serviceIDs.indexOf(id) != -1) {
                    // found a new serviceID - record it and use it
                    classifier = (FileClassifier) items[m].service;
                    if (classifier == null) {
                        System.out.println("Classifier null");
                        continue;
                    }

                    serviceIDs.add(id);
```

```
                    MIMEType type;
                    try {
                        type = classifier.getMIMEType("file1.txt");
                        System.out.println("Type is " + type.toString());
                    } catch(java.rmi.RemoteException e) {
                        System.err.println(e.toString());
                    }
                }
            }
        }
    }

    public void discarded(DiscoveryEvent evt) {
        // empty
    }
} // TestFileClassifier
```

6. Leased Changes to a Service

Sometimes a service may allow changes to its state to be made by external (remote) objects. This happens all the time to service locators, which have services added and removed. A service may wish to behave in the same manner as the locators, and just grant a lease for the change. After the lease has expired, the service will remove the change. Such a situation may occur with file classification, where a new service starts that can handle a particular MIME type: it can register the filename mapping with a file classifier service. However, the file classifier service will just timeout the mapping unless the new service keeps it renewed.

The example of this section follows the lease management section of the chapter on ‘‘Leases’’. It gives a concrete illustration of that section now that there is enough background to do so.

6.1 Leased FileClassifier

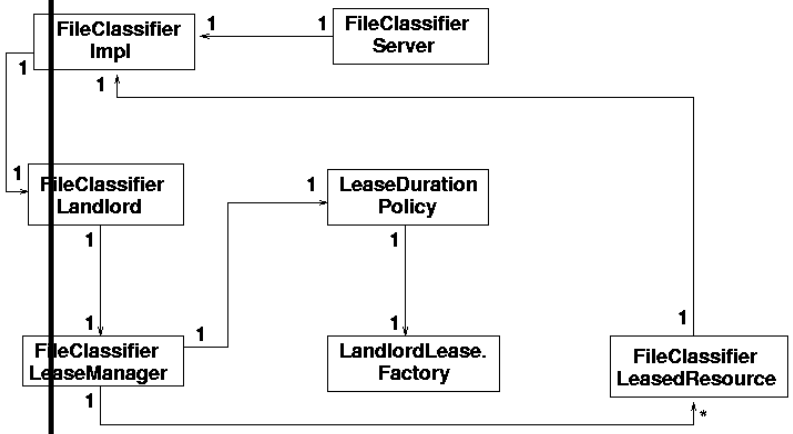
A dynamically extensible version of a file classification will have methods to add and remove MIME mappings:

```
package common;

import java.io.Serializable;

/**
 * LeaseFileClassifier.java
 *
 *
 * Created: Thu Jun 02 14:14:36 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

import net.jini.core.lease.Lease;

public interface LeaseFileClassifier extends Serializable {

    public MIMEType getMIMEType(String fileName)
```

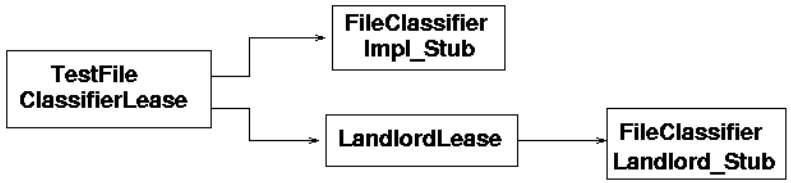


```
        throws java.rmi.RemoteException;

    /*
     * Add the MIME type for the given suffix.
     * The suffix does not contain '.' e.g. "gif".
     * @exception net.jini.core.lease.LeaseDeniedException
     * a previous MIME type for that suffix exists.
     * This type is removed on expiration or cancellation
     * of the lease.
     */
    public Lease addType(String suffix, MIMEType type)
        throws java.rmi.RemoteException,
               net.jini.core.lease.LeaseDeniedException;

    /**
     * Remove the MIME type for the suffix.
     * This shouldn't be public, but I haven't
     * figured out the best way of handling a
     * "rollback" method yet. Jini Transactions?
     */
    public void removeType(String suffix)
        throws java.rmi.RemoteException;
} // LeaseFileClasssifier
```

The `addType()` method returns a lease. We shall use the landlord leasing system discussed in an earlier chapter. The client and the service will be in different Java VM's, probably on different machines. On the service side, the object structure will be



On the client side, the lease object will be a copy of the lease created on the server (normally RMI semantics), but the other objects from the service will be stubs that call into the real objects on the service:



6.2 FileClassifier Leased Resource

The `FileClassifierLeasedResource` acts as a wrapper around the actual resource. It adds cookie and time expiration fields around the resource. It adds a unique cookie mechanism, in addition to making the wrapped resource visible.

```
/**
 * FileClassifierLeasedResource.java
 *
 *
 * Created: Mon Jun 14 18:04:33 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */
package lease;
```



```
import common.LeaseFileClassifier;
import com.sun.jini.lease.landlord.LeasedResource;

public class FileClassifierLeasedResource implements LeasedResource  {

    static protected int cookie = 0;
    protected int thisCookie;
    protected LeaseFileClassifier fileClassifier;
    protected long expiration = 0;
    protected String suffix = null;

    public FileClassifierLeasedResource(LeaseFileClassifier fileClassifier,
                                        String suffix) {
        this.fileClassifier = fileClassifier;
        this.suffix = suffix;
        thisCookie = cookie++;
    }

    public void setExpiration(long newExpiration) {
        this.expiration = newExpiration;
    }

    public long getExpiration() {
        return expiration;
    }
    public Object getCookie() {
        return new Integer(thisCookie);
    }

    public LeaseFileClassifier getFileClassifier() {
        return fileClassifier;
    }

    public String getSuffix() {
        return suffix;
    }
} // FileClassifierLeasedResource
```

6.3 FileClassifier Lease Manager

```
/**
 * FileClassifierLeaseManager.java
 *
 *
 * Created: Mon Jun 14 19:14:45 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */
package lease;

import java.util.*;
import common.LeaseFileClassifier;

import net.jini.core.lease.Lease;
import com.sun.jini.lease.landlord.LeaseManager;
```

```
import com.sun.jini.lease.landlord.LeasedResource;
import com.sun.jini.lease.landlord.LeaseDurationPolicy;
import com.sun.jini.lease.landlord.Landlord;
import com.sun.jini.lease.landlord.LandlordLease;
import com.sun.jini.lease.landlord.LeasePolicy;

public class FileClassifierLeaseManager implements LeaseManager {

    protected static long DEFAULT_TIME = 30*1000L;

    protected Vector fileClassifierResources = new Vector();
    protected LeaseDurationPolicy policy;

    public FileClassifierLeaseManager(Landlord landlord) {
        policy = new LeaseDurationPolicy(Lease.FOREVER,
                                         DEFAULT_TIME,
                                         landlord,
                                         this,
                                         new LandlordLease.Factory());
        new LeaseReaper().start();
    }

    public void register(LeasedResource r, long duration) {
        fileClassifierResources.add(r);
    }

    public void renewed(LeasedResource r, long duration, long olddur) {
        // no smarts in the scheduling, so do nothing
    }

    public void cancel(Object cookie) {
        for (int n = fileClassifierResources.size(); --n >= 0; ) {
            FileClassifierLeasedResource r = (FileClassifierLeasedResource) fileClassifierResources.elementAt(n);
            if (!policy.ensureCurrent(r)) {
                System.out.println("Lease expired for cookie = " +
                                   r.getCookie());
                try {
                    r.getFileClassifier().removeType(r.getSuffix());
                } catch(java.rmi.RemoteException e) {
                    e.printStackTrace();
                }
                fileClassifierResources.removeElementAt(n);
            }
        }
    }

    public LeasePolicy getPolicy() {
        return policy;
    }

    public LeasedResource getResource(Object cookie) {
        for (int n = fileClassifierResources.size(); --n >= 0; ) {
            FileClassifierLeasedResource r = (FileClassifierLeasedResource) fileClassifierResources.elementAt(n);
            if (r.getCookie().equals(cookie)) {
                return r;
            }
        }
        return null;
    }

    class LeaseReaper extends Thread {
        public void run() {
            while (true) {
                try {
                    Thread.sleep(DEFAULT_TIME) ;
                }
                catch (InterruptedException e) {
                }
                for (int n = fileClassifierResources.size()-1; n >= 0; n--) {
                    FileClassifierLeasedResource r = (FileClassifierLeasedResource)
                                        fileClassifierResources.elementAt(n)
;
                    if (!policy.ensureCurrent(r)) {
                        System.out.println("Lease expired for cookie = " +
                                           r.getCookie()) ;
                        try {
                            r.getFileClassifier().removeType(r.getSuffix());
                        } catch(java.rmi.RemoteException e) {
                            e.printStackTrace();
                        }
                        fileClassifierResources.removeElementAt(n);

                    }
                }
            }
        }
    }

} // FileClassifierLeaseManager
```

6.4 FileClassifier Landlord

```
/**
 * FileClassifierLandlord.java
 *
 *
 * Created: Mon Jun 14 22:48:11 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package lease;

import common.LeaseFileClassifier;

import com.sun.jini.lease.landlord.*;
import net.jini.core.lease.LeaseDeniedException;
import net.jini.core.lease.Lease;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.Remote;

public class FileClassifierLandlord extends UnicastRemoteObject implements Landlord, Remote {

    FileClassifierLeaseManager manager = null;

    public FileClassifierLandlord() throws java.rmi.RemoteException {
        manager = new FileClassifierLeaseManager(this);
    }

    public void cancel(Object cookie) {
        manager.cancel(cookie);
    }

    public void cancelAll(Object[] cookies) {
        // empty
    }

    public long renew(java.lang.Object cookie,
                      long extension)
        throws net.jini.core.lease.LeaseDeniedException,
               net.jini.core.lease.UnknownLeaseException {
        LeasedResource resource = manager.getResource(cookie);
        if (resource != null) {
            return manager.getPolicy().renew(resource, extension);
        }
```

```
        return -1;
    }

    public Lease newFileClassifierLease(LeaseFileClassifier fileClassifier,
                                        String suffixKey, long duration)
        throws LeaseDeniedException {
        FileClassifierLeasedResource r = new FileClassifierLeasedResource(fileClassifier,
                                                                           suffixKey);
        return manager.getPolicy().leaseFor(r, duration);
    }

    public Landlord.RenewResults renewAll(java.lang.Object[] cookie,
                                          long[] extension) {
        return null;
    }
} // FileClassifierLandlord
```

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Remote Events

Contents

1. Event Models
2. Remote Events
3. Event Registration
4. Listener List
    1. Single Listener
    2. Multiple Listeners
5. Listener Source
6. File Classifier with Events
7. Monitoring Changes in Services
8. Summary

This chapter looks at the distributed event model that is part of Jini. It looks at how remote event listeners are registered with objects, and how these objects notify their listeners of changes. It also looks at how leases are managed by event sources.

1. Event Models

Java has a number of event models, differing in various subtle ways. All of these involve an object generating an event in response to some change of state, either in the object itself (someone has changed a field, say), or in the external environment (a user has moved the mouse). At some earlier stage, a listener (or set of listeners) will have registered interest in this event and will have suitable methods called on them with the event as parameter. The event models all have their origin in the `Command` pattern from the Gang of Four book, but this is modified by other pressures, such as Java Beans.

There are low-level *input events*, which are generated by user actions in controlling an application with a graphical user interface. These events - of type `KeyEvent` and `MouseEvent` - are placed in an *event queue*. These are removed from the queue by a separate thread and dispatched to the relevant objects. In this case, the object that is responsible for generating the event is not responsible for dispatch to listeners, and creation and dispatch of events occurs in different threads.

Input events are a special case, caused by the need to listen to user interactions and always deal with them without loss of response time. Most events are dealt with in a simpler manner: an object maintains its own list of listeners, generates its own events, and dispatches it directly to its listeners. In this category fall all the *semantic events* generated by the AWT and Swing toolkits, such as `ActionEvent`, `ListSelectionEvent`, *etc*. There is a large range of these event types, and they all call different methods in the listeners, based on the event name. For example, an `ActionEvent` is used in a listener's `actionPerformed()` method of an `ActionListener`. There are naming conventions in this, caused by Java Beans.

Java Beans is also the influence behind `PropertyChange` events, which get delivered whenever a bean changes a "bound" or "constrained" property value. These are delivered to `PropertyChangeListener`'s `propertyChange()` method and to `VetoableChangeListener`'s `vetoableChange()` method. These are usually used to signal a change in a field of an object, where this change may be of interest to others either for information or for vetoing.

Jini objects may also be interested in changes in other Jini objects, and would like to be listeners for such changes. The networked nature of Jini has led to a particular event model which differs slightly from the other models already in Java. The differences are caused by factors such as

1. Network delivery is unreliable: messages may be lost. Synchronous methods requiring a reply may not work here
2. Network delivery is time-dependent: messages may arrive at different times to different listeners. So the state of an object as perceived by a listener at any time may be inconsistent with the state perceived by others. Passing complex object state across the network may be more complex to manage than passing simpler information
3. A remote listener may have disappeared by the time the event occurs. Listeners have to be allowed to "time out", like services do.
4. Java Beans can require method names and event types that vary. This requires availablity of classes across the network, which is more complex than a single method on a single event type (the original `Command` pattern used a single method of a single event type, for simplicity).

2. Remote Events

Unlike the large number of event classes in the AWT and Swing (for example), Jini typically uses events of one type, the `RemoteEvent` or a small number of subclasses. The class has public methods

```
package net.jini.core.event;

public class RemoteEvent {
    public long getID();
    public long getSequenceNumber();
    public java.rmi.MarshalledObject getRegistrationObject();
}
```

Events in Beans and AWT convey complex object state information. Jini events avoid this, and convey just enough information to allow state information to be found if needed. A remote event is serializable and can be moved around the network to its listeners.

AWT Events such as `MouseEvent` contain an `id` field that is set to values such as `MOUSE_PRESSED` or `MOUSE_RELEASED`. These are not seen by the AWT programmer because the AWT event dispatch system uses this field to choose appropriate methods such as `mousePressed()` or `mouseReleased()`. Jini does not make these assumptions about event dispatch, and just gives you the identifier. Either the source or the listener (or both) will know what this value means. For example, a file classifier that can update its knowledge of MIME types could have message types `ADD_TYPE` and `REMOVE_TYPE` to reflect the sort of changes it is going through.

In a synchronous system with no losses both sides of an interaction can keep consistent ideas of state. In a network system this is not so easy. The event source supplies a sequence number that could be used to generate state information if needed. This generalises things such as time-stamps on mouse events. For example, a message with id of `ADD_TYPE` and sequence number of `10` could correspond to the state change "added MIME type `text/xml` for files with suffix `.xml`". The event source should be able to supply this state information upon request.

An idea borrowed from systems such as the Xt Intrinsics and Motif is for registration of interest by a client in an event to include a piece of data from the client called a *handback*, which is returned with each event. This can be a reminder of *client* state at the time of registration. For example, a Jini taxi-driver might register interest in taxi bookings while passing through an area. As part of its registration it could include its current location. Then when it receives a booking event it is told about its old location, and it could check to see if it is still interested in events from that old location. A more novel possibility is that one object can register another object for events. So your stock-broker could register you for events about stock movements, and when you receive an event you also get a reminder about who registered your interest (plus a request for commission...).

3. Event Registration

Jini does not say how to register listeners with objects that can generate events. This is unlike other event models in Java that specify methods such as

```
public void addActionListener(ActionListener listener);
```

for `ActionEvent` generators. What Jini does do is to specify a convenience class as a *return* value from this registration. This convenience class is

```
package net.jini.core.event;
import net.jini.core.lease.Lease;

public class EventRegistration implements java.io.Serializable {
    public EventRegistration(long eventID, Object source,
                             Lease lease, long seqNum);
    public long getID();
    public Object getSource();
    public Lease getLease();
    public long getSequenceNumber();
}
```

This return object contains information that *may* be of value to the object that registered a listener. Each registration will typically only be for a limited amount of time, and this information may be returned in the `Lease` object. If the event registration was for a particular type, this may be returned in the id field. A sequence number may also be given. The meaning of these may depend on the particular system - in other words, Jini gives you a class that is optional in use, and whose fields are not tightly specified. This gives you the freedom to choose your own meanings to some extent.

This means that as the programmer of a event producer, you have to define (and implement) methods such as

```
public EventRegistration addRemoteEventListener(RemoteEventListener listener);
```

There is no standard interface for this.

4. Listener List

Each listener for remote events must implement the `RemoteEventListener` interface

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

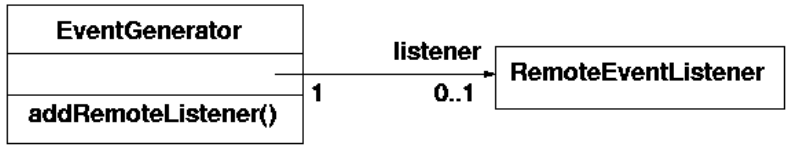
Because it extends `Remote` it means that the listener will most likely be something like an RMI stub for a remote object, so that calling `notify()` will result in a call on the remote object, with the event being passed across to it.

In event generators, there are multiple implementations for handling lists of event listeners all the way through the Java core and extensions. This is tedious, re-inventing the same thing. There are basically two cases

1. Only one listener can be in the list
2. Any number of listeners can be in the list

4.1 Single Listener

The case where there is only one listener allowed can be done by using a single-valued variable.



The simplest case of event registration is

```
protected RemoteEventListener listener = null;

public EventRegistration addRemoteListener(RemoteEventListener listener)
       throws java.util.TooManyListenersException {
    if (this.listener == null {
        this.listener = listener;
    } else {
        throw new java.util.TooManyListenersException();
    }
    return new EventRegistration(0L, this, null, 0L);
}
```

This is closest to the ordinary Java event registration: no really useful information is returned that wasn't known before. In particular, there is no lease object, so one could probably assume that the lease is being granted "forever", as would be the case with non-networked objects.

When an event occurs, the listener can be informed by the event generator calling `fireNotify()`:

```
protected void fireNotify(long eventID,
                          long seqNum) {
    if (listener == null) {
        return;
    }

    RemoteEvent remoteEvent = new RemoteEvent(this, eventID,
                                              seqNum, null);
    listener.notify(remoteEvent);
}
```

It is easy to add a "handback" to this: just add another field to the object, and set and return this in the registration and notify methods. Far more complex is the addition of a non-null lease. Firstly, the event source has to decided on a "lease policy", that is, for what periods of time is it going to grant leases. Then it has to implement a time-out mechanism to discard listeners when their leases expire. And finally, it has to handle lease renewal and cancellation requests, possibly using its lease policy again to make decisions. The landlord package would be of use here.

4.2 Multiple Listeners

For the case of any number of listeners, the convenience class `javax.swing.event.EventListenerList` can be used. The object delegates all list handling to the convenience class.

A version suitable for ordinary events is

```
import javax.swing.event.EventListenerList;

EventListenerList listenerList = new EventListenerList();

public EventRegistration addRemoteListener(RemoteEventListener l) {
    listenerList.add(RemoteListener.class, l);
    return new EventRegistration(0L, this, null, 0L);
}

public void removeRemoteListener(RemoteEventListener l) {
    listenerList.remove(RemoteListener.class, l);
}


// Notify all listeners that have registered interest for
// notification on this event type.  The event instance
// is lazily created using the parameters passed into
// the fire method.

protected void fireNotify(long eventID,
                          long seqNum) {
    RemoteEvent remoteEvent = null;

    // Guaranteed to return a non-null array
    Object[] listeners = listenerList.getListenerList();

    // Process the listeners last to first, notifying
    // those that are interested in this event
    for (int n = listeners.length - 2; n >= 0; n -= 2) {
        if (listeners[n] == RemoteEventListener.class) {
            RemoteEventListener listener =
                            (RemoteEventListener) listeners[n+1];
            if (remoteEvent == null) {
                remoteEvent = new RemoteEvent(this, eventID,
                                              seqNum, null);
            }
            try {
                listener.notify(remoteEvent);
```

```
            } catch(UnknownEventException e) {
                e.printStackTrace();
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Then a source object need only call `fireNotify()` to send the event to all listeners. (You may decide that it is easier to simply use a `Vector` of listeners!)

It is again straightforward to add handbacks to this. The only tricky point is that each listener can have its own handback, so they will need to be stored in some kind of map (say a `HashMap`) keyed on the listener. Then before `notify()` is called for each listener, the handback will need to be retrieved for the listener and a new remote event created with that handback.

5. Listener Source

Jini is a networked federation of objects. The ordinary Java event model has all objects in a single address space, so that registration of event listeners and notifying these listeners all takes place using objects in the one space. We have already seen that this is not the case with Jini, and in many cases one is dealing with proxy objects, not the "real" ones. This happens just the same with remote events, except that now we often have the *direction* of proxies reversed. To see what we mean by this, consider what happens if a client wants to monitor any changes in the service. The client will already have a proxy object for the service. It will use this proxy to register itself as a listener. But the service proxy will most likely just hand this listener back off to the service itself (that is what proxies such as RMI proxies do). So we need to get a proxy for the *client* over to the service.

Consider the file classification problems of earlier. There a file classifier had a "hard-coded' set of filename extensions built in. However, it may be possible to extend these, if applications come along that know how to define (and maybe handle) such extensions. In this case an application would locate the file classification server, and using an exported method from the file classification interface would add the new MIME type and file extension. This is no departure from any standard Java or earlier Jini stuff. It only affects the implementation level of the file classifier, changing it from a static list of filename extensions to a more dynamic one.

What it does affect is the poor application that has been blocked (probably sleeping) on an unknown filename extension. When the classifier installs a new type, it can send an event saying so. The blocked application can then try again to see if the extension is now known. If so, it uses it, if not it blocks again. Note that we don't bother with the actual state change, since it is just as easy to make another query knowing that the state *has* changed. More complex situations may require more information to be maintained. In order to get to this situation, the application must have registered its interest in events, and the event producer must be able to find it.

How this gets resolved is for the client to find the service in the same way as we have already discussed. It ends up with a proxy object in the client's address space. One of the methods on the proxy will be to add an event listener, which will be called by the client. For simplicity, assume that it is the client which is being added. The proxy will then call the "real" object's add listener method, back on its server side. But in doing this, we have made a remote call across the network, and the client, which was *local* to the call on the proxy, is now *remote* to the "real" object! So what the "real" object is getting is a proxy to the client! Then, when it makes notification calls, the client's proxy can make a remote call back to the client itself.

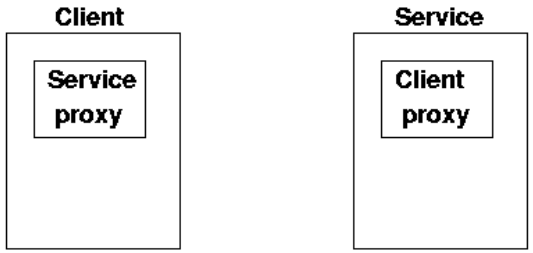


Figure 1: Proxies for services and listeners

6. File Classifier with Events

Let's make this more concrete by looking at a new file classifier that can have its set of mappings dynamically updated. The first to be modified is the interface `MutableFileClassifier`, known to all objects. This adds methods to add and remove types, and also to register listeners for events. The event types are labelled by two constants. The listener model is simple, and does not include handbacks or leases. We don't even bother to include the "timestamp" sequence identifier, since it is easier to just make MIME type queries again than it is to maintain lists of changes.

```
package common;

import java.io.Serializable;

/**
 * MutableFileClassifier.java
 *
 *
 * Created: Thu Jun 02 14:14:36 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.EventRegistration;

public interface MutableFileClassifier extends FileClassifier {

    static final public long ADD_TYPE = 1;
    static final public long REMOVE_TYPE = 2;

    /*
     * Add the MIME type for the given suffix.
     * The suffix does not contain '.' e.g. "gif".
     * Overrides any previous MIME type for that suffix
     */
```

```
    public void addType(String suffix, MIMEType type)
        throws java.rmi.RemoteException;

    /*
     * Delete the MIME type for the given suffix.
     * The suffix does not contain '.' e.g. "gif".
     * Does nothing if the suffix is not known
     */
    public void removeMIMEType(String suffix, MIMEType type)
        throws java.rmi.RemoteException;

    public EventRegistration addRemoteListener(RemoteEventListener listener)
        throws java.rmi.RemoteException;


} // MutableFileClasssifier
```

The `RemoteFileClassifier` just changes its package and inheritance for this

```
package mutable;

import common.MutableFileClassifier;
import java.rmi.Remote;

/**
 * RemoteFileClassifier.java
 *
 *
 * Created: Thu Jun 02
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public interface RemoteFileClassifier extends MutableFileClassifier, Remote {

} // RemoteFileClasssifier
```

The implementation changes from a static list of `if...then` statements to a dynamic map keyed on file suffixes. It manages the event listener list for multiple listeners in the simple way discussed earlier. It generates events whenever a new suffix/type is added or successfully removed.

```
package mutable;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.MarshalledObject;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.EventRegistration;
import java.rmi.RemoteException;
import net.jini.core.event.UnknownEventException ;
```



```
import javax.swing.event.EventListenerList;

import common.MIMEType;
import common.MutableFileClassifier;
import java.util.Map;
import java.util.HashMap;

/**
 * FileClassifierImpl.java
 *
 *
 * Created: Thu Jun 02
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class FileClassifierImpl extends  UnicastRemoteObject
                                 implements RemoteFileClassifier {

    /**
     * Map of String extensions to MIME types
     */
    protected Map map = new HashMap();

    /**
     * Listeners for change events
     */
    protected EventListenerList listenerList = new EventListenerList();

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        System.out.println("Called with " + fileName);

        MIMEType type;
        String fileExtension;
        int dotIndex = fileName.lastIndexOf('.');

        if (dotIndex == -1 || dotIndex + 1 == fileName.length()) {
            // can't find suitable suffix
            return null;
        }

        fileExtension= fileName.substring(dotIndex + 1);
        type = (MIMEType) map.get(fileExtension);
        return type;

    }

    public void addType(String suffix, MIMEType type)
        throws java.rmi.RemoteException {
        map.put(suffix, type);
        fireNotify(ADD_TYPE);
    }
```

```
    public void removeMIMEType(String suffix, MIMEType type)
        throws java.rmi.RemoteException {
        if (map.remove(suffix) != null) {
            fireNotify(REMOVE_TYPE);
        }
    }

    public EventRegistration addRemoteListener(RemoteEventListener listener)
        throws java.rmi.RemoteException {
        listenerList.add(RemoteEventListener.class, listener);

        return new EventRegistration(0, this, null, 0);
    }

    // Notify all listeners that have registered interest for
    // notification on this event type.  The event instance
    // is lazily created using the parameters passed into
    // the fire method.

    protected void fireNotify(long eventID) {
        RemoteEvent remoteEvent = null;

        // Guaranteed to return a non-null array
        Object[] listeners = listenerList.getListenerList();

        // Process the listeners last to first, notifying
        // those that are interested in this event
        for (int i = listeners.length - 2; i >= 0; i -= 2) {
            if (listeners[i] == RemoteEventListener.class) {
                RemoteEventListener listener = (RemoteEventListener) listeners[i+1];
                if (remoteEvent == null) {
                    remoteEvent = new RemoteEvent(this, eventID,
                                                  0L, null);
                }
                try {
                    listener.notify(remoteEvent);
                } catch(UnknownEventException e) {
                    e.printStackTrace();
                } catch(RemoteException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public FileClassifierImpl()  throws java.rmi.RemoteException {
        // load a predefined set of MIME type mappings
        map.put("gif", new MIMEType("image", "gif"));
        map.put("jpeg", new MIMEType("image", "jpeg"));
        map.put("mpg", new MIMEType("video", "mpeg"));
        map.put("txt", new MIMEType("text", "plain"));
        map.put("html", new MIMEType("text", "html"));
    }
} // FileClassifierImpl
```


The proxy changes its inheritance, and as a result has more methods to implement which it just delegates to its server object.

```
package mutable;

import common.MutableFileClassifier;
import common.MIMEType;

import java.io.Serializable;
import java.io.IOException;
import java.rmi.Naming;

import net.jini.core.event.EventRegistration;
import net.jini.core.event.RemoteEventListener;

/**
 * FileClassifierProxy
 *
 *
 * Created: Thu Jun 10 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class FileClassifierProxy implements MutableFileClassifier, Serializable {

    RemoteFileClassifier server = null;

    public FileClassifierProxy(FileClassifierImpl serv) {
        this.server = serv;
        if (serv==null) System.err.println("server is null");
    }

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        return server.getMIMEType(fileName);
    }

    public void addType(String suffix, MIMEType type)
        throws java.rmi.RemoteException {
        server.addType(suffix, type);
    }

    public void removeMIMEType(String suffix, MIMEType type)
        throws java.rmi.RemoteException {
        server.removeMIMEType(suffix, type);
    }

    public EventRegistration addRemoteListener(RemoteEventListener listener)
        throws java.rmi.RemoteException {
        return server.addRemoteListener(listener);
```

```
    }


} // FileClassifierProxy
```

7. Monitoring Changes in Services

Services will start and stop. When they start they will inform the lookup services, and sometime after they stop they will be removed from the lookup services. But there are a lot of times when other services or clients will want to know when services start or are removed. For example: the editor that wants to know if a disk service has started so that it can save its file; the graphics display program that wants to know when printer services start up; the user interface for a camera that wants to track changes in disk and printer services so that it can update the ‘‘Save’’ and ‘‘Print’’ buttons.

A service registrar acts as a generator of events of type `ServiceEvent` which subclass from `RemoteEvent`. These events are generated in response to changes of state of services which match (or fail to match) a template pattern for services. This event type has three categories from the `ServiceEvent.getTransition()` method:

1. `TRANSITION_NOMATCH_MATCH`: a service has changed state so that whereas it previously did not match the template, now it does. In particular, if it didn’t exist before now it does. This transition type can be used to spot new services starting
2. `TRANSITION_MATCH_NOMATCH`: a service has changed state so that whereas it previously *did* match the template, now it *doesn’t*. This can be used to detect when services are removed from a lookup service
3. `TRANSITION_MATCH_NOMATCH`: a service has changed state, but it matched both before and after. This typically happens when an `Entry` value changes, and is used to monitor changes of state such as a printer running out of paper, or a piece of hardware signalling that it is due for maintenance work

A client that wants to monitor changes of services on a lookup service must first create a template for the types of service it is interested in. A client that want to monitor all changes could prepare a template such as

```
ServiceTemplate templ = new ServiceTemplate(null, null, null); // or
ServiceTemplate templ = new ServiceTemplate(null, new Class[] {}, new Entry[] {}); // or
ServiceTemplate templ = new ServiceTemplate(null, new Class[] {Object.class}, null);
```

It then sets up a transition mask as a bit-wise OR of the three service transitions, and then calls `notify()` on the `ServiceRegistrar` object. A program to monitor all changes is

```
/**
 * RegistrarObserver.java
 *
 *
 * Created: Fri Oct 22 15:32:13 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    uses Jini 1.1 LeaseRenewalManager
 */

package observer;

import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.lookup.ServiceEvent;
```



```
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lease.Lease;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceID;
import net.jini.core.event.EventRegistration;
// import com.sun.jini.lease.LeaseRenewalManager; // Jini 1.0
import net.jini.lease.LeaseRenewalManager;        // Jini 1.1
import net.jini.core.lookup.ServiceMatches;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import net.jini.core.entry.Entry;
import net.jini.core.event.UnknownEventException;

public class RegistrarObserver extends UnicastRemoteObject implements RemoteEventListener {

    protected static LeaseRenewalManager leaseManager = new LeaseRenewalManager();
    protected ServiceRegistrar registrar;

    protected final int transitions = ServiceRegistrar.TRANSITION_MATCH_NOMATCH |
                                      ServiceRegistrar.TRANSITION_NOMATCH_MATCH |
                                      ServiceRegistrar.TRANSITION_MATCH_MATCH;

    public RegistrarObserver() throws RemoteException {
    }

    public RegistrarObserver(ServiceRegistrar registrar) throws RemoteException {
        this.registrar = registrar;
        ServiceTemplate templ = new ServiceTemplate(null, null, null);
        EventRegistration reg = null;
        try {
            // eventCatcher = new MyEventListener();
            reg = registrar.notify(templ,
                                   transitions,
                                   this,
                                   null,
                                   Lease.ANY);
            System.out.println("notifed id " + reg.getID());
        } catch(RemoteException e) {
            e.printStackTrace();
        }
        leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, null);
    }

    public void notify(RemoteEvent evt)
        throws RemoteException, UnknownEventException {
        try {
            ServiceEvent sevt = (ServiceEvent) evt;
            int transition = sevt.getTransition();
            System.out.println("transition " + transition);
            switch (transition) {
            case ServiceRegistrar.TRANSITION_NOMATCH_MATCH:
                System.out.println("nomatch -> match");
                break;
            case ServiceRegistrar.TRANSITION_MATCH_MATCH:
                System.out.println("match -> match");
                break;
            case ServiceRegistrar.TRANSITION_MATCH_NOMATCH:
                System.out.println("match -> nomatch");
                break;
            }
```

```
            System.out.println(sevt.toString());
            if (sevt.getServiceItem() == null) {
                System.out.println("now null");
            } else {
                Object service = sevt.getServiceItem().service;
                System.out.println("Service is " + service.toString());
            }
        } catch(Exception e) {
            e.printStackTrace();
        }

    }

} // RegistrarObserver
```

A suitable driver for this is

```
package client;

import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceMatches;

import java.util.Vector;
import observer.RegistrarObserver;

/**
 * ReggieMonitor.java
 *
 *
 * Created: Wed Mar 17 14:29:15 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class ReggieMonitor implements DiscoveryListener {

    protected Vector observers = new Vector();

    public static void main(String argv[]) {
        new ReggieMonitor();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(100000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public ReggieMonitor() {
        System.setSecurityManager(new RMISecurityManager());

        LookupDiscovery discover = null;
        try {
```



```
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);

    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Service found");
            ServiceRegistrar registrar = registrars[n];
            if (registrar == null) {
                System.out.println("registrar null");
                continue;
            }
System.out.println("here1");
            try {
                observers.add(new RegistrarObserver(registrar));
            } catch(RemoteException e) {
                System.out.println("adding observer failed");
            }
System.out.println("here3");
            ServiceTemplate templ = new ServiceTemplate(null, new Class[] {Object.class}, null);
            ServiceMatches matches = null;
            try {
                matches = registrar.lookup(templ, 10);
            } catch(RemoteException e) {
                System.out.println("lookup failed");
            }
System.out.println("here5");
            for (int m = 0; m < matches.items.length; m++) {
                if (matches.items[m] != null && matches.items[m].service != null) {
                    System.out.println("Reg konws about " + matches.items[m].service.toString() +
                                       " with id " + matches.items[m].serviceID);
                }
            }

        }
    }

    public void discarded(DiscoveryEvent evt) {
        // remove observer
    }
} // ReggieMonitor
```

8. Summary

This chapter has looked at how the remote event differs from the other event models in Java, and looked at how to create and use them.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior

permission is obtained from the copyright holder.

ClientLookupManager

Contents

1. Introduction
2. ServiceItemFilter
3. Finding a Service Immediately
4. Using a Filter
5. Building a Cache of Services
    1. Running the CachedClientLookup
6. Monitoring Changes to the Cache

The class `ClientLookupManager` is only available in Jini 1.1.

1. Introduction

Both clients and services need to find lookup services. Both can do this using low-level core classes, or discovery utilities such as `LookupDiscoveryManager`. Once a lookup service is found, a service just needs to register with it and try to keep the lease alive for as long as it wants to. A service can make use of the `JoinManager` class for this.

Clients are a little more complex in their requirements:

1. A client may wish to use a service immediately or later
2. A client may want to use multiple services
3. A client will want to find services by their interface, but may also want to apply additional criteria, such as being a “fast enough” printer
4. A client may just wish to use a service if it is available at the time of request, but alternatively may want to be informed of new services becoming available and respond to this new availability (for example, a service browser)

The class `ClientLookupManager` is a utility class designed to help in the various possible cases that can occur. Due to the variety of these, it is more complex than `JoinManager`. Its interface includes

```
package net.jini.lookup;

public class ClientLookupManager {
    public ClientLookupManager(DiscoveryManagement discoveryMgr,
                               LeaseRenewalManager leaseMgr)
           throws IOException;

    LookupCache createLookupCache(ServiceTemplate tmpl,
                                  ServiceItemFilter filter,
                                  ServiceDiscoveryListener listener);

    ServiceItem[] lookup(ServiceTemplate tmpl,
                         int maxMatches, ServiceItemFilter filter);

    ServiceItem lookup(ServiceTemplate tmpl,
                       ServiceItemFilter filter);

    ServiceItem lookup(ServiceTemplate tmpl,
```

```
                      ServiceItemFilter filter, long wait);

    ServiceItem[] lookup(ServiceTemplate tmpl,
                         int minMaxMatch, int maxMatches,
                         ServiceItemFilter filter, long wait);

    void terminate();
}
```

2. ServiceItemFilter

Most methods of the client lookup manager require a `ServiceItemFilter`. This is a simple interface, designed to be an additional filter on the client side to help in finding services. The primary mechanisms for a client to find a service are to ask for an instance of an interface, possibly with additional entry attributes. This matching is performed on the lookup service, and only implements a form of exact pattern matching. It allows one to ask for a toaster that will handle two slices of toast exactly, but not for one which will toast two *or more*. To perform arbitrary boolean matching on the lookup service is a security issue, plus possibly a performance issue on the lookup service. The `ServiceItemFilter` allows additional boolean filtering to be performed on the client side, by client code, so these issues are local to the client only.

The interface is

```
package net.jini.lookup;

public interface ServiceItemFilter {
    boolean check(ServiceItem item);
}
```

A client filter will implement this to perform additional checking.

Client-side filtering will not answer all of the problems of locating the ‘‘best’’ service. Some questions will still need to use other services which know ‘‘local’’ information such as distances in a building

3. Finding a Service Immediately

The simplest scenario for a client is that it wants to find a service immediately, use it and then (perhaps) terminate. It will be prepared to wait a certain amount of time before giving up. All issues of discovery can be given to the `ClientLookupManager`, and finding a service can be given to a method such as `lookup()` with a `wait` parameter. The `lookup()` method will block until a suitable service is found or the time limit is reached. If the time limit is reached, a `null` object will be returned otherwise a non-`null` service object will be returned.

```
package client;

import common.FileClassifier;
import common.MIMEType;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ClientLookupManager;
import net.jini.core.lookup.ServiceItem;
```



```
import net.jini.lease.LeaseRenewalManager;

/**
 * ImmediateClientLookup.java
 *
 *
 * Created: Mon Dec 20 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class ImmediateClientLookup {

    private static final long WAITFOR = 100000L;

    public static void main(String argv[]) {
        new ImmediateClientLookup();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(WAITFOR);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public ImmediateClientLookup() {
        ClientLookupManager clientMgr = null;

        System.setSecurityManager(new RMISecurityManager());

        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            clientMgr = new ClientLookupManager(mgr,
                                                new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        // Class [] classes = new Class[] {FileClassifier.class};
        Class [] classes = new Class[] {net.jini.lease.LeaseRenewalService.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        ServiceItem item = null;
        try {
            item = clientMgr.lookup(template,
                                    null, /* no filter */
                                    WAITFOR /* timeout */);
```

```
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        if (item == null) {
            // couldn't find a service in time
            System.out.println("no service");
            System.exit(1);
        }
        FileClassifier classifier = (FileClassifier) item.service;

        if (classifier == null) {
            System.out.println("Classifier null");
            System.exit(1);
        }

        // Now we have a suitable service, use it
        MIMEType type;
        try {
            String fileName;

            fileName = "file1.txt";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);

            fileName = "file2.rtf";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);

            fileName = "file3.abc";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);
        } catch(java.rmi.RemoteException e) {
            System.err.println(e.toString());
        }
        System.exit(0);
    }

    private void printType(String fileName, MIMEType type) {
        System.out.print("Type of " + fileName + " is ");
        if (type == null) {
            System.out.println("null");
        } else {
            System.out.println(type.toString());
        }
    }
} // ImmediateClientLookup
```


4. Using a Filter

An example of an earlier chapter discussed how to select a printer with a speed greater than a certain value. This type of problem is well suited to the `ClientLookupManager` using a `ServiceItemFilter`.

```
package client;

import common.Printer;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ClientLookupManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.ServiceItemFilter;
/**
 * TestPrinterSpeedFilter.java
 *
 *
 * Created: Mon Dec 20 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class TestPrinterSpeedFilter implements ServiceItemFilter {
    private static final long WAITFOR = 100000L;

    public TestPrinterSpeedFilter() {
        ClientLookupManager clientMgr = null;

        System.setSecurityManager(new RMISecurityManager());

        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            clientMgr = new ClientLookupManager(mgr,
                                                new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        Class[] classes = new Class[] {Printer.class};

        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
        ServiceItem item = null;
        try {
```

```
            item = clientMgr.lookup(template,
                                    this, /* filter */
                                    WAITFOR /* timeout */);
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        if (item == null) {
            // couldn't find a service in time
            System.exit(1);
        }

        Printer printer = (Printer) item.service;
        // Now use the printer
        // ...
    }

    public boolean check(ServiceItem item) {
        // This is the filter
        Printer printer = (Printer) item.service;
        if (printer.getSpeed() > 24) {
            return true;
        } else {
            return false;
        }
    }

    public static void main(String[] args) {

        TestPrinterSpeed f = new TestPrinterSpeed();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(WAITFOR);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

} // TestPrinterSpeed
```

5. Building a Cache of Services

A client may wish to make use of a service multiple times. If it just found a suitable reference to a service, then before each use it would have to check that the reference was still valid, and if not, find another one. It may also want ot use minor variants of a service, such as a fast printer one time, a slow one the next. While this management can be done easily enough for each case, the `ClientLookupManager` can supply a cache of services which will do it for you. This cache will look after monitoring lookup services to keep the cache as up-to-date as possible with services.

The cache is defined as an interface

```
package net.jini.lookup;

public interface LookupCache {
    public ServiceItem lookup(ServiceItemFilter filter);
    public ServiceItem[] lookup(ServiceItemFilter filter,
                                int maxMatches);
    public void addListener(ServiceDiscoveryListener l);
    public void removeListener(ServiceDiscoveryListener l);
    public void discard(Object serviceReference);
    void terminate();
}
```

and a suitable implementation object may be created by the `ClientLookupManager` method

```
    LookupCache createLookupCache(ServiceTemplate tmpl,
                                  ServiceItemFilter filter,
                                  ServiceDiscoveryListener listener);
```

We shall ignore the `ServiceDiscoveryListener` until the next section. It can be set to `null` in `createLookupCache()`.

The `LookupCache` created by `createLookupCache()` takes a template for matching against interface and entry attributes. In addition it also takes a filter to perform additional client-side boolean filtering of services. The cache will then maintain a set of local references to services matching the template and passing the filter. Searching for a service can then be done by local methods `LookupCache.lookup()`. These can take an additional filter which can be used to further refine the set of services returned to the client.

There are versions of the `ClientLookupManager.lookup()` with a time parameter, which block until a service is found or the method times out. These methods do not use polling, but instead use event notification. This is because they are trying to find services based on remote calls to lookup services. The `lookup()` methods of `LookupCache` do not implement such a blocking call, because the methods run purely locally and it is reasonable to poll the cache for a short time if need be.

A version of the file classifier client that creates and examines the cache for suitable service is

```
package client;

import common.FileClassifier;
import common.MIMEType;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ClientLookupManager;
import net.jini.lookup.LookupCache;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;

/**
 * CachedClientLookup.java
 *
```

```
 *
 * Created: Mon Dec 20 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class CachedClientLookup {

    private static final long WAITFOR = 100000L;

    public static void main(String argv[]) {
        new CachedClientLookup();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(WAITFOR);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public CachedClientLookup() {
        ClientLookupManager clientMgr = null;
        LookupCache cache = null;

        System.setSecurityManager(new RMISecurityManager());

        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            clientMgr = new ClientLookupManager(mgr,
                                                new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        Class [] classes = new Class[] {FileClassifier.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        try {
            cache = clientMgr.createLookupCache(template,
                                                null, /* no filter */
                                                null /* no listener */);
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        ServiceItem item = null;
```



```
        while (item == null) {
            System.out.println("no service yet");
            try {
                Thread.currentThread().sleep(1000);
            } catch(java.lang.InterruptedException e) {
                // do nothing
            }
            // see if a service is there now
            item = cache.lookup(null);
        }
        FileClassifier classifier = (FileClassifier) item.service;

        if (classifier == null) {
            System.out.println("Classifier null");
            System.exit(1);
        }

        // Now we have a suitable service, use it
        MIMEType type;
        try {
            String fileName;

            fileName = "file1.txt";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);

            fileName = "file2.rtf";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);

            fileName = "file3.abc";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);
        } catch(java.rmi.RemoteException e) {
            System.err.println(e.toString());
        }
        System.exit(0);
    }

    private void printType(String fileName, MIMEType type) {
        System.out.print("Type of " + fileName + " is ");
        if (type == null) {
            System.out.println("null");
        } else {
            System.out.println(type.toString());
        }
    }
} // CachedClientLookup
```

5.1 Running the CachedClientLookup

While it is okay to poll the local cache, the cache itself must get its contents from lookup services, and in general it is not okay to poll these since that involves possibly heavy network traffic. The cache itself gets its information by registering itself as a listener for service events from the lookup services. The lookup services will then call `notify()` on the cache listener. This call is a remote call from the remote lookup service to the local cache, done (probably) using an RMI stub. In fact, the Sun implementation of `ClientLookupManager` uses a nested class `ClientLookupManager.LookupCacheImpl.LookupListener` which has an RMI stub.

In order that the cache actually works, it is necessary to set the RMI codebase property `java.rmi.server.codebase` to a suitable location for the class files (such as an HTTP server), and to make sure that the class `net/jini/lookup/ClientLookupManager$LookupCacheImpl$LookupListener_Stub.class` is accessible from this codebase. The stub file may be found in the library `lib/jini-ext.jar` in the Jini 1.1 distribution. It has to be extracted from there and placed in the codebase (using e.g. `unzip jini-ext.jar 'net/jini/lookup/ClientLookupManager$LookupCacheImpl$LookupListener_Stub.class' -d /home/WWW/htdocs/classes`). Note that the specification just says this type of thing has to be done, but does not descend to details about the class name - that is left to the documentation of the `ClientLookupManager` as implemented by Sun. If another implementation is made of the Jini classes, then it would probably use a different remote class.

6. Monitoring Changes to the Cache

The cache uses remote events to monitor the state of lookup services. It includes a local mechanism to pass some of these changes onto a client by means of the `ServiceDiscoveryListener` interface

```
package net.jini.lookup;
interface ServiceDiscoveryListener {
    void serviceAdded(ServiceDiscoveryEvent event);
    void serviceChanged(ServiceDiscoveryEvent event);
    void serviceRemoved(ServiceDiscoveryEvent event);
}
```

with event

```
package net.jini.lookup;

class ServiceDiscoveryEvent extends EventObject {
    ServiceItem getPostEventServiceItem();
    ServiceItem getPreEventServiceItem();
}
```

Clients are not likely to be interested in all events generated by lookup services, even for services in which they are interested. For example, if a new service registers itself with ten lookup services, they will all generate transition events from `NO_MATCH` to `MATCH`, but the client will usually only be interested in seeing the first of these - the other nine are just repeated information. Similarly, if a service's lease expires from one lookup service, then that doesn't matter much; but if it expires from *all* lookup services that the client knows of, then it does matter as the service is no longer available to it. The cache consequently prunes events so that the client gets information about the real services rather than information about the lookup services.

IN the chapter on ''Events'' an example was given on monitoring changes to services from a lookup service viewpoint, reporting each change to lookup services. A client-oriented view is just to monitor changes in services themselves, which can be done easily using `ServiceDiscoveryEvent`'s:


```
package client;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.lookup.ServiceDiscoveryListener;
import net.jini.lookup.ServiceDiscoveryEvent;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceItem;
import net.jini.lookup.ClientLookupManager;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.LookupCache;

/**
 * ServiceMonitor.java
 *
 *
 * Created: Wed Mar 17 14:29:15 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class ServiceMonitor implements ServiceDiscoveryListener {

    public static void main(String argv[]) {
        new ServiceMonitor();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(100000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public ServiceMonitor() {
        ClientLookupManager clientMgr = null;
        LookupCache cache = null;

        System.setSecurityManager(new RMISecurityManager());

        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            clientMgr = new ClientLookupManager(mgr,
                                                new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
```

```
        ServiceTemplate template = new ServiceTemplate(null, null,
                                                       null);
        try {
            cache = clientMgr.createLookupCache(template,
                                                null, /* no filter */
                                                this /* listener */);
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    // methods for ServiceDiscoveryListener
    public void serviceAdded(ServiceDiscoveryEvent evt) {
        // evt.getPreEventServiceItem() == null
        ServiceItem postItem = evt.getPostEventServiceItem();
        System.out.println("Service appeared: " +
                           postItem.service.getClass().toString());
    }

    public void serviceChanged(ServiceDiscoveryEvent evt) {
        ServiceItem preItem = evt.getPostEventServiceItem();
        ServiceItem postItem = evt.getPreEventServiceItem() ;
        System.out.println("Service changed: " +
                           postItem.service.getClass().toString());
    }
    public void serviceRemoved(ServiceDiscoveryEvent evt) {
        // evt.getPostEventServiceItem() == null
        ServiceItem preItem = evt.getPreEventServiceItem();
        System.out.println("Service disappeared: " +
                           preItem.service.getClass().toString());
    }

} // ServiceMonitor
```

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Transactions

Contents

1. Introduction
2. Transactions
3. TransactionManager
4. TransactionParticipant
5. Mahalo
6. Example
    1. PayableFileClassifierImpl
    2. AccountsImpl
    3. Client
7. Summary

Transactions are needed to coordinate changes of state across multiple clients and services. The Jini transaction model uses a simple model of transactions, with details of semantics left to the clients and services. The Jini distribution supplies a transaction manager that can be used.

1. Introduction

Transactions are a neccessary part of many distributed operations. Frequently two or more objects may need to synchronize changes of state so that they all occur, or none occur. This happens in situations such as control of ownership, where one party has to give up ownership at the ‘‘same’’ time as another asserts ownership. What has to be avoided is only one party performing the action, which could result in either no owners or two owners.

The *theory* of transactions often talks about the ‘‘ACID’’ properties:

Atomicity
: All the operations of a transaction must take place, or none of them do

Consistency
: The completion of a transaction must leave the participants in a ‘‘consistent’’ state, whatever that means. For example, the number of owners of a resource must remain at one

Isolation
: The activities of one transaction must not affect any other transactions

Durability
: The results of a transaction must be persistent

The *practice* of transactions is that they use the *two-phase commit protocol*. This requires that participants in a transaction be asked to ‘‘vote’’ on a transaction. If all agree to go ahead, then the transaction ‘‘commits’’, which is binding on all the participants. If any ‘‘abort’’ during this voting stage then it forces abortion of the transaction on all participants.

Jini has adopted the syntax of the two-phase commit method. It is up to the clients and services within a transaction to observe the ACID properties if they desire. Jini essentially supplies the *mechanism* of two-phase commit, and leaves the *policy* of meaning to the participants in a transaction.

2. Transactions

Restricting Jini transactions to a two-phase commit model without associating a particular semantics to this means that a transaction can be represented in a simple way, just as a `long` identifier. This identifier is obtained from a transaction manager, and will uniquely label the transaction to that manager. (It is not guaranteed unique *between* managers, though - unlike service ID's.) All participants in the transaction communicate with the transaction manager, using this identifier to label which transaction they belong too.

The participants in a transaction may disappear; the transaction manager may disappear. So transactions are managed by a lease, which will expire unless it is renewed. When a transaction manager is asked for a new transaction it returns a `TransactionManager.Created` object, containing the transaction identifier and lease:

```
public interface TransactionManager {
    public static class Created {
        public final long id;
        public final Lease lease;
    }
    ...
}
```

A `Created` object may be passed around between participants in the lease. One of them will need to look after lease renewals. All the participants will use the transaction identifier in communication with the transaction manager.

3. TransactionManager

A transaction manager looks after the two-phase commit protocol for all the participants in a transaction. It is responsible for creating a new transaction by its `create()` method. Any of the participants may force the transaction to abort by `abort()`, or can force it to the two-phase commit stage by calling `commit()`.

```
public interface TransactionManager {

    Created create(long leaseFor) throws ...;
    void join(long id, TransactionParticipant part,
              long crashCount) throws ...;
    void commit(long id) throws ...;
    void abort(long id) throws ...;
    ...
}
```

When a participant joins a transaction, it registers a listener of type `TransactionParticipant`. If any participant calls `commit()`, the transaction manager starts the voting process using all of these listeners. If all of these are prepared to commit, then the manager moves all of these listeners to the commit stage. Alternatively, any of the participants may call `abort()`, which forces all of the listeners to abort.

4. TransactionParticipant

When an object becomes a participant listener in a transaction, it allows the transaction manager to call various methods


```
public interface TransactionParticipant ... {

    int prepare(TransactionManager mgr, long id) throws ...;
    void commit(TransactionManager mgr, long id) throws ...;
    void abort(TransactionManager mgr, long id) throws ...;
    int prepareAndCommit(TransactionManager mgr, long id) throws ...;
}
```

These are triggered by calls made upon the transaction manager. For example, if one client calls the transaction manager to abort, then it calls all the listeners to abort.

The "normal" mode of operation (that is, when nothing goes wrong with the transaction) is for a call to be made on the transaction manager to commit. It then enters the two-phase commit stage where it asks each participant listener to firstly `prepare()` and then to either `commit()` or `abort()`.

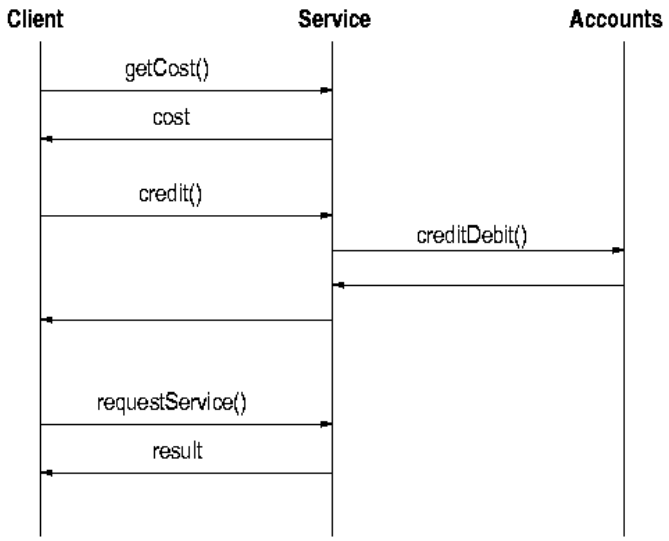
5. Mahalo

Mahalo is a transaction manager supplied by Sun as part of the Jini distribution. It can be used without change. Just like all Jini services, it has two parts: the part that runs as a server, needing its own set of class files in `mahalo.jar`, and the set of class files that need to be available to clients in `mahalo-dl.jar`. It also needs a security policy, and HTTP server and log files. It can be started using a command line such as

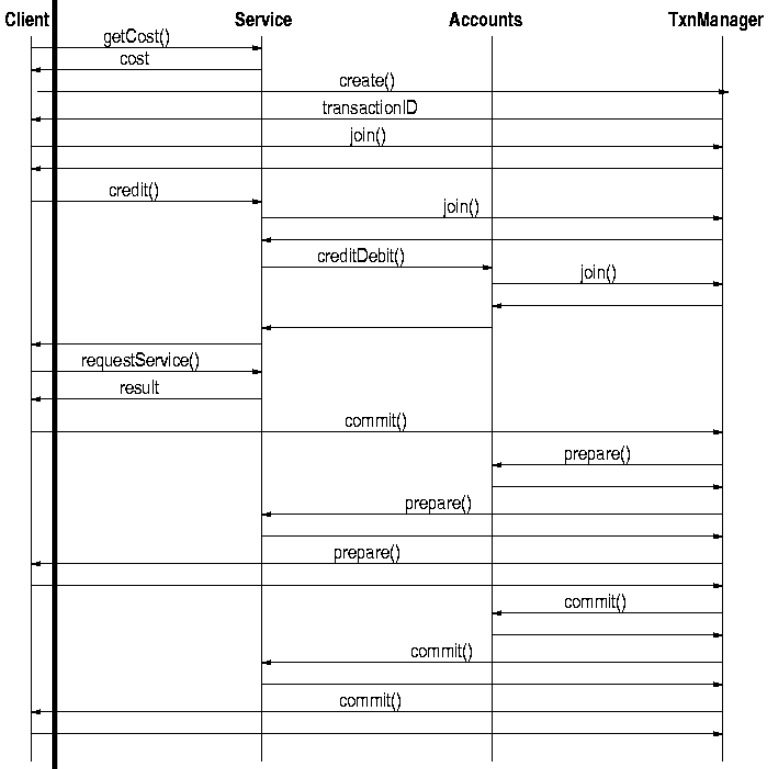
```
java  -Djava.security.policy=policy.all \
      -Dcom.sun.jini.mahalo.managerName=TransactionManager \
      -jar /home/jan/tmpdir/jini1_0/lib/mahalo.jar \
      http://`hostname`:8080/mahalo-dl.jar \
      /home/jan/projects/jini/doc/policy.all \
      /tmp/mahalo_log public &
```

6. Example

The classic use of transactions is to handle money transfers between accounts. In this there are two accounts, one of which is debited and the other credited. This is not too exciting as an example, so we shall try a more complex situation. A service may decide to charge for its use. If a client decides this cost is reasonable, it will first credit the service and then request that the service be performed. The actual accounts will be managed by an accounts service, which will need to be informed of the credits and debits that occur. A simple accounts model is that the service gets, say, a customer ID from the client, and passes its own ID and the customer ID to the accounts service which manages both accounts. Simple, prone to all sorts of e-commerce issues that we have no intention of going into, and similar to the way credit cards work! This might give rise to the following messages in a normal sequence diagram:



There are a number of problems with this that can benefit by use of a transaction model. The steps of `credit()` and `creditDebit()` should certainly be performed either both together or not at all. But in addition there is the issue of the quality of the service: suppose the client is not happy with the results from the service. It would like to reclaim its money, or better yet, not spend it in the first case! So if we include the delivery of the service in the transaction then there is the opportunity to for the client to abort the transaction before it is committed. This leads to the following messages in the sequence diagram:



The points of failure in this transaction include

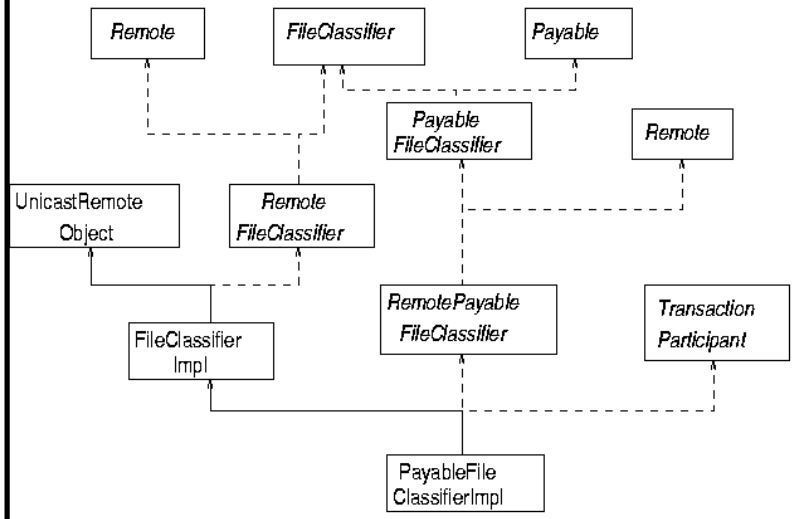
1. The cost may be too high for the client. However, at this stage it has not created or joined a transaction, so this doesn't matter
2. The client may offer too little by way of payment to the service. The service can signal this by joining the transaction and then aborting it. This will ensure that the client has to rollback the transaction. (Of course, it could instead throw a `NotEnoughPayment` exception - joining and aborting is used for illustrating transaction possibilities.)
3. There may be a time delay between finding the price and asking for the service. The price may have gone up in the meantime! The service would then abort the transaction, forcing the client and the accounts to rollback
4. After the service is performed, the client may decide that the result was not good enough, and refuse to pay. Aborting the transaction at this stage would cause the service and accounts to rollback

5. The accounts service may abort the transaction if sufficient client funds are unavailable

6.1 PayableFileClassifierImpl

The service is a version of the familiar file classifier that requires a payment before it will divulge the MIME type for a file name. A bit unrealistic, perhaps, but that doesn't matter for purposes here. There will be an interface `PayableFileClassifier`, which extends the `FileClassifier` interface. We will also make it extend the `Payable` interface, just in case we want to charge for other services. In line with other interfaces, we shall extend this to a `RemotePayableFileClassifier`, and then implement this by a `PayableFileClassifierImpl`.

The `PayableFileClassifierImpl` can use the implementation of the `option3.FileClassifierImpl`, so we shall make it extend this class. We also want it to be a participant in a transaction, so it must implement the `TransactionParticipant` interface. This leads to the following inheritance diagram, which isn't really as complex as it looks:



The new elements in this hierarchy are firstly the interface `Payable`

```
package common;

import java.io.Serializable;
import net.jini.core.transaction.server.TransactionManager;

/**
 * Payable.java
 *
```



```
 *
 * Created: Tue Aug  3 13:58:01 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public interface Payable extends Serializable {

    void credit(long amount, long accountID,
                TransactionManager mgr,
                long transactionID)
        throws java.rmi.RemoteException;

    long getCost() throws java.rmi.RemoteException;
} // Payable
```

Extending this is the interface `PayableFileClassifier`. This will be used by the client to search for the service

```
package common;

/**
 * PayableFileClassifier.java
 *
 *
 * Created: Tue Aug  3 14:14:44 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public interface PayableFileClassifier extends FileClassifier, Payable {

} // PayableFileClassifier
```

with a simple extension to the remote form

```
package txn;

import common.PayableFileClassifier;
import java.rmi.Remote;

/**
 * RemotePayableFileClassifier.java
 *
 *
```

```
 * Created: Wed Mar 17 14:14:36 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public interface RemotePayableFileClassifier extends PayableFileClassifier, Remote {

} // RemotePayableFileClasssifier
```

The implementation of this joins the transaction, finds an `Accounts` service from a known location (using unicast lookup), registers the money transfer, and then performs the service. There is no real state information kept by this implementation which is altered by the transaction. When asked to `prepare()` by the transaction manager it can just return `NOTCHANGED`. If there was state, the `prepare()` and `commit()` methods would have more content.

```
package txn;

import common.MIMEType;
import common.Accounts;
import option3.FileClassifierImpl;
//import common.PayableFileClassifier;
//import common.Payable;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.transaction.server.TransactionParticipant;
import net.jini.core.transaction.server.TransactionConstants;
import net.jini.core.transaction.UnknownTransactionException;
import net.jini.core.transaction.CannotJoinException;
import net.jini.core.transaction.CannotAbortException;
import net.jini.core.transaction.server.CrashCountException;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;

/**
 * PayableFileClassifierImpl.java
 *
 *
 * Created: Tue Aug  3 14:39:48 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class PayableFileClassifierImpl extends FileClassifierImpl
    implements RemotePayableFileClassifier, TransactionParticipant {

    protected TransactionManager mgr = null;
```



```
    protected Accounts accts = null;
    protected long crashCount = 0; // ???
    protected long cost = 10;
    protected final long myID = 54321;

    public PayableFileClassifierImpl() throws java.rmi.RemoteException {
        super();

        System.setSecurityManager(new RMISecurityManager());
    }

    public void credit(long amount, long accountID,
                       TransactionManager mgr,
                       long transactionID) {
        System.out.println("crediting");

        this.mgr = mgr;

        // before findAccounts
        System.out.println("Joining txn");
        try {
            mgr.join(transactionID, this, crashCount);
        } catch(UnknownTransactionException e) {
            e.printStackTrace();
        } catch(CannotJoinException e) {
            e.printStackTrace();
        } catch(CrashCountException e) {
            e.printStackTrace();
        } catch(RemoteException e) {
            e.printStackTrace();
        }
        System.out.println("Joined txn");


        findAccounts();

        if (accts == null) {
            try {
                mgr.abort(transactionID);
            } catch(UnknownTransactionException e) {
                e.printStackTrace();
            } catch(CannotAbortException e) {
                e.printStackTrace();
            } catch(RemoteException e) {
                e.printStackTrace();
            }
        }

        try {
            accts.creditDebit(amount, accountID, myID,
                              transactionID, mgr);
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
        }
```

```
}

public long getCost() {
    return cost;
}

protected void findAccounts() {
    // find a known account service
    LookupLocator lookup = null;
    ServiceRegistrar registrar = null;

    try {
        lookup = new LookupLocator("jini://localhost");
    } catch(java.net.MalformedURLException e) {
        System.err.println("Lookup failed: " + e.toString());
        System.exit(1);
    }

    try {
        registrar = lookup.getRegistrar();
    } catch (java.io.IOException e) {
        System.err.println("Registrar search failed: " + e.toString());
        System.exit(1);
    } catch (java.lang.ClassNotFoundException e) {
        System.err.println("Registrar search failed: " + e.toString());
        System.exit(1);
    }
    System.out.println("Registrar found");

    Class[] classes = new Class[] {Accounts.class};
    ServiceTemplate template = new ServiceTemplate(null, classes,
                                                   null);
    try {
        accts = (Accounts) registrar.lookup(template);
    } catch(java.rmi.RemoteException e) {
        System.exit(2);
    }
}

public MIMEType getMIMEType(String fileName) throws RemoteException {

    if (mgr == null) {
        // don't process the request
        return null;
    }

    return super.getMIMEType(fileName);
}


public int prepare(TransactionManager mgr, long id) {
    System.out.println("Preparing...");
```

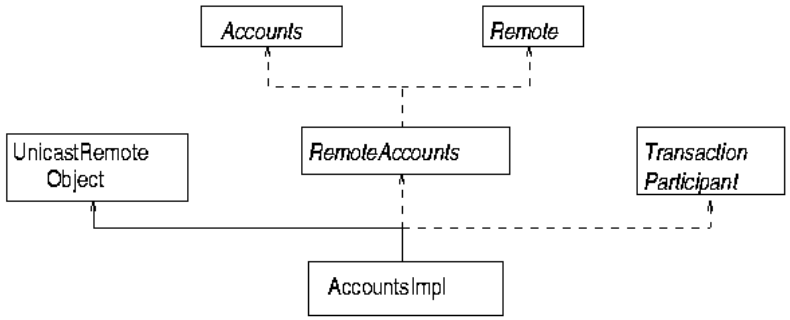


```
        return TransactionConstants.PREPARED;
    }

    public void commit(TransactionManager mgr, long id) {
        System.out.println("committing");

    }


    public void abort(TransactionManager mgr, long id) {
        System.out.println("aborting");
    }

    public int prepareAndCommit(TransactionManager mgr, long id) {
        int result = prepare(mgr, id);
        if (result == TransactionConstants.PREPARED) {
            commit(mgr, id);
            result = TransactionConstants.COMMITTED;
        }
        return result;
    }


} // PayableFileClassifierImpl
```

6.2 AccountsImpl

We shall assume that all accounts in this example are managed by a single `Accounts` service, that knows about all accounts by a `long` identifier. These should be stored in permanent form, and there should be proper crash recovery mechanisms, *etc.* For simplicity, we shall just have a hash table of accounts, with uncommitted transactions kept in a "pending" list. When commitment occurs, the pending transaction takes place.

The accounts class diagram looks like



The `Accounts` interface is

```
/**
 * Accounts.java
 *
 *
 * Created: Thu Jul 29 17:33:14 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package common;

import net.jini.core.transaction.server.TransactionManager;

public interface Accounts  {

    void creditDebit(long amount, long creditorID,
                     long debitorID, long transactionID,
                     TransactionManager tm)
        throws java.rmi.RemoteException;

} // Accounts
```

and the implementation is

```
/**
 * AccountsImpl.java
 *
 *
 * Created: Thu Jul 29 17:36:35 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package txn;

// import common.Accounts;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.transaction.server.TransactionParticipant;
import net.jini.core.transaction.server.TransactionConstants;
import java.rmi.server.UnicastRemoteObject;
import java.util.Hashtable;
// import java.rmi.RMISecurityManager;
// debug
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;
// end debug

public class AccountsImpl extends UnicastRemoteObject
    implements RemoteAccounts, TransactionParticipant, java.io.Serializable {

    protected long crashCount = 0; // value??
```



```
    protected Hashtable accountBalances = new Hashtable();
    protected Hashtable pendingCreditDebit = new Hashtable();

    public AccountsImpl() throws java.rmi.RemoteException {
        // System.setSecurityManager(new RMISecurityManager());
    }

    public void creditDebit(long amount, long creditorID,
                            long debitorID, long transactionID,
                            TransactionManager mgr) {

        java.rmi.Remote stub = null;
        try {
            stub = toStub(this);
        } catch(Exception e) {
            System.out.println("To stub failed");
            e.printStackTrace();
        }
        System.out.println("To stub found");
        String annote = java.rmi.server.RMIClassLoader.getClassAnnotation(stub.getClass());
        System.out.println("from " + annote);
        try {
            Class cl = java.rmi.server.RMIClassLoader.loadClass(annote, "txn.AccountsImpl_Stub");
        } catch(Exception e) {
            System.out.println("To stub class failed");
            e.printStackTrace();
        }
        System.out.println("To stub class ok");

        // mgr = findManager();
        try {
            System.out.println("Trying to join");
            mgr.join(transactionID, this, crashCount);
        } catch(net.jini.core.transaction.UnknownTransactionException e) {
            e.printStackTrace();
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
        } catch(net.jini.core.transaction.server.CrashCountException e) {
            e.printStackTrace();
        } catch(net.jini.core.transaction.CannotJoinException e) {
            e.printStackTrace();
        }
        System.out.println("joined");
        pendingCreditDebit.put(new TransactionPair(mgr,
                                                   transactionID),
                               new CreditDebit(amount, creditorID,
                                               debitorID));
    }

    // findmanager debug hack
    protected TransactionManager findManager() {
        // find a known account service
        LookupLocator lookup = null;
        ServiceRegistrar registrar = null;
        TransactionManager mgr = null;

        try {
            lookup = new LookupLocator("jini://localhost");
        } catch(java.net.MalformedURLException e) {
            System.err.println("Lookup failed: " + e.toString());
            System.exit(1);
        }

        try {
```

```
            registrar = lookup.getRegistrar();
        } catch (java.io.IOException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        } catch (java.lang.ClassNotFoundException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        }
        System.out.println("Registrar found");

        Class[] classes = new Class[] {TransactionManager.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
        try {
            mgr = (TransactionManager) registrar.lookup(template);
        } catch(java.rmi.RemoteException e) {
            System.exit(2);
        }
        return mgr;
    }

    public int prepare(TransactionManager mgr, long id) {
        System.out.println("Preparing...");
        return TransactionConstants.PREPARED;
    }

    public void commit(TransactionManager mgr, long id) {
        System.out.println("committing");

    }


    public void abort(TransactionManager mgr, long id) {
        System.out.println("aborting");
    }

    public int prepareAndCommit(TransactionManager mgr, long id) {
        int result = prepare(mgr, id);
        if (result == TransactionConstants.PREPARED) {
            commit(mgr, id);
            result = TransactionConstants.COMMITTED;
        }
        return result;
    }

    class CreditDebit {
        long amount;
        long creditorID;
        long debitorID;

        CreditDebit(long a, long c, long d) {
            amount = a;
            creditorID = c;
            debitorID = d;
        }
    }

    class TransactionPair {

        TransactionPair(TransactionManager mgr, long id) {

        }
    }
} // AccountsImpl
```


6.3 Client

The final component in this is the client which starts the transaction. It uses a nested class which extends `Thread`. Because of this, it cannot extend `UnicastRemoteObject`, and so is not automatically exported. In order to export itself, it has to call `UnicastRemoteObject.exportObject)`. This must be done before the call to join the transaction, which expects a remote object.

```
package client;

import common.PayableFileClassifier;
import common.MIMEType;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.transaction.server.TransactionConstants;
import net.jini.core.transaction.server.TransactionParticipant;
// import com.sun.jini.lease.LeaseRenewalManager;
import net.jini.lease.LeaseRenewalManager;
import net.jini.core.lease.Lease;
import net.jini.lookup.entry.Name;
import net.jini.core.entry.Entry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

/**
 * TestTxn.java
 *
 *
 * Created: Tue Aug 3 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    uses Jini 1.1 LeaseRenewalManager
 */

public class TestTxn implements DiscoveryListener {

    PayableFileClassifier classifier = null;
    TransactionManager mgr = null;

    long myClientID; // my account id

    public static void main(String argv[]) {
        new TestTxn();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(100000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestTxn() {
        System.setSecurityManager(new RMISecurityManager());
```

```
    LookupDiscovery discover = null;
    try {
        discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
    } catch(Exception e) {
        System.err.println(e.toString());
        System.exit(1);
    }

    discover.addDiscoveryListener(this);

}

public void discovered(DiscoveryEvent evt) {

    ServiceRegistrar[] registrars = evt.getRegistrars();

    for (int n = 0; n < registrars.length; n++) {
        System.out.println("Service found");
        ServiceRegistrar registrar = registrars[n];

        new LookupThread(registrar).start();
    }

        // System.exit(0);
}

public void discarded(DiscoveryEvent evt) {
    // empty
}

public class LookupThread extends Thread implements TransactionParticipant, java.io.Serializable {

    ServiceRegistrar registrar;
    long crashCount = 0; // ???

    LookupThread(ServiceRegistrar registrar) {
        this.registrar = registrar;
    }


    public void run() {
        long cost = 0;

        // try to find a classifier if we haven't already got one
        if (classifier == null) {
            System.out.println("Searching for classifier");
            Class[] classes = new Class[] {PayableFileClassifier.class};
            ServiceTemplate template = new ServiceTemplate(null, classes,
                                                           null);

            try {
                Object obj = registrar.lookup(template);
                System.out.println(obj.getClass().toString());
                Class cls = obj.getClass();
                Class[] clss = cls.getInterfaces();
                for (int n = 0; n < clss.length; n++) {
                    System.out.println(clss[n].toString());
                }
                classifier = (PayableFileClassifier) registrar.lookup(template);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(2);
            }
            if (classifier == null) {
                System.out.println("Classifier null");
            } else {
                System.out.println("Getting cost");
```



```
                try {
                    cost = classifier.getCost();
                } catch(java.rmi.RemoteException e) {
                    e.printStackTrace();
                }
                if (cost > 20) {
                    System.out.println("Costs too much: " + cost);
                    classifier = null;
                }
            }


        }

        // try to find a transaction manager if we haven't already got one
        if (mgr == null) {
            System.out.println("Searching for txnmgr");

            Class[] classes = new Class[] {TransactionManager.class};
            ServiceTemplate template = new ServiceTemplate(null, classes,
                                                           null);


            /*
            Entry[] entries = {new Name("TransactionManager")};
            ServiceTemplate template = new ServiceTemplate(null, null,
                                                           entries);
            */

            try {
                mgr = (TransactionManager) registrar.lookup(template);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(2);
            }
            if (mgr == null) {
                System.out.println("Manager null");
                return;
            }

        }

        if (classifier != null && mgr != null) {
            System.out.println("Found both");
            TransactionManager.Created tcs = null;


            System.out.println("Creating transaction");
            try {
                tcs = mgr.create(Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                mgr = null;
                return;
            } catch(net.jini.core.lease.LeaseDeniedException e) {
                mgr = null;
                return;
            }

            long transactionID = tcs.id;

            // join in ourselves
            System.out.println("Joining transaction");

            // but first, export ourselves since we don't extend UnicastRemoteObject
            try {
                UnicastRemoteObject.exportObject(this);
            } catch(RemoteException e) {
```

```
            e.printStackTrace();
        }

        try {
            mgr.join(transactionID, this, crashCount);
        } catch(net.jini.core.transaction.UnknownTransactionException e) {
            e.printStackTrace();
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
        } catch(net.jini.core.transaction.server.CrashCountException e) {
            e.printStackTrace();
        } catch(net.jini.core.transaction.CannotJoinException e) {
            e.printStackTrace();
        }

        new LeaseRenewalManager().renewUntil(tcs.lease,
                                             Lease.FOREVER,
                                             null);
        System.out.println("crediting...");
        try {
            classifier.credit(cost, myClientID,
                              mgr, transactionID);
        } catch(Exception e) {
            System.err.println(e.toString());
        }

        System.out.println("classifying...");
        MIMEType type = null;
        try {
            type = classifier.getMIMEType("file1.txt");
        } catch(java.rmi.RemoteException e) {
            System.err.println(e.toString());
        }

        // if we get a good result, commit, else abort
        if (type != null) {
            System.out.println("Type is " + type.toString());
            System.out.println("Calling commit");
            // new CommitThread(mgr, transactionID).run();

            try {
                System.out.println("mgr state " + mgr.getState(transactionID));
                mgr.commit(transactionID);
            } catch(Exception e) {
                e.printStackTrace();
            }

        } else {
            try {
                mgr.abort(transactionID);
            } catch(java.rmi.RemoteException e) {
            } catch(net.jini.core.transaction.CannotAbortException e) {
            } catch( net.jini.core.transaction.UnknownTransactionException e) {
            }
        }
    }
}

public int prepare(TransactionManager mgr, long id) {
    System.out.println("Preparing...");
    return TransactionConstants.PREPARED;
}

public void commit(TransactionManager mgr, long id) {
    System.out.println("committing");
}
```



```
    public void abort(TransactionManager mgr, long id) {
        System.out.println("aborting");

    }

    public int prepareAndCommit(TransactionManager mgr, long id) {
        int result = prepare(mgr, id);
        if (result == TransactionConstants.PREPARED) {
            commit(mgr, id);
            result = TransactionConstants.COMMITTED;
        }
        return result;
    }

} // LookupThread

class CommitThread extends Thread {
    TransactionManager mgr;
    long transactionID;

    public CommitThread(TransactionManager m, long id) {
        mgr = m;
        transactionID = id;
        try {
            Thread.sleep(1000);
        } catch(Exception e) {
        }
    }

    public void run() {
        try {
            mgr.abort(transactionID);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
} // CommitThread

} // TestTxn
```

7. Summary

Transactions are needed to coordinate changes of state across multiple clients and services. The Jini transaction model uses a simple model of transactions, with details of semantics left to the clients and services. The Jini distribution supplies a transaction manager that can be used.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Lego MindStorms

Contents

1. Making Hardware into Jini Services
2. MindStorms
3. MindStorms as a Jini Service
4. RCXPort
5. RCX Programs
6. Jini classes
7. Entry objects for a robot
8. Adding entries to a server
9. A client-side RCX class
10. Higher level mechanisms: Not Quite C

Lego MindStorms is a ''Robotics Invention System'' that allows you to build Lego toys with a programmable computer. This chapter looks at the issues in interfacing with a specialised hardware device, using MindStorms as example

1. Making Hardware into Jini Services

Hardware and software services can equally be turned into Jini services. A software service can have a ''wrapper'' placed around it that mediates Jini calls into the service calls. Hardware devices are a little more complex because they are defined at a lower level, and often have resource constraints that do not apply to software services.

There are two major categories of hardware service: those that can run a Java Virtual Machine and those that do not have enough memory or an adequate processor (e.g. an 8086 with 20-bit addressing and only 1M of addressable memory would not be an adequate processor). Devices capable of running a JVM may be further subdivided into those that are capable of running a standard JDK 1.2 JVM and core libraries and those that have to run some stripped-down version.

A device that runs Jini natively will need to support

1. The Jini Discovery and Lookup services
2. The particular implementations of `Lease` and `ServiceRegistrar` used by the lookup service, since these are downloaded to run locally in the service
3. Enough of the RMI features used by the above services
4. Enough of the language and core packages used by the above services

There is the possibility of a very small JVM and libraries to support particular implementations of the lookup service. None of the current embedded or small JVM's are yet capable of this, primarily because they cannot handle the JDK 1.2 extended features of RMI required by Sun's `reggie` lookup service.

Anything above this level will be more flexible, upto the level of a full JDK 1.2 with Jini. Whatever, one is looking at a device with 8M of RAM or more, with networking capabilities, on a 32-bit processor.

If the device cannot run a JVM, then something else must run the JVM and act as a proxy for the device. Your blender is unlikely to have 32M RAM, but your home control center (possibly located on the front of the fridge) may have this capability. In that case the blender service would be located in this JVM, and the fridge would have some means of sending commands to the blender.

2. MindStorms

Lego MindStorms (http://www.legomindstorms.com) is a ''Robotics Invention System'' which consists of a number of Lego parts and a microcomputer called the RCX, plus an infra-red transmitter (connected to the serial port of an ordinary computer) and various sensors and motors. Using this, one can build an almost indefinite variety of Lego robots that can be controlled by the RCX. This computer can be sent ''immediate'' commands, or can have a (small) program downloaded and then run.

MindStorms is a pretty cool system, that can be driven at a number of levels. A primary audience for programming this is children, and there is a visual programming environment to help in this. This visual environment only runs on Windows or Macintosh machines which are connected to the RCX by their serial port and the infrared transmitter. Behind this environment is a Visual Basic set of procedures captured in an OCX, and behind *that* is the machine code of the RCX which can be sent as byte codes on the serial port.

The RCX computer is completely incapable of running Jini. It is a 16-bit processor with a mere 32k of RAM, and the default firmware will only allow 32 variables. It can only be driven by a service running on, say, an ordinary PC.

3. MindStorms as a Jini Service

A MindStorms robot can be programmed and run from an infrared transmitter attached to the serial port of a computer. There is no security or real location for the RCX: it will accept commands from any transmitter in range. We will assume a ''home'' computer for it.

There must be a way of communicating with this device. For a MindStorms robot this is by the serial port, but other devices may have different mechanisms. Communication may be by Java code or by native code. Even if Java code is used, at some stage it must drop down to the native code level in order to communicate with the device - the only question is whether you write the native code or someone else does it for you and wraps it up in Java object methods.

For the serial port, Sun has an extension package - the `commAPI` - to talk to serial and parallel ports (http://java.sun.com/products/javacomm/index.html). This gives platform-independent Java code, and also platform specific native code libraries supplied as DLL's for Windows and Solaris. I am running Linux on my laptop, so I need a Linux version of the DLL. This has been made by Trent Jarvi (jarvi@ezlink.com), and can be found at http://jarvi.ezlink.com/rxtx/. The native code part of communicating to the device has been done for us, and it is all wrapped up in a set of portable Java classes.

The RCX expects particular message formats, such as starting with standard headers. A Java package to make this easier is available by Dario Laverde at http://www.escape.com/~dario/java/rcx. There are other packages that will do the same thing: see the ''Lego Mindstorms Internals'' page by Russell Nelson at http://www.crynwr.com/lego-robotics/.

With this as background, we can look at how to make an RCX into a Jini service. It will involve being able to construct an RCX program on a client and send this back to the server where it can be sent on to the RCX via the serial port. This will then allow a client to control a Mindstorms robot remotely. Actually, the Jini part is pretty easy - the hard part was tracking down all the bits and pieces needed to drive the RCX from Java. With your own lumps of hardware, the hard part will be writing the JNI and Java code to drive it.

4. RCXPort

Version 1.1 of the package by Dario Laverde defines various classes, of which the most important is `RCXPort`:

```
package rcx;

public class RCXPort {
    public RCXPort(String port);
    public void addRCXListener(RCXListener rl);
    public boolean open();
    public void close();
    public boolean isOpen();
    public OutputStream getOutputStream();
    public InputStream getInputStream();
    public synchronized boolean write(byte[] bArray);
    public String getLastError();
}
```

The class `RCXOpcode` has a useful method for creating byte code

```
package rcx;

public class RCXOpcode {
    public static byte[] parseString(String str);
}
```

(There are minor changes from version 1.0 of this package.)

We may not need all of these in this project, but never mind - we can just ignore the ones we don't want. The ones we do want are

1. The constructor `RCXPort()`. This takes the name of a port as parameter, and this should be something like `COM1` for Windows and `/dev/ttyS0` for Linux.
2. The method `write()` is used to send an array of opcodes and their arguments to the RCX. This is *machine code* and you can only read it by a dis-assembler or a Unix tool like octal dump (`od -t xC`).
3. The *static* method `parseString()` of `RCXOpcode` can be used to translate a string of instructions in readable form to an array of byte for sending to the RCX. It isn't as good as an assembler, as you have to give strings such as `"21 81"` to start the A motor. To use this for Jini, we will have to use a non-static method in our interface since static methods are not allowed.
4. To handle responses from the RCX, a listener may be added by `addRCXListener()`. The listener must implement the interface

```
package rcx;

import java.util.*;

/*
 * RCXListener
 * @author Dario Laverde
 * @version 1.1
 * Copyright 1999 Dario Laverde, under terms of GNU LGPL
 */
public interface RCXListener extends EventListener {
    public void receivedMessage(byte[] message);
    public void receivedError(String error);
}
```


5. RCX Programs

At the lowest level, the RCX is controlled by machine-code programs sent via the infrared link. It will respond to these by stopping and starting motors, changing speed, etc. As it completes commands or receives information from sensors, it can send replies back to the host computer. The RCX can handle instructions sent directly, or have a program downloaded into firmware and run from there.

Kekoa Proudfoot has produced a list of the opcodes understood by the RCX which is available at http://graphics.stanford.edu/~kekoa/rcx. Using these and the `rcx` package from Dario Laverde means we can control the RCX from the "home" computer by standalone programs such as

```
/**
 * TestRCX.java
 *
 *
 * Created: Wed Jun  2 13:34:12 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package standalone;

import rcx.*;

public class TestRCX implements RCXListener {
    static final String PORT_NAME = "/dev/ttyS0"; // Linux

    public TestRCX() {
        RCXPort port = new RCXPort(PORT_NAME);

        port.addRCXListener(this);

        byte[] byteArray;

        // send ping message, reply should be e7 or ef
        byteArray = RCXOpcode.parseString("10"); // Alive
        port.write(byteArray);

        // beep twice
        byteArray = RCXOpcode.parseString("51 01"); // Play sound
        port.write(byteArray);

        // turn motor A on (forwards)
        byteArray = RCXOpcode.parseString("e1 81"); // Set motor direction
        port.write(byteArray);
        byteArray = RCXOpcode.parseString("21 81"); // Set motor on
        port.write(byteArray);
        try {
            Thread.currentThread().sleep(1000);
        } catch(Exception e) {
        }
```

```
        // turn motor A off
        byteArray = RCXOpcode.parseString("21 41"); // Set motor off
        port.write(byteArray);

        // turn motor A on (backwards)
        byteArray = RCXOpcode.parseString("e1 41"); // Set motor direction
        port.write(byteArray);
        byteArray = RCXOpcode.parseString("21 81"); // Set motor on
        port.write(byteArray);
        try {
            Thread.currentThread().sleep(1000);
        } catch(Exception e) {
        }

        // turn motor A off
        byteArray = RCXOpcode.parseString("21 41"); // Set motor off
        port.write(byteArray);
    }

    /**
     * listener method for messages from the RCX
     */
    public void receivedMessage(byte[] message) {
        if (message == null) {
            return;
        }
        StringBuffer sbuffer = new StringBuffer();
        for(int n = 0; n < message.length; n++) {
            int newbyte = (int) message[n];
            if (newbyte < 0) {
                newbyte += 256;
            }
            sbuffer.append(Integer.toHexString(newbyte) + " ");
        }
        System.out.println("response: " + sbuffer.toString());
    }

    /**
     * listener method for error messages from the RCX
     */
    public void receivedError(String error) {
        System.err.println("Error: " + error);
    }

    public static void main(String[] args) {
        new TestRCX();
    }

} // TestRCX
```


6. Jini classes

A simple Jini service can use and RMI proxy, where the service just remains in the server and the client makes remote method calls on it. The service will hold an `RCXPort` and will feed the messages through it. This means constructing a hierarchy of classes

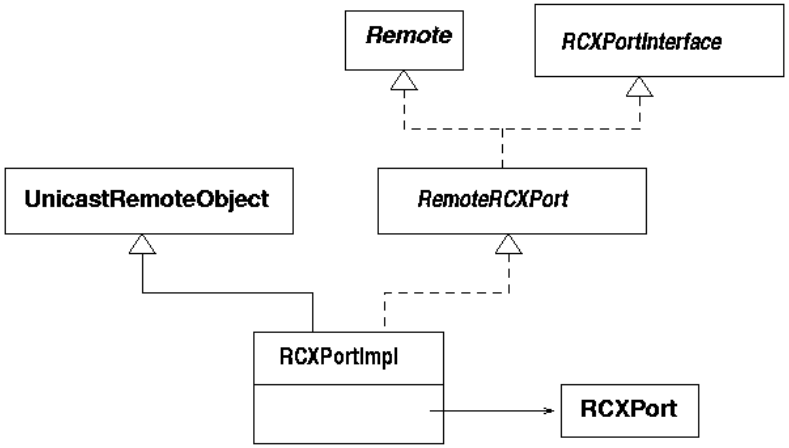


Figure 1: Jini classes for RCX

The `RCXPortInterface` just defines the methods we shall be making available from the Jini service. It doesn't have to follow the `RCXPort` methods completely, because these will be wrapped up in implementation classes such as `RCXPortImpl`. The interface is defined as

```
/**
 * RCXPortInterface.java
 *
 *
 * Created: Wed Jun  2 23:08:12 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package rcx.jini;

import net.jini.core.event.RemoteEventListener;

public interface RCXPortInterface extends java.io.Serializable {
```

```
    /**
     * constants to distinguish message types
     */
    public final long ERROR_EVENT = 1;
    public final long MESSAGE_EVENT = 2;

    /**
     * Write an array of bytes that are RCX commands
     * to the remote RCX.
     */
    public boolean write(byte[] byteCommand) throws java.rmi.RemoteException;

    /**
     * Parse a string into a set of RCX command bytes
     */
    public byte[] parseString(String command) throws java.rmi.RemoteException;

    /**
     * Add a RemoteEvent listener to the RCX for messages and errors
     */
    public void addListener(RemoteEventListener listener)
        throws java.rmi.RemoteException;

    /**
     * The last message from the RCX
     */
    public byte[] getMessage(long seqNo)
        throws java.rmi.RemoteException;

    /**
     * The error message from the RCX
     */
    public String getError(long seqNo)
        throws java.rmi.RemoteException;

} // RCXPortInterface
```

We have chosen to make it a sub-package of the `rcx` package to make its role clearer. Note that it has no static methods, but makes `parseString()` into an ordinary instance method.

This interface contains two types of methods: those used to prepare and send messages to the RCX (`write()` and `parseString()`), and those for handling messages sent from the RCX (`addListener()`, `getMessage()` and `getError()`). Adding a listener means that it will be informed of events generated by implementations of this interface by having the listener's `notify()` method called. However, a `RemoteEvent` does not contain detailed information about what has happened, only an event type (`MESSAGE_EVENT` or `ERROR_EVENT`). It is up to the listener to make queries back into the object to discover what the event meant, which it does by `getMessage()` and `getError()`.

The interface `RemoteRCXPort` adds the `Remote` interface as before

```
/**
 * RemoteRCXPort.java
 *
```



```
 *
 * Created: Wed Jun  2 23:13:17 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package rcx.jini;

import java.rmi.Remote;

public interface RemoteRCXPort extends RCXPortInterface, Remote {

} // RemoteRCXPort
```

The `RCXPortImpl` constructs its own `RCXPort` object and feeds methods through to it, such as `write()`. Since it extends `UnicastRemoteObject` it also adds exceptions to each method, which cannot be done to the original `RCXPort` class. In addition, it picks up the value of the port name from the `port` property. (This follows the example of the `RCXLoader` in the `rcx` package which gives a GUI interface to driving the RCX.) It looks for this property in a file `parameters.txt` which should have lines such as

```
port=/dev/ttyS0
```

Note that the parameters file exists of course on the server side - no client would know this information!

The `RCXPortImpl` also acts as a listener for ''ordinary'' RCX events signalling messages from the RCX. It uses the callback methods `receivedMessage()` and `receivedError()` to create a new `RemoteEvent` object and send it to the implementation's listener object (if any) by calling its `notify()` method.

The implementation looks like

```
/**
 * RCXPortImpl.java
 *
 *
 * Created: Wed Jun  2 23:14:58 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package rcx.jini;

import java.rmi.server.UnicastRemoteObject;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import rcx.*;
import java.io.*;
import java.util.*;

public class RCXPortImpl extends UnicastRemoteObject
    implements RemoteRCXPort, RCXListener {

    protected String error = null;
    protected byte[] message = null;
```

```
protected RCXPort port = null;
protected RemoteEventListener listener = null;
protected long messageSeqNo, errorSeqNo;

public RCXPortImpl()
    throws java.rmi.RemoteException {

    Properties parameters;
    String portName = null;
    File f = new File("parameters.txt");
    if (!f.exists()) {
        f = new File(System.getProperty("user.dir")
                     + System.getProperty("path.separator")
                     + "parameters.txt");
    }
    if (f.exists()) {
        try {
            FileInputStream fis = new FileInputStream(f);
            parameters = new Properties();
            parameters.load(fis);
            fis.close();
            portName = parameters.getProperty("port");
        } catch (IOException e) { }
    } else {
        System.err.println("Can't find parameters.txt with \"port=...\" specified");
        System.exit(1);
    }

    port = new RCXPort(portName);
    port.addRCXListener(this);

}

public boolean write(byte[] byteCommands)
    throws java.rmi.RemoteException {
    return port.write(byteCommands);
}

public byte[] parseString(String command)
    throws java.rmi.RemoteException {
    return RCXOpcode.parseString(command);
}

/**
 * Received a message from the RCX.
 * Send it to the listener
 */
public void receivedMessage(byte[] message) {

    this.message = message;

    // Send it out to listener
    if (listener == null) {
        return;
    }

    RemoteEvent evt = new RemoteEvent(this, MESSAGE_EVENT, messageSeqNo++, null);
    try {
```



```
        listener.notify(evt);
    } catch(net.jini.core.event.UnknownEventException e) {
        e.printStackTrace();
    } catch(java.rmi.RemoteException e) {
        e.printStackTrace();
    }
}

/**
 * Received an error message from the RCX.
 * Send it to the listener
 */
public void receivedError(String error) {
    // System.err.println(error);

    // Send it out to listener
    if (listener == null) {
        return;
    }
    this.error = error;
    RemoteEvent evt = new RemoteEvent(this, ERROR_EVENT, errorSeqNo, null);
    try {
        listener.notify(evt);
    } catch(net.jini.core.event.UnknownEventException e) {
        e.printStackTrace();
    } catch(java.rmi.RemoteException e) {
        e.printStackTrace();
    }
}

/**
 * Expected use: the RCX has returned a message,
 * and we have informed the listeners. They query
 * this method to find the message for the message
 * seqence number they were given in the RemoteEvent.
 * We could use this as an index into a table of messages.
 */
public byte[] getMessage(long msgSeqNo) {
    return message;
}

/**
 * Expected use: the RCX has returned an error message,
 * and we have informed the listeners. They query
 * this method to find the error message for the error message
 * seqence number they were given in the RemoteEvent.
 * We could use this as an index into a table of messages.
 */
public String getError(long errSeqNo) {
    return error;
}

/**
 * Add a listener for RCX messages.
 * Should allow more than one, or throw
 * TooManyListeners if more than one registers
 */
public void addListener(RemoteEventListener listener) {
```

```
        this.listener = listener;
        messageSeqNo = 0;
        errorSeqNo = 0;
    }
} // RCXPortImpl
```

7. Getting it running

To make use of these classes, we need to provide a server to get the service put onto the network, and some clients to make use of the service. This section will just look at a simple way of doing this, and later sections will try to put in more structure.

A simple server follows earlier servers using RMI proxies, just substituting `RCXPort` for `FileClassifier`, and using a `JoinManager`. It creates an `RCXPortImpl` object and registers it (rather, the RMI proxy) with lookup services:

```
package rcx.jini;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.core.lookup.ServiceID;
// import com.sun.jini.lease.LeaseRenewalManager;
// import com.sun.jini.lookup.JoinManager;
// import com.sun.jini.lookup.ServiceIDListener;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.JoinManager;
import net.jini.lookup.ServiceIDListener;

/**
 * RCXServer.java
 *
 *
 * Created: Wed Mar 17 14:23:44 1999
 *
 * @author Jan Newmarch
 * @version 1.2
 *    removed second proxy, now just use RMI proxy
 *    added LeaseRenewalManager
 *    moved sleep() from constructor to main()
 *    uses Jini 1.1 JoinManager
 */

public class RCXServer implements ServiceIDListener {

    protected RCXPortImpl impl;
    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

    public static void main(String argv[]) {
        new RCXServer();
        // remember to keepalive
    }
```



```
    public RCXServer() {
        try {
            impl = new RCXPortImpl();
        } catch(Exception e) {
            System.err.println("New impl: " + e.toString());
            System.exit(1);
        }

        // set RMI security manager
        System.setSecurityManager(new RMISecurityManager());

        // find, register, lease, etc
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            JoinManager joinMgr = new JoinManager(impl,
                                                  null,
                                                  this,
                                                  mgr,
                                                  new LeaseRenewalManager());
        } catch(java.io.IOException e) {
            e.printStackTrace();
        }
    }

    public void serviceIDNotify(ServiceID serviceID) {
        System.out.println("Got service ID " + serviceID.toString());
    }
} // RCXServer
```

Why is it simplistic *as a service*? Well, it doesn't contain any information to allow a client to distinguish one Lego Mindstorms robot from another, so that if there are many robots on the network then a client could ask the wrong one to do things!

An equally simplistic client to make the RCX do a few actions is given below. In addition to sending a set of commands to the RCX, it must also listen for replies from the RCX. We separate out this listener as an `EventHandler` for readability. The listener will act as a remote event listener, with its `notify()` method called from the server. This can be done by letting it run an RMI stub on the server, so we subclass it from `UnicastRemoteObject`.

This particular client is designed to drive a particular robot: the ''RoverBot'' described in the Lego MindStorms ''Constructopedia''. This has motors to drive tracks or wheels on either side. The client can send instructions to make the RoverBot move forwards or backwards, stop, or turn to the left or right. The set of commands (and their implementation as RCX instructions) depends on the robot, and on what you want to do with it.

```
package client;

import rcx.jini.*;

import java.awt.*;
```

```
import java.awt.event.*;
import javax.swing.*;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.UnknownEventException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
/**
 * TestRCX.java
 *
 *
 * Created: Wed Mar 17 14:29:15 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *    added GUI front-end
 */

public class TestRCX implements DiscoveryListener {

    public static final int STOPPED = 1;
    public static final int FORWARDS = 2;
    public static final int BACKWARDS = 4;

    protected int state = STOPPED;

    public static void main(String argv[]) {
        new TestRCX();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(10000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestRCX() {
        System.setSecurityManager(new RMISecurityManager());

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }
```



```
        discover.addDiscoveryListener(this);

    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();
        Class [] classes = new Class[] {RCXPortInterface.class};
        RCXPortInterface port = null;
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Service found");
            ServiceRegistrar registrar = registrars[n];
            try {
                port = (RCXPortInterface) registrar.lookup(template);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(2);
            }
            if (port == null) {
                System.out.println("port null");
                continue;
            }

            // add an EventHandler as an RCX Port listener
            try {
                port.addListener(new EventHandler(port));
            } catch(Exception e) {
                e.printStackTrace();
            }
        }
    }

    public void discarded(DiscoveryEvent evt) {
        // empty
    }

    class EventHandler extends UnicastRemoteObject
                       implements RemoteEventListener, ActionListener {

        protected RCXPortInterface port = null;

        JFrame frame;
        JTextArea text;

        public EventHandler(RCXPortInterface port) throws RemoteException {
            super() ;
            this.port = port;

            frame = new JFrame("Lego MindStorms");
            Container content = frame.getContentPane();
```

```
        JLabel label = new JLabel(new ImageIcon("images/mindstorms.jpg"));
        JPanel pane = new JPanel();
        pane.setLayout(new GridLayout(2, 3));

        content.add(label, "North");
        content.add(pane, "Center");

        JButton btn = new JButton("Forward");
        pane.add(btn);
        btn.addActionListener(this);

        btn = new JButton("Stop");
        pane.add(btn);
        btn.addActionListener(this);

        btn = new JButton("Back");
        pane.add(btn);
        btn.addActionListener(this);

        btn = new JButton("Left");
        pane.add(btn);
        btn.addActionListener(this);

        label = new JLabel("");
        pane.add(label);

        btn = new JButton("Right");
        pane.add(btn);
        btn.addActionListener(this);

        frame.pack();
        frame.setVisible(true);
    }

    public void sendCommand(String comm) {
        byte[] command;
        try {
            command = port.parseString(comm);
            if (! port.write(command)) {
                System.err.println("command failed");
            }
        } catch(RemoteException e) {
            e.printStackTrace();
        }
    }

    public void forwards() {
        sendCommand("e1 85");
        sendCommand("21 85");
        state = FORWARDS;
    }

    public void backwards() {
        sendCommand("e1 45");
```



```
        sendCommand("21 85");
        state = BACKWARDS;
    }

    public void stop() {
        sendCommand("21 45");
        state = STOPPED;
    }

    public void restoreState() {
        if (state == FORWARDS)
            forwards();
        else if (state == BACKWARDS)
            backwards();
        else
            stop();
    }

    public void actionPerformed(ActionEvent evt) {
        String name = evt.getActionCommand();

        if (name.equals("Forward")) {
            forwards();
        } else  if (name.equals("Stop")) {
            stop();
        } else  if (name.equals("Back")) {
            backwards();
        } else  if (name.equals("Left")) {
            sendCommand("e1 81");
            sendCommand("21 81");
            sendCommand("21 44");
            try {
                Thread.sleep(100);
            } catch(InterruptedException e) {
            }
            restoreState();

        } else  if (name.equals("Right")) {
            sendCommand("e1 84");
            sendCommand("21 84");
            sendCommand("21 41");
            try {
                Thread.sleep(100);
            } catch(InterruptedException e) {
            }
            restoreState();
        }
    }

    public void notify(RemoteEvent evt) throws UnknownEventException,
                                               java.rmi.RemoteException {
        // System.out.println(evt.toString());

        long id = evt.getID();
```

```
            long seqNo = evt.getSequenceNumber();
            if (id == RCXPortInterface.MESSAGE_EVENT) {
                byte[] message = port.getMessage(seqNo);
                StringBuffer sbuffer = new StringBuffer();
                for(int n = 0; n < message.length; n++) {
                    int newbyte = (int) message[n];
                    if (newbyte < 0) {
                        newbyte += 256;
                    }
                    sbuffer.append(Integer.toHexString(newbyte) + " ");
                }
                System.out.println("MESSAGE: " + sbuffer.toString());
            } else if (id == RCXPortInterface.ERROR_EVENT) {
                System.out.println("ERROR: " + port.getError(seqNo));
            } else {
                throw new UnknownEventException("Unknown message " + evt.getID());
            }
        }
    }

} // TestRCX
```

Why is one simplistic *as a client*? It tries to find all robots on the local network, and sends the same set of commands to it. Worse, if a robot has registered with, say, half-a-dozen service locators, and the client finds all of these, then it will send the same set of commands six times to the same robot! Some smarts are needed here...

8. Entry objects for a robot

The RCX was not designed for network visibility. It has no concept of identity or location. The closest it comes to this is when it communicates to other RCX's by the infrared transmitter: then one RCX may have to decide if it is the master, which it does by setting a local variable to 'master' if it broadcasts before it receives, while the other RCX's will set the variable to 'slave' if they receive before broadcasting. Crude, but it works.

In a Jini environment, there may be many RCX devices. These devices are not 'tied' to any particular computer, as they will respond to any infrared transmitter on the correct frequency talking the right protocol. All the devices within range of a transmitter will accept signals from the transmitter, although this can cause problems as the source computers tend to assume that there is only one target at a time and can get confused by responses from multiple RCXs. The advice is to ''turn off all but one RCX when a program is being downloaded'' to avoid this confusion. Then turn on the next, and download to it, etc.

An RCX may also be mobile - it can control motors, so if it is placed in a mobile robot it can drive itself out of the range of one PC and (maybe) into the range of another. There are no mechanisms to signal either passing out of range, or coming into range.

The RCX is a poorly behaved animal from a network viewpoint. However, we will need to distinguish between different RCX's in order to drive the correct ones. An `Entry` class to distinguish them should contain information such as

1. An identifier for robot type, such as ''Robo 1'', ''Acrobot 1'', etc. This will allow the robot that the RCX is built into to be determined. The RCX will have no knowledge of this - it must be externally supplied.
2. The RCX can be driven by direct commands, or by executing a program already downloaded (there may be upto five of these). An identifier for each program downloaded should be available for choice.

3. The RCX will have some sort of location, although it may move around to a limited extent. This information may be available from the controlling computer, using the Jini `Location` or `Address` classes.

There may be other useful attributes, and there are certainly issues about how the information could be stored and accessed from an RCX. However, they stray beyond the bounds of this chapter.

9. Adding entries to a server

See next installment :-)

10. A client-side RCX class

In the simplistic client given earlier, there were many steps that will be the same for all clients that can drive the RCX. In a similar way that `JoinManager` simplifies repetitive code on the server side, we can define a ''convenience'' class for the RCX that will do the same on the client side. The aim is to supply a class that will make remote RCX programming as easy as local RCX programming.

A class to encapsulate client-side behaviour may as well look as much as possible like the local `RCXPort` class. We define its (public) methods as

```
public class JiniRCXPort {
    public JiniRCXPort();
    public void addRCXListener(RCXListener l);
    public boolean write(byte[] bArray);
    public byte[] parseString(String str);
}
```

This class should have some control over how it looks for services, by including entry information, group information about locators, and any specific locators it should try. There are a variety of constructors, all ending up in

```
    public JiniRCXPort(Entry[] entries,
                       java.lang.String[] groups,
                       LookupLocator[] locators)
```

The class is also concerned with *uniqueness* issues, as it should not attempt to send the same instructions to an RCX more than once. However, it could send the same instructions to more than one RCX if it matches the search criteria. So this class maintains a list of RCX's, and does not add to the list if it has already seen the RCX from another service locator. This implies that a single RCX is registered with the same `ServiceID` with all locators, which it has because the RCX server uses `JoinManager`.

11. Higher level mechanisms: Not Quite C

''Not Quite C'' is a language and a compiler from David Baum, designed for the RCX. It defines a language with C-like syntax which defines *tasks* which can be executed concurrently. The RCX API also defines a number of constants, functions and macros targetted specifically to the RCX. These include constants such as `OUT_A` (for output 'A') and functions such as `OnFwd` to turn a motor on forwards.

A trivial NQC program to turn motor 'A' on for 1 second (units are 1/100th of a second) is

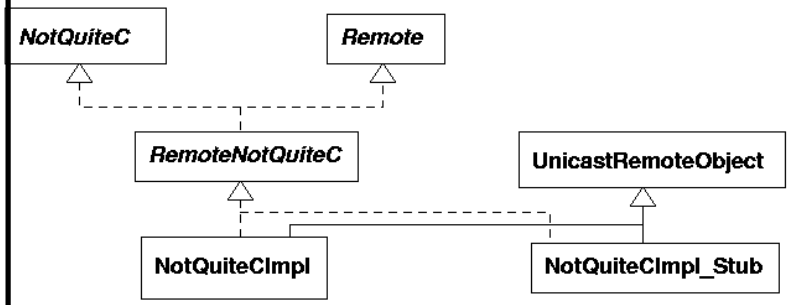
```
task main() {
        OnFwd(OUT_A);
        Wait(100);
        Off(OUT_A);
}
```

Writing programs using a higher-level language such as this is clearly preferable to writing in Assembler!

NQC is not the only higher-level language for programming the RCX. There are links to many others on the alternative MindStorms site http://www.crynwr.com/lego-robotics/. It is one of the earliest and more popular ones, though, and forms a typical example of a stand-alone, non-GUI program written in a language other than Java that can still be used as a Jini service.

The NQC compiler is written in C++ and needs to be compiled for each platform that it will run on. Pre-compiled versions are available for a number of systems such as Windows and Linux. Once compiled, it is tied to a particular computer (at least, to computers with a particular O/S and shared library configuration). It is software, not hardware like the MindStorms robots, but is nevertheless not mobile. It cannot be moved around like Java code can. However, it can be turned into a Jini service in exactly the same way as MindStorms, by wrapping it in a Java class that can be exported as a Jini service. This also fits the RMI proxy model, with the client-side using a thin proxy that makes calls to a service that invokes the NQC compiler.

The class diagram follows other RMI proxy diagrams



The interfaces are defined by

```
/**
 * NotQuiteC.java
 *
 *
 * Created: Wed Oct 27 21:00:57 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package rcx.jini;
```



```
import java.rmi.RemoteException;
import java.io.Serializable;

public interface NotQuiteC extends Serializable {

    public byte[] compile(String program)
        throws RemoteException, CompileException;

} // NotQuiteC
```

and

```
/**
 * RemoteNotQuiteC.java
 *
 *
 * Created: Wed Oct 27 21:06:52 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package rcx.jini;

import java.rmi.Remote;

public interface RemoteNotQuiteC extends NotQuiteC, Remote {

} // RemoteNotQuiteC
```

The compile exception is thrown when things go wrong, and is

```
/**
 * CompileException.java
 *
 *
 * Created: Wed Oct 27 21:03:55 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package rcx.jini;

public class CompileException extends Exception {

    protected String error;

    public CompileException(String err) {
        error = err;
```

```
    }

    public String toString() {
        return error;
    }
} // CompileException
```

The implementation needs to encapsulate a traditional application running in an environment of just reading and writing files. GUI applications, or those nuisance Unix ones that insist on using an interactive terminal (such as `telnet`) will need more complex encapsulation methods. The `nqc` type of application will read from standard input or from a file, often depending on command line flags. Similarly, it will write to a file or to standard output, again depending on command line flags. Applications either succeed or fail in their task: this should be indicated by what is know as an *exit code*, which by convention is zero for success, something else for failure. If a failure occurs, an application will usually write diagnostic output to the standard error channel.

The current version of `nqc` (version 2.0.2) is badly behaved for reading from standard input (it crashes) and writing to standard output (no way of doing this). So we can't create a `Process` to run `nqc` and feed into its input and output. Instead, we need to create temporary files and write to and read from these files, so that the Jini wrapper can communicate wtih `nqc`. These files also need to be cleaned up on termination, whether the normal or exception routes are followed! On the other hand, if errors occur they will be reported on the error channel of the process, and this needs to be captured in some way - here we do it via an exception constructor.

The hard part in this example is ploughing your way through the Java I/O maze, and deciding exactly how to hook up I/O streams and/or files to the external process. The following code uses temporary files for ordinary I/O with the process, and the standard error stream for compile errors.

```
/**
 * NotQuiteCImpl.java
 *
 *
 * Created: Wed Oct 27 21:08:01 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package rcx.jini;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import java.io.*;

public class NotQuiteCImpl extends UnicastRemoteObject
    implements RemoteNotQuiteC {

    protected final int SIZE = 1<<15; // 32k - the size of the RCX memory

    public NotQuiteCImpl() throws RemoteException {
    }

    public byte[] compile(String program)
        throws CompileException {
```



```
        // This is the input file we read from - it is the output from nqc
        File inFile = null;
        //  This is the output file that we write to - it is the input to nqc
        File outFile = null;
        byte[] buff = new byte[SIZE];

        try {
            outFile = File.createTempFile("jini", ".nqc");
            inFile = File.createTempFile("jini", ".rcx");
            OutputStreamWriter out = new OutputStreamWriter(
                                        new FileOutputStream(
                                            outFile));

            out.write(program);
            out.close();

            Process p = Runtime.getRuntime().exec("nqc -O" +
                                                  inFile.getAbsolutePath() +
                                                  " " + outFile.getAbsolutePath());

            int status = p.waitFor();
            if (status != 0) {
                BufferedReader err = new BufferedReader(
                                        new InputStreamReader(p.
                                             getErrorStream()));
                StringBuffer errBuff = new StringBuffer();
                String line;
                while ((line = err.readLine()) != null) {
                    errBuff.append(line + '\n');
                }
                throw new CompileException(errBuff.toString());
            }

            DataInputStream compiled = new DataInputStream(new
                                                FileInputStream(outFile));
            int nread = compiled.read(buff);
            byte[] result = new byte[nread];
            System.arraycopy(buff, 0, result, 0, nread);
            return result;
        } catch(IOException e) {
            throw new CompileException(e.toString());
        } catch(InterruptedException e) {
            throw new CompileException(e.toString());
        } finally {
            // cleanup files even if exceptions thrown
            if (inFile != null) {
                inFile.delete();
            }
            if (outFile != null) {
                outFile.delete();
            }
        }
    }
```

```
    public static void main(String[] argv) {
        String program = "task main() {\n" +
                         "     OnFwd(OUT_A);\n" +
                         "     Wait(100);\n" +
                         "     Off(OUT_A);\n" +
                         "}";
        NotQuiteCImpl compiler = null;
        try {
            compiler = new NotQuiteCImpl();
            byte[] bytes = compiler.compile(program);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
} // NotQuiteCImpl
```

This section does not give server and client implementations: the server is the same as servers delivering other RMI services (is there a design pattern here - or just an implementation pattern?). A client will make a call on this service, specifying the program to be compiled. It can then write the byte stream to the RCX using the classes given earlier. *To be honest, I am assuming here - this hasn't been tested. My RCX died after being in the hold of several airplanes. The robot came to pieces, the battery case fell off and the batteries got dislodged (so the stored programs got lost), and the RCX won't respond to the infra-red port any more to download new programs :-(((*

12. Summary

This chapter has considered some of the issues involved in making a piece of hardware with a Jini service. This was illustrated with Lego MindStorms, where a large part of the base work of native code libraries and encapsulation in Java classes has already been done. Even then, there is still much work to make it into a suitable Jini service, and these have been discussed. This not yet complete, and more work remains to be done for Lego MindStorms.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Corba and Jini

Contents

1. CORBA
2. CORBA to Java Mapping
3. Jini Proxies
4. Simple Example
    1. Corba Server in Java
    2. Corba Client in Java
    3. Jini Service
    4. Jini Server and Client
    5. Building the Example
    6. Running the Example
5. Room Booking Example
    1. CORBA Objects
    2. Multiple Objects
    3. Exceptions
    4. Interfaces for Single Thin Proxy
    5. RoomBookingBridge Implementation
    6. Other Classes
    7. Building the Example
    8. Running the Example
6. Migrating a CORBA Client to Jini
7. Jini Service as a CORBA Object

1. CORBA

CORBA is also an infrastructure for distributed systems. It is targeted at a slightly different "space" to Jini, and has some minor and major differences.

1. It allows specification of objects which can be distributed. The concentration is on *distributed objects* rather than on *distributed services*
2. It is language-independent, using an Interface Definition Language (IDL) for specification of interfaces
3. CORBA objects can be implemented in a number of languages, including C, C++, SmallTalk and Java
4. Current versions of CORBA pass remote object references, rather than complete object instances. Each CORBA object lives within a server, and the object can only act within this server. This is more restricted than Jini, where an object can have instance data and class files sent to a remote location and execute there. This limitation in CORBA may change in future with "pass by value" parameters to methods

IDL is a language that allows the programmer to specify the interfaces of a distributed object system. The syntax is similar to C++, but does not include any implementation-level constructs. So it allows definitions of data types (such as structures and unions), constants, enumerated types, exceptions and interfaces. Within interfaces it allows declaration of attributes and operations (methods). The complete IDL specification can be found on the Object Management Group (OMG) site http://www.omg.org.

The book "Java Programming with CORBA" by Andreas Vogel and Keith Duddy (http://www.wiley.com/compbooks/vogel) contains an example of a room booking service specified in CORBA IDL and implemented in Java. This defines interfaces for `Meeting`, a factory to produce them, `MeetingFactory` and a `Room`. A room may have a number of meetings in slots (hourly throughout the day), and there are support constants, enumerations and typedefs to support this. In addition, exceptions may be

thrown under various error conditions. The IDL that follows differs slightly from that given in the book, in that definitions of some data types that occur within interfaces have been "lifted" to a more global level, since the mapping from IDL to Java has changed slightly for elements nested within interfaces since that book was written.

```
module corba {
module RoomBooking {

    interface Meeting {

        // A meeting has two read-only attributes which describes
        // the purpose and the participants of that meeting.

        readonly attribute string purpose;
        readonly attribute string participants;

        oneway void destroy();
    };

    interface MeetingFactory {

        // A meeting factory creates meeting objects.

        Meeting CreateMeeting( in string purpose, in string participants);
    };

    // Meetings can be held between the usual business hours.
    // For the sake of simplity there are 8 slots at which meetings
    // can take place.

    enum Slot { am9, am10, am11, pm12, pm1, pm2, pm3, pm4 };

    // since IDL does not provide means to determine the cardinality
    // of an enum, a corresponding constant MaxSlots is defined.

    const short MaxSlots = 8;

    exception NoMeetingInThisSlot {};
    exception SlotAlreadyTaken {};

    interface Room {

        // A Room provides operations to view, make and cancel bookings.
        // Making a booking means associating a meeting with a time-slot
        // (for this particular room).

        // Meetings associates all meetings (of a day) with time slots
        // for a room.

        typedef Meeting Meetings[ MaxSlots ];

        // The attribute name names a room.

        readonly attribute string name;
```



```
        // View returns the bookings of a room.
        // For simplicity, the implementation handles only bookings
        // for one day.

        Meetings View();

        void Book( in Slot a_slot, in Meeting  a_meeting )
            raises(SlotAlreadyTaken);

        void Cancel( in Slot  a_slot )
            raises(NoMeetingInThisSlot);
    };
};
};
```

2. CORBA to Java Mapping

CORBA has bindings to a number of languages. The most recent addition is Java, and this binding is still under active development (that is, the core is basically settled, but some parts are still changing). This binding must cover all elements of IDL. A horribly brief summary is:

Module
: A module is translated to a java package. All elements within the module becomes classes or interfaces within the package

Basic types
: Most of the basic types map in a straightforward manner: a CORBA `int` becomes a Java `int`, a CORBA `string` becomes a Java `java.lang.String`, and so on. Some are a little tricky, such as the unsigned types which have no Java equivalent.

Constant
: Constants within a CORBA IDL interface are mapped to constants within the corresponding Java interface. Constants which are "global" have no direct equivalent in Java, and so are mapped to Java interfaces with a single field which is the value.

Enum
: Enumerated types have no direct Java equivalent, and so are mapped into a Java interface with the enumeration as a set of integer constants.

Struct
: A CORBA IDL structure is implemented as a Java class with instance variables for all fields.

Interface
: A CORBA IDL interface translates into a Java interface.

Exception
: A CORBA IDL exception maps to a final Java class.

This mapping does not conform to naming conventions such as those established for Java Beans. For example, the IDL declaration `readonly string purpose` becomes the Java accessor method `String purpose()` rather than `String getPurpose()`. Where Java code is generated, the generated names will be used. But in "ordinary" methods, I shall use the more accepted naming forms.

3. Jini Proxies

A Jini service exports a proxy object that acts within the client, on behalf of the service. The proxy may be "fat" or "thin", depending on circumstances. In the chapter on MindStorms, the proxy has to be thin, since all it does is pass on requests to the real service, which is linked to the hardware device. The service cannot move since it has to talk to a particular serial port, so the proxy is thin. (The proxy may have an extensive user interface, but the Jini community seesm to feel that any user interface should be in `Entry` objects rather than in the proxy itself.) Proxy objects created as RMI proxies are similarly thin, just passing on method calls to the service.

CORBA services can be delivered to any accessible client. Each service is limited to the server on which it is running, so they are essentially immobile. They can be *found* by a variety of methods, such as a CORBA naming or trading service. These search methods can be run by any client, anywhere. A search will return a reference to a remote object, which is essentially a thin proxy to the CORBA service. Similarly, if a CORBA method call creates and returns an object, then it will return a remote reference to that object which will continue to exist on the server where it was created. (The new CORBA standards will allow objects to be returned by *value*. This is not yet commonplace, and will probably be restricted to a few languages such as C++ and Java.)

The simplest way to make a CORBA object available to a Jini federation is to build a Jini *service* that is at the same time a CORBA *client*. The service acts as a *bridge* between the two protocols. Really, this is just the same as MindStorms - anything that requires talking a different protocol (hardware or software) will require a bridge between the two.

Most CORBA implementations use IIOP - that is, they are TCP based. The current Jini implementation is also TCP based. So there is a confluence of transport methods, which normally would not occur. A bridge would usually be fixed to a particular piece of hardware, but here it is not necessary. A Jini proxy need not report back to its server since it can negotiate directly with the CORBA service over the TCP network. In many instances it can be a "fat" proxy. This is similar in concept to the fat proxy in the chapter on RMI: there the fat proxy used HTTP to talk to a service within another protocol, but this can also be done with the CORBA protocols. A proxy may need to be thin in some cases such as when it is used by an applet: security restrictions may mean that it cannot talk to any host but only to the one the applet came from. In such a case it would have to perform all Corba client activities back on its 'home' server.

4. Simple Example

The CORBA IDL tutorial at http://java.sun.com/products/jdk/1.2/docs/guide/idl/ gives a "hello world" example. This is defined by the IDL

```
module corba {
    module HelloApp {
        interface Hello {
            string sayHello();
        };
    };
};
```

This can be compiled into Java using a compiler such as Sun's `idltojava` (or other Corba 2.2 compliant compiler). This results in a package `corba.HelloApp` containing a number of classes and interfaces. `Hello` is an interface which is used by a CORBA client (in Java).


```
/*
 * File: ./corba/HelloApp/Hello.java
 * From: Hello.idl
 * Date: Tue Aug 24 11:30:21 1999
 *   By: idltojava Java IDL 1.2 Aug 11 1998 02:00:18
 */

package corba.HelloApp;
public interface Hello
    extends org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity {
    String sayHello()
;
}
```

4.1 Corba Server in Java

A server for the hello IDL can be written in any language with a CORBA binding, such as C++. Rather than get diverted into other languages, though, we shall stick to a Java implementation. However, this language choice is not forced on us by CORBA.

The server must create an object that implements the `Hello` interface. This is done by creating a *servant* that inherits from the `_HelloImplBase`, and then registering it with the CORBA ORB (Object Request Broker - the CORBA "backplane"). The server must also find a name server and register the name and the implementation. The servant implements the `Hello` interface. The server can just sleep to continue existence after registering the servant.

```
/**
 * CorbaHelloServer.java
 *
 *
 * Created: Thu Aug 19 22:37:39 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */
package corba;

import corba.HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class CorbaHelloServer {

    public CorbaHelloServer() {

    }

    public static void main(String[] args) {
        try {
            // create a Hello implementation object
            ORB orb = ORB.init(args, null);
```

```
            HelloServant helloRef = new HelloServant();
            orb.connect(helloRef);

            // find the name server
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);

            // bind the Hello service to the name server
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            ncRef.rebind(path, helloRef);

            // sleep
            java.lang.Object sleep = new java.lang.Object();
            synchronized(sleep) {
                sleep.wait();
            }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

} // CorbaHelloServer

class HelloServant extends _HelloImplBase {
    public String sayHello() {
        return("hello world");
    }
}
```

4.2 Corba Client in Java

A standalone client finds a proxy implementing the `Hello` interface by methods such as looking up a CORBA name server. The name server returns a `org.omg.CORBA.Object` which is cast to the interface type by the `HelloHelper` method `narrow()` (the Java cast method is not used). This proxy object can then be used to call methods back in the CORBA server

```
/**
 * CorbaHelloClient.java
 *
 *
 * Created: Thu Aug 19 22:28:44 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */
package corba;

import corba.HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
```

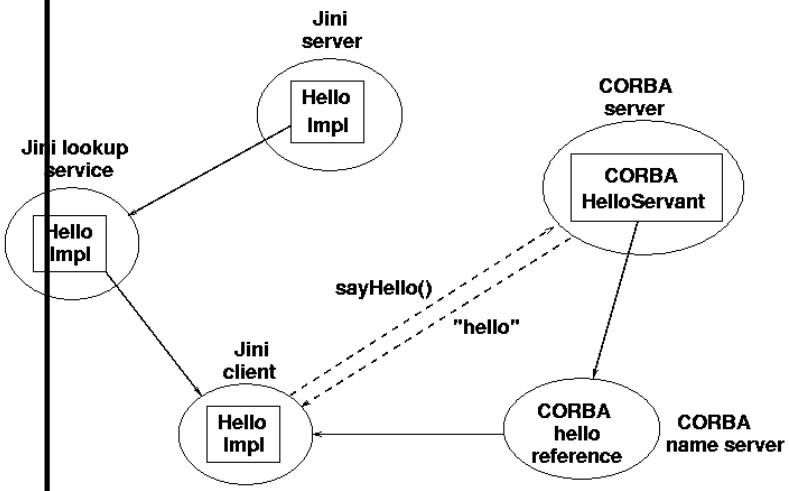


```
public class CorbaHelloClient {

    public CorbaHelloClient() {

    }

    public static void main(String[] args) {
        try {
            ORB orb = ORB.init(args, null);

            // find the name server
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);

            // find the Hello proxy
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            org.omg.CORBA.Object obj = ncRef.resolve(path);
            Hello helloRef = HelloHelper.narrow(obj);

            // now invoke methods on the CORBA proxy
            String hello = helloRef.sayHello();
            System.out.println(hello);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

} // CorbaHelloClient
```

4.3 Jini Service

In order to make the CORBA object accessible to the Jini world, it must be turned into a Jini service. At the same time it must remain in a CORBA server, so that it can be used by ordinary CORBA clients. So we do nothing to the CORBA server. Instead, we need to build a Jini service that will act as a CORBA *client*. This service will then be able to deliver the CORBA service to Jini clients.

The Jini service can be implemented as a fat proxy delivered to a Jini client and performing most if its activity within this Jini client. The processes that run in this, with their associated Jini and CORBA objects, look like:



The Java interface for this service is quite simple, and basically just copies the interface for the CORBA service, as

```
/**
 * JavaHello.java
 *
 *
 * Created: Tue Aug 24 11:12:36 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */
package corba;

import java.io.Serializable;

public interface JavaHello extends Serializable {

    public String sayHello();
} // JavaHello
```

The method `sayHello()` for the CORBA IDL returns a `string`. In the Java binding this becomes an ordinary Java `String`, and the Jini service can just use this type. The next example will show a more complex case where CORBA objects may get returned. Note that as a "fat" service, any implementation will get moved across to a Jini client and will run there. So the service only needs to implement `Serializable`, and its methods do not need to throw `Remote` exceptions since they will run locally in the client.

The implementation of this Jini interface will basically act as a CORBA client. Its `sayHello()` method will contact the CORBA naming service, find a reference to the CORBA `Hello` object and call *its* `sayHello()` method. The Jini service can just return the string it gets from the CORBA service.

```
/**
 * JavaHelloImpl.java
 *
 *
 * Created: Tue Aug 24 11:14:00 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */
package corba;

import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import corba.HelloApp.*;

public class JavaHelloImpl implements JavaHello {

    protected Hello helloRef = null;
    protected String[] argv;

    public JavaHelloImpl(String[] argv) {
        this.argv = argv;
    }

    public String sayHello() {
        // Hello helloRef = null;

        if (helloRef == null) {
            helloRef = getHelloRef();
        }
        // now invoke methods on the CORBA proxy
        String hello = helloRef.sayHello();
        return hello;
    }

    protected Hello getHelloRef() {
        ORB orb = null;
        // Act like a CORBA client
        try {
            orb = ORB.init(argv, null);

            // find the CORBA name server
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);

            // find the CORBA Hello proxy
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            org.omg.CORBA.Object obj = ncRef.resolve(path);
```

```
            Hello helloRef = HelloHelper.narrow(obj);
            return helloRef;
        } catch(Exception e) {
            e.printStackTrace();
            return null;
        }
    }
} // JavaHelloImpl
```

4.4 Jini Server and Client

The Jini server which exports the service doesn't contain anything new

```
package corba;


// import com.sun.jini.lookup.JoinManager;
import net.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
// import com.sun.jini.lookup.ServiceIDListener;
// import com.sun.jini.lease.LeaseRenewalManager;
import net.jini.lookup.ServiceIDListener;
import net.jini.lease.LeaseRenewalManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.LookupDiscoveryManager;

/**
 * JavaHelloServer.java
 *
 *
 * Created: Wed Mar 17 14:23:44 1999
 *
 * @author Jan Newmarch
 * @version 1.1
 *   uses Jini 1.1 JoinManager
 */

public class JavaHelloServer implements ServiceIDListener {

    public static void main(String argv[]) {
        new JavaHelloServer(argv);

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(1000000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }

    }

    public JavaHelloServer(String[] argv) {
```



```
        JoinManager joinMgr = null;
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            joinMgr = new JoinManager(new JavaHelloImpl(argv),
                                      null,
                                      this,
                                      mgr,
                                      new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void serviceIDNotify(ServiceID serviceID) {
        System.out.println("got service ID " + serviceID.toString());
    }

} // JavaHelloServer
```

Similarly, the Jini client doesn't contain anything new

```
package client;

import corba.JavaHello;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;


/**
 * TestCorbaHello.java
 *
 *
 * Created: Tue Aug 24
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class TestCorbaHello implements DiscoveryListener {

    public static void main(String argv[]) {
        new TestCorbaHello();
```

```
        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(10000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestCorbaHello() {
        System.setSecurityManager(new RMISecurityManager());

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);

    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Service found");
            ServiceRegistrar registrar = registrars[n];

            new LookupThread(registrar).start();
        }

            // System.exit(0);
    }

    public void discarded(DiscoveryEvent evt) {
        // empty
    }

    class LookupThread extends Thread {

        ServiceRegistrar registrar;

        LookupThread(ServiceRegistrar registrar) {
            this.registrar = registrar;
        }


        public void run() {

            Class[] classes = new Class[] {JavaHello.class};
            JavaHello hello = null;
```



```
            ServiceTemplate template = new ServiceTemplate(null, classes,
                                                           null);

            try {
                hello = (JavaHello) registrar.lookup(template);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(2);
            }
            if (hello == null) {
                System.out.println("hello null");
                return;
            }
            String msg;
            try {
                msg = hello.sayHello();
                System.out.println(msg);
            } catch(Exception e) {
                // we may get a CORBA runtime error
                System.err.println(e.toString());
            }
        }
    }

} // TestCorbaHello
```

4.5 Building the Example

The major additional step is to build the Java classes from the IDL specification. There are a number of CORBA IDL to Java compilers. One of these is the Sun compiler `idltojava` which is available from `java.sun.com`. This (or another compiler) needs to be run on the IDL file to produce the Java files in the package `corba.HelloApp`. These are standard Java files, and can be compiled using your normal Java compiler. They may need some CORBA files in the classpath if required by your vendor's implementation of CORBA. Files produced by `idltojava` do not need any extra classes.

The Jini server, service and client are also normal Java files, and can be compiled like earlier Jini files, with the classpath set to include the Jini libraries.

4.6 Running the Example

There are a large number of elements and processes that must be set running to get this running satisfactorily

1. A CORBA name server must be set running. In the JDK 1.2 distribution is a server, `tnameserv`. By default this runs on TCP port 900. Under Unix access to this port is restricted to system supervisors. It can be set running on this port by a supervisor, or started during boot time. An ordinary user will need to use the option `-ORBInitialPort port` to run it on a port above 1024. For example,

   ```
   tnameserv -ORBInitialPort 1055
   ```

   All CORBA services and clients should also use this port number
2. The Java version of the CORBA service can then be started by

```
java corba.CorbaHelloServer -ORBInitialPort 1055
```

3. Typical Jini support services will need to be running, such as a Jini lookup service, the RMI daemon `rmid`, and HTTP servers to move class definitions around
4. The Jini service can be started by

```
java corba.JavaHelloServer -ORBInitialPort 1055
```

5. Finally, the Jini client can be run by

```
java client.TestCorbaHello -ORBInitialPort 1055
```

5. Room Booking Example

The IDL for a room booking problem was briefly discussed in an earlier section. This has a few more complexities over the previous example. The problem is to have a set of rooms, and for each room a set of bookings that can be made on that room. The bookings may be made on the hour, from 9am till 4pm (this only covers the bookings for one day). Bookings once made may be cancelled. A room can be queried for the set of bookings it has: it returns an array of meetings, which are `null` if no booking has been made, or a non-null meeting which has details of the participants and the purpose of the meeting.

The additional features of this example are

1. Each room is implemented as a separate CORBA object. There is also a ''meeting factory'' which produces more objects. This is a system with multiple CORBA objects residing on many CORBA servers. There are several possibilities for dealing with multiple objects
2. Some of the methods return CORBA objects. These may need to be exposed to clients. This is not a problem if the client is a CORBA client, but here we will have Jini clients
3. Some of the methods throw user-defined exceptions, in addition to CORBA-defined exceptions. Both of these need to be handled appropriately

5.1 CORBA Objects

CORBA defines a set of ''primitive'' types such as integers of various sizes, chars, *etc*. The language bindings specify the primitive types they turn into for each language. For example, the CORBA wide character `wchar` becomes a Java Unicode `char`. Things are different for non-primitive objects, though.

The IDL for the room booking defines CORBA interfaces for `Meeting`, `MeetingFactory` and `Room`. The implementation of these can be in any suitable language, and need not be in Java. The Java binding will convert these into Java interfaces. A CORBA client written in Java will get objects that implement these interfaces, but these will essentially be just references to remote CORBA objects. Two things are certain about these references:

1. CORBA interfaces generate Java interfaces such as `Hello`. These inherit from `org.omg.CORBA.portable.IDLEntity`, which implements `Serializable`. So the references can be moved around like Jini objects. But they lose their link to the CORBA ORB that created them, and may end up in a different ''namespace'' where the reference makes no sense. So CORBA references cannot be usefully moved around. The closest one can come at present is to convert them to ''string-ified'' form and move that around. This may change when CORBA pass-by-value objects become common. Note that the serialization method that gives a string representation of a CORBA object is not the same as the Java one: the CORBA method serializes the remote reference, the Java method serializes the object's instance data
2. They do not subclass from `UnicastRemoteObject` or `Activatable`. The Java runtime will not use an

RMI stub for them

If a Jini client gets local references to these objects, and keeps them local, then it can use them *via* their Java interfaces. If they need to be moved around the network, then appropriate ''mobile'' classes will need to be defined and the information copied across to them from the local objects. For example, the CORBA `Meeting` interface generates the Java interface

```
/*
 * File: ./corba/RoomBooking/Meeting.java
 * From: RoomBooking.idl
 * Date: Wed Aug 25 11:30:25 1999
 *   By: idltojava Java IDL 1.2 Aug 11 1998 02:00:18
 */

package corba.RoomBooking;
public interface Meeting
    extends org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity {
    String purpose();
    String participants();
    void destroy()
;
}
```

To make the *information* from a CORBA `Meeting` available as a mobile Jini object, we would need an interface

```
/**
 * JavaMeeting.java
 *
 *
 * Created: Tue Aug 24 20:00:31 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */
package corba.common;

import java.io.Serializable;
import org.omg.CORBA.*;
import corba.RoomBooking.*;
import java.rmi.RemoteException;

public interface JavaMeeting extends Serializable {
    String getPurpose();
    String getParticipants();
    Meeting getMeeting(ORB orb);
} // JavaMeeting
```

The first two methods allow information about a meeting to be accessible to applications that do not want to contact the CORBA service, and the third allows a CORBA object reference to be reconstructed within a new ORB. A suitable implementation is

```
/**
 * JavaMeetingImpl.java
 *
 *
 * Created: Thu Aug 26 15:33:24 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */
package corba.RoomBookingImpl;

import corba.RoomBooking.*;
import org.omg.CORBA.*;
import corba.common.*;

/**
 * A portable Java object representing a CORBA object.
 * It carries a stringified version of the CORBA reference
 * for reconstruction on JVM's which have a CORBA orb.
 * It exposes the attributes of the CORBA object to non-CORBA
 * applications by copying the attribute values on construction.
 */
public class JavaMeetingImpl implements JavaMeeting {
    protected String purpose;
    protected String participants;
    protected String corbaObj;

    /**
     * get the purpose of a meeting for a Java client
     * unaware of CORBA
     */
    public String getPurpose() {
        return purpose;
    }

    /**
     * get the participants of a meeting for a Java client
     * unaware of CORBA
     */
    public String getParticipants() {
        return participants;
    }

    /**
     * reconstruct a meeting using a CORBA orb in the target JVM
     */
    public Meeting getMeeting(ORB orb) {
        org.omg.CORBA.Object obj = orb.string_to_object(corbaObj);
        Meeting m = MeetingHelper.narrow(obj);
        return m;
    }

    /**
```



```
     * construct a portable Java representation of the CORBA
     * Meeting, using the CORBA orb on the source JVM
     */
    public JavaMeetingImpl(Meeting m, ORB orb) {
        purpose = m.purpose();
        participants = m.participants();
        corbaObj = orb.object_to_string(m);
    }

} // JavaMeetingImpl
```

might be appropriate.

5.2 Multiple Objects

The implementation of the room booking problem in the Vogel and Duddy book runs each room as a separate CORBA object, each with its own server. A meeting factory creates meeting objects which are kept within the factory server, and passed around by reference. So for a distributed application with ten rooms, there will be eleven CORBA servers running.

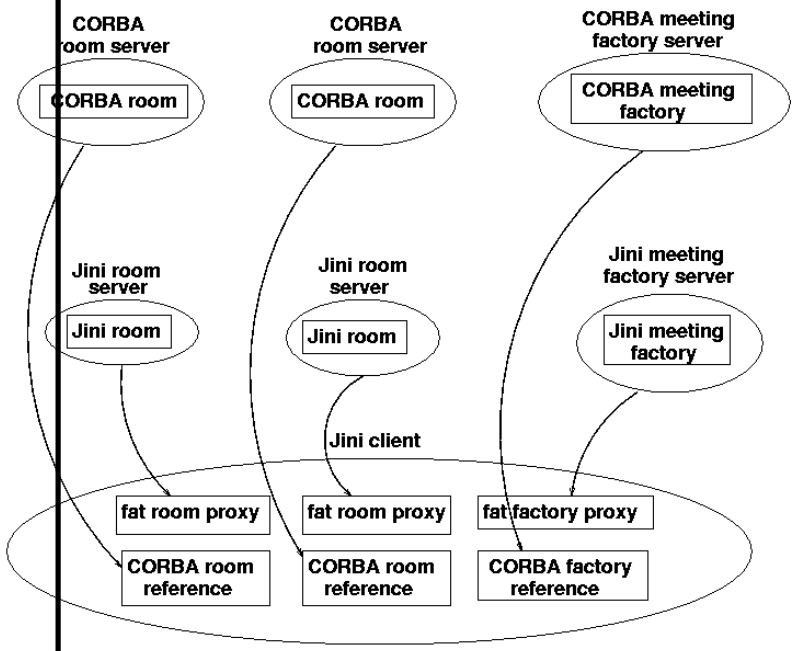
There are several possible ways of bringing this set of objects into the Jini world so that they are accessible to a Jini client, including

1. A Jini server may exist for each CORBA server
    1. Each Jini server may export fat proxies, which build CORBA references in the same Jini client
    2. Each Jini server may export a thin proxy, with a CORBA reference held in each of these servers
2. A single Jini server may be built for the federation of CORBA objects
    1. The single Jini server exports a fat proxy, which builds CORBA references in the Jini client
    2. The single Jini server exports a thin proxy, with all CORBA references within this single server

5.2.1 Many fat proxies

We can have one Jini server for each of the CORBA servers. The Jini servers can be running on the same machines as the CORBA ones, but there is no necessity to do so. Applet security restrictions, for example, may mean that all the Jini servers are running on a single machine, the same one as an applet's HTTP server.

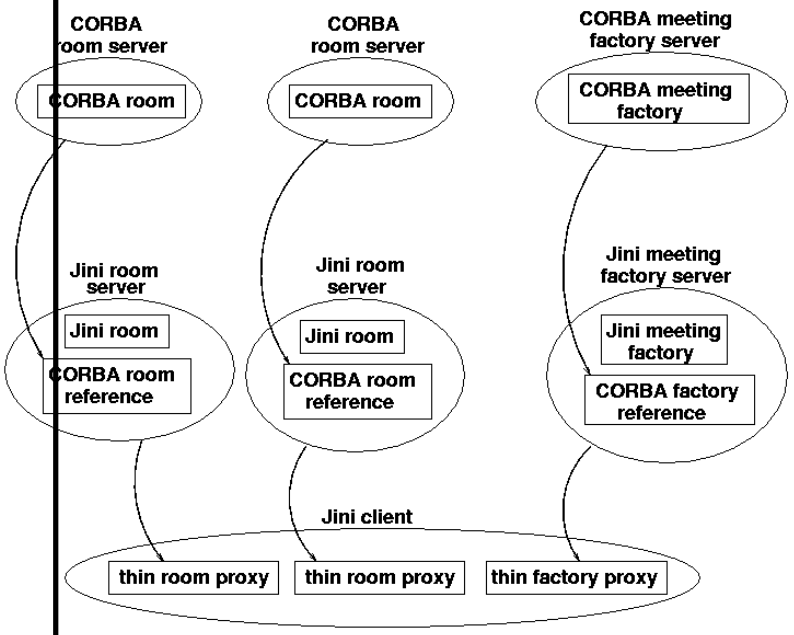
The Jini proxy objects exported by each Jini server may be fat ones, which connect directly to the CORBA server. Thus each proxy becomes a CORBA client, as was dealt with in the "hello world" example. Within the Jini client, we do not just have one proxy, but many proxies. Because they are all running within the same address space, they can share CORBA references - there is no need to package a CORBA reference as a portable Jini object. In addition, the Jini client can just use all of these CORBA references directly, as instance objects of interfaces. Diagramatically, this appears as



The CORBA servers are all accessed from within the Jini client. This may rule out this arrangement if the client is an applet, and the servers are on different machines.

5.2.2 Many thin proxies

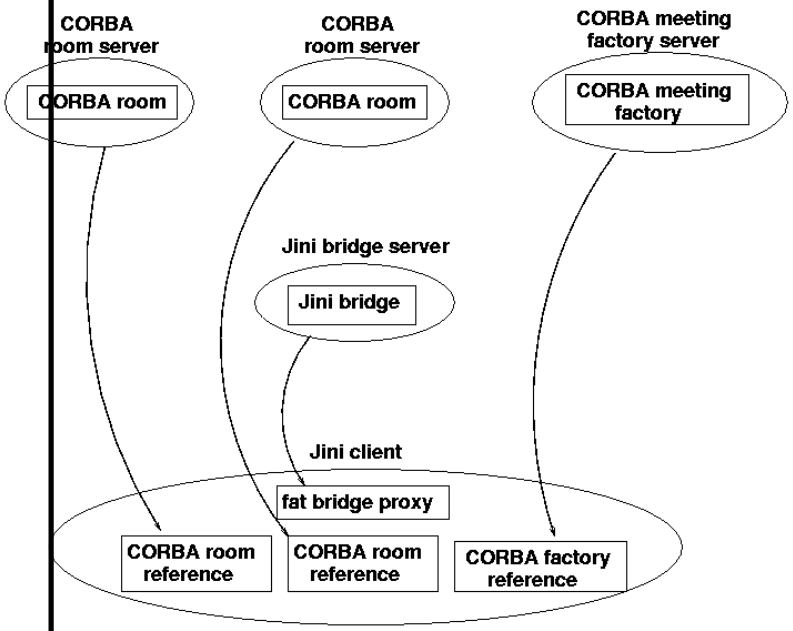
The proxies exported can be thin, such as RMI stubs. In this case each Jini server is acting as a CORBA client.



If all the Jini servers are collocated on the same machine, then this becomes a possible architecture suitable for applets. The downside of this approach is that all the CORBA references are within different JVMs. In order to move the reference for a meeting from the Jini meeting factory to one of the Jini rooms, it may be necessary to wrap it in a portable Jini object, as discussed above. The Jini client will also need to get information about the CORBA objects, which can be gained from these portable Jini objects.

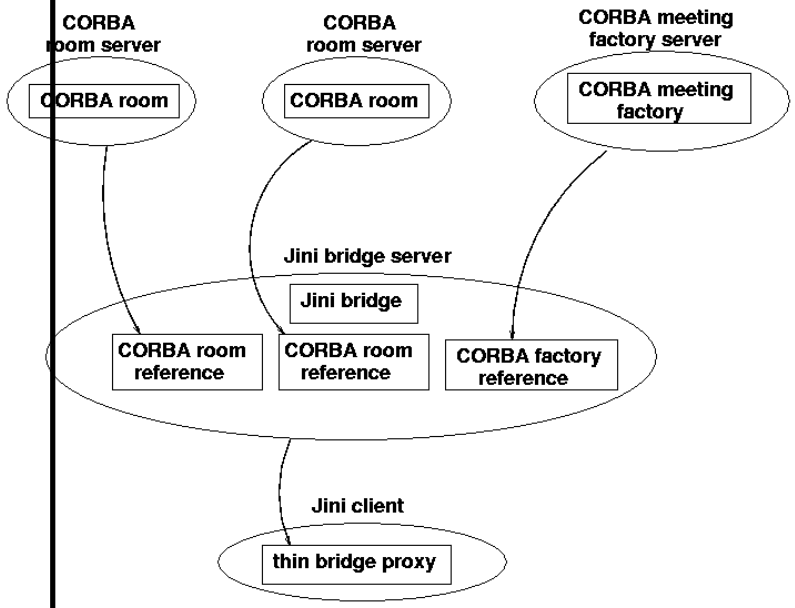
5.2.3 Single fat proxy

An alternative to Jini servers for each CORBA server is to have a single Jini server into the CORBA federation. This can be a feasible alternative when the set of CORBA objects form a complete system or module, and it makes sense to treat them as a unit. There are then the choices again of where to locate the CORBA references, either in the Jini server or in a proxy. Placing them in a fat proxy looks like



5.2.4 Single thin proxy

Placing all the CORBA references on the server side of a Jini service means that a Jini client only needs to make one network connection to the service. This is probably the best option from a security viewpoint of a Jini client.



5.3 Exceptions

CORBA methods can throw exceptions of two types: system exceptions and user exceptions. System exceptions subclass from `RuntimeException`, and so are *unchecked*. These do not need to have explicit `try...catch` clauses around them. If an exception is thrown, it will be caught by the Java runtime, and generally halt the process with an error message. This would result in a CORBA client dying, which would generally be undesirable. Many of these system exceptions will be caused by the distributed nature of CORBA objects, and probably should be caught explicitly. If they cannot be handled directly, then to bring them into line with the Jini world, they can be wrapped as ''nested exceptions'' within a `Remote` exception and thrown again.

User exceptions are declared in the IDL for the CORBA interfaces and methods. These exceptions are *checked*, and need to be explicitly caught (or re-thrown) by Java methods. If a user exception is thrown, this will be because of some semantic error within one of the objects, and will be unrelated to any networking or remote issues. They should be treated as they are, without wrapping them in `Remote` exceptions.

5.4 Interfaces for Single Thin Proxy

This and the following sections consider issues in building a single thin proxy for a federation of CORBA objects. The Vogel and Duddy book gives a CORBA client to interact with the CORBA federation, and this is used as the basis for the Jini services and clients.

Using a thin proxy means that all CORBA-related calls will be placed in the service object, and made available to Jini clients only by means of portable Jini versions of the CORBA objects. These portable objects are defined by the interfaces

```
/**
 * JavaRoom.java
 *
 *
 * Created: Tue Aug 24 20:00:31 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */
package corba.common;

import corba.RoomBooking.*;
import java.io.Serializable;
import org.omg.CORBA.*;
import java.rmi.RemoteException;

public interface JavaRoom extends Serializable {
    String getName();
    Room getRoom(ORB orb);
} // JavaRoom
```

and

```
/**
 * JavaMeeting.java
 *
 *
 * Created: Tue Aug 24 20:00:31 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */
package corba.common;

import java.io.Serializable;
import org.omg.CORBA.*;
import corba.RoomBooking.*;
import java.rmi.RemoteException;

public interface JavaMeeting extends Serializable {
    String getPurpose();
    String getParticipants();
    Meeting getMeeting(ORB orb);
} // JavaMeeting
```


The bridge interface between the CORBA federation and the Jini clients has to provide methods for making changes to objects within the CORBA federation, and to obtain information from them. For the room booking system this requires the ability to book and cancel meetings within rooms, and also the ability to view the current state of the system. Viewing is accomplished by three methods: updating the current state, and getting the list of rooms and of bookings for a room.

```
/**
 * RoomBookingBridge.java
 *
 *
 * Created: Sun Aug 29 11:11:46 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package corba.common;

import java.rmi.RemoteException;
import corba.RoomBooking.*;
import org.omg.CORBA.*;

public interface RoomBookingBridge extends java.io.Serializable {

    public void cancel(int selected_room, int selected_slot)
        throws RemoteException, NoMeetingInThisSlot;
    public void book(String purpose, String participants,
                     int selected_room, int selected_slot)
        throws RemoteException, SlotAlreadyTaken;
    public void update()
        throws RemoteException, UserException;
    public JavaRoom[] getRooms()
        throws RemoteException;
    public JavaMeeting[] getMeetings(int room_index)
        throws RemoteException;
} // RoomBookingBridge
```

There is a slight legacy in this that comes from the original ‘‘monoblock’’ CORBA client. In there, because the GUI interface elements and the CORBA references were all in the one client, simple shareable structures such as arrays of rooms and arrays of meetings were used. Meetings and rooms could be indentified simply by their index into the appropriate array. In splitting the client apart into multiple (and remote) classes this is not really a good idea anymore as it assumes a commonality of implementation across objects which may not occur. It doesn’t seem worthwhile being too fussy about that here, though.

5.5 RoomBookingBridge Implementation

The room booking Jini bridge has to perform all CORBA activities, and to wrap these up as portable Jini objects. A major part of this is locating the CORBA services, which here are the meeting factory and all of the rooms. We do not want to get too involved in these issues here. The meeting factory can be found in essentially the same way as the hello server was earlier, by looking up its name. Finding the rooms is harder as these are not known in advance. Essentially, the equivalent of a ‘‘directory’’ has to be set up on the name server, which is known as a ‘‘naming context’’. Rooms are registered within this naming context by their servers, and the client

gets this context and then does a search for its contents. The Jini component of this object is that it subclasses from `UnicastRemoteObject`, and implements a `RemoteRoomBookingBridge` which is a remote version of `RoomBookingBridge`. It is also worthwhile noting how CORBA exceptions are caught and wrapped in `Remote` exceptions.

```
/**
 * RoomBookingBridgeImpl.java
 *
 *
 * Created: Wed Aug 25 22:54:38 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */
package corba.RoomBookingImpl;

import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import corba.RoomBooking.*;
import corba.common.*;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class RoomBookingBridgeImpl extends UnicastRemoteObject implements RemoteRoomBookingBridge {

    private MeetingFactory meeting_factory;
    private Room[] rooms;
    private Meeting[] meetings;
    private ORB orb;
    private NamingContext room_context;

    public RoomBookingBridgeImpl(String[] args)
        throws RemoteException, UserException {
        try {
            // initialise the ORB
            orb = ORB.init(args, null);
        }
        catch(SystemException system_exception ) {
            throw new RemoteException("constructor RoomBookingBridge: ",
                                      system_exception);
        }
        init_from_ns();
        update();
    }

    public void init_from_ns()
        throws RemoteException, UserException {

        // initialise from Naming Service
        try {
            // get room context
            String str_name = "/BuildingApplications/Rooms/";
            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent(str_name,  " ");
            NameComponent path[] = {nc};

            org.omg.CORBA.Object roomRef = ncRef.resolve(path);
            room_context = NamingContextHelper.narrow(roomRef);
            if( room_context == null ) {
                System.err.println( "Room context is null," );
                System.err.println( "exiting ..." );
```



```
                System.exit( 1 );
            }

            // get MeetingFactory from Naming Service
            str_name = "/BuildingApplications/MeetingFactories/MeetingFactory";
            nc = new NameComponent(str_name, " ");
            path[0] = nc;
            meeting_factory = MeetingFactoryHelper.narrow(ncRef.resolve(path));
            if( meeting_factory == null ) {
                System.err.println(
                    "No Meeting Factory registred at Naming Service" );
                System.err.println( "exiting ..." );
                System.exit( 1 );
            }
        }
        catch(SystemException system_exception ) {
            throw new RemoteException("Initialise ORB", system_exception);
        }
    }

    public void update()
        throws RemoteException, UserException {

        try {
            // list rooms
            // initialise binding list and binding iterator
            // Holder objects for out parameter
            BindingListHolder blHolder = new BindingListHolder();
            BindingIteratorHolder biHolder = new BindingIteratorHolder();
            BindingHolder bHolder = new BindingHolder();
            Vector roomVector = new Vector();
            Room aRoom;

            // we are 2 rooms via the room list
            // more rooms are available from the binding iterator
            room_context.list( 2, blHolder, biHolder );

            // get rooms from Room context of the Naming Service
            // and put them into the roomVector
            for(int i = 0; i < blHolder.value.length; i++ ) {
                aRoom = RoomHelper.narrow(
                    room_context.resolve( blHolder.value[i].binding_name ));
                roomVector.addElement( aRoom );
            }

            // get remaining rooms from the iterator
            if( biHolder.value != null ) {
                while( biHolder.value.next_one( bHolder ) ) {
                    aRoom = RoomHelper.narrow(
                        room_context.resolve( bHolder.value.binding_name ) );
                    if( aRoom != null ) {
                        roomVector.addElement( aRoom );
                    }
                }
            }

            // convert the roomVector into a room array
            rooms = new Room[ roomVector.size() ];
            roomVector.copyInto( rooms );

            // be fiendly with system resources
            if(  biHolder.value != null )
                biHolder.value.destroy();
        }
```

```
        catch(SystemException system_exception) {
            throw new RemoteException("View", system_exception);
            // System.err.println("View: " + system_exception);
        }
    }

    public void cancel(int selected_room, int selected_slot)
        throws RemoteException, NoMeetingInThisSlot {
        try {
            rooms[selected_room].Cancel(
                Slot.from_int(selected_slot) );
            System.out.println("Cancel called" );
        }
        catch(SystemException system_exception) {
            throw new RemoteException("Cancel", system_exception);
        }
    }

    public void book(String purpose, String participants,
                     int selected_room, int selected_slot)
        throws RemoteException, SlotAlreadyTaken {
        try {
            Meeting meeting =
                meeting_factory.CreateMeeting(purpose, participants);
            System.out.println( "meeting created" );
            String p = meeting.purpose();
            System.out.println("Purpose: "+p);
            rooms[selected_room].Book(
                Slot.from_int(selected_slot), meeting );
            System.out.println( "room is booked" );
        }
        catch(SystemException system_exception ) {
            throw new RemoteException("Booking system exception", system_exception);
        }
    }

    /**
     * return a list of the rooms as portable JavaRooms
     */
    public JavaRoom[] getRooms() {
        int len = rooms.length;
        JavaRoom[] jrooms = new JavaRoom[len];
        for (int n = 0; n < len; n++) {
            jrooms[n] = new JavaRoomImpl(rooms[n]);
        }
        return jrooms;
    }

    public JavaMeeting[] getMeetings(int room_index) {
        Meeting[] meetings = rooms[room_index].View();
        int len = meetings.length;
        JavaMeeting[] jmeetings = new JavaMeeting[len];
        for (int n = 0; n < len; n++) {
            if (meetings[n] == null) {
                jmeetings[n] = null;
            } else {
                jmeetings[n] = new JavaMeetingImpl(meetings[n], orb);
            }
        }
        return jmeetings;
    }
} // RoomBookingBridgeImpl
```


5.6 Other Classes

The Java classes and servers implementing the CORBA objects are mainly unchanged from the implementations given in the Vogel and Duddy book. They can continue to act as CORBA servers to the original clients. I replaced the "easy naming" naming service by a later one with the slightly more complex standard mechanism for creating contexts and placing names within this context. This can use the `tnameserv` CORBA naming server, for example.

The Vogel and Duddy room booking client was messed around a little bit, but its essential structure remained unchanged. The GUI elements, for example, were not altered. All CORBA-related code was hived off into the bridge classes.

The Vogel and Duddy code samples can all be downloaded from a public Web site, and come with no author attribution or copyright claim. The client is also quite lengthy since it has plenty of GUI inside. So I won't complete the code listing here. The code for all my classes, and the modified code of the Vogel and Duddy classes, is given in the subdirectory `corba` of the `programs.zip` file.

5.7 Building the Example

The IDL interface `RoomBooking.idl` needs to have an IDL to Java run on it. This produces classes in the package `corba.RoomBooking`. These can all be compiled using the standard Java classes and any CORBA classes needed.

The Jini server, service and client are also normal Java files, and can be compiled like earlier Jini files, with the classpath set to include the Jini libraries.

5.8 Running the Example

There are a large number of elements and processes that must be set running to get this running satisfactorily

1. A CORBA name server must be set running, as in the earlier problem. For example,

   ```
   tnameserv -ORBInitialPort 1055
   ```

   All CORBA services and clients should also use this port number
2. A CORBA server should be started for each room, with the first parameter being the "name" of the room, as in

   ```
   java corba.RoomBookingImpl.RoomServer "freds room" -ORBInitialPort 1055
   ```

3. A CORBA server should be started for the meeting factory

   ```
   java corba.RoomBookingImpl.MeetingFactoryServer -ORBInitialPort 1055
   ```

4. Typical Jini support services will need to be running, such as a lookup service, the RMI daemon `rmid`, and HTTP servers to move class definitions around
5. The Jini service can be started by

   ```
   java corba.RoomBookingImpl.RoomBookingBridgeServer -ORBInitialPort 1055
   ```

6. Finally, the Jini client can be run by

```
java corba.RoomBookingImpl.RoomBookingClientApplication -ORBInitialPort 1055
```

6. Migrating a CORBA Client to Jini

Both of the examples in this chapter started life as pure CORBA systems written by other authors, with CORBA objects delivered by servers to a CORBA client. The clients were both migrated to Jini clients of a Jini service acting as a front-end to CORBA objects, in a series of steps. For those in a similar situation, it may be worthwhile to spell out the steps I went through in doing this for the room booking problem.

1. The original client was a single ''monoblock'' client, mixing GUI elements, CORBA calls and glue to hold it all together. The first step was to decide on the architectural constraint: one Jini service, or many.
2. A single Jini service was chosen (for no other reason than it looked to offer more complexities). This implied that all CORBA related calls had to be collected into a single object, the `RoomBookingBridgeImpl`. At this stage, the `RoomBookingBridge` *interface* was not defined: that came after the implementation was completed (ok, I hang my head in shame, but I was trying to adapt existing code rather than starting from scratch). At this time, the client was still running as a pure CORBA client - no Jini mechanisms had been introduced.
3. Once all the CORBA related code was isolated into one class, another architectural decision had to be made, as to whether this was to function as a fat or thin proxy. The decision to make it thin in this case was again based on interest rather than functional reasons.
4. The GUI elements left behind in the client needed to access information from the CORBA objects. In the thin proxy model, this meant that portable Jini objects had to be built to carry information out of the CORBA world. This led to the interfaces such as `JavaRoom` and the implementations as `JavaRoomImpl`. The GUI code in the client had no need to directly modify fields in these objects, so they ended up as ''readonly'' versions of their CORBA sources. (If a fat proxy had been used, this step of creating portable Jini objects would not have been necessary.)
5. The client was modified to use these portable Jini objects, and the `RoomBookingBridgeImpl` was changed to return these from its methods. Again, this was all still done within the CORBA world, and no Jini services were yet involved. This looked like a good time to define the `RoomBookingBridge` interface, when everything had settled down.
6. Finally, the `RoomBookingBridgeImpl` was turned into a `UnicastRemoteObject` and placed into a Jini server. The client was changed to *lookup* a `RoomBookingBridge` service rather than *create* a `RoomBookingBridgeImpl` object.

At the end of this, I had an implementation of a Jini service, with a thin RMI proxy. The CORBA objects and servers had not been changed at all. The original CORBA client had been split into two, with the Jini service implementing all of the CORBA lookups. These were exposed to the client through a set of ''facades'' that gave it the information it needed.

The client was still responsible for all of the GUI aspects, and so was acting as a ''knowledgeable'' client. If needed, these GUI elements could be placed into `Entry` objects, and also exported as part of the service.

7. Jini Service as a CORBA Service

We have looked at making CORBA objects into Jini services. Is it possible to go the other way, and make a Jini service appear as a CORBA object in a CORBA federation? Well, it should be. Just as there is a mapping from CORBA IDL to Java, there is also a mapping of a suitable subset of Java into CORBA IDL. So a Jini service interface can be written as a CORBA interface. A Jini client could then be written as the implementation of a CORBA serve to this IDL.

At present, with a paucity of Jini services, it does not seem worthwhile yet to explore this in detail. This may change in the future, though.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

User Interfaces for Jini Services

Contents

1. User Interfaces as Entries
2. User Interfaces from Factory Objects
3. Objects Returned from getUI()
4. Support Classes
5. Images
6. MindStorms
    1. RCXLoaderFrame
    2. RCXLoaderFrameFactory
    3. Exporting the RCXLoaderFrameFactory
    4. Customised User Interfaces
    5. CarJFrame
    6. CarJFrameFactory
    7. Exporting the CarJFrameFactory
    8. The RCX Client

1. User Interfaces as Entries

Interaction with a service is specified by its interface. This will be the same across all implementations of the interface. This doesn't allow any flexibility, and of course it shouldn't. It is the bottom level for using this type of service. But service implementations do differ, and there is a need to allow for this. The mechanism used in Jini is to put these differences in `Entry` objects. Typical objects supplied by vendors may include `Name` and `ServiceInfo`.

Clients can make use of the interface and these additional entry items, primarily in the selection of a service. But once they have the service, are they just constrained to use it via the type interface? Many services will probably benefit from some sort of user interface. For example, a printer may supply a method to print a file, but it may have the capability to print multiple copies of the same file. Rather than relying on the client to be smart enough to figure this out, the printer vendor may want to call attention to this by supplying a user-interface object with a special component for "number of copies".

The user interface for a service cannot expect to have all details supplied by the client - at best a client could only manage a fairly generic user interface. It should come from the vendor, or maybe even third parties. When your video player becomes Jini enabled, it would be a godsend for *someone* to supply a decent user interface for it, since the video player vendors seem generally incapable of doing so! The `Entry` objects need not just provide static data - as Java objects they are perfectly capable of running as user-interface objects.

User interfaces are not part of the Jini standard. But the Jini community (with a semi-formal organisation as the "Jini Community") is coming towards a standard way of specifying many things, including user-interface standards and guidelines. Guideline number one from the "serviceUI" group is: *user interfaces for a service should be given in `Entry` objects*.

2. User Interfaces from Factory Objects

In the chapter on "More Complex Examples" some discussion was given to the location of code, using user-interface components as examples. The chapter suggested that user interfaces should not be created on the server side but on the client side. So the user-interface should be exported as a *factory* object that can create the user-interface on the client side.

More arguments can be given to support this:

1. A service exported from a low resource computer, such as an embedded Java engine may not have the classes on the service side needed to create the user-interface (it may not have the Swing or even the AWT libraries)
2. There may be many potential user-interfaces for any particular service: the Palm Pilot (with small grey-scale screen) requires a different interface to a high-end workstation with huge screen and enormous numbers of colours. It is not reasonable to expect the service to create every one of these, but it could export a factory capable of doing so
3. Localisation of internationalised services cannot be done on the service side, only on the client side

The service should export a user-interface *factory*, with a method to create the interface, such as `getUI()`. The service and its user-interface factory entry will both be retrieved by the client. The client will then create the user-interface. Note that the factory will not know the service object - if it was given one during *its* construction (on the service side) it would end with another copy of the service. So when it is asked for a user-interface (on the client side), it should be passed the service as well. The actual creation may fail in lots of ways: no suitable libraries, out of memory, screen too small, *etc*. So this method may need to throw an exception.

The factory will probably know many attributes of the user-interface: preferred and minimum screen sizes, required class libraries *etc*. These can all be stored as a set of features of the factory. Two have been singled out as being of general importance: the `name` and the `role` (such as `main` and `admin`). The *current* definition of a user-interface factory is

```
public class UIFactory extends AbstractEntry {
    public String name;
    public String role;
    public Map properties;

    public UIFactory(String name, String role);
    public UIFactory(String name, String role, Map properties);

    public Object getUI(Object service) throws UICreationException;
}
```

A service may have lots of these factories, each capable of creating a different user interface object.

3. Objects Returned from getUI()

There are many kinds of user-interface objects that can be returned. They could be individual components such as buttons or lists, containers such as panels, or components with their own toplevel windows such as frames and dialogs. Dialogs may be modal or non-modal. These will all need to be handled by a client in different ways: a component needs to be placed within a container, a frame is shown and processing independently continues, a modal dialog blocks execution of the calling process until it is dismissed. The client will need to know these broad categories.

A user interface may just use AWT components. It may use Swing components. But it may also use the 2-D or 3-D classes, or even things like the MPEG class from the Media API. Perhaps it doesn't have a "traditional" screen interface, but uses a speech interface instead. There may even be possibilities of some other description apart from Java objects: JACL scripts, perhaps (JACL is a `tcl` interpreter written in Java).

The client needs to have a way of figuring out which types of objects a factory will produce, without actually going all the way through the actual creation. There are two parts to this:

1. What type of object is returned from `getUI()`?
2. What classes are needed by the client before it can succeed it creating the complete set of objects required by the user-interface?

The first problem (what type of object is returned from `getUI()`) can be answered by using suitable subclasses of `UIFactory`:

1. `AWTFrameFactory` returns an AWT `Frame`
2. `SwingFrameFactory` returns a Swing `JFrame`
3. `AWTPanelFactory` returns an AWT `Panel`
4. `SwingPanelFactory` returns a Swing `JPanel`

and so on. The list of these classes, and any subclassing relations between them is not fixed yet, and may change in later versions of the ''serviceui'' project. Nevertheless, it is likely that this type of scheme will be eventually decided upon.

The second problem (what classes are needed) is even less settled. A set of ''required libraries'' is currently favoured. This would be one of the properties stored in the map of additional properties in each factory. Each element of this could contain the full package name of the library (such as `"javax.swing"`) plus other information such as version number. There are still some tricky elements in this: for example, must the client supply these libraries, or could they be downloaded from an HTTP server? Some libraries such as the Java Media Framework contain objects written in pure Java, while others in this same package require native code underneath (such as the MPEG player). Must a native library be available even if the widget requiring the native library isn't used?

Processes to define standards acceptable to the Jini Community are being set up. Standards within this are also being defined. Right now things are unsettled, but this will change.

4. Support Classes

The ''serviceUI'' will eventually decide on a standard set of support classes. In the meantime you will have to ''wing it'' and be aware that you are writing/using unstable code. For the purposes of the rest of this chapter this section will define classes that will approximate the final classes of the ''serviceUI'' specification.

A possible implementation of the `UIFactory` class is

```
/**
 * UIFactory.java
 *
 *
 * Created: Sun Nov 14 18:07:12 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package ui;

import net.jini.entry.AbstractEntry;
import java.util.Map;
import java.util.HashMap;

public class UIFactory extends AbstractEntry {
```



```
    public String name;
    public String role;
    public Map properties;

    public UIFactory() {
        this(null, null, null);
    }

    public UIFactory(String name, String role) {
        this(name, role, null);
    }

    public UIFactory(String name, String role, Map properties) {
        this.name = name;
        this.role = role;
        if (properties == null) {
            this.properties = null;
        } else {
            this.properties = new HashMap(properties);
        }
    }

    public Object getUI(Object service) throws UICreationException {
        throw new UICreationException();
    }
} // UIFactory
```

(The package name is convenient for this book: it will not be the final package name.)

A factory that can be used to return AWT `Frame`'s is the `AWTFrameFactory`. It does not enforce this return type, and users of this class are expected to examine the class type in order to determine what to do with it.

```
/**
 * AWTFrameFactory.java
 *
 *
 * Created: Sun Nov 14 18:18:32 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package ui;

import java.util.Map;

public class AWTFrameFactory extends UIFactory {

    public AWTFrameFactory() {
        super();
    }
```

```
    public AWTFrameFactory(String name, String role) {
        super(name, role);
    }

    public AWTFrameFactory(String name, String role, Map properties) {
        super(name, role, properties);
    }

} // AWTFrameFactory
```

5. Images

User interfaces often contain images. They may be used as icons in toolbars, for general images on the screen, or for the icon image when the application is iconified. When a user interface is created on the client, these images will also need to be created and installed in the relevant part of the application. Images are not serializable, so they cannot be created on the server and exported as live objects in some manner. They need to be created from scratch on the client.

The Swing package contains a convenience class `ImageIcon`. This can be instantiated from a byte array, a filename, or most interestingly here, from a URL. So if an image is stored where an HTTP server can find it, then the `ImageIcon` constructor can use this directly. There may be failures in this: the URL may be incorrect or malformed, or the image may fail to exist on the HTTP server. Suitable code is

```
        ImageIcon icon = null;
        try {
            icon = new ImageIcon(new URL("http://localhost/images/mindstorms.jpg"));
            switch (icon.getImageLoadStatus()) {
            case MediaTracker.ABORTED:
            case MediaTracker.ERRORED:
                System.out.println("Error");
                icon = null;
                break;
            case MediaTracker.COMPLETE:
                System.out.println("Complete");
                break;
            case MediaTracker.LOADING:
                System.out.println("Loading");
                break;
            }
        } catch(java.net.MalformedURLException e) {
            e.printStackTrace();
        }
        // icon is null or is a valid image
```

6. MindStorms

6.1 RCXLoaderFrame

A MindStorms robot is primarily defined by the `RCXPort` interface. The Jini version is defined by the `RCXPortImplementation` interface:


```
/**
 * RCXPortInterface.java
 *
 *
 * Created: Wed Jun  2 23:08:12 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package rcx.jini;

import net.jini.core.event.RemoteEventListener;

public interface RCXPortInterface extends java.io.Serializable {

    /**
     * constants to distinguish message types
     */
    public final long ERROR_EVENT = 1;
    public final long MESSAGE_EVENT = 2;

    /**
     * Write an array of bytes that are RCX commands
     * to the remote RCX.
     */
    public boolean write(byte[] byteCommand) throws java.rmi.RemoteException;

    /**
     * Parse a string into a set of RCX command bytes
     */
    public byte[] parseString(String command) throws java.rmi.RemoteException;

    /**
     * Add a RemoteEvent listener to the RCX for messages and errors
     */
    public void addListener(RemoteEventListener listener)
        throws java.rmi.RemoteException;

    /**
     * The last message from the RCX
     */
    public byte[] getMessage(long seqNo)
        throws java.rmi.RemoteException;

    /**
     * The error message from the RCX
     */
    public String getError(long seqNo)
        throws java.rmi.RemoteException;

} // RCXPortInterface
```

This allows programs to be downloaded and run, and instructions to be sent for direct execution. As it stands, the client needs to call these interface methods directly. To make it more useable for the human trying to drive a robot, some sort of user interface would be useful.

There can be several general purpose user interfaces for the RCX robot, including:

1. Enter machine code (somehow) and download that
2. Enter RCX assembler code in the form of strings, assemble and download them
3. Enter NQC (Not Quite C) code, compile and download it

The set of RCX classes by Laverde includes a standalone application called `RCXLoader` which does the second of these. We can steal code from this and some of his other classes, to define a class `RCXLoaderFrame`

```
package rcx.jini;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.UnknownEventException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import rcx.*;

/*
 * RCXLoaderFrame
 * @author Dario Laverde
 * @author Jan Newmarch
 * @version 1.1
 * Copyright 1999 Dario Laverde, under terms of GNU LGPL
 */

public class RCXLoaderFrame extends Frame
    implements ActionListener, WindowListener, RemoteEventListener
{
    private String      portName;
    private RCXPortInterface     rcxPort;
    private Panel       textPanel;
    private Panel       topPanel;
    private TextArea    textArea;
    private TextField   textField;
    private Button      tableButton;
    private Properties  parameters;
    private int inByte;
    private int charPerLine = 48;
    private int lenCount;
    private StringBuffer sbuffer;
    private byte[] byteArray;
    private Frame opcodeFrame;
    private TextArea opcodeTextArea;

    public static Hashtable Opcodes=new Hashtable(55);

    static {
        Opcodes.put(new Byte((byte)0x10),"PING            ,void, void,P");
        Opcodes.put(new Byte((byte)0x12),"GETVAL          ,byte src byte arg, short val,P");
        Opcodes.put(new Byte((byte)0x13),"SETMOTORPOWER   ,byte motors byte src byte arg, void,CP");
        Opcodes.put(new Byte((byte)0x14),"SETVAL          ,byte index byte src byte arg, void,CP");
        Opcodes.put(new Byte((byte)0x15),"GETVERSIONS     ,byte key[5], short rom[2] short firmware[2],P");
        Opcodes.put(new Byte((byte)0x17),"CALLSUB         ,byte subroutine, void,C");
        Opcodes.put(new Byte((byte)0x20),"GETMEMMAP       ,void, short map[94],P");
        Opcodes.put(new Byte((byte)0x21),"SETMOTOR        ,byte code, void,CP");
        Opcodes.put(new Byte((byte)0x22),"SETTIME         ,byte hours byte minutes, void,CP");
        Opcodes.put(new Byte((byte)0x23),"PLAYTONE        ,short frequency byte duration, void,CP");
        Opcodes.put(new Byte((byte)0x24),"ADDVAR          ,byte index byte src short arg, void,CP");
        Opcodes.put(new Byte((byte)0x25),"DOWNLOADTASK    ,byte unknown short task short length, byte error,P");
        Opcodes.put(new Byte((byte)0x27),"BRANCHALWAYSNEAR,byte offset, void,C");
        Opcodes.put(new Byte((byte)0x30),"GETBATTERYPOWER ,void, short millivolts,P");
        Opcodes.put(new Byte((byte)0x31),"SETTRANSMITRANGE,byte range, void,CP");
        Opcodes.put(new Byte((byte)0x32),"SETSENSORTYPE   ,byte sensor byte type, void,CP");
        Opcodes.put(new Byte((byte)0x33),"SETDISPLAY      ,byte src short arg, void,CP");
        Opcodes.put(new Byte((byte)0x34),"SUBFROMVAR      ,byte index byte src short arg, void,CP");
        Opcodes.put(new Byte((byte)0x35),"DOWNLOADSUB     ,byte unknown short subroutine short length, byte error,P");
        Opcodes.put(new Byte((byte)0x40),"DELALLTASKS     ,void, void,CP");
        Opcodes.put(new Byte((byte)0x42),"SETSENSORMODE   ,byte sensor byte code, void,CP");
        Opcodes.put(new Byte((byte)0x43),"WAIT            ,byte src short arg, void,C");
```



```
        Opcodes.put(new Byte((byte)0x44),"DIVIDEVAR       ,byte index byte src short arg, void,CP");
        Opcodes.put(new Byte((byte)0x45),"TRANSFERDATA    ,short index short length byte data[length] byte checksum, byte error,P");
        Opcodes.put(new Byte((byte)0x50),"STOPALLTASKS    ,void, void,CP");
        Opcodes.put(new Byte((byte)0x51),"PLAYSOUND       ,byte sound, void,CP");
        Opcodes.put(new Byte((byte)0x52),"SETDATALOGSIZE  ,short size, byte error,CP");
        Opcodes.put(new Byte((byte)0x54),"MULTIPLYVAR     ,byte index byte src short arg, void,CP");
        Opcodes.put(new Byte((byte)0x60),"POWEROFF        ,void, void,CP");
        Opcodes.put(new Byte((byte)0x61),"DELTASK         ,byte task, void,CP");
        Opcodes.put(new Byte((byte)0x62),"DATALOGNEXT     ,byte src byte arg, byte error,CP");
        Opcodes.put(new Byte((byte)0x64),"SIGNVAR         ,byte index byte src short arg, void,CP");
        Opcodes.put(new Byte((byte)0x65),"DELFIRMWARE     ,byte key[5], void,CP");
        Opcodes.put(new Byte((byte)0x70),"DELALLSUBS      ,void, void,CP");
        Opcodes.put(new Byte((byte)0x71),"STARTTASK       ,byte task, void,CP");
        Opcodes.put(new Byte((byte)0x72),"BRANCHALWAYSFAR ,byte offset byte extension, void,C");
        Opcodes.put(new Byte((byte)0x74),"ABSVAL          ,byte index byte src short arg, void,CP");
        Opcodes.put(new Byte((byte)0x75),"DOWNLOADFIRMWARE,short address short checksum byte unknown, void,P");
        Opcodes.put(new Byte((byte)0x81),"STOPTASK        ,byte task, void,CP");
        Opcodes.put(new Byte((byte)0x82),"SETLOOPCOUNTER  ,byte src byte arg, void,C");
        Opcodes.put(new Byte((byte)0x84),"ANDVAR          ,byte index byte src byte arg, void,CP");
        Opcodes.put(new Byte((byte)0x90),"CLEARMESSAGE    ,void, void,C");
        Opcodes.put(new Byte((byte)0x91),"SETPROGRAMNUM   ,byte program, void,CP");
        Opcodes.put(new Byte((byte)0x92),"DECCNTRANDBRANCH,short offset, void,C");
        Opcodes.put(new Byte((byte)0x94),"ORVAR           ,byte index byte src byte arg, void,CP");
        Opcodes.put(new Byte((byte)0x95),"TESTANDBRANCH   ,byte opsrc1 byte src2 short arg1 byte arg2 short offset,void,C");
        Opcodes.put(new Byte((byte)0xa1),"CLEARTIME       ,byte timer, void,CP");
        Opcodes.put(new Byte((byte)0xa4),"UPLOADDATALOG   ,short first short count, dlrec data[length],P");
        Opcodes.put(new Byte((byte)0xa5),"UNLOCKFIRMWARE  ,byte key[5], byte data[25],P");
        Opcodes.put(new Byte((byte)0xb1),"SETPOWERDOWN    ,byte minutes, void,CP");
        Opcodes.put(new Byte((byte)0xb2),"SENDMESSAGE     ,byte src byte arg, void,C");
        Opcodes.put(new Byte((byte)0xc1),"DELSUB          ,byte subroutine, void,CP");
        Opcodes.put(new Byte((byte)0xd1),"CLEARSENSOR     ,byte sensor, void,CP");
        Opcodes.put(new Byte((byte)0xe1),"SETMOTORDIR     ,byte code, void,CP");
        Opcodes.put(new Byte((byte)0xf7),"SETMESSAGE      ,byte message, void,PC");
    }

    // added port interface parameter to Dario's code
    public RCXLoaderFrame(RCXPortInterface port) {
        super("RCX Loader");

        // changed from Dario's code
        rcxPort = port;

        addWindowListener(this);

        topPanel  = new Panel();
        topPanel.setLayout(new BorderLayout());

        tableButton = new Button("table");
        tableButton.addActionListener(this);

        textField = new TextField();
        // textField.setEditable(false);
        // textField.setEnabled(false);
        // tableButton.setEnabled(false);
        textField.addActionListener(this);

        textPanel = new Panel();
        textPanel.setLayout(new BorderLayout(5,5));

        topPanel.add(textField,"Center");
        topPanel.add(tableButton,"East");
        textPanel.add(topPanel,"North");

        textArea = new TextArea();
        // textArea.setEditable(false);
        textArea.setFont(new Font("Courier",Font.PLAIN,12));
        textPanel.add(textArea,"Center");

        add(textPanel, "Center");

        textArea.setText("initializing...\n");

        Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
        setBounds(screen.width/2-370/2,screen.height/2-370/2,370,370);
        // setVisible(true);

        // changed listener type from Dario's code
        try {
            // We are remote to the object we are listening to
            // (the RCXPort), so the RCXPort must get a stub object
            // for us. We have subclassed from Frame, not from
            // UnicastRemoteObject. So we must export ourselves
            // for the remote references to work
            UnicastRemoteObject.exportObject(this);
            rcxPort.addListener(this);
        } catch(Exception e) {
            textArea.append(e.toString());
```

```
    }
    tableButton.setEnabled(true);

    /*
    if(rcxPort.isOpen()) {
        textArea.append("RCXPort initialized.\n");
        String lasterror = rcxPort.getLastError();
        if(lasterror!=null)
            textArea.append(lasterror+"\n");
        textField.setEditable(true);
        textField.setEnabled(true);
        textField.requestFocus();
    }
    else {
        if(portName!=null) {
            textArea.append("Failed to create RCXPort with "+portName+"\n");
            textArea.append("Port "+portName+" is invalid or may be ");
            textArea.append("currently used.\nTry another port.\n");
            textArea.append("Edit or create a file named parameters.txt ");
            textArea.append("that has:\nport=COM1\n(replace COM1 with ");
            textArea.append("the correct port name)\n");
        }
        else
            textArea.append("Please specify a port in parameters.txt\n");
            textArea.append("Create a file that has:\nport=COM1\n");
            textArea.append("(replace COM1 with the correct port name)\n");
    }
    */
}

/*
public void receivedMessage(byte[] responseArray) {
    if(responseArray==null)
        return;
    for(int loop=0;loop<responseArray.length;loop++) {
        int newbyte = (int)responseArray[loop];
        if(newbyte<0) newbyte=256+newbyte;
        sbuffer = new StringBuffer(Integer.toHexString(newbyte));

        if(sbuffer.length()<2)
            sbuffer.insert(0,'0');
        textArea.append(sbuffer+" ");
        lenCount+=3;
        if(lenCount==charPerLine) {
            lenCount=0;
            textArea.append("\n");
        }
    }
    if(lenCount!=charPerLine)
        textArea.append("\n");
}
*/

/*
public void receivedError(String error) {
    textArea.append(error+"\n");
}
*/

public void actionPerformed(ActionEvent evt) {
    Object obj = evt.getSource();
    if(obj==textField) {
        String strInput = textField.getText();
        textField.setText("");
        textArea.append("> "+strInput+"\n");
        try {
            byteArray = rcxPort.parseString(strInput);
        } catch(RemoteException e) {
            textArea.append(e.toString());
        }
        // byteArray = RCXOpcode.parseString(strInput);
        if(byteArray==null) {
            textArea.append("Error: illegal hex character or length\n");
            return;
        }
        if(rcxPort!=null) {
            try {
                if(!rcxPort.write(byteArray)) {
                    textArea.append("Error: writing data to port "+portName+"\n");
                }
            } catch(Exception e) {
                textArea.append(e.toString());
            }
        }
    }
    else if(obj==tableButton) {
        // make this all in the ui side
```



```
            showTable();
            setLocation(0,getLocation().y);
        }
    }

    public void windowActivated(WindowEvent e) { }
    public void windowClosed(WindowEvent e) { }
    public void windowDeactivated(WindowEvent e) { }
    public void windowDeiconified(WindowEvent e) { }
    public void windowIconified(WindowEvent e) { }
    public void windowOpened(WindowEvent e) { }
    public void windowClosing(WindowEvent e) {
        /*
        if(rcxPort!=null)
                rcxPort.close();
        */
        System.exit(0);
    }

    public void notify(RemoteEvent evt) throws UnknownEventException,
                                               java.rmi.RemoteException {

        long id = evt.getID();
        long seqNo = evt.getSequenceNumber();
        if (id == RCXPortInterface.MESSAGE_EVENT) {
            byte[] message = rcxPort.getMessage(seqNo);
            StringBuffer sbuffer = new StringBuffer();
            for(int n = 0; n < message.length; n++) {
                int newbyte = (int) message[n];
                if (newbyte < 0) {
                    newbyte += 256;
                }
                sbuffer.append(Integer.toHexString(newbyte) + " ");
            }
            textArea.append(sbuffer.toString());
            System.out.println("MESSAGE: " + sbuffer.toString());
        } else if (id == RCXPortInterface.ERROR_EVENT) {
            textArea.append(rcxPort.getError(seqNo));
        } else {
            throw new UnknownEventException("Unknown message " + evt.getID());
        }
    }

    public void showTable()
    {
        if(opcodeFrame!=null)
        {
            opcodeFrame.dispose();
            opcodeFrame=null;
            return;
        }
        opcodeFrame = new Frame("RCX Opcodes Table");
        Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
        opcodeFrame.setBounds(screen.width/2-70,0,screen.width/2+70,screen.height-25);
        opcodeTextArea = new TextArea("  Opcode          ,parameters, response, C=program command P=remote command\n",60,100);
        opcodeTextArea.setFont(new Font("Courier",Font.PLAIN,10));
        opcodeFrame.add(opcodeTextArea);
        Enumeration k = Opcodes.keys();
        for (Enumeration e = Opcodes.elements(); e.hasMoreElements();) {
            String tmp = Integer.toHexString(((Byte)k.nextElement()).intValue());
            tmp = tmp.substring(tmp.length()-2)+" "+(String)e.nextElement()+"\n";
            opcodeTextArea.append(tmp);
        }
        opcodeTextArea.setEditable(false);
        opcodeFrame.setVisible(true);
    }
}
```

6.2 RCXLoaderFrameFactory

The factory object for the RCX is now easy to define, it just returns a `RCXLoaderFrame` in the `getUI()` method:

```
/**
 * RCXLoaderFrameFactory.java
 *
 *
 * Created: Sun Nov 14 18:22:46 1999
```

```
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package rcx.jini;

import ui.AWTFrameFactory;
import ui.UICreationException;
import java.util.Map;

public class RCXLoaderFrameFactory extends AWTFrameFactory {

    public RCXLoaderFrameFactory() {
        super();
    }

    public RCXLoaderFrameFactory(String name, String role) {
        super(name, role);
    }

    public RCXLoaderFrameFactory(String name, String role, Map properties) {
        super(name, role, properties);
    }

    public Object getUI(Object obj) throws UICreationException {
        if (obj instanceof RCXPortInterface) {
            return new RCXLoaderFrame((RCXPortInterface) obj);
        } else {
            throw new UICreationException();
        }
    }

} // RCXLoaderFrameFactory
```

6.3 Exporting the FrameFactory

The factory object is exported by making it one of the `Entry` objects with a name and role that is agreed on by all services and clients:

```
Entry[] entries = {new RCXLoaderFrameFactory("generic", "user")};
JoinManager joinMgr = new JoinManager(impl,
                                      entries,
                                      this,
                                      new LeaseRenewalManager());
```

6.4 Customised User Interfaces

The `RCXLoaderFrame` is a general interface to any RCX robot. Of course, there could be many other such interfaces, differing in the classes used, the amount of internationalisation support, the appearance, *etc.* All the variations, however, will just use the standard `RCXPortInterface`, as that is all they know about.

The Lego pieces can be combined in a huge variety of ways, and the RCX itself is programmable. So you can build an RCX car, an RCX crane, an RCX maze-runner, and so on. Each different robot can be driven by the general interface, but most could benefit from a custom-built interface for that type of robot. This is typical: for example, every blender could be driven from a general blender user interface (using the - possibly - forthcoming standard blender interface :-). But the blenders from individual vendors would have their own customised user interface for their brand of blender.

I have been using an RCX car. While it can do lots of things, for demonstrations it has been convenient to use five commands: *forward*, *stop*, *back*, *left* and *right*, with a user interface looking like

In the MindStorms chapter this appearance was hard-coded into the client - which really shouldn't know about this sort of detail.

6.5 CarJFrame

The `CarJFrame` class produces the user interface as a Swing `JFrame`, with the buttons generating specific RCX code for this model.

```
package rcx.jini;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.UnknownEventException;
import java.net.URL;

class CarJFrame extends JFrame
    implements RemoteEventListener, ActionListener {

    protected RCXPortInterface port = null;

    JFrame frame;
    JTextArea text;

    public CarJFrame(RCXPortInterface port) {
        super() ;
        this.port = port;

        frame = new JFrame("Lego MindStorms");
        Container content = frame.getContentPane();
        JLabel label = null;
        ImageIcon icon = null;
        try {
            icon = new ImageIcon(new
                                          URL("http://localhost/images/mindstorms.jpg"));
            switch (icon.getImageLoadStatus()) {
```

```
            case MediaTracker.ABORTED:
            case MediaTracker.ERRORED:
                System.out.println("Error");
                icon = null;
                break;
            case MediaTracker.COMPLETE:
                System.out.println("Complete");
                break;
            case MediaTracker.LOADING:
                System.out.println("Loading");
                break;
            }
        } catch(java.net.MalformedURLException e) {
            e.printStackTrace();
        }
        if (icon != null) {
            label = new JLabel(icon);
        } else {
            label = new JLabel("MINDSTORMS");
        }

        JPanel pane = new JPanel();
        pane.setLayout(new GridLayout(2, 3));

        content.add(label, "North");
        content.add(pane, "Center");

        JButton btn = new JButton("Forward");
        pane.add(btn);
        btn.addActionListener(this);

        btn = new JButton("Stop");
        pane.add(btn);
        btn.addActionListener(this);

        btn = new JButton("Back");
        pane.add(btn);
        btn.addActionListener(this);

        btn = new JButton("Left");
        pane.add(btn);
        btn.addActionListener(this);

        label = new JLabel("");
        pane.add(label);

        btn = new JButton("Right");
        pane.add(btn);
        btn.addActionListener(this);

        frame.pack();
        frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent evt) {
        String name = evt.getActionCommand();
        byte[] command;
```



```
        try {
            if (name.equals("Forward")) {
                command = port.parseString("e1 81");
                if (! port.write(command)) {
                    System.err.println("command failed");
                }
                command = port.parseString("21 81");
                if (! port.write(command)) {
                    System.err.println("command failed");
                }
            } else  if (name.equals("Stop")) {
                command = port.parseString("21 41");
                if (! port.write(command)) {
                    System.err.println("command failed");
                }
            } else  if (name.equals("Back")) {
                command = port.parseString("e1 41");
                if (! port.write(command)) {
                    System.err.println("command failed");
                }
                command = port.parseString("21 81");
                if (! port.write(command)) {
                    System.err.println("command failed");
                }

            } else  if (name.equals("Left")) {
                command = port.parseString("e1 04");
                if (! port.write(command)) {
                    System.err.println("command failed");
                }
                command = port.parseString("21 84");
                if (! port.write(command)) {
                    System.err.println("command failed");
                }
                Thread.sleep(100);
                command = port.parseString("21 44");
                if (! port.write(command)) {
                    System.err.println("command failed");
                }

            } else  if (name.equals("Straight")) {

            } else  if (name.equals("Right")) {
                command = port.parseString("e1 84");
                if (! port.write(command)) {
                    System.err.println("command failed");
                }
                command = port.parseString("21 84");
                if (! port.write(command)) {
                    System.err.println("command failed");
                }
                Thread.sleep(100);
                command = port.parseString("21 44");
                if (! port.write(command)) {
                    System.err.println("command failed");
                }
            }
        } catch(Exception e) {
```

```
            e.printStackTrace();
        }
    }

    public void notify(RemoteEvent evt) throws UnknownEventException,
        java.rmi.RemoteException {
        // System.out.println(evt.toString());

        long id = evt.getID();
        long seqNo = evt.getSequenceNumber();
        if (id == RCXPortInterface.MESSAGE_EVENT) {
            byte[] message = port.getMessage(seqNo);
            StringBuffer sbuffer = new StringBuffer();
            for(int n = 0; n < message.length; n++) {
                int newbyte = (int) message[n];
                if (newbyte < 0) {
                    newbyte += 256;
                }
                sbuffer.append(Integer.toHexString(newbyte) + " ");
            }
            System.out.println("MESSAGE: " + sbuffer.toString());
        } else if (id == RCXPortInterface.ERROR_EVENT) {
            System.out.println("ERROR: " + port.getError(seqNo));
        } else {
            throw new UnknownEventException("Unknown message " + evt.getID());
        }
    }
}
```

6.6 CarJFrameFactory

The factory generates a `CarJFrame` object

```
/**
 * CarJFrameFactory.java
 *
 *
 * Created: Fri Oct 29 23:27:16 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

package rcx.jini;

import ui.SwingFrameFactory;
import ui.UICreationException;
import java.util.Map;
import rcx.*;

public class CarJFrameFactory extends SwingFrameFactory {
```



```
    public CarJFrameFactory() {
        super();
    }

    public CarJFrameFactory(String name, String role) {
        super(name, role);
    }

    public CarJFrameFactory(String name, String role, Map properties) {
        super(name, role, properties);
    }

    public Object getUI(Object service)
        throws UICreationException {

        if (service instanceof RCXPortInterface) {
            return new CarJFrame((RCXPortInterface) service);
        } else {
            throw new UICreationException();
        }
    }

} // CarJFrameFactory
```

6.7 Exporting the FrameFactory

Both of the user interfaces discussed can be exported by expanding the set of `Entry` objects.

```
Entry[] entries = {new RCXLoaderFrameFactory("generic", "user"),
                   new CarJFrameFactory("car", "user")};
JoinManager joinMgr = new JoinManager(impl,
                                      entries,
                                      this,
                                      new LeaseRenewalManager());
```

6.8 The RCX Client

The following client will start up all user interfaces that it understands

```
package client;

import rcx.jini.*;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
```

```
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.UnknownEventException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.entry.Entry;
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.lookup.ServiceItem;

import ui.UIFactory;
import ui.AWTFrameFactory;
import ui.SwingFrameFactory;
import ui.UICreationException;

/**
 * TestRCX2.java
 *
 *
 * Created: Wed Mar 17 14:29:15 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class TestRCX2 implements DiscoveryListener {

    public static void main(String argv[]) {
        new TestRCX2();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(1000000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestRCX2() {
        System.setSecurityManager(new RMISecurityManager());

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
```



```
    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();
        Class [] classes = new Class[] {RCXPortInterface.class};
        RCXPortInterface port = null;
        Entry[] entries = {new UIFactory(null, null)};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       entries);

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Service found");
            ServiceRegistrar registrar = registrars[n];
            ServiceMatches matches = null;
            try {
                matches = registrar.lookup(template, 10);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(2);
            }
System.out.println("match count; " + matches.items.length);
            for (int nn = 0; nn < matches.items.length; nn++) {
                ServiceItem item = matches.items[nn];
                port = (RCXPortInterface) item.service;
                if (port == null) {
                    System.out.println("port null");
                    continue;
                }

                Entry[] attributes = item.attributeSets;
System.out.println("Attribute count: " + attributes.length);
                for (int m = 0; m < attributes.length; m++) {
System.out.println(attributes[m].toString());
                    Entry attr = attributes[m];
                    if (attr instanceof UIFactory) {
                        showUI(port, (UIFactory) attr);
                    }
                }
            }



        }
    }

    public void discarded(DiscoveryEvent evt) {
        // empty
    }

    private void showUI(RCXPortInterface port, UIFactory attr) {
System.out.println("showing ui");
        if (attr instanceof AWTFrameFactory) {
            Frame frame = null;
```

```
            try {
                frame = (Frame) attr.getUI(port);
            } catch(UICreationException e) {
                e.printStackTrace();
                return;
            }
            frame.setVisible(true);
        } else if (attr instanceof SwingFrameFactory) {
            JFrame frame = null;
            try {
                frame = (JFrame) attr.getUI(port);
            } catch(UICreationException e) {
                e.printStackTrace();
                return;
            }
            frame.setVisible(true);
        } else {
            System.out.println("non-gui entry");
        }
    }
} // TestRCX
```

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Activation

Contents

1. Introduction
2. A Service using Activation
    1. Service
    2. Server
    3. Running the Service
    4. Security
    5. Non-lazy Services
    6. Maintaining State
3. LeaseRenewalService
    1. Norm
    2. Using the LeaseRenewalService
4. LookupDiscoveryService
    1. Fiddler
    2. Using the LookupDiscoveryService

1. Introduction

Many of the examples in earlier chapters use RMI proxies for services. These services subclass `UnicastRemoteObject`, and live within a server whose principal task is to keep the service alive and registered with lookup services. If the server fails to renew leases then lookup services will eventually discard it; if it fails to keep itself and its service alive then the service will not be available when a client wants to use it.

This results in a server and a service which most of the time will be idle, probably swapped out to disk but still using virtual memory. In JDK 1.2, the memory requirements can be enormous (hopefully this will get fixed, but at the moment this is a severe embarrassment to Java, and a potential threat to the success of Jini). In JDK 1.2, there is an extension to RMI called *activation* which allows an idle object to die, and be recalled to life when needed. In this way, it does not occupy virtual memory while idle. Of course, a process needs to be alive to restore such objects, and RMI supplies a daemon `rmid` to manage this. In effect, `rmid` acts as another virtual memory manager as it stores information about dormant Java objects in its own files and restores them from there as needed.

There is a serious limitation to `rmid`: it is a Java program itself and when running also uses enormous amounts of memory! So it only makes sense to use this technique when you expect to be running a number of largely idle services on the same machine. When a service is recalled to life, or *activated*, a new JVM may be started to run the object. This again increases memory use.

If memory use was the only concern, then there are a variety of other systems such as `echidna` which run multiple applications within a single JVM. These may be adequate to solve memory issues. However, RMI Activation is also designed to work with distributed objects, and allows JVM's to hold remote references to objects which are no longer active. Instead of throwing a remote exception on trying to access these objects, the Activation system tries to resurrect the object using `rmid` to give a valid (and new) reference. Of course, if it fails to do this, it will throw an exception anyway.

The principal place that this is used in the standard Jini distribution is with the `reggie` lookup service. `reggie` is an activatable service that starts, registers itself with `rmid` and then exits. Whenever lookup services are required, `rmid` restarts `reggie` in a new JVM. Clients of the lookup service are unaware of this mechanism: they simply make calls on their proxy `ServiceRegistration` object and the Activation system looks after the rest. The main problem is for the system administrator: getting `reggie` to work in the first place!

2. A Service using Activation

The major concepts in Activation are the activatable object itself (which extends `java.rmi.activation.Activatable`) and the environment in which it runs, an `ActivationGroup`. A JVM may have an activation group associated with it. If an object needs to be activated and there is already a JVM running its group then it is restarted within that JVM. Otherwise, a new JVM is started. An activation group may hold a number of co-operating objects.

2.1 Service

Making an object into an activable object requires subclassing from `Activatable`, and using a special two-argument constructor which will be called when the object needs to be reconstructed. There is a standard implementation of this constructor which just passes up to the super class:

```
public ActivatableImpl(ActivationID id, MarshalledObject data)
    throws RemoteException {
    super(id, 0);
}
```

(The use of the marshalled data is discussed later). This is the only change normally needed for a service. For example, the file classifier becomes

```
package activation;

import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;

import common.MIMEType;
import common.FileClassifier;
import option3.RemoteFileClassifier;

/**
 * FileClassifierImpl.java
 *
 *
 * Created: Wed Mar 17 14:22:13 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class FileClassifierImpl extends Activatable
                             implements RemoteFileClassifier {

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        System.out.println("Called with " + fileName);
        try {Runtime.getRuntime().exec("/usr/bin/whoami");} catch(Exception e){e.printStackTrace();}
        java.util.Properties props = System.getProperties();
        System.out.println("user " + props.getProperty("user.name"));
        if (new java.io.File("/etc/at.deny").canRead())
            System.out.println("can read at.deny");
        else
            System.out.println("can't read at.deny");
        if (fileName.endsWith(".gif")) {
            return new MIMEType("image", "gif");
        } else if (fileName.endsWith(".jpeg")) {
            return new MIMEType("image", "jpeg");
        } else if (fileName.endsWith(".mpg")) {
            return new MIMEType("video", "mpeg");
        } else if (fileName.endsWith(".txt")) {
            return new MIMEType("text", "plain");
        } else if (fileName.endsWith(".html")) {
            return new MIMEType("text", "html");
        } else
            // fill in lots of other types,
            // but eventually give up and
            return new MIMEType(null, null);
    }


    public FileClassifierImpl(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException {
        super(id, 0);
    }

} // FileClassifierImpl
```

Note that an activatable object cannot have a ''normal'' constructor to initialize itself, since this new constructor is how the object is constructed by the activation system.

2.2 Server

The server needs to create an activation group for the objects to run in. The main issue in this is to set a security policy file. There are two security policies in activatable objects: the policy used to *create* the server and export the service, and the policy used to *run* the service. The activation group uses its policy file for *running* methods of the service object. The policy file for the server is set using the normal `-Djava.security.policy=...` argument to starting the server. After setting various parameters, the activation group is set for the JVM by `ActivationGroup.createGroup()`.

The ''standard'' remote objects that subclass `UnicastRemoteObject` are created in the normal way using a constructor. Activatable objects are not constructed in the server but are instead registered with `rmid`, which will look after construction on an as-needed basis. `rmid` needs to know the class name and the location of the class files. These are wrapped up in an `ActivationDesc`, and registered with `rmid` by `Activatable.register()`. This returns an RMI stub object that can be registered with lookup services using the `ServiceRegistrar.register()` methods.

The changes for servers from the unicast remote objects are

1. An activation group has to be created with a security policy file
2. The service is not created explicitly but instead registered with `rmid`
3. The return object from this registration is a stub that can be registered with lookup services
4. Leasing vanishes: the server just exits. The service will just expire after a while. Oh well...see a later section

The file classifier server which uses an activatable service is

```
package activation;

import option3.RemoteFileClassifier;

import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
```

```
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
import java.rmi.RMISecurityManager;
import java.rmi.MarshalledObject;
import java.rmi.activation.ActivationDesc;
import java.rmi.activation.ActivationGroupDesc;
import java.rmi.activation.ActivationGroupDesc.CommandEnvironment;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationGroup;
import java.rmi.activation.ActivationGroupID;

import java.util.Properties;

import java.rmi.activation.UnknownGroupException;
import java.rmi.activation.ActivationException;
import java.rmi.RemoteException;

/**
 * FileClassifierServer.java
 *
 *
 * Created: Wed Dec 22 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class FileClassifierServer implements DiscoveryListener {

    static final protected String SECURITY_POLICY_FILE =
        "/home/jan/projects/jini/doc/policy.all";
    // Don't forget the trailing '/'!
    static final protected String CODEBASE = "http://localhost/classes/";

    // protected FileClassifierImpl impl;
    protected RemoteFileClassifier stub;

    public static void main(String argv[]) {
        new FileClassifierServer(argv);
        // stick around while lookup services are found
        try {
            Thread.sleep(10000L);
        } catch(InterruptedException e) {
            // do nothing
        }
        // the server doesn't need to exist anymore
        System.exit(0);
    }

    public FileClassifierServer(String[] argv) {
        // install suitable security manager
        System.setSecurityManager(new RMISecurityManager());
```



```
        // Install an activation group
        Properties props = new Properties();
        props.put("java.security.policy",
                SECURITY_POLICY_FILE);
        ActivationGroupDesc.CommandEnvironment ace = null;
        ActivationGroupDesc group = new ActivationGroupDesc(props, ace);
        ActivationGroupID groupID = null;
        try {
            groupID = ActivationGroup.getSystem().registerGroup(group);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }

        try {
            ActivationGroup.createGroup(groupID, group, 0);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }

        String codebase = CODEBASE;
        MarshalledObject data = null;
        ActivationDesc desc = null;
        try {
            desc = new ActivationDesc("activation.FileClassifierImpl",
                                                codebase, data);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }
System.out.println("Group ID " + ActivationGroup.currentGroupID().toString());

        try {
            stub = (RemoteFileClassifier) Activatable.register(desc);
        } catch(UnknownGroupException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        }

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
```

```
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();
        RemoteFileClassifier service;

        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];

            // export the proxy service
            ServiceItem item = new ServiceItem(null,
                                               stub,
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.print("Register exception: ");
                e.printStackTrace();
                // System.exit(2);
                continue;
            }
            try {
                System.out.println("service registered at " +
                                   registrar.getLocator().getHost());
            } catch(Exception e) {
            }
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }
} // FileClassifierServer
```

2.3 Running the Service

The service and the server must be compiled as usual, and in addition an RMI stub object must be created for the service using `rmic`. The class files for the stub must be copied to somewhere that an HTTP server can deliver them to clients. This is the same as for any other RMI stubs. There is an extra step that must be performed for Activatable objects: the activation server `rmid` must be able to reconstruct a copy of the *service* (the client must be able to reconstruct a copy of the services's *stub*). This means that `rmid` must have access to the class files of the service, either from an HTTP server or from the file system. In the example, the `codebase` property in the `ActivationDesc` is set to an HTTP url, so the class files for the service must be accessible to an HTTP server. The server need not be on the same machine as the service.

Before starting the server, an `rmid` process must be set running on the same machine as the service. An HTTP server must be running on a machine as specified by the `codebase` property on the service. The server can then be started. This will register the service with `rmid`, and will copy a stub object up to any lookup services that are found. The server can then terminate.

While the service remains registered with lookup services, clients can download its RMI stub. The service will be created on demand by `rmid`. You only need to run the server *once*, since `rmid` keeps information about the service in it own log files.

2.4 Security

Note that the JVM for the service will be created by `rmid`, and will be running in the same environment as `rmid`. In particular, such things as the current directory for the service will be the same as for `rmid`, not from where the server ran. Similarly, the user id for the service will be the user id of `rmid`. This is a potential security problem in multi-user systems. For example, any user on a Unix system could write a service which attempts to read the shadow password file on the system, as an activatable service. Once registered with `rmid`, this same user could write a client that calls the appropriate methods on the service. If `rmid` is running in privileged mode, owned by the superuser of the system, then the service will run in that same mode and will happily read any file in the entire file system! For safety, `rmid` should probably be run using the user id `nobody`, much like the recommendations for HTTP servers.

An additional level of security can be added by specifying a security policy file to `rmid` by

```
rmid -J-Djava.security.policy="..."
```

I haven't got this to work properly yet :-(

2.5 Non-lazy Services

The type of service discussed above are "lazy" services, activated on demand when their methods are called. This reduces memory use, at the expense of starting up a new JVM when required. Some services need to be continuously alive, but can still benefit from the log mechanism of `rmid`. If `rmid` crashes and is restarted, or the machine is rebooted and `rmid` restarts, then it is able to use its log files to restart any "active" services registered with it, as well as restore "lazy" services on demand. This can avoid messing around with boot configuration files, by making services non-lazy and just ensuring that `rmid` is started on reboot.

2.6 Maintaining State

An activatable object is created afresh each time a method is called on it, using its two argument constructor. The default action, calling `super(id, 0)` will result in the object being created in the same state on each activation. However, method calls on objects (apart from `get...()` methods) usually result in a change of state of the object. Activatable objects will need some way of reflecting this change on each activation, and this is typically done by saving and restoring state using a disk file.

When an object is activated, one of the parameters passed to it is a `MarshalledObject` instance. This is the same object that was passed to the activation system in the `ActivationDesc` parameter to `Activation.register()`. This object does not change between different activations. So it cannot hold changing state, but only data which is fixed for all activations. A simple use for it is to hold the name of a file that can be used for state. Then on each activation, the object can restore state by reading stored information. On each subsequent method call that changes state, the information in the file can be overwritten.

The "mutable file classifier" was discussed in an earlier chapter, which could be sent `addType()` and `removeType()` messages. It begins with a given set of MIME type/file extension mappings. State here is very simple, just storing all the file extensions and their corresponding MIME type in a `Map`. If we turn this into an activatable object, we store the state by just storing the map. This can be saved to disk using

`ObjectOutputStream.writeObject()`, and retrieved by `ObjectInputStream.readObject()`. More complex cases might need more complex storage methods.

The very first time a mutable file classifier starts on a particular host, it should build its initial state file. There are a variety of methods that could be used. For example, if the state file does not exist, then the first activation could detect this and construct the initial state at that time. Alternatively, a method such as `init()` could be defined, to be called once after the object has been registered with the activation system.

The "normal" way of instantiating an object - through a constructor - doesn't work too well with activatable objects. If a constructor for a class doesn't start by calling another constructor by `this(...)` or `super(...)`, then the no argument superclass constructor `super()` is called. But the class `Activatable` doesn't have a no-args constructor. So you can't subclass from `Activatable` *and* have a constructor such as `FileClassifierMutable(String stateFile)` that doesn't use the activation system. You can avoid this by not inheriting from `Activatable`, and register explicitly with the activation system by *e.g.*

```
public FileClassifierMutable(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException {
        Activatable.exportObject(this, id, 0);
        // continue with instantiation
```

Nevertheless, this is a bit clumsy in use: you create an object solely to build up initial state, and then discard it since the activation system will recreate it on demand.

The technique adopted in this example is to create initial state if the attempt to restore state from the state file fails for any reason as the object is activated. This is done in the method `restoreMap()`, called from the constructor `FileClassifierMutable(ActivationID id, MarshalledObject data)`. The name of the file is extracted from the marshalled object passed in as parameter.

```
package activation;

import java.io.*;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;

import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.EventRegistration;
import java.rmi.RemoteException;
import net.jini.core.event.UnknownEventException ;

import javax.swing.event.EventListenerList;

import common.MIMEType;
import common.MutableFileClassifier;
import mutable.RemoteFileClassifier;
import java.util.Map;
import java.util.HashMap;

/**
 * FileClassifierMutable.java
 *
 *
 * Created: Wed Dec 27 1999
 *
```



```
 * @author Jan Newmarch
 * @version 1.0
 */

public class FileClassifierMutable extends  Activatable
                                implements RemoteFileClassifier {

    /**
     * Map of String extensions to MIME types
     */
    protected Map map = new HashMap();

    /**
     * Permanent storage for the map while inactive
     */
    protected String mapFile;

    /**
     * Listeners for change events
     */
    protected EventListenerList listenerList = new EventListenerList();

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        System.out.println("Called with " + fileName);

        MIMEType type;
        String fileExtension;
        int dotIndex = fileName.lastIndexOf('.');

        if (dotIndex == -1 || dotIndex + 1 == fileName.length()) {
            // can't find suitable suffix
            return null;
        }

        fileExtension= fileName.substring(dotIndex + 1);
        type = (MIMEType) map.get(fileExtension);
        return type;

    }

    public void addType(String suffix, MIMEType type)
        throws java.rmi.RemoteException {
        map.put(suffix, type);
        fireNotify(MutableFileClassifier.ADD_TYPE);
        saveMap();
    }

    public void removeMIMEType(String suffix, MIMEType type)
        throws java.rmi.RemoteException {
        if (map.remove(suffix) != null) {
            fireNotify(MutableFileClassifier.REMOVE_TYPE);
            saveMap();
        }
    }
```

```
public EventRegistration addRemoteListener(RemoteEventListener listener)
    throws java.rmi.RemoteException {
    listenerList.add(RemoteEventListener.class, listener);

    return new EventRegistration(0, this, null, 0);
}

// Notify all listeners that have registered interest for
// notification on this event type.  The event instance
// is lazily created using the parameters passed into
// the fire method.

protected void fireNotify(long eventID) {
    RemoteEvent remoteEvent = null;

    // Guaranteed to return a non-null array
    Object[] listeners = listenerList.getListenerList();

    // Process the listeners last to first, notifying
    // those that are interested in this event
    for (int i = listeners.length - 2; i >= 0; i -= 2) {
        if (listeners[i] == RemoteEventListener.class) {
            RemoteEventListener listener = (RemoteEventListener) listeners[i+1];
            if (remoteEvent == null) {
                remoteEvent = new RemoteEvent(this, eventID,
                                              0L, null);
            }
            try {
                listener.notify(remoteEvent);
            } catch(UnknownEventException e) {
                e.printStackTrace();
            } catch(RemoteException e) {
                e.printStackTrace();
            }
        }
    }
}

/**
 * Restore map from file.
 * Install default map if any errors occur
 */
public void restoreMap() {
    try {
        FileInputStream istream = new FileInputStream(mapFile);
        ObjectInputStream p = new ObjectInputStream(istream);
        map = (Map) p.readObject();

        istream.close();
    } catch(Exception e) {
        e.printStackTrace();
        // restoration of state failed, so
        // load a predefined set of MIME type mappings
        map.put("gif", new MIMEType("image", "gif"));
```



```
        map.put("jpeg", new MIMEType("image", "jpeg"));
        map.put("mpg", new MIMEType("video", "mpeg"));
        map.put("txt", new MIMEType("text", "plain"));
        map.put("html", new MIMEType("text", "html"));

        this.mapFile = mapFile;
        saveMap();
    }
}

/**
 * Save map to file.
 */
public void saveMap() {
    try {
        FileOutputStream ostream = new FileOutputStream(mapFile);
        ObjectOutputStream p = new ObjectOutputStream(ostream);
        p.writeObject(map);
        p.flush();
        ostream.close();
    } catch(Exception e) {
        e.printStackTrace();
    }
}

public FileClassifierMutable(ActivationID id, MarshalledObject data)
    throws java.rmi.RemoteException {
    super(id, 0);
    try {
        mapFile = (String) data.get();
    } catch(Exception e) {
        e.printStackTrace();
    }
    restoreMap();
}
} // FileClassifierMutable
```

The difference between the server for this service and the last one, is that we now have to prepare a marshalled object for the state file, and register this with the activation system. Here the filename is hard-coded, but it could be given as a command-line argument (like services such as `reggie` do).

```
package activation;

import mutable.RemoteFileClassifier;

import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
import java.rmi.RMISecurityManager;
```

```
import java.rmi.MarshalledObject;
import java.rmi.activation.ActivationDesc;
import java.rmi.activation.ActivationGroupDesc;
import java.rmi.activation.ActivationGroupDesc.CommandEnvironment;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationGroup;
import java.rmi.activation.ActivationGroupID;

import java.util.Properties;

import java.rmi.activation.UnknownGroupException;
import java.rmi.activation.ActivationException;
import java.rmi.RemoteException;

/**
 * FileClassifierServerMutable.java
 *
 *
 * Created: Wed Dec 22 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class FileClassifierServerMutable implements DiscoveryListener {

    static final protected String SECURITY_POLICY_FILE =
        "/home/jan/projects/jini/doc/policy.all";
    // Don't forget the trailing '/'!
    static final protected String CODEBASE = "http://localhost/classes/";
    static final protected String LOG_FILE = "/tmp/file_classifier";

    // protected FileClassifierImpl impl;
    protected RemoteFileClassifier stub;

    public static void main(String argv[]) {
        new FileClassifierServerMutable(argv);
        // stick around while lookup services are found
        try {
            Thread.sleep(10000L);
        } catch(InterruptedException e) {
            // do nothing
        }
        // the server doesn't need to exist anymore
        System.exit(0);
    }

    public FileClassifierServerMutable(String[] argv) {
        // install suitable security manager
        System.setSecurityManager(new RMISecurityManager());

        // Install an activation group
        Properties props = new Properties();
        props.put("java.security.policy",
```



```
                SECURITY_POLICY_FILE);
        ActivationGroupDesc.CommandEnvironment ace = null;
        ActivationGroupDesc group = new ActivationGroupDesc(props, ace);
        ActivationGroupID groupID = null;
        try {
            groupID = ActivationGroup.getSystem().registerGroup(group);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }

        try {
            ActivationGroup.createGroup(groupID, group, 0);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }

        String codebase = CODEBASE;
        MarshalledObject data = null;
        try {
            data = new MarshalledObject(LOG_FILE);
        } catch(java.io.IOException e) {
            e.printStackTrace();
        }

        ActivationDesc desc = null;
        try {
            desc = new ActivationDesc("activation.FileClassifierMutable",
                                                codebase, data);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }
System.out.println("Group ID " + ActivationGroup.currentGroupID().toString());

        try {
            stub = (RemoteFileClassifier) Activatable.register(desc);
        } catch(UnknownGroupException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        }

        LookupDiscovery discover = null;
        try {
```

```
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();
        RemoteFileClassifier service;

        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];

            // export the proxy service
            ServiceItem item = new ServiceItem(null,
                                               stub,
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.print("Register exception: ");
                e.printStackTrace();
                // System.exit(2);
                continue;
            }
            try {
                System.out.println("service registered at " +
                                   registrar.getLocator().getHost());
            } catch(Exception e) {
            }
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }
} // FileClassifierServerMutable
```

This example used a simple way to store state. Sun uses a far more complex system in many of its services such as `reggie`. This is a "reliable log", in package `com.sun.jini.reliableLog`. However, this package is not a part of standard Jini, so may change or even be removed in later versions of Jini. However, there is nothing to stop you using it if you need a robust storage mechanism.

3. LeaseRenewalService

Activatable objects are one example of services that are not continuously alive. Mobile services, such as those that will exist on mobile phones are another. These services will be brought to life on demand (as activatable objects), or will join the network on occasions. These services raise a number of problems, and one was skirted around in the last section: how to renew leases when the object is not alive?

Activatable objects are brought back to life when methods are invoked on them. The expiration of a lease does not cause any methods to be invoked. There is no "lease expiration event" generated that could cause a listener method to be invoked, either. If a server is alive, then it can use a `LeaseRenewalManager` to keep leases alive, but firstly the renewal manager works by sleeping and waking up just in time to renew the leases, and secondly, if the server exits then no `LeaseRenewalManager` will continue to run.

Jini 1.1 supplies a lease renewal *service* that partly avoids these problems. Since it runs as a service, it has an independent existence that does not depend on the server for any other service. It can act like a `LeaseRenewalManager` in keeping track of leases registered with it, and renewing them as needed. In general, it can keep leases alive without waking the service itself, which can slumber till activated by clients calling methods.

There is a small hiccup in this: how long should the `LeaseRenewalService` keep renewing leases for a service? The `LeaseRenewalManager` utility has a simple solution: keep renewing while the server for that service is alive. If the server dies, taking down a service, then it will also take down the `LeaseRenewalManager`, so leases will expire as expected after an interval.

But this mechanism won't work for `LeaseRenewalService` because the managed service can disappear without the `LeaseRenewalService` knowing about it. So the lease renewal must be done on a leased basis itself! The `LeaseRenewalService` will renew leases for a service only for a particular amount of time, specified by a lease. The service will still have to renew its lease, even though it is with a `LeaseRenewalService` instead of a bunch of lookup services! The lease granted by this service should be of much longer duration than those granted by the lookup services for this to be of value.

Activatable services can only be woken by calling one of their methods. The `LeaseRenewalService` accomplishes this by generating renewal events and calling a `notify()` method on a listener. If the listener is the activatable object, this will wake it up so that it can perform the renewal. If the `rmid` process managing the service has died or is unavailable, then the event will not be delivered and the `LeaseRenewalService` can remove this service from its renewal list.

This is not quite satisfactory for other types of "dormant" services such as might exist on mobile phones, since there is no equivalent of `rmid` to handle activation. Instead, the mobile phone service might say that it will connect once a day and renew the lease, as long as the `LeaseRenewalService` agrees to keep the lease for at least a day. This is still "negotiable", in that the service asks for a duration and the `LeaseRenewalService` replies with a value that might not be so long. Still, it should be better than dealing with the lookup services.

3.1 Norm

Jini 1.1 supplies an implementation of `LeaseRenewalService` called `norm`. This is a non-lazy Activatable service, which requires `rmid` to be running. This is run by

```
java -jar [setup_jvm_options] executable_jar_file
          codebase_arg  norm_policy_file_arg
          log_directory_arg
          [groups] [server_jvm] [server_jvm_args]
```

as in

```
java -jar \
        -Djava.security.policy=/files/jini1_1/example/txn/policy.all \
        /files/jini1_1/lib/norm.jar \
        http://`hostname`:8080/norm-dl.jar \
        /files/jini1_1/example/books/policy.all /tmp/norm_log
```

The first security file defines the policy that will be used for the server startup. The file `norm.jar` contains the class files for the `norm` service. This exports RMI stubs and the class definitions for these are in `norm-dl.jar`. The second security file defines the policy file that will be used in execution of the `LeaseRenewalService` methods. Finally, the log file is used to keep state, so that it can keep track of the leases it is managing.

The `norm` service will maintain a set of leases for a period of upto 2 hours. The `reggie` lookup service only grants leases for 5 minutes, so that using this service increases the amount of time between renewing leases by a factor of over twenty.

3.2 Using the LeaseRenewalService

The `norm` service exports an object of type `LeaseRenewalService` which is defined by the interface

```
package net.jini.lease;

public interface LeaseRenewalService {
    LeaseRenewalSet createLeaseRenewalSet(long leaseDuration)
        throws java.rmi.RemoteException;
}
```

The `leaseDuration` is a requested value in milliseconds for the lease service to manage a set of leases. The lease service creates a lease for this request, and in order for it to continue to manage the set beyond the lease's expiry, the lease must be renewed before expiration. Since the service may be inactive around the time of expiry, the `LeaseRenewalSet` can be asked to register a listener object that will receive an event containing the lease. This will activate a dormant listener so that it can renew the lease in time. If the lease for the `LeaseRenewalSet` is allowed to lapse, then eventually all the leases for the services it was managing will also expire, making the services unavailable.

The `LeaseRenewalSet` returned from `createLeaseRenewalSet` has interface including

```
package net.jini.lease;

public interface LeaseRenewalSet {
    public void renewFor(Lease leaseToRenew,
                         long membershipDuration)
                throws RemoteException;
    public EventRegistration setExpirationWarningListener(
                         RemoteEventListener listener,
                         long minWarning,
                         MarshalledObject handback)
                throws RemoteException;
    ....
}
```

The `renewFor()` method adds a new lease to the set being looked after. The `LeaseRenewalSet` will keep renewing the lease until either the requested `membershipDuration` expires or the lease on the whole `LeaseRenewalSet` expires (or an exception happens, like a lease being refused).

Setting an expiration warning listener means that its `notify()` method will be called at least `minWarning` millseconds before the lease for the set expires. The event argument to this will actually be an `ExpirationWarningEvent`

```
package net.jini.lease;

public class ExpirationWarningEvent extends RemoteEvent {
    Lease getLease();
}
```

This allows the listener to get the lease for the `LeaseRenewalSet`, and (probably) renew it. A simple activatable class that can renew the lease is

```
package activation;

import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.lease.Lease;
import net.jini.lease.ExpirationWarningEvent;

public class RenewLease extends Activatable
    implements RemoteEventListener {

    public RenewLease(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException {
        super(id, 0);
    }

    public void notify(RemoteEvent evt) {
        System.out.println("expiring... " + evt.toString());
        ExpirationWarningEvent eevt = (ExpirationWarningEvent) evt;
        Lease lease = eevt.getLease();
        try {
            // This is short, for testing. Try 2+ hours
            lease.renew(20000L);
        } catch(Exception e) {
            e.printStackTrace();
        }
        System.out.println("Lease renewed for " +
                           (lease.getExpiration() -
                            System.currentTimeMillis()));
    }
}
```

The server will need to register the service and export it as an activatable object. This is done in exactly the same way as in the first example of this chapter. The changes are

1. It will also need to register the lease listener with the activation system as an activatable object
2. It will need to find a `LeaseRenewalService` from a lookup service
3. It will need to register all leases from services with the `LeaseRenewalService`. Since it may find services before it finds the renewal service, it will need to keep a list of services found before finding the service, in order to register them with it

```
package activation;

import option3.RemoteFileClassifier;

import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.lease.Lease;
import net.jini.lease.LeaseRenewalService;
import net.jini.lease.LeaseRenewalSet;
import java.rmi.RMISecurityManager;
import java.rmi.MarshalledObject;
import java.rmi.activation.ActivationDesc;
import java.rmi.activation.ActivationGroupDesc;
import java.rmi.activation.ActivationGroupDesc.CommandEnvironment;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationGroup;
import java.rmi.activation.ActivationGroupID;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;

import java.util.Properties;
import java.util.Vector;

import java.rmi.activation.UnknownGroupException;
import java.rmi.activation.ActivationException;
import java.rmi.RemoteException;

/**
 * FileClassifierServer.java
 *
 *
 * Created: Wed Dec 22 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class FileClassifierServerLease
    implements DiscoveryListener {
```



```
    static final protected String SECURITY_POLICY_FILE =
        "/home/jan/projects/jini/doc/policy.all";
    // Don't forget the trailing '/'!
    static final protected String CODEBASE = "http://localhost/classes/";

    protected RemoteFileClassifier stub;

    protected RemoteEventListener leaseStub;

    // Lease renewal management
    protected LeaseRenewalSet leaseRenewalSet = null;

    // List of leases not yet managed by a LeaseRenewalService
    protected Vector leases = new Vector();

    public static void main(String argv[]) {
        new FileClassifierServerLease(argv);
        // stick around while lookup services are found
        try {
            Thread.sleep(10000L);
        } catch(InterruptedException e) {
            // do nothing
        }
        // the server doesn't need to exist anymore
        System.exit(0);
    }

    public FileClassifierServerLease(String[] argv) {
        // install suitable security manager
        System.setSecurityManager(new RMISecurityManager());

        // Install an activation group
        Properties props = new Properties();
        props.put("java.security.policy",
                  SECURITY_POLICY_FILE);
        ActivationGroupDesc.CommandEnvironment ace = null;
        ActivationGroupDesc group = new ActivationGroupDesc(props, ace);
        ActivationGroupID groupID = null;
        try {
            groupID = ActivationGroup.getSystem().registerGroup(group);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }

        try {
            ActivationGroup.createGroup(groupID, group, 0);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }
```

```
        String codebase = CODEBASE;
        MarshalledObject data = null;
        ActivationDesc desc = null;
        ActivationDesc descLease = null;
        try {
            desc = new ActivationDesc("activation.FileClassifierImpl",
                                      codebase, data);
            descLease = new ActivationDesc("activation.RenewLease",
                                      codebase, data);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }

        try {
            stub = (RemoteFileClassifier) Activatable.register(desc);
            leaseStub = (RemoteEventListener) Activatable.register(descLease);
        } catch(UnknownGroupException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        }

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();
        RemoteFileClassifier service;

        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];

            // export the proxy service
            ServiceItem item = new ServiceItem(null,
                                               stub,
                                               null);
            ServiceRegistration reg = null;
            try {
```



```
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.print("Register exception: ");
                e.printStackTrace();
                // System.exit(2);
                continue;
            }
            try {
                System.out.println("service registered at " +
                                   registrar.getLocator().getHost());
            } catch(Exception e) {
            }

            Lease lease = reg.getLease();
            // if we have a lease renewal manager, use it
            if (leaseRenewalSet != null) {
                try {
                    leaseRenewalSet.renewFor(lease, Lease.FOREVER);
                } catch(RemoteException e) {
                    e.printStackTrace();
                }
            } else {
                // add to the list of unmanaged leases
                leases.add(lease);
                // see if this lookup service has a lease renewal manager
                findLeaseService(registrar);
            }
        }
    }

    public void findLeaseService(ServiceRegistrar registrar) {
        System.out.println("Trying to find a lease service");
        Class[] classes = {LeaseRenewalService.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
        LeaseRenewalService leaseService = null;
        try {
            leaseService = (LeaseRenewalService) registrar.lookup(template);
        } catch(RemoteException e) {
            e.printStackTrace();
            return;
        }
        if (leaseService == null) {
            System.out.println("No lease service found");
            return;
        }
        try {
            // This time is unrealistically small - try 10000000L
            leaseRenewalSet = leaseService.createLeaseRenewalSet(20000);
            System.out.println("Found a lease service");
            // register a timeout listener
            leaseRenewalSet.setExpirationWarningListener(leaseStub, 5000,
                                                         null);
            // manage all the leases found so far
```

```
            for (int n = 0; n < leases.size(); n++) {
                Lease ll = (Lease) leases.elementAt(n);
                leaseRenewalSet.renewFor(ll, Lease.FOREVER);
            }
            leases = null;
        } catch(RemoteException e) {
            e.printStackTrace();
        }
        Lease renewalLease = leaseRenewalSet.getRenewalSetLease();
        System.out.println("Lease expires in " +
                           (renewalLease.getExpiration() -
                            System.currentTimeMillis()));
    }

    public void discarded(DiscoveryEvent evt) {

    }
} // FileClassifierServerLease
```

4. LookupDiscoveryService

It is easy enough for a server to discover all of the lookup services within reach at the time it is started, using `LookupDiscovery`. While the server continues to stay alive, any new lookup services that start will also be found by `LookupDiscovery`. But if the server terminates, which it will for activable services, then these extra lookup services will probably never be found. This will result in the service not being registered with them, which could mean in turn that clients may not find it. This is analogous to leases not being renewed if the server terminates

Jini 1.1 supplies a service, the `LookupDiscoveryService`, that can be used to continuously monitor the state of lookup services. It will monitor these on behalf of a service that will most likely want to register with each new lookup service as it starts. If the service is an activatable one, the the server that would have done this will have terminated, as its role would have just been to register the service with `rmid`.

When there is a change to lookup services, the `LookupDiscoveryService` need to notify an object about this, by sending it a remote event (actually of type `RemoteDiscoveryEvent`). But again, we do not want to have a process lying around waiting for such notification, so the listener object will probably also be an activatable object.

The `LookupDiscoveryService` interface has specification

```
package
public interface LookupDiscoveryService {
    LookupDiscoveryRegistration register(String[] groups,
                                         LookupLocator[] locators,
                                         RemoteEventListener listener,
                                         MarshalledObject handback,
                                         long leaseDuration);
}
```


Calling the `register()` method will begin a multicast search for the groups, and unicast lookup for the locators. The registration is leased, and will need to be renewed before expiry (a lease renewal service can be used for this). Note that the listener *cannot* be `null` - this is simple sanity checking, for if the listener was `null` then the service could never do anything useful!

A lookup service in one of the groups can start or terminate. Or it can change its group membership in such a way that it now does (or doesn't) meet the group criteria. A lookup service in the locators list can also start or stop. These will generate `RemoteDiscoveryEvent` events can call the `notify()` method of the listener. The event interface includes

```
package net.jini.discovery;

public interface RemoteDiscoveryEvent {
    ServiceRegistrar[] getRegistrars();
    boolean isDiscarded();
    ...
}
```

The list of registrars is the set that triggered the event. The `isDiscarded()` method is used to check if it is a "discovered" lookup service or a "discarded" lookup service. An initial event is not posted when the listener is registered: the set of lookup services that are initially found can be retrieved from the `LookupDiscoveryRegistration` object returned from the `register()` method, by its `getRegistrars()`.

4.1 Fiddler

The Jini 1.1 release includes an implementation of the lookup discovery service, called `fiddler`. This is started in a similar way to other services such as `reggie`. It is also a non-lazy activatable service.

```
java -jar [setup_jvm_options] executable_jar_file
          codebase_arg  fiddler_policy_file_arg
      log_directory_arg
          [groups and
    locators] [server_jvm] [server_jvm_args]
```

as in

```
java -jar \
        -Djava.security.policy=/files/jini1_1/example/txn/policy.all \
        /files/jini1_1/lib/fiddler.jar \
        http://`hostname`:8080/norm-dl.jar \
        /files/jini1_1/example/books/policy.all /tmp/fiddler_log
```

4.2 Using the LookupDiscoveryService

WARNING: the program in this section does not yet work, I think due to a bug in Jini 1.1. An activatable service can make use of a lease renewal service to look after the leases for lookup services discovered. How it finds these lookup services can be by means of a lookup discovery service. The logic to manage this is a little tricky. For this to be of any use, the service itself must be a transient service, such as an activatable one. While lease management can be done by the lease renewal service, the lease renewal set will also be leased and will need to be renewed on occasions. The lease renewal service can call an activatable `RenewLease` object to do this, as in the last section.

The lookup discovery service is also a leased service - it will only report changes to lookup services while its *own* lease is current. So the lease from this service will have to be managed by the lease renewal service, in addition to the leases for any lookup services discovered.

The primary purpose of the lookup discovery service is to call the `notify()` method of some object when information about lookup services changes. This object should also be an activatable object. This object (here a `DiscoveryChange` object) will need to know the service that it is registering with a new lookup service, and will also need to know the lease renewal service to register the new lease with it. These two objects can be passed in an array of two objects as the `MarshalledObject` to the activation constructor. The class itself can be implemented as

```
package activation;

import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.lease.Lease;
import net.jini.lease.ExpirationWarningEvent;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.lease.LeaseRenewalSet;
import net.jini.discovery.RemoteDiscoveryEvent;
import java.rmi.RemoteException;
import  net.jini.discovery.LookupUnmarshalException;

import option3.RemoteFileClassifier;

public class DiscoveryChange extends Activatable
    implements RemoteEventListener {

    protected LeaseRenewalSet leaseRenewalSet;
    protected RemoteFileClassifier service;

    public DiscoveryChange(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException {
        super(id, 0);
        Object[] objs = null;
        try {
            objs = (Object []) data.get();
        } catch(ClassNotFoundException e) {
            e.printStackTrace();
        } catch(java.io.IOException e) {
            e.printStackTrace();
        }
        service = (RemoteFileClassifier) objs[0];
        leaseRenewalSet= (LeaseRenewalSet) objs[1];
    }

    public void notify(RemoteEvent evt) {
        System.out.println("lookups changing... " + evt.toString());
```



```
        RemoteDiscoveryEvent revt = (RemoteDiscoveryEvent) evt;

        if (! revt.isDiscarded()) {
            // The event is a discovery event
            ServiceItem item = new ServiceItem(null, service, null);
            ServiceRegistrar[] registrars = null;
            try {
                registrars = revt.getRegistrars();
            } catch(LookupUnmarshalException e) {
                e.printStackTrace();
                return;
            }
            for (int n = 0; n < registrars.length; n++) {
                ServiceRegistrar registrar = registrars[n];

                ServiceRegistration reg = null;
                try {
                    reg = registrar.register(item, Lease.FOREVER);
                    leaseRenewalSet.renewFor(reg.getLease(), Lease.FOREVER);
                } catch(java.rmi.RemoteException e) {
                    System.err.println("Register exception: " + e.toString());
                }
            }
        }
    }
}
```

The server must install an activation group, and then find activation proxies for the service itself and also for our lease renewal object. After this, it can use a `ClientLookupManager` to find the lease service, and register our lease renewal object with it. Now that it has a proxy for the ervice object, and also a lease renewal service, it can create the marshalled data for the lookup discovery service and registert this with `rmid`. Now we can find the lookup discovery service, and register our discovery change listener `DiscoveryChange` with it. At the same time, we have to register the service with all the lookup services the lookup discovery service finds on initialisation. This all leads to

```
package activation;

import option3.RemoteFileClassifier;

import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.LookupDiscoveryService;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.discovery.LookupDiscoveryRegistration;
import net.jini.discovery.LookupUnmarshalException;

import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.lease.Lease;
```

```
import net.jini.lease.LeaseRenewalService;
import net.jini.lease.LeaseRenewalSet;
import net.jini.lease.LeaseRenewalManager;

import net.jini.lookup.ClientLookupManager;

import java.rmi.RMISecurityManager;
import java.rmi.activation.ActivationDesc;
import java.rmi.activation.ActivationGroupDesc;
import java.rmi.activation.ActivationGroupDesc.CommandEnvironment;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationGroup;
import java.rmi.activation.ActivationGroupID;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;
import java.rmi.activation.UnknownGroupException;
import java.rmi.activation.ActivationException;
import java.rmi.RemoteException;

import java.util.Properties;
import java.util.Vector;



/**
 * FileClassifierServerDiscovery.java
 *
 *
 * Created: Wed Dec 22 1999
 *
 * @author Jan Newmarch
 * @version 1.0
 */

public class FileClassifierServerDiscovery
    /* implements DiscoveryListener */ {
    private static final long WAITFOR = 10000L;

    static final protected String SECURITY_POLICY_FILE =
        "/home/jan/projects/jini/doc/policy.all";
    // Don't forget the trailing '/'!
    static final protected String CODEBASE = "http://localhost/classes/";

    protected RemoteFileClassifier serviceStub;

    protected RemoteEventListener leaseStub,
                                  discoveryStub;

    // Services
    protected LookupDiscoveryService discoveryService = null;
    protected LeaseRenewalService leaseService = null;

    // Lease renewal management
    protected LeaseRenewalSet leaseRenewalSet = null;

    // List of leases not yet managed by a LeaseRenewalService
    protected Vector leases = new Vector();

    protected ClientLookupManager clientMgr = null;

    public static void main(String argv[]) {
        new FileClassifierServerDiscovery();
```



```
        // stick around while lookup services are found
        try {
            Thread.sleep(20000L);
        } catch(InterruptedException e) {
            // do nothing
        }
        // the server doesn't need to exist anymore
        System.exit(0);
    }

    public FileClassifierServerDiscovery() {
        // install suitable security manager
        System.setSecurityManager(new RMISecurityManager());

        installActivationGroup();

        serviceStub = (RemoteFileClassifier)
                      registerWithActivation("activation.FileClassifierImpl", null);

        leaseStub = (RemoteEventListener)
                      registerWithActivation("activation.RenewLease", null);

        initClientLookupManager();

        findLeaseService();

        // the discovery change listener needs to know the service and the lease service
        Object[] discoveryInfo = {serviceStub, leaseRenewalSet};
        MarshalledObject discoveryData = null;
        try {
            discoveryData = new MarshalledObject(discoveryInfo);
        } catch(java.io.IOException e) {
            e.printStackTrace();
        }
        discoveryStub = (RemoteEventListener)
                        registerWithActivation("activation.DiscoveryChange",
                                               discoveryData);

        findDiscoveryService();

    }

    public void installActivationGroup() {

        Properties props = new Properties();
        props.put("java.security.policy",
                  SECURITY_POLICY_FILE);
        ActivationGroupDesc.CommandEnvironment ace = null;
        ActivationGroupDesc group = new ActivationGroupDesc(props, ace);
        ActivationGroupID groupID = null;
        try {
            groupID = ActivationGroup.getSystem().registerGroup(group);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }

        try {
            ActivationGroup.createGroup(groupID, group, 0);
```

```
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public Object registerWithActivation(String className, MarshalledObject data) {
        String codebase = CODEBASE;
        ActivationDesc desc = null;
        Object stub = null;

        try {
            desc = new ActivationDesc(className,
                                      codebase, data);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }

        try {
            stub = Activatable.register(desc);
        } catch(UnknownGroupException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        }
        return stub;
    }

    public void initClientLookupManager() {
        LookupDiscoveryManager lookupDiscoveryMgr = null;
        try {
            lookupDiscoveryMgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            clientMgr = new ClientLookupManager(lookupDiscoveryMgr,
                                                new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void findLeaseService() {
        leaseService = (LeaseRenewalService) findService(LeaseRenewalService.class);
        if (leaseService == null) {
            System.out.println("Lease service null");
        }
        try {
            leaseRenewalSet = leaseService.createLeaseRenewalSet(20000);
            leaseRenewalSet.setExpirationWarningListener(leaseStub, 5000,
                                                         null);
        } catch(RemoteException e) {
            e.printStackTrace();
        }
    }
```



```
    public void findDiscoveryService() {
        discoveryService = (LookupDiscoveryService) findService(LookupDiscoveryService.class);
        if (discoveryService == null) {
            System.out.println("Discovery service null");
        }
        LookupDiscoveryRegistration registration = null;
        try {
            registration =
                discoveryService.register(LookupDiscovery.ALL_GROUPS,
                                          null,
                                          discoveryStub,
                                          null,
                                          Lease.FOREVER);
        } catch(RemoteException e) {
            e.printStackTrace();
        }
        // manage the lease for the lookup discovery service
        try {
            leaseRenewalSet.renewFor(registration.getLease(), Lease.FOREVER);
        } catch(RemoteException e) {
            e.printStackTrace();
        }

        // register with the lookup services already found
        ServiceItem item = new ServiceItem(null, serviceStub, null);
        ServiceRegistrar[] registrars = null;
        try {
            registrars = registration.getRegistrars();
        } catch(RemoteException e) {
            e.printStackTrace();
            return;
        } catch(LookupUnmarshalException e) {
            e.printStackTrace();
            return;
        }

        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
                leaseRenewalSet.renewFor(reg.getLease(), Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.println("Register exception: " + e.toString());
            }
        }
    }

    public Object findService(Class cls) {
        Class [] classes = new Class[] {cls};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        ServiceItem item = null;
        try {
            item = clientMgr.lookup(template,
                                    null, /* no filter */
                                    WAITFOR /* timeout */);
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
```

```
        }
        if (item == null) {
            // couldn't find a service in time
            System.out.println("No service found for " + cls.toString());
            return null;
        }
        return item.service;
    }
} // FileClassifierServerDiscovery
```

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Architecture

Contents

1. Threads
    1. LookupDiscovery threads
    2. LeaseRenewalManager threads

This chapter looks at internal structures of Jini. This is heavily based on the 1.0 implementation by Sun, and may not be valid for any future implementations either by Sun or other vendors.

1. Threads

Jini uses many threads in its internal workings. The knowledge of these is not important to using Jini, but some may find it valuable to know what is going on under the hood.

1.1 LookupDiscovery threads

A multicast request is made by creating a new `LookupDiscovery` object with a non-empty set of groups, or by calling `setGroups()`. This must broadcast the service, looking for service locators. This is done using a thread of type `Requestor`. This creates a `MulticastSocket`, sets its time-to-live field and sends out a number (currently eight) of announcements

A class `ResponseListener` is used to handle replies. This runs as its own thread. It listens on a socket for responses and adds them to a list of service locators. If a new locator is found, then it calls each `DiscoveryListener`. (The implementation is slightly more complex due to timing issues - see the source code.)

Both of these run as *daemon* threads. That means they run as background threads. A user thread is needed to keep the application alive: an application will terminate if there are only daemon threads alive.

1.2 LeaseRenewalManager threads

A `LeaseRenewalManager` uses a thread of class `RenewThread`. This thread looks after all aspects of renewal.

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Changes

1. Version 0.01, released 17 March, 1999

1. Added a section ‘‘What needs to be where?’’ to SimpleExample
2. Added section ‘‘Other Resources’’ to Jini
3. Added section ‘‘Uploading a Proxy’’ to SimpleExample
4. Added an early version of a chapter on ‘‘Security’’
5. Added section on ‘‘Convenience Classes’’ to Entry
6. Fixed Java source -> XML converter to change ’<’ to ’<’ rather than ’&<’

2. Version 0.02, released 22 March, 1999

1. Added Jini and RMI home pages to Resources in ‘‘Jini’’
2. All document source files plus Perl converters made available for downloading
3. Added sections on running the programs
4. Added header material to each chapter, so that first section in each chapter can have its own header
5. Update information added
6. Added reference to Roger Whitney’s lecture notes

3. Version 0.03, released 24 March, 1999

1. Replaced erroneous references to RMI with correct ones to sockets
2. Extended Entry information to include template matching
3. Added document location to Jini.xml
4. Added more resources to list of ‘‘Other Resources’’
5. Sections added on the information that can be obtained from various objects
6. Fixed error in assuming maxMatches == items.length in client lookup()
7. Added ‘‘Time to live’’ section to LookupDiscovery
8. Changed "href" attribute of "ulink" to "url" in conformance to DocBook DTD
9. Added initial version of chapter on Join Manager
10. Added initial version of chapter on Leasing

4. Version 0.04, released 29 March, 1999

1. Added section on information obtainable from JoinManager
2. Revised discussion on client search to emphasise that it should use an interface rather than superclass (Brian Jeltema, mailto:jeltema@richdist.east.sun.com)
3. Moved Lease before SimpleExamples because it is needed for proper examples, and modified examples to use LeaseRenewalManager
4. Removed more erroneous RMI references
5. Added some more figures
6. Added summary section to each chapter
7. Renamed UnicastRegistrar example to UnicastRegister and MulticastRegistrar as MulticastRegister
8. Added sections on service and client structure to Overview
9. Added section ‘‘Staying Alive’’ to Lookup Discovery and corrected/expanded time discussions in Simple Examples
10. Added contents table to each chapter

5. Version 0.05, released 7 April, 1999

1. Added initial version of ‘‘Architecture’’ chapter
2. Added section ‘‘What classes need to be where?’’ to Simple Examples section on option3 - now that I have a clearer idea on what rmiregistry requires!
3. Changed package structure of examples to fit logical structure rather than document structure
4. Fixed JoinManager example/discussion by adding in group information
5. Added missed subsection headers to SimpleExample
6. Removed sleep() from option3 server because RMI keeps it alive, and discussed this
7. Added section on Feedback to main page
8. Added disclaimer that TCP/IP is not strictly necessary for Jini
9. Security discussion added to ServiceRegistrar in ‘‘Client Search’’, for lookup() method
10. Fixed ‘‘class locations’’ and ‘‘running things’’ for option2
11. Fixed ‘‘class locations’’ and ‘‘running things’’ for option3
12. Added new chapter on ‘‘More Complex Examples’’ with three initial sections, ‘‘Where is the Code’’ and ‘‘Using threads’’ and ‘‘Reusing RMI Proxies’’
13.

6. Version 0.06, released 12 April, 1999

1. Added section on inexact matching
2. Added section on matching using a local service

7. Version 0.07, released 21 April, 1999

1. Fixed erroneous export instruction in SimpleExample option 3 server description (Robbert van den Beld, rbe@mms-dresden.de)
2. Fixed wording in Leasing to refer to lookup service rather than lookup locator (Chitrarasu Muthaiyan chitrarasu@cswl.com)
3. Split figures in Overview to show sequencing better (Stuart Remphrey Stuart.Remphrey@Aus.Sun.COM)
4. Replaced Class.forName("XXX") with XXX.class, simplifying code (JJ Larrea jlarrea@redtop.com)
5. Added images to docs.zip
6. Added chapter on Lego MindStorms

8. Version 0.08, released 4 June, 1999

1. Added more explanation as to why the lookup matching isn’t smarter, based on a mailing-list posting by Bob Scheiffler rws@east.sun.com
2. Replaced LookupDiscovery etc with JoinManager in MindStorms server
3. Version 7 cleaned up the code for option3 by giving an RMI object to the exported proxy rather than a name. Now cleaned up the description around this, since we no longer have calls to Naming(), etc. This also involved the removal of rmiregistry references, since we don’t need this anymore.
4. Made rcx files from Dario available on my HTTP server machine, so they get imported correctly into documents
5. Added Jini Community site www.jini.org to ‘‘Other Resources’’
6. Added section to ‘‘More Complex’’ on ‘‘Finding a service once only’’ to ensure uniqueness of services - so we only send commands to our robot once, rather than many times
7. Added new chapter on ‘‘Remote Events’’. It doesn’t cover leases yet
8. Added code and description to MindStorms for event handling back from robot

9. Version 0.09, released 11 June, 1999

1. Section added to ‘‘Lease’’ chapter about lease management on the lease granting side. This includes discussion on the landlord package.
2. Overdue maintenance on Makefile. I have no Windows version of the Makefile, though
3. Added section using MIMEType which uses the landlord lease system
4. Added classes.zip to distribution, so you don’t have to build your own to see the examples

10. Version 0.10, released 23 June, 1999

1. Found decent DocBook info, so removed some HTML/new tags from XML source and replaced them with proper DocBook tags. So far, cleaned "chapintro", "contents", "img" (not Figure yet)
2. Fixed broken link from Entryl.xml to Entry.xml in Overview.xml (Chris Garbers chris@babbelgamm.de)
3. New introductory section in Overview
4. New chapter on ‘‘Running Jini’’. Basically a collection of environmental problems and solutions. Much too much to be done on this chapter :-(
5. Swapped order of Enterprise Java Beans reference (Yu Liu, yuliu@dstc.monash.edu.au)
6. Removed unnecessary references to reggie.jar from compilation and running of examples
7. Fixed foo-landlord package diagram to show correct multiplicities, and added some dialog about it. More information about the structure and rationale of the package is added. Other tidy ups done (John McClain, john.mcclain@sun.com)
8. Pulled RMI-related stuff into its own chapter, as there are enough issues in that to justify it
9. Added new section on _not_ using RMI to RMI chapter
10. Major expansion to chapter on security issues - no longer so mickey-mouse
11. Added new resource of Benot Marchal’s Jini articles
12. Upgraded copyright license to the new Open Publication License

11. Version 0.11, released 27 July, 1999

1. Added new chapter on transactions
2. Added some more items to ‘‘Running Jini’’

12. Version 0.12, released 1 August, 1999

1. Added more UML diagrams throughout
2. New section ‘‘Partitioning an Application’’ added to Overview
3. New section ‘‘Support Services’’ added to Overview
4. New section on running reggie added to LookupDiscovery
5. More discussion added to Simple Examples on Remote versus RemoteException
6. Interfaces and common classes in Simple Examples moved to their own section before Client
7. Renamed "A Jini Tutorial" to "Jan Newmarch’s Guide to JINI Technologies", in line with proposed book title
8. Added chapter on CORBA and Jini
9. Changed title of chapter ‘‘Running Jini’’ to ‘‘TroubleShooting’’. Still needs more content :-(
10. Signed book contract, this is no longer a tutorial!

13. Version 1.00, released 31 August, 1999

1. Fixed broken links to Noel’s Nuggets and Eran’s timeservice Sebestian Voges (voges@netplace.de)
2. Changed all renewFor(..., FOREVER, ...) to renewUntil() Sebestian Voges (voges@netplace.de)
3. Fixed TestFileClassifier to handle null returns
4. Renamed "RMI" chapter to "Choices in Service Architecture", reworked content and extended it. Moved stuff out of "Simple Example" to "Choices" so that it just looks at one example only. Renamed "Some Simple Examples" to "A Simple Example"

14. Version 1.01, released 27 October, 1999

1. Revised MindStorms chapter because the RCXPort stuff has changed versions and interfaces
2. Completed MindStorms section on NQC compiler

15. Version 1.02, released 2 November, 1999

1. New chapter on user interfaces
2. New section: monitoring changes to services registered in lookup services

16. Version 1.03, released 17 December, 1999

1. Modified examples to use Jini 1.1 (alpha). The main changes are to the package name for LeaseRenewalManager, LeaseListener and LeaseListener; the JoinManager has changed interface enough to warrant "old" and "new" sections
2. Added chapter on Jini 1.1 ClientLookupManager

17. Version 2.00, released 22 December, 1999

1. Added security warning in using rmid
2. New chapter on Activation, including Jini 1.1. LeaseRenewalService and LookupDiscoveryService

18. Version 2.01

This file is Copyright (©) 1999 by Jan Newmarch (http://pandonia.canberra.edu.au) jan@ise.canberra.edu.au. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v0.4 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.