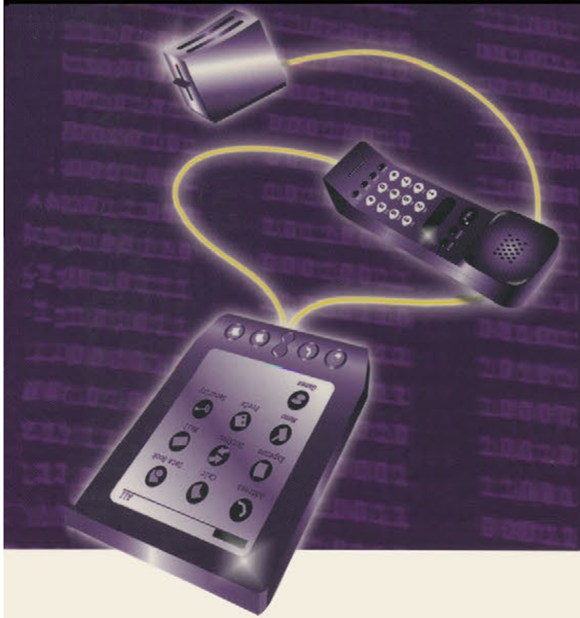
BOOKS FOR PROFESSIONALS BY PROFESSIONALS™



Jan Newmarch

A Programmer's Guide to Jini™ Technology

Up-to-date coverage of the newest Jini™ features announced by Sun this year

Addresses important topics such as application architecture, user interfaces for Jini™ services, and how hardware devices and CORBA fit in with the Jini™ framework

Tech reviewed by master Java™ programmer and well-known columnist Bill Venner

APress Media, LLC

A Programmer's Guide to Jini™ Technology

JAN NEWMARCH

APress Media, LLC

libdocdel@fr.com

A Programmer's Guide to Jini™ Technology

Copyright ©2000 by Jan Newmarch

Originally published by Apress in 2000

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN 978-1-893115-80-4 ISBN 978-1-4302-0860-0 (eBook)
DOI 10.1007/978-1-4302-0860-0

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Editorial Directors: Dan Appleman, Gary Cornell, Karen Watterson

Editor: Andy Carroll

Production Editor: Kari Brooks

Page Composition: Tony Jonick—Rappid Rabbit

Artist: Karl Miyajima

Indexer: Carol Burbo

Cover: Karl Miyajima

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

libdocdel@fr.com

Contents at a Glance

Introduction .... *xix*

Chapter 1 Overview of Jini *1*

Chapter 2 Troubleshooting Jini Configuration Problems *17*

Chapter 3 Discovering a Lookup Service *23*

Chapter 4 Entry Objects *43*

Chapter 5 Service Registration *49*

Chapter 6 Client Search *57*

Chapter 7 Leasing *63*

Chapter 8 A Simple Example *83*

Chapter 9 Choices for Service Architecture *109*

Chapter 10 Discovery Management *153*

Chapter 11 Join Manager *161*

Chapter 12 Security *169*

Chapter 13 More Complex Examples *193*

Chapter 14 Remote Event *235*

Chapter 15 ServiceDiscoveryManager *255*

Chapter 16 Transaction *271*

Chapter 17 LEGO MINDSTORMS *295*

Chapter 18 CORBA and Jini *323*

Chapter 19 User Interfaces for Jini Services *355*

Chapter 20 Activation *393*

Index *433*

libdocdel@fr.com

Contents

Introduction xix

Chapter 1 Overview of Jini 1

Jini 1

Components 2

Service Registration 3

Client Lookup 5

Proxies 7

Client Structure 8

Server Structure 10

Partitioning an Application 11

Support Services 13

- HTTP Server 13
- RMI Daemon 15

Summary 15

Chapter 2 Troubleshooting Jini Configuration Problems 17

Java Packages 17

Jini Versions 18

libdocdel@fr.com

Jini Packages 19

Lookup Service 20

RMI Stubs 20

Debugging 22

Summary 22

Chapter 3 Discovering a Lookup Service 23

Running a Lookup Service 23

Reggie 23

rmid and JDK 1.3 26

Unicast Discovery 26

LookupLocator 27

InvalidLookupLocator 27

Running the InvalidLookupLocator 29

Information from the LookupLocator 29

getRegistrar 30

Running the UnicastRegister 32

Broadcast Discovery 32

Groups 33

LookupDiscovery 33

DiscoveryListener 34

DiscoveryEvent 35

Staying Alive 37

Running the MulticastRegister 38

Broadcast Range 39

ServiceRegistrar 39

libdocdel@fr.com

Information from the ServiceRegistrar 41

Summary 42

Chapter 4 Entry Objects 43

Entry Class 43

Attribute Matching Mechanism 45

Restrictions on Entries 46

Convenience Classes 46

Further Uses of Entries 47

Summary 48

Chapter 5 Service Registration 49

ServiceRegistrar 49

ServiceItem 49

Registration 51

ServiceRegistration 51

The SimpleService Program 52

Running the SimpleService 53

Information from the ServiceRegistration 54

Service ID 54

Entries 55

Summary 55

libdocdel@fr.com

Chapter 6 Client Search 57

Searching for Services with the ServiceRegistrar 57

Receiving the ServiceMatches Object 60

Matching Services 61

Summary 62

Chapter 7 Leasing 63

Requesting and Receiving Leases 63

Cancellation 65

Expiration 65

Renewing Leases 65

Granting and Handling Leases 66

Abstract Lease 67

Landlord Lease Package 68

Summary 81

Chapter 8 A Simple Example 83

Problem Description 83

Service Specification 86

Common Classes 87

MIMEType 87

FileClassifier Interface 89

The Client 90

libdocdel@fr.com

Unicast Client 90
Multicast Client 94
Exception Handling 96
The Service Proxy 97
Uploading a Complete Service 98
FileClassifier Implementation 99
FileClassifierServer Implementation 99
Client Implementation 104
What Classes Need to Be Where? 104
Running the FileClassifier 106
Summary 107

Chapter 9 Choices for Service Architecture 109

Proxy Choices 109
Proxy Is the Service 109
RMI Proxy 110
Non-RMI Proxy 112
RMI and Non-RMI Proxies 114
RMI Proxy for FileClassifier 115
What Doesn't Change 115
RemoteFileClassifier 116
FileClassifierImpl 116
FileClassifierServer 117
What Classes Need to Be Where? 120
Running the RMI Proxy FileClassifier 122
Non-RMI Proxy for FileClassifier 123
FileClassifierProxy 124

libdocdel@fr.com

FileServerImpl 126
Service Provider 128
What Classes Need to Be Where? 131
Running the RMI Proxy FileClassifier 132

RMI and non-RMI Proxies for FileClassifier 133
FileClassifierProxy 133
ExtendedFileClassifier 134
ExtendedFileClassifierImpl 135
FileClassifierServer 137
What Classes Need to Be Where? 139

Using Other Services 140
Heart Interface 142
HeartServer 142
HeartClient 145
Heart Implementation 147

Summary 152

Chapter 10 Discovery Management 153

Finding Lookup Locators 153

LookupLocatorDiscovery 155

LookupDiscoveryManager 157

Summary 159

Chapter 11 Join Manager 161

Jini 1.1 JoinManager 161

libdocdel@fr.com

Jini 1.0 JoinManager 163
Getting Information from JoinManager 166
Summary 167

Chapter 12 Security 169

Getting Going with No Security 169
Why AllPermission Is Bad 170
Removing AllPermission 172
Jini with Protection 173
Service Requirements 174
Client Requirements 176
RMI Parameters 178
ServiceRegistrar 179
Transaction Manager and Other Activatable Services 180
rmid 182
rmid and JDK 1.3 183
Being Paranoiac 186
Protection Domains 186
Signing Standard Files 187
Signing Other Services 188
Permissions 88
Putting It Together 189
Summary 191

libdocdel@fr.com

Chapter 13 More Complex Examples 193

Where Are the Class Files? 193

Problem Domain 193

NameEntry Interface 195

Naive Implementation 196

Factory Implementation 199

Using Multiple Class Files 201

Running Threads from Discovery 204

Server Threads 204

Join Manager Threads 207

Client Threads 207

Inexact Service Matching 209

Matching Using Local Services 213

Finding a Service Once Only 221

Leasing Changes to a Service 225

Leased FileClassifier 226

The FileClassifierLeasedResource Class 228

The FileClassifierLeaseManager Class 229

The FileClassifierLandlord Class 231

Summary 233

Chapter 14 Remote Events 235

Event Models 235

Remote Events 236

Event Registration 238

libdocdel@fr.com

Listener List 239

Single Listener 239

Multiple Listeners 241

Listener Source 242

File Classifier with Events 244

Monitoring Changes in Services 249

Summary 254

Chapter 15 ServiceDiscoveryManager 255

ServiceDiscoveryManager Interface 255

ServiceItemFilter Interface 256

Finding a Service Immediately 257

Using a Filter 259

Building a Cache of Services 262

Running the CachedClientLookup 265

Monitoring Changes to the Cache 266

Summary 269

Chapter 16 Transactions 271

Transaction Identifiers 271

TransactionManager 272

TransactionParticipant 273

Mahalo 273

libdocdel@fr.com

A Transaction Example 274

PayableFileClassifierImpl 276

AccountsImpl 282

Client 287

Summary 294

Chapter 17 LEGO MINDSTORMS 295

Making Hardware into Jini Services 295

MINDSTORMS 296

MINDSTORMS as a Jini Service 296

RCXPort 297

RCX Programs 299

Jini Classes 301

Getting It Running 307

Entry Objects for a Robot 315

A Client-Side RCX Class 316

Higher-Level Mechanisms: Not Quite C 317

Summary 322

Chapter 18 CORBA and Jini 323

CORBA 323

CORBA to Java Mapping 325

Jini Proxies 326

libdocdel@fr.com

A Simple CORBA Example 328
CORBA Server in Java 328
CORBA Client in Java 330
Jini Service 331
Jini Server and Client 334
Building the Simple CORBA Example 334
Running the Simple CORBA Example 335
CORBA Implementations 335

Room-Booking Example 336
CORBA Objects 337
Multiple Objects 340
Exceptions 344
Interfaces for Single Thin Proxy 345
RoomBookingBridge Implementation 347
Other Classes 351
Building the Room-Booking Example 352
Running the Room-Booking Example 352

Migrating a CORBA Client to Jini 353

Jini Service as a CORBA Service 354

Summary 354

Chapter 19 User Interfaces for Jini Services 355

User Interfaces as Entries 355

User Interfaces from Factory Objects 356

Current Factories 358

Marshalling Factories 358

libdocdel@fr.com

UIDescriptor..................360
Toolkit..................360
Role..................361
Attributes..................362
File Classifier UI Example..................363
Images..................372
ServiceType..................373
MINDSTORMS UI Example..................374
RCXLoaderFrame..................374
RCXLoaderFrameFactory..................380
Exporting the FrameFactory..................381
Customized User Interfaces..................382
CarJFrame..................383
CarJFrameFactory..................387
Exporting the FrameFactory..................388
The RCX Client..................389
Summary..................392

Chapter 20 Activation..................393

A Service Using Activation..................394
The Service..................394
The Server..................395
Running the Service..................400
Security..................401
Non-Lazy Services..................402
Maintaining State..................402
LeaseRenewalService..................411

libdocdel@fr.com

The Norm Service.......412
Using the LeaseRenewalService.......413
LookupDiscoveryService 420
The Fiddler Service 422
Using the LookupDiscoveryService 422
Summary 431
Index.......433

libdocdel@fr.com

Introduction

THE BUSINESS AND ACADEMIC WORLDS HAVE LONG ACCEPTED the use of networking technologies, allowing users to share files and applications and to exchange information using network services such as email. The explosive growth of the Internet has made everyone conscious of the importance of networked applications, and this importance is set to grow at an enormous rate with the emergence of home and mobile networks.

For the programmer, building distributed applications can be a complex business. There are issues related to network stability and accessibility in addition to partitioning applications into portions that can run separately but still be linked into larger functional units. A large variety of frameworks—experimental and commercial—have been devised to make it easier to build and deploy distributed applications.

Jini is one of the latest frameworks for building distributed applications. Created by Sun Microsystems, it builds upon previous experiences but also introduces new concepts that fit into the modern object-oriented world. Jini is written in Java and distributes and organizes applications based on the distributed object-oriented principles supported by Java. It allows the programmer to build type-safe applications with distributed garbage collection, which results in applications that are resilient to network failures and can that discover and use distributed services at need.

This book is written for programmers/architects who have a working knowledge of Java and of network programming and who want to come up to speed with Jini quickly. It assumes you are comfortable with network concepts such as remote procedure calls, are familiar with Java syntax, and have a working knowledge of the Java core classes.

This is a hands-on, study-the-code book. My intention is to introduce you to code that can be readily understood, and that can be copied and used in your own programs. The book covers the full range of Jini concepts, and it also deals with a number of advanced topics such as linking Jini and CORBA systems and using Jini to make hardware devices available across the network. The book has been available on the Internet in various forms for nearly two years, and has benefited from user feedback while it has been aiding many new Jini programmers.

The first eight chapters cover the basics of Jini programming, leading to a complete, but simple, application. The subsequent chapters discuss more advanced material, such as event handling, security, transactions, and activation, and it also covers the new helper classes of Jini 1.1. In addition, I have included topics not normally covered in Jini books, such as user interfaces, links to other distributed systems such as CORBA, and using hardware devices with Jini.

libdocdel@fr.com

The book uses Jini version 1.1, released late in 2000, and the code works with both JDK 1.2 and JDK 1.3

Files

The source for the programs in the book is available as a zip file: `programs.zip`. The compiled classes are also available as a zip file: `classes.zip`. These files are on this Web site: `http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml`.

Other Resources

- Jini FAQ: `http://www.artima.com/jini/faq.html`
- Jini Community Web site: `http://www.jini.org/`
- Jini home page: `http://www.sun.com/jini/index.html`
- RMI home page: `http://java.sun.com/products/jdk/rmi/index.html`
- Jini mailing list: `jini-users@java.sun.com`
- Jini mailing list archives: `http://archives.java.sun.com/archives/jini-users.html`
- RMI mailing list: `rmi-users@java.sun.com`
- RMI mailing list archives: `http://archives.java.sun.com/archives/rmi-users.html`
- Jini Interface Repository for standardized service interfaces (empty at present): `http://www.artima.com/jini/interrepo/`

libdocdel@fr.com

Acknowledgments

The author is grateful for comments on this tutorial from

Brian Jeltema, jeltema@richdist.east.sun.com

Roger Whitney, whitney@cs.sdsu.edu

Robbert van den Beld, rbe@mms-dresden.de

Chitrarasu Muthaiyan, chitrarasu@cswl.com

Stuart Remphrey, Stuart.Remphrey@Aus.Sun.COM

JJ Larrea, jlarrea@redtop.com

John McClain, John.McClain@East.Sun.COM

Bob Scheifler, rws@east.sun.com

Much of the work for this book was done while the author was on a sabbatical program at the CRC for Distributed Systems Technology, http://www.dstc.edu.au, and the work reported in this book 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.

libdocdel@fr.com

CHAPTER 1

Overview of Jini

JINI IS MIDDLEWARE FOR building distributed systems in Java. It builds upon the distributed computing mechanisms of sockets and Remote Method Invocation. The intent is to offer "network plug and work," where new services can join a network of other services and be immediately useful, and where clients can search for and use these services. Jini has only been released for a little over a year as this is being written, and it introduces novel ideas and technologies for building distributed systems. This chapter gives a brief overview of the components of a Jini system and the relationships between them.

Jini

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

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

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

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

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

libdocdel@fr.com

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

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

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. One of the later chapters in this book discusses bridging between Jini and CORBA, as an example of linking these different distributed architectures.

There are other Java frameworks from Sun that might appear to overlap Jini, such as Enterprise Java Beans (EJBs). EJBs make it easier to build business logic servers, whereas Jini would be better used to distribute those services in a "network plug and play" manner.

You should be aware that Jini is only one competitor in a non-empty market. The success or failure of Jini will result partly from 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 a service. Third, there is a lookup service (service locator), which acts as a broker/trader/locator between services and clients. There is one additional component, and that is a network connecting all three of these. 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.) See Figure 1-1.

Code will be moved around between these three pieces, and this is done by marshalling the objects. This involves serializing the objects in such a way that they can be moved around the network, stored in this "freeze-dried" form, and later reconstituted by using instance data and included information about the class files. Movement around the network is done using Java's socket support to send and receive objects.

libdocdel@fr.com

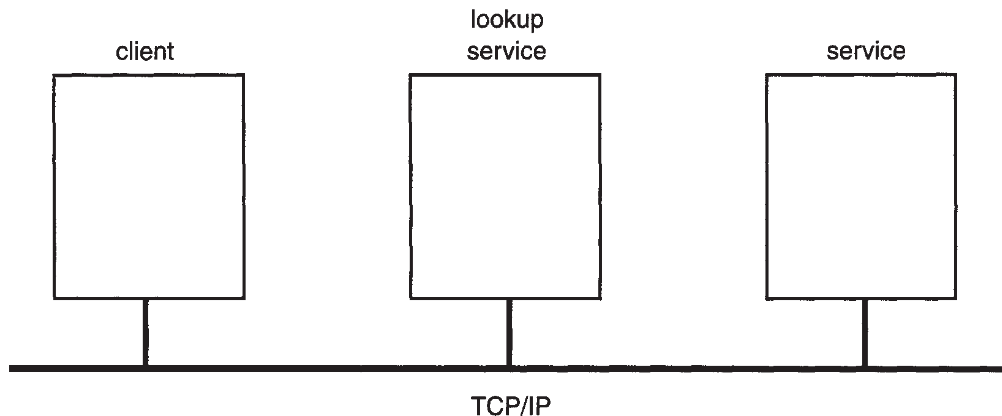


Figure 1-1. Components of a Jini system

In addition, objects in one JVM (Java Virtual Machine) may need to invoke methods on an object in another JVM. Often this will be done using RMI (Remote Method Invocation), although the Jini specification does not require this and there are many other possibilities.

Service Registration

A service is a logical concept and can be anything, such as a blender, a chat service, a disk. A service is usually defined by a Java interface, and this interface is used to advertise the service. This interface is also used to locate a service. Each service can be implemented in many ways, by many different vendors. For example, there may be Joe's dating service, Mary's dating service, and many others. What makes them the same service is that they implement the same interface; what distinguishes one from another is that each different implementation uses a different set of objects (or maybe just one object) belonging to different classes.

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

- It creates the objects that implement the service.
- It registers one of these—the service object—with lookup services. The service object is the publicly visible part of the service, and it will be downloaded to clients.
- It stays alive in a server role, performing various tasks such as keeping the service "alive."

libdocdel@fr.com

In order for the service provider to register the service object with a lookup service, the server must first find the lookup service. This can be done in two ways. If the location of the lookup service is known, then the service provider can use unicast TCP to connect directly to it. If the location is not known, the service provider will make UDP multicast requests, and lookup services may respond to these requests. Lookup services will be listening on port 4160 for both the unicast and multicast requests. (4160 is the decimal representation of hexadecimal (CAFE - BABE). Oh well, these numbers have to come from somewhere.) This process is illustrated in Figure 1-2.

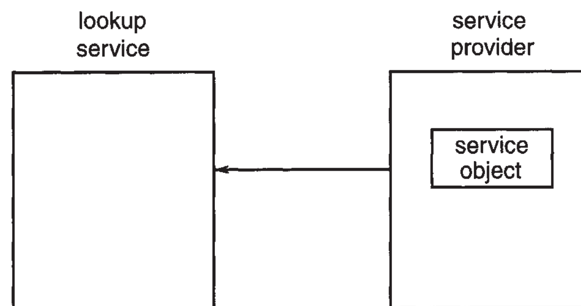


Figure 1-2. Querying for a service locator

When the lookup service gets a request on this port, it sends an object back to the server, as shown in Figure 1-3. This object, known as a registrar, acts as a proxy to the lookup service and runs in the service's JVM. Any requests that the service provider needs to make of the lookup service are made through this proxy registrar. Any suitable protocol may be used to do this, but in practice the implementations of the lookup service that you get (such as those from Sun) will probably use RMI.

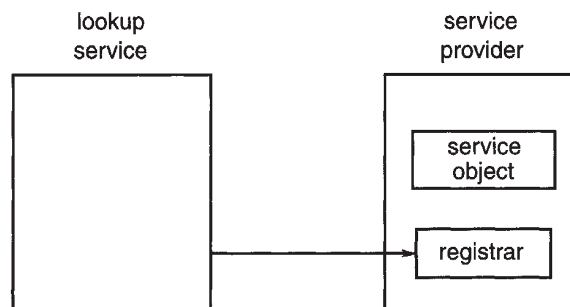


Figure 1-3. Registrar returned

libdocdel@fr.com

What the service provider does with the registrar is register the service with the lookup service. This involves taking a copy of the service object and storing it on the lookup service, as shown in Figure 1-4.

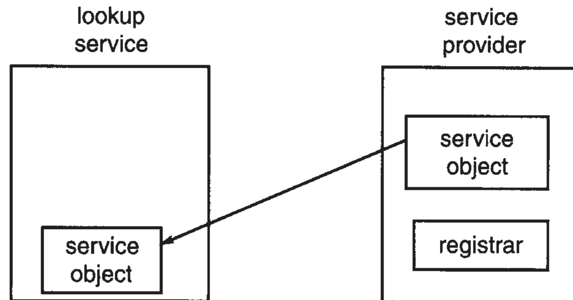


Figure 1-4. Service uploaded

Client Lookup

The client on the other hand, is trying to get a copy of the service object into its own JVM. It goes through the same mechanism to get a registrar from the lookup service, as shown in Figures 1-5 and 1-6.

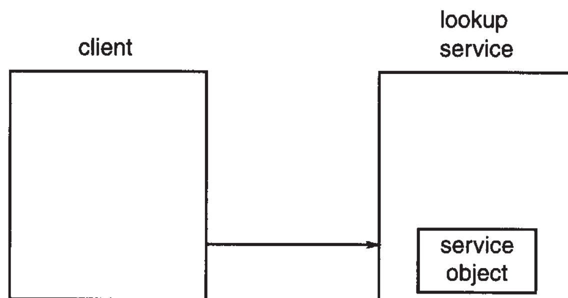


Figure 1-5. Querying for a service locator

libdocdel@fr.com

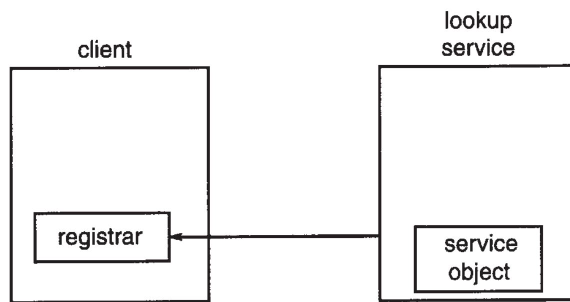


Figure 1-6. Registrar returned

However, the client does something different with the registrar. It requests that the service object be copied across to it. See Figures 1-7 and 1-8.

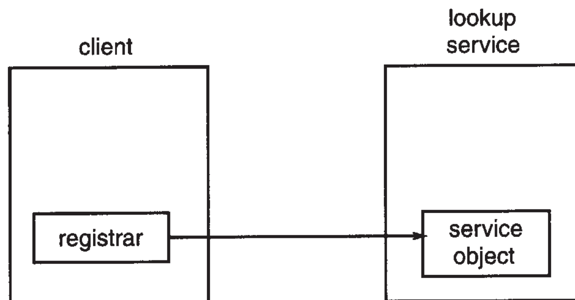


Figure 1-7. Asking for a service

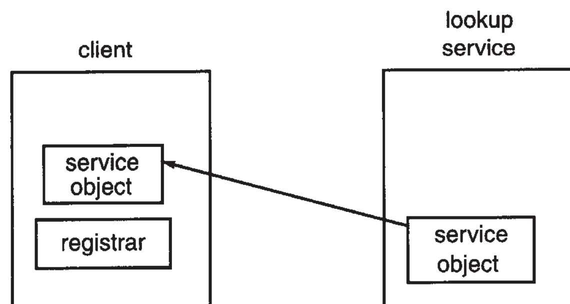


Figure 1-8. Service object returned

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

libdocdel@fr.com

Proxies

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

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

The motivation for discussing proxies is the situation in which a service object needs to control a remote piece of hardware that is not directly accessible to the service object. However, this need not involve hardware—there could be files accessible to the service provider that are not available to objects running in clients. There could be applications local to the service provider that are useful in implementing the service. Or it could simply be easier to program the service in ways that involve objects on the service provider, with the service object being just a proxy. The majority of service implementations end up with the service object being just a proxy to service backend objects, and it is quite common to see the service object being referred to as a service proxy. It is sometimes referred to as the service proxy even if the implementation doesn't use a proxy at all!

The proxy needs to communicate with other objects in the service provider, but this begins to look like a chicken-and-egg situation: how does the proxy find the service backend objects in its service provider? Use a Jini lookup? No, when the proxy is created it is "primed" with its own service provider's location so that when it is run it can find its own "home," as illustrated in Figure 1-9.

How is the proxy primed? This isn't specified by Jini, and it can be done in many 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 activable objects. Another option is that 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.

libdocdel@fr.com

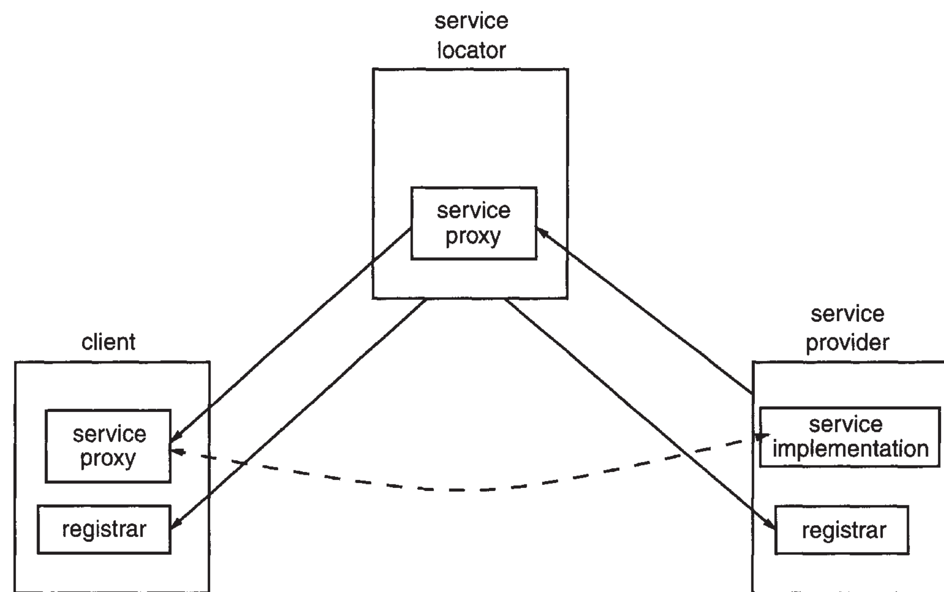


Figure 1-9. A proxy service

Client Structure

Now that we've looked at how the various pieces interact, we'll take a look at what is going on inside clients and services. Internally a client will look like this:

PSEUDOCODE	WHERE DISCUSSED
Prepare for discovery	Chapter 3, "Discovering a Lookup Service"
Discover a lookup service	Chapter 3, "Discovering a Lookup Service"
Prepare a template for lookup search	Chapter 4, "Entry Objects," and Chapter 6, "Client Search"
Look up a service	Chapter 6, "Client Search"
Call the service	Chapter 8, "A Simple Example"

The "prepare for discovery" step involves setting up a list of service locators that will be looked for. The "discover a lookup service" step is where the unicast or multicast search for lookup services is performed. "Prepare a template for lookup search" involves creating a description of the service so that it can be found. "Look up a service" is when a service locator is queried to see if it has such a service. Once a suitable service has been found, then "call the service" will invoke methods on this service.

libdocdel@fr.com

The following code has been simplified from the real case by omitting various checks on exceptions and other conditions. It attempts to find a `FileClassifier` service, and then calls the `getMIMEType()` method on this service. The full version of the code is given in Chapter 8. I won't give detailed explanations right now—this is just to show how the preceding 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
```


libdocdel@fr.com

Server Structure

A server application will internally look like this:

PSEUDOCODE	WHERE DISCUSSED
Prepare for discovery	Chapter 3, "Discovering a Lookup Service"
Discover a lookup service	Chapter 3, "Discovering a Lookup Service"
Create information about a service	Chapter 4, "Entry Objects"
Export a service	Chapter 5, "Service Registration"
Renew leasing periodically	Chapter 7, "Leasing"

Again, the following code has been simplified 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 of the code is given in Chapter 8. I won't give detailed explanations right now—this is just to show how the preceding 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 (almost) 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);
```


libdocdel@fr.com

```
    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
```

Partitioning an Application

Jini uses a *service* view of applications, 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 as individual services, which will communicate via their proxy objects. The service view will show these services as they exist on their servers, without any detail about their implementation by objects. This leads to a different way of partitioning an application, not into its component objects, but into its component services. The Jini specification claims that in many monolithic applications there are one or more services waiting to be released, and that making them into services increases their possible uses.

To support this claim, we can look at a smart file viewer application. This application 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 application, using a standard object-oriented approach, might look like Figure 1-10.

There are a number of services that could be extracted from this smart file viewer application. Classifying a file into types is one service that can be used in lots of different situations, not just when you want to view file contents. Each of the different viewer classes is another service.

libdocdel@fr.com

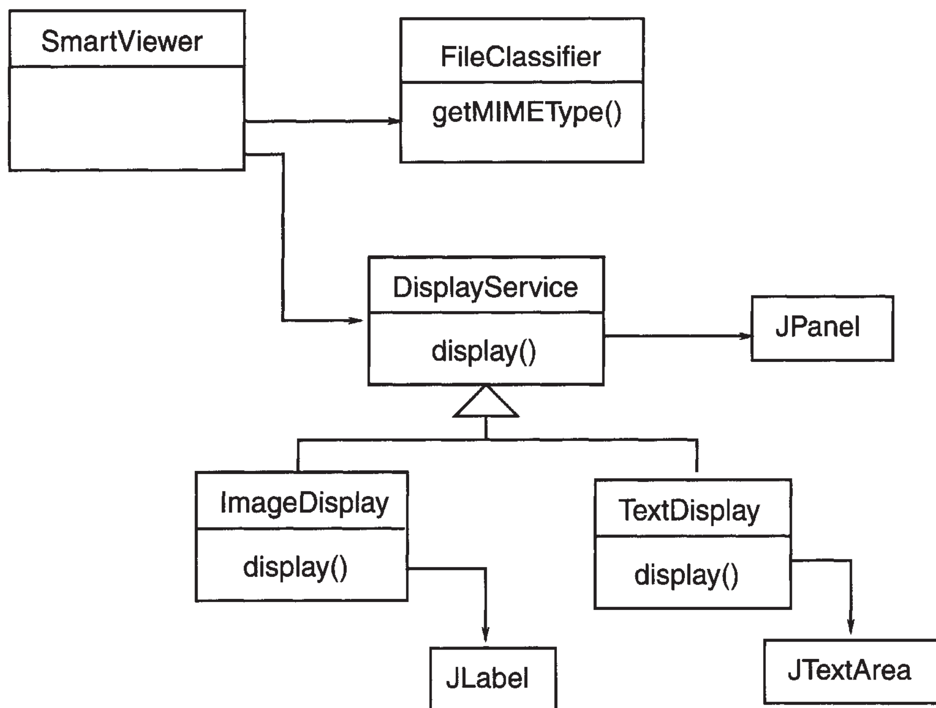


Figure 1-10. UML diagram for an application

However, 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 all

- have a simple interface
- are useful in more than one situation
- can be replaced or varied

They are reusable, and this is what makes them good candidates for services. They do not require high-bandwidth communication, and they are not completely trivial.

If the application is reorganized as a collection of services, then it could look like Figure 1-11.

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.

libdocdel@fr.com

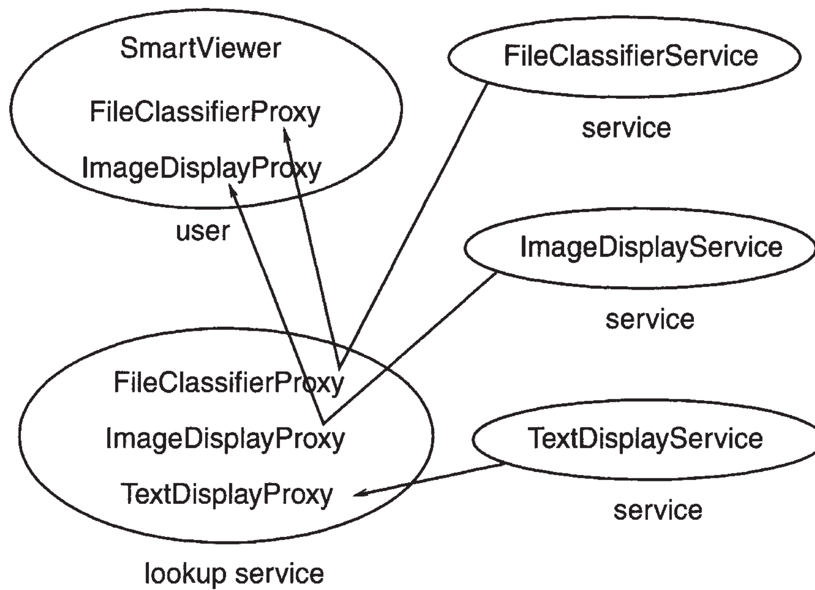


Figure 1-11. Application as a collection of services

Support Services

The three components of a Jini system are clients, services, and service locators, each of which can run anywhere on the network. These will be implemented using Java code running in Java Virtual Machines (JVMs). The implementation may be in pure Java but it could make use of native code using JNI (Java Native Interface) or make external calls to other applications. Often, each of these applications will run in its own JVM on its own computer, though they could run on the same machine or even share the same JVM. When they run, they will need access to Java class files, just like any other Java application. Each component will use the `CLASSPATH` environment variable or use the `CLASSPATH` option with the runtime to locate the classes it needs to run.

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

HTTP Server

A Java object running as a service has a proxy component exported to the service locators and then onto a client. The proxy passes through a service locator's JVM in "passive" form and is activated (brought to life) in the client's JVM. Essentially, a

libdocdel@fr.com

snapshot of the object's state is taken using serialization, and this snapshot 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. So, where is the code? This is where a distributed Jini application differs from a standalone application or a client-server application: 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.

This means that 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 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 on this server. This often means that there is 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 than 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. These files and support services are shown in Figure 1-12.

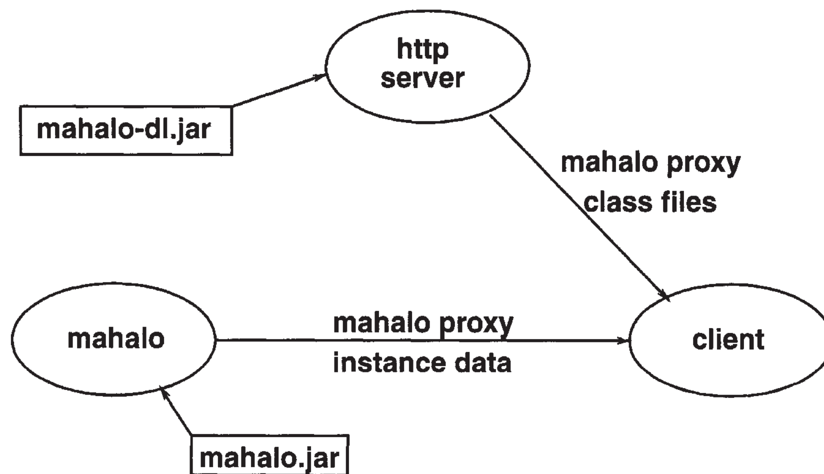


Figure 1-12. Support services for `mahalo`

libdocdel@fr.com

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.

RMI Daemon

As mentioned earlier, a proxy service gets exported to the client, and in most cases it will need to communicate with its host service. There are many ways to do this, which are discussed in full in later chapters. One mechanism is the Java Remote Method Invocation (RMI) system. This comes in two flavors in JDK 1.2: the original UnicastRemoteObject and the newer Activatable class. Whereas UnicastRemoteObject requires a process to remain alive and running, Activatable 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 that can accept method calls and activate the objects. This server is called an *RMI daemon,* and Sun supplies such a server, called rmid.

This is really 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 supplies a service locator called reggie, and this is really just another Jini service that plays a special role. It exports proxy objects—the registrar objects. What makes this complex is that reggie uses Activatable 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.

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.

Summary

A Jini system is made up of three parts:

- Service
- Client
- Service locator

libdocdel@fr.com

Code is moved between these parts of 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. Services may require support from other non-Jini servers, such as an HTTP server.

libdocdel@fr.com

CHAPTER 2

Troubleshooting Jini Configuration Problems

JINI IS ADVERTISED AS "network plug and work," which 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 work *does* indeed work, but if you lose at any stage, then it can be all uphill!

The difficulty is getting the right files in the right places with the right permissions. About 50 percent of the messages in the Jini mailing list are about these configuration problems. They shouldn't occur, and that is why this is "The Chapter That Shouldn't Exist." 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 is the second chapter in the book, so right now you shouldn't have managed to fail at anything! In the following chapters, the sections contains instructions on what to do to get the example programs working, and include step-by-step instructions, so skip on to the next chapters, but come back here when things go wrong. Your luck may vary: I got a reasonable way into my first attempts without problems, and some people are even luckier. Some aren't. . . .

Java Packages

A typical Java packages error looks like this:

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

Most of the code in this tutorial is organized 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 `basic` package. The program must be run using the fully qualified path name, like this:

```
java basic.InvalidLookupLocator
```


libdocdel@fr.com

Note the use of the period (.), not a slash (/).

In order to find this class, the CLASSPATH must be set correctly for the Java runtime. If you have copied the classes.zip file, 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 the 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 /home/jan/classes/basic/InvalidLookupLocator.class, then set the 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, like this:

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

Jini Versions

At the time of writing, there are two versions of Jini: 1.0 and a version of 1.1. The core classes are all the same for versions 1.0 and 1.1. The only changes in version 1.1 for the programmer are that some classes from Jini 1.0 have been better specified and are in different packages, and some classes are new.

These are the main classes that have changed:

- JoinManager
- LeaseRenewalManager
- ServiceIDListener

These are the main new classes:

- LookupLocatorDiscovery
- LookupDiscoveryManager
- ClientLookupManager

libdocdel@fr.com

If you get syntax or runtime errors relating to these classes, then it is possible that you are using Jini 1.0 instead of Jini 1.1. If you get "deprecated" warnings, then it is likely that you are using the Jini 1.0 classes in a Jini 1.1 environment. The old classes are supported for now, but are not approved.

Jini Packages

A typical Jini package error looks like this:

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

The Jini class files are all in jar files. The Jini distribution puts them in a `lib` subdirectory when they are 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	reggie.jar
jini-ext.jar		

The `jini-core.jar` jar file 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` file is in your `CLASSPATH`.

libdocdel@fr.com

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

```
net.jini.admin      net.jini.discovery
net.jini.entry      net.jini.lease
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` file is in your `CLASSPATH`.

The `sun-util.jar` jar file 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 this:

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

Lookup Service

A typical lookup service error looks like this:

```
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.

RMI Stubs

A typical RMI stubs error looks like this:

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


libdocdel@fr.com

Many of the examples in this book 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 a proxy for the object (which continues to run back in the server). The stub has to be created using the `rmic` compiler, like this:

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

This will create a `FileClassifierImpl_Stub.class` in the `rmi` 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 this:

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

This error arises when an object is trying to get a remote reference to `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`, as shown in Figure 2-1.

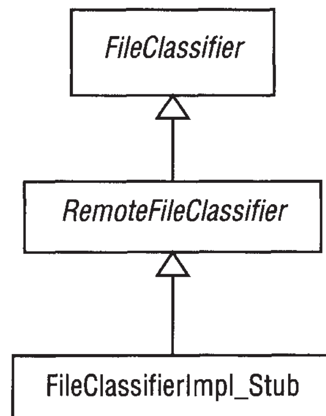


Figure 2-1. Interfaces and superclasses for an exported stub

Class files for all of these classes and interfaces have to be available! The `FileClassifier` interface may be "well known," with a class file on each client and server. However, an interface such as `RemoteFileClassifier`, as well as the

libdocdel@fr.com

implementation files for FileClassifierImpl, may only be known to a particular server. 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—in this case, for RemoteFileClassifier as well as FileClassfier.

Debugging

Debugging a Jini application is difficult because there are so many bits to it, and these bits are all running separately: the server for a service, the client, the lookup services, the remote activation daemons, and the HTTP servers. There are a few (not many) errors within the Jini objects themselves, but more importantly, many of these objects are implemented using multiple threads, and the flow of execution is not always clear. There are no magic "debug" flags that can be turned on to show what is happening.

On either the client or service side, a debugger such as jdb can be used to step through or trace execution of a client or server. Lots of print statements help too. There are also three flags that can be turned on to help:

```
java -Djava.security.debug=access \
   -Dnet.jini.discovery.debug=1 \
   -Djava.rmi.server.logCalls=true ...
```

These flags don't give complete information, but they do give some, and they can at least tell you if the application's parts are still living! If the java.security.debug property is set to access, then every time the application needs to check a security access (such as making a network connection, opening a file, etc.) it will print a message. If net.jini.discovery.debug is set to any non-null value, then any exceptions thrown during the discovery process will be printed. The final property will set on logging of RMI calls.

Summary

Setting up and running a Jini system is complex at present, with many things that can go wrong. This chapter looked at some of the problems that can occur and some of the solutions. The list is not complete, but it may help in the most common situations.

libdocdel@fr.com

CHAPTER 3

Discovering a Lookup Service

JINI USES A LOOKUP SERVICE in much the same way as other distributed systems use naming services and traders. Services register with lookup services, and clients use them to find services they are interested in. Jini lookup services are designed to be an integral part of the Jini system, and they have their own set of classes and methods. This chapter looks at what is involved in discovering a lookup service or service locator. This is common to both services and clients. The chapter also discusses issues particular to the Sun lookup service, `reggie`.

Running a Lookup Service

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

The initial task for 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. Unicast means that you know the address of the lookup service and can contact it directly. Multicast is used when you do not know where a lookup service is and have to broadcast a message across the network so that any lookup service can respond. In fact, the lookup service is just another Jini service, but it is one that is specialized to store services and pass them on to clients looking for them.

Reggie

Sun supplies a lookup service called `reggie` as part of the standard Jini distribution. The specification of a lookup service is public, and in future we can expect to see other implementations of lookup services.

There may be any number of these lookup services running in a network. A LAN may run many lookup services to provide redundancy in case one of them crashes. Similarly, across the internet, people may run lookup services for a variety

libdocdel@fr.com

of reasons: a public lookup service is running on http://www.jini.canberra.edu.au to aid people trying Jini clients and services so that they don't need to also set up a lookup service. Other lookup services may act as coordination centers, such as a repository of locations for all the atomic clock servers in the world.

Anybody can start a lookup service (depending on access permissions), but it will usually be started by an administrator, or started at boot time.

Reggie requires support services: an HTTP server and an RMI daemon, rmid. These need to be already running by the time reggie is started. If there is already an HTTP server running, it 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 it 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 tools.jar file, and it can be started with this command:

```
java -jar tools.jar
```

This Jini HTTP server 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 the 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 the full path /home/jan/tmpdir/jini1_0/lib/reggie-dl.jar. I set the 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 it is a part of the standard Java distribution. Vendors could implement other RMI daemons, but this is unlikely to happen. rmid must be run on the same machine as reggie. The following command is a Unix command that runs rmid as a background process:

```
rmid &
```

This command also has major options:

```
rmid [-port num] [-log dir]
```


libdocdel@fr.com

These options can specify the TCP port used (which defaults to 4160). You can also specify the location for the log files that `rmid` uses to store its state—they default to being in the `log` subdirectory.

There is a security issue with `rmid` on multiuser systems such as Unix. The activation system that it supports allows anyone on the same machine to run programs using the user ID that `rmid` is running under. That means you should never run `rmid` using a sensitive user ID such as `root`, but instead should run it as the least privileged user, `nobody`.

Once the HTTP server and `rmid` are running, `reggie` can be started with a number of compulsory parameters:

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

The parameters are as follows:

- The `lookup-server-jarfile` will be `reggie.jar` or some path to it.

- The `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* that means `jannote.dstc.edu.au`. To the *client* it would be a different machine altogether, because to the client `localhost` is their own machine, not `jannote.dstc.edu.au`! The client would then fail to find `reggie-dl.jar` on its own machine. Even an abbreviated address, such as `jannote`, would fail to be resolved if the client is external to the local network.

- The `lookup-policy-file` controls security accesses. Initially you can set this to the `policy.all` path in the Jini distribution, but for deployment, use a less dangerous policy file. The topic of security is discussed in Chapter 12, but in brief, Jini code mobility allows code from other sources to run within the client machine. If you trust the other code, then that may be fine, but can you *really* trust it? If not, you don't want to run it, and Jini security can control this. However, in the debugging and testing phases, this security can cause extra complications, so you should turn off security while testing other aspects of your code by using a weak security policy. Then make sure you turn it back on later!

- The `output-log-dir` can be set to any (writable) path to store the log files.

- The `lookup-service-group` can be set to the public group `public`.

libdocdel@fr.com

As an example, on my own machine, I start reggie like this:

```
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.all \
    /tmp/reggie_log public
```

After starting, reggie will promptly exit! Don't worry about this—it is actually kept in a 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, since it keeps information about reggie in its log files.

rmid and JDK 1.3

rmid is responsible for starting (or restarting) services such as reggie. It will create a new JVM on demand to run the service. rmid may look after a number of services, not just reggie, and they will all be run in their own JVMs. In JDK 1.2 there was no difference in handling these different JVMs. However, in JDK 1.3, the ability to set different security policies was introduced. This topic is dealt with in detail in Chapter 12.

In JDK 1.3, starting rmid requires an extra parameter to set the sun.rmi.activation.execPolicy policy. It is simplest to set it so that rmid behaves the same way as it did in JDK 1.2. This can be done with the following command:

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

This setting ignores the new security mechanism, and it is not recommended as a long-term or production solution.

Unicast Discovery

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

Unicast discovery relies on a single class, LookupLocator, which is described in the next section. Basic use of this class is illustrated in the sections on the InvalidLookupLocator program. The InvalidLookupLocator should be treated as an

libdocdel@fr.com

introductory Jini program that you can build and run without having to worry about network issues. Connecting to a lookup service using the network is done with the `getRegistrar()`method of `LookupLocator`, and an example program using this is shown in the `UnicastRegistrar` program in the "Get Registrar" section.

LookupLocator

The `LookupLocator` class in the `net.jini.core.discovery` package is used for unicast discovery of a lookup service. 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` parameter follows the standard URL syntax of "protocol://host" or "protocol://host:port". The protocol is `jini`. If no port is given, it defaults to 4160. The host should be a valid DNS name (such as `pandonia.canberra.edu.au` or an IP address (such as `137.92.11.13`). So for example, `jini://pandonia.canberra.edu.au:4160` may be given as the URL for the first constructor. No unicast discovery is performed at this stage, though, so any rubbish could be entered. Only a check for the syntactic validity of the URL is performed. The first constructor will throw an exception if it discovers a syntax error. This syntactic check is not even done for the second constructor, which takes a host name and port separately.

InvalidLookupLocator

The following program creates some objects with valid and invalid host/URLs. They are only checked for syntactic validity rather than existence as URLs. That is, no network lookups are performed. This should be treated as a basic example to get you started building and running a simple Jini program.

```
package basic;

import net.jini.core.discovery.LookupLocator;
```


libdocdel@fr.com

```
/**
 * InvalidLookupLocator.java
 */

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, and should throw an exception
    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, but no check is made anyway
    lookup = new LookupLocator("localhost", 80);
    System.out.println("Fourth lookup creation succeeded");
    }

} // InvalidLookupLocator
```


libdocdel@fr.com

Running the InvalidLookupLocator

All Jini 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).

The `InvalidLookupLocator` program defines the `InvalidLookupLocator` class in the `basic` package. The source code will be in the `InvalidLookupLocator.java` file in the `basic` subdirectory. From the parent directory, this can be compiled by a command such as this:

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

This will leave the class file also in the `basic` subdirectory.

When you compile the source code, the `CLASSPATH` will need to include the `jini-core.jar` Jini file. Similarly, when a service is run, this Jini file will need to be in its `CLASSPATH`, and when a client runs, it will also need this file in its `CLASSPATH`. The reason for this repetition is that the service and the client are two separate applications, running in two separate JVMs, and quite likely will be on two separate computers.

The `InvalidLookupLocator` has no additional requirements. It does not perform any network calls and does not require any additional service to be running. It can be run simply by entering this command:

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

Information from the LookupLocator

Two of the methods of `LookupLocator` are these:

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

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

libdocdel@fr.com

getRegistrar

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

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

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

UML sequence diagrams are useful for showing the timelines of object existence and the method calls that are made from one object to another. The timeline reads down, and the method calls and their returns read across. A UML sequence diagram augmented with a jagged arrow showing the network connection is shown in Figure 3-1. The UnicastRegister object makes a new() call to create a LookupLocator, and this call returns a lookup object. The getRegistrar() method call is then made on the lookup object, and this causes network activity. As a result of this, a ServiceRegistrar object is created in some manner by the lookup object, and this is returned from the method as the registrar.

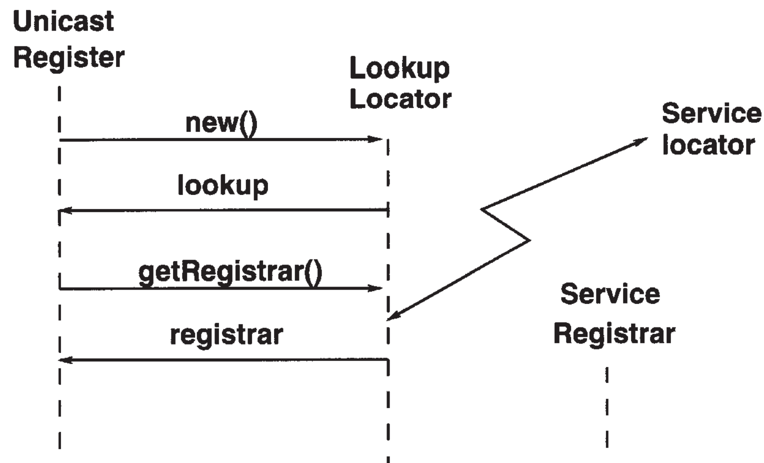


Figure 3-1. UML sequence diagram for lookup

The UnicastRegistrar program that implements Figure 3-1 and performs the network connection to get a ServiceRegistrar object is as follows:

```
package basic;

import net.jini.core.discovery.LookupLocator;
```


libdocdel@fr.com

```
import net.jini.core.lookup.ServiceRegistrar;

/**
 * UnicastRegistrar.java
 */

public class UnicastRegister  {

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

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

        try {
            lookup = new LookupLocator("jini://www.jini.canberra.edu.au");
        } 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: the services will use it to register themselves, and the clients will use it to locate services.

libdocdel@fr.com

> **NOTE** *This program might not run as is, due to security issues. If that is the case, see the first section of Chapter 12.*

Running the UnicastRegister

When the `UnicastRegistrar` program in the previous section program needs to be compiled and run, it has to have the file `jini-code.jar` in its `CLASSPATH`. When run, it will attempt to connect to the service locator, so obviously the service locator needs to be running on the machine specified in order for this to happen. Otherwise, the program will throw an exception and terminate. In this case, the host specified is `www.jini.canberra.edu.au`. 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 for `LookupLocator` would be `jini://pandonia.canberra.edu.au`.

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

```
java  basic.UnicastRegister
```

The `CLASSPATH` for the `UnicastRegister` program should contain the Jini jar files as well as the path to `basic/UnicastRegister.class`.

Broadcast Discovery

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

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

libdocdel@fr.com

Groups

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

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

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

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

LookupDiscovery

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

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

The parameter in the `LookupDiscovery` constructor can take three possible values:

- `null`, or `LookupDiscovery.ALL_GROUPS`, means that the object should attempt to discover all reachable lookup services, no matter which group they belong to. This will be the normal case.
- An empty list of strings, or `LookupDiscovery.NO_GROUPS`, means that the object is created but no search is performed. In this case, the method `setGroups()` will need to be called in order to perform a search.
- A non-empty array of strings can be given. This will attempt to discover all lookup services in that set of groups.

libdocdel@fr.com

DiscoveryListener

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

```
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, the discovered() method is the natural place for a service to register, and it is also the natural place for a client to ask if there is a service available and to invoke the 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, and 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 until replies come in. One way of handling this is for the application to have a GUI interface, in which case the application will stay until the user dismisses it. Another possibility is that the application may be prepared to wait for a while before giving up. In that case, the main() method could sleep for, say, ten seconds and then exit. This will depend on what the application should do if no lookup service is discovered.

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

libdocdel@fr.com

DiscoveryEvent

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

```
package net.jini.discovery;

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

This has one public method, getRegistrars(), which returns an array of ServiceRegistrar objects. Each one of these implements the ServiceRegistrar interface, just like the object returned from a unicast search for a lookup service. More than one ServiceRegistrar object can be returned if a set of replies have come in before the listener was registered—they are collected in an array and returned in a single call to the listener. A UML sequence diagram augmented with jagged arrows showing the network broadcast and replies is shown in Figure 3-2.

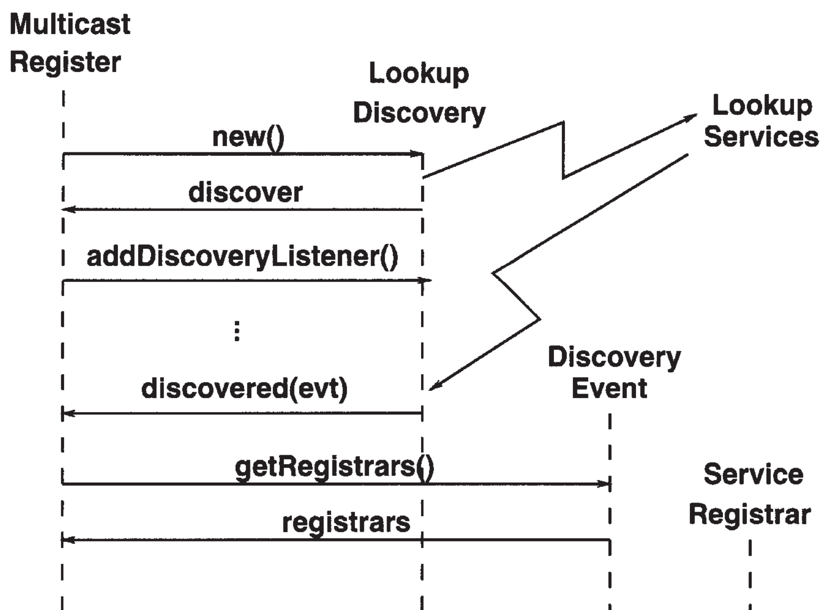


Figure 3-2. UML sequence diagram for discovery

In Figure 3-2, the creation of a LookupDiscovery object starts the broadcast search, and it returns the discover object. The MulticastRegister adds itself as a listener to the discover object. The search continues in a separate thread, and when a

libdocdel@fr.com

new lookup service replies, the `discover` object invokes the `discovered()` method in the `MulticastRegister`, passing it a newly created `DiscoveryEvent`. The `MulticastRegister` object can then make calls on the `DiscoveryEvent`, such as `getRegistrars()`, which will return suitable `ServiceRegistrar` objects. There is no line connecting to the `ServiceRegistrar` because the `DiscoveryEvent` creates the `ServiceRegistrar` somehow, but the actual mechanism that is used is hidden in the implementation of the `DiscoveryEvent`.

A `MulticastRegister` program that implements multicast searches for lookup services would look like this:

```
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
 */

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() {
    System.setSecurityManager(new java.rmi.RMISecurityManager());
        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            e.printStackTrace();
            System.exit(1);
        }
```


libdocdel@fr.com

```
        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
```

Staying Alive

In the preceding constructor for the MulticastRegister program, 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 for LookupDiscovery actually starts up a number of threads to broadcast the service and to listen for replies (see Chapter 21).

When replies come in, the listener thread will call the discovered() method of the MulticastRegister. However, these threads are daemon threads. Java has two types of threads—daemon and user threads—and at least one user thread must be running or the application will terminate. All these other threads are not enough to keep the application alive, so it keeps a user thread running in order to continue to exist.

The sleep() method 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 requires either increasing the sleep time or creating another user thread in the discovered() method. In Chapter 7, use is made of a useful constant, Lease.FOREVER. It is tempting to use the FOREVER constant if you want a thread to sleep forever. While the "leasing" system understands this FOREVER constant, the standard Java sleep() method does not treat it any special way and merely uses its

libdocdel@fr.com

Long.MAX_VALUE value and treats it as the maximum value of a long, so that it just sleeps for a very lengthy period.

I have placed the sleep() call 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 in the constructor, because it looks like the constructor does not terminate (so is the object created or not?), so I prefer this placement. Note that although the constructor for MulticastRegister 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 have a GUI interface running, another simple way to keep them alive is to create an object and then wait for another thread to notify() it. Since nothing will, the thread (and hence the application) stays alive. Essentially, this is an unsatisfied wait that will never terminate—usually an erroneous thing to do, but here it is deliberate:

```
Object keepAlive = new Object();
synchronized(keepAlive) {
    try {
        keepAlive.wait();
    } catch(InterruptedException e) {
        // do nothing
    }
}
```

This will keep the service alive indefinitely, and it will not terminate unless interrupted. This is unlike sleep(), which will terminate eventually.

Running the MulticastRegister

The MulticastRegister program needs to be compiled and run with jini-core.jar and jini-ext.jar in its CLASSPATH. The extra jar file is needed because it contains the class files from the net.jini.discovery package. When run, the program will attempt to find all the service locators that it can. If there are none, it will find none—pretty boring. So one or more service locators should be set running in the network or on the local machine. Service locators running in the network must be accessible by multicast calls or they will not be found. This usually means that they will have to be on the same LAN as the MulticastRegister program.

libdocdel@fr.com

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

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 two things:

- the time-to-live (TTL) field on the packets
- the network administrator settings on routers and gateways

By default, the current implementation of `LookupDiscovery` sets the TTL to 15. Common network administrative settings restrict such packets to the local network. However, the TTL can be changed by giving the system property `net.jini.discovery.ttl` a different value. However, be careful about setting this; many people will get irate if you flood the networks with multicast packets.

ServiceRegistrar

The `ServiceRegistrar` is an abstract class that 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 JVM to another by Jini. It is shipped from the lookup service to the application looking for the lookup service, using a socket connection. From then on, 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. The implementation used by Sun's `reggie` uses RMI to communicate, but the application does not need to know this, and anyway, it could be done in different ways. This proxy object should not cache any information on the application side, but instead should get "live" information from the lookup service as needed. The implementation of the lookup service supplied by Sun does exactly this.

libdocdel@fr.com

The ServiceRegistrar object has two major methods. One is used by a service attempting to register:

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

The other method (with two forms) is used 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 methods are given in Chapter 5 and Chapter 6. For now, an overview will suffice.

A service provider will register a service 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 provider may register a singleton object that completely implements the service, but more likely it will register a service proxy that will communicate back to other objects in the service provider. Note carefully: the registered object will be shipped around the network, and 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 use a class object, and try to find instances of this class lying around in service locators. As discussed later in Chapter 6, 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 at how matching will be performed. This is done using Jini Entry objects, which are discussed in Chapter 4. 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 Chapter 5 or to Chapter 6 and come back to the discussion of entries in Chapter 4 later.

libdocdel@fr.com

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. So, in addition to using this object to register a service or to look up a service, you can use it to find out about the lookup locator. The major methods are these:

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

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

The second method, getLocator(), is more interesting. This returns exactly the same type of object as is used in the unicast lookup, but now its fields are filled in by the discovery process. You can find out which host the locator is running on, and its hostname, by using the following statement:

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

The following code shows how this can be used in the discovered() method to print information about each lookup service that replies to the multicast request:

```
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 the discovered() method to find out 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 IDs are used to give a globally unique identifier for the service (different services should not 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.

libdocdel@fr.com

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.

libdocdel@fr.com

CHAPTER 4

Entry Objects

A SERVICE IS EXPORTED TO LOOKUP SERVICES based on its class. Clients search for services using class information, typically using an interface. There is often additional information about a service that is not part of its class information, such as who owns the service, who maintains it, where it is located, and so on. Entries are used to pass this kind of additional information about services to clients. The clients can then use that information—as well as class type—to decide if a particular service is what it wants.

Entry Class

When a service provider registers a service, it places a copy of the service object (or a service proxy) on the lookup service. This copy is an instance of a class, albeit in serialized form. The server can optionally register sets of attributes along with the service object. Each set is described by an `Entry` object. What is stored on each service locator is an instance of a class along with a set of `Entry` objects, each of which describes some special additional attributes of the service.

For example, a set of file editors may be available as services. Each editor is capable of editing different types of files, as shown in Figure 4-1.

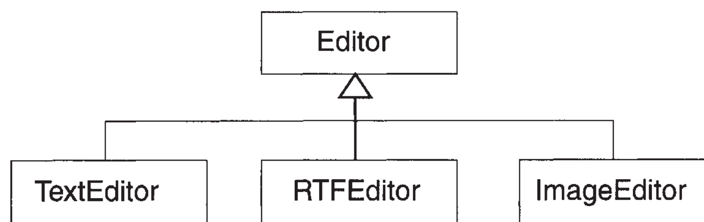


Figure 4-1. Editor class diagram

> **NOTE** *These classes would probably be interfaces, rather than instantiable classes.*

libdocdel@fr.com

In this situation, a client could search for a suitable editor in two ways:

- By asking for an instance of a specific class, such as `ImageEditor`
- 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, as defined by the services and possible clients. Services advertise themselves by exporting an object that is of a particular class and by exporting additional information along with this object. Jini clients can then search for specific types of services by asking for an object that implements the specific class, such as `ImageEditor`. They can also search for superclass objects and include extra objects to narrow the search, based on additional information advertised by the service. This additional information is given in objects belonging to subclasses of the `Entry` class.

The `Entry` class allows services to advertise their capabilities in very flexible ways. For example, suppose an editor was capable of handling a number of file types, such as plain text *and* RTF files. It could do so by exporting a service object implementing `Editor` along with an `Entry` object saying that it can handle plain text and another `Entry` object saying that it can handle RTF files. The service implementation can just add more and more information about its capabilities without altering the basic interface.

To manage this way of adding information, we could have a `FileType` class, which would give information about the types of files 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 an RTF editor, the attribute set would be `FileType("application/rtf")`.

For an editor capable of handling both plain text and RTF files, its capabilities would be given by using an array of entries:

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


libdocdel@fr.com

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

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

For example, to search for a plain text editor, an entry like this would be used:

```
Entry[] entries = new Entry[] {new FileType("plain/text")};
```

If any editor would do, the following entry could be used:

```
Entry[] entries = new Entry[] {new FileType(null)};
```

Attribute Matching Mechanism

The attribute matching mechanism is pretty basic. For example, a printer typically has the capacity to print a certain number of pages per minute, but if it specifies this using an `Entry`, it actually becomes rather hard to find. A client can request a printer service in which it does not care about speed, or it can request 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 with 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 the actual speed. This would be done outside of the Jini mechanisms, using whatever interface has been agreed on for the description of printers. A similar problem, that of finding a physically "close" service, is taken up in Chapter 13.

The attribute matching mechanism that was chosen by the Jini designers, 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, that would involve unmarshalling the service on the locator in order to run the `matches()` method, and this would slow the lookup service down and generate a couple of awkward questions:

- What security permissions should the filter run with?

libdocdel@fr.com

- What would happen if the filter modifies its arguments (deep copying to avoid this would cause further slowdowns)?

The ServiceDiscoveryManager—discussed in Chapter 15—has the ability to do client-side filtering to partly rectify this problem.

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 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 one of the primitive types, such as int or char, as a field. If one of these fields is required, then it must be wrapped up in a class such as Integer or Character. This make it easier to perform "wildcarding" for matching (see Chapter 5 for details). Jini uses null in the fields of Entry objects from the client to act as a wildcard. This will work for any class, including wrapper classes such as Boolean. The primitive types, such as boolean, have no values that can be used as a wildcard pattern, since all possible values (true and false) could be valid request values.

Jini places some further restrictions on the fields of Entry objects. They must be public, non-static, non-transient, and non-final. In addition, an Entry class must have a no-args constructor.

Convenience Classes

The AbstractEntry class implements the Entry interface, and it is designed as a convenience class. It implements methods such as equals() and toString(). An application would probably want to subclass this instead of implementing Entry.

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

- Address—The address of the physical component of a service.
- Comment—A free-form comment about a service.

libdocdel@fr.com

- `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 toward allowing someone to determine what a service (for example, a printer) does and if 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 can be derived.

For example, the `Address` class contains the following:

```
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.

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 the client wants to use. An expectation in this is that the information in an entry is primarily static. However, entries are objects, and they could also implement behavior as well as state.

libdocdel@fr.com

This should not be used to extend the behavior of a service, since all service behavior should be captured in the service interface specification.

A good example of a "non-static" entry is `ServiceType`, which is an abstract subclass of `AbstractEntry`. This contains "human oriented" information about a service, and contains abstract methods, such as `String getDisplayName()`. This method is intended to provide a localized name for the service. Localization (for example, producing an appropriate French name for the service for French-speaking communities) can only be done on the client side and will require code to be executed in the client to examine the locale and produce a name.

Another use is to define the user interface for a service. Services do not have or require user interfaces for human users, since they are defined by Java interfaces that can be called by any other Java objects. However, some services may wish to offer a way of interacting with themselves by means of a user interface, and this involves much executable code. Since it is not part of the service itself, this should be left in suitable `Entry` objects. This topic is looked at in detail in Chapter 19.

Summary

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

libdocdel@fr.com

CHAPTER 5

Service Registration

THIS CHAPTER LOOKS AT HOW SERVICES REGISTER themselves with lookup locators so that they can later be found by clients. From the service locator, the server will get a `ServiceRegistrar` object. The server will prepare a description of the service in a `ServiceItem` and will then call the `ServiceRegistrar`'s `register()` method with the `ServiceItem` as a parameter. The `ServiceItem` can contain additional information about a service in addition to its type, and this information is stored in `Entry` objects.

ServiceRegistrar

A server for a service finds a service locator using either a unicast lookup with a `LookupLocator` or a multicast search using `LookupDiscovery`. In both cases, a `ServiceRegistrar` object is returned to act as a proxy for the lookup service.

The server then registers the service with the service locator using the `ServiceRegistrar`'s `register()` method:

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

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

The second parameter here, `leaseDuration`, 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 Chapter 7.

The first parameter is of `ServiceItem` type.

ServiceItem

The service provider will create a `ServiceItem` object by using the constructor, shown here:

libdocdel@fr.com

```
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);
}
```

Once the service provider has created the `ServiceItem` object, it is passed into `register()`. The first parameter, `serviceID`, is set to `null` when the service is registered for the first time. The lookup service will set a non-null value as it registers the service. On subsequent registrations or re-registrations, this non-null value should be used. The `serviceID` is used as a globally unique identifier for the service.

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

- The object must be serializable. Some objects, such as the graphical user interface `JTextArea` object are not serializable 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, so it may need to be a proxy for the actual service. For example, the object may be able to show a set of toaster controls, but might have to send messages across the network to the real toaster service, which is connected to the physical toaster.

- If the service object is an RMI proxy, then the object in the `ServiceItem` is given by the programmer as the `UnicastRemoteObject` for the proxy stub, not the proxy itself. The Java runtime substitutes the proxy. This subtlety is explored in Chapter 6.

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

libdocdel@fr.com

Registration

The server attempts to register the service by calling `register()`. This may throw a `java.rmi.RemoteException`, which must be caught. The second parameter to the `register()` method is a request to the service locator for the length of time to store the service. The time requested may or may not be honored.

The return value is of type `ServiceRegistration`.

ServiceRegistration

The `ServiceRegistration` object is created by the lookup service and is returned to run in the service provider. This object acts as a proxy object that will maintain the state information for the service object exported to the lookup service.

Actually, the `ServiceRegistration` object can be used to make changes to the entire `ServiceItem` stored on the lookup service. The `ServiceRegistration` object maintains a `serviceID` field, which is used to identify the `ServiceItem` on the lookup service. The `serviceID` value can be retrieved by `getServiceID()` for reuse by the server if it needs to do so (which it should, so that it can use as the same identifier for the service across all lookup services). These objects are shown in Figure 5-1.

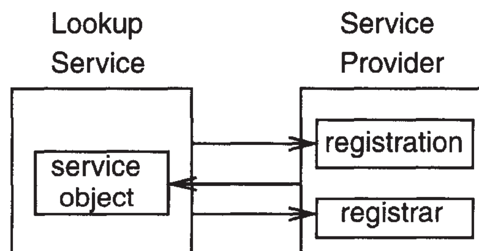


Figure 5-1. Objects in service registration

Other methods can be used to change the entry attributes stored on the lookup service, such as the following:

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

The final public method for the `ServiceRegistration` class is `getLease()`, which returns a `Lease` object that allows renewal or cancellation of the lease. This is discussed in more detail in Chapter 7.

libdocdel@fr.com

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

The SimpleService Program

A unicast server that exports its service and does nothing else is shown in the following `SimpleService` 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
 */

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 {
```


libdocdel@fr.com

```
            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
```

Running the SimpleService

The `SimpleService` 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.

The instance data for the service object is transferred in serialized form across socket connections. This instance data is kept in this serialized form by the lookup services. Later, when a client asks for the service to be reconstituted, it will use this instance data and also will need the class files. At this point, the class files will also need to be transferred, probably by an HTTP server. There is no need for additional RMI support services, such as `rmiregistry` or `rmid`, since all registration is done by the `register()` method.

libdocdel@fr.com

Information from the ServiceRegistration

The `ServiceRegistrar` object's `register()` method 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 these:

```
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 it will be discussed in more detail in Chapter 7. For now, we can just use it to find out how long the lease has been granted for by using its `getExpiration()` method:

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

Service ID

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

The `ServiceID` plays the role of unique identifier for a service. It is a 128-bit number generated in a pseudo-random manner and is effectively unique—the chance that the generator might duplicate this number is vanishingly small. Services do not generate this identifier because 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 means that it should save the ID in persistent storage and retrieve it on restarting. The previous code is not well-behaved in this respect.

libdocdel@fr.com

Entries

A server can announce a number of entry attributes when it registers a service 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. There is no limitation to the amount of information the server can include about the service, and it is better if the server gives more information than less; in later searches by clients, each entry is treated as though it were *OR*'ed with the other entries. In other words, the more entries that are given by the server, the more information is available to clients, and the greater the chance of matching a client's requirements.

For example, suppose we have a coffee machine on the seventh level in room 728 of our building, which is known as both "GP South Building" and "General Purpose South Building." Information such as this, and general information about the coffee machine, can be encapsulated in the convenience classes `Location` and `Comment` from the `net.jini.lookup.entry` package. If this were on our network as a service, it would advertise itself as follows:

```
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, ...);
```

Summary

A service uses the `ServiceRegistrar` object, which is returned as a proxy from a locator, to register itself with that locator. The server prepares a `ServiceItem` that contains a service object and a set of entries, and the service object may be a proxy for the real service. The server registers this service and entry information using the `register()` method of the `ServiceRegistrar` object.

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

libdocdel@fr.com

CHAPTER 6

Client Search

THIS CHAPTER LOOKS AT WHAT THE CLIENT has to do once it has found a service locator and wishes to find a service. From the service locator, the client will get a `ServiceRegistrar` object. To find a service from the locator, the client needs to prepare a description of the service, which it does using a `ServiceTemplate` object. The client will then call one of two methods on the `ServiceRegistrar` to return either a single matching service or a set of matching services.

Searching for Services with the ServiceRegistrar

A client gets a `ServiceRegistrar` object from the lookup service, and it uses the `lookup()` method to search for a service stored on that lookup service. Here is 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 (as many as `maxMatches`).

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;
```


libdocdel@fr.com

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

Although each service should have been assigned a `serviceID` by a lookup service, a client might not know the `serviceID` (it could be the first time the client has looked for this service, for example). In this case, the `serviceID` is set to `null`. If the client does know the `serviceID`, then it can set the value to find the service. The `attributeSetTemplates` is a set of `Entry` elements used to match attributes, and it will be discussed in the "Matching Services" section, later in this chapter.

The major parameter of the `lookup()` methods is a `ServiceTemplate`, which contains a list of `serviceTypes`. We know that services export instances of a class, but how does the client ask so that it gets a suitable instance delivered from the lookup locator?

Although the lookup services keep instances of objects for the service, the client will only know about a service from its specification (unless it already has a `serviceID` for the service), and the specification will almost certainly be a Java interface. Therefore, the client needs to ask using this interface. An interface can have a class object just like ordinary classes, so the list of `serviceTypes` will typically be a list of class objects for service interfaces. Thus, the client will usually request an interface object.

To be more concrete, suppose a toaster is defined by this interface:

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

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

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


libdocdel@fr.com

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 it will make a request like this:

```
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 lookup() can throw an exception. This can occur if, for example, the service requested cannot be de-serialized.

As a result of calling the lookup() method, an object (an instance of a class implementing the Toaster interface) has been transported across to the client, and the object has been coerced to be of this Toaster type. This object has two methods: setDarkness() and startToasting(). No other information is available about the toaster's 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 either of the two methods:

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

Before leaving this discussion, you might wonder what the role of System.setSecurityManager(new RMISecurityManager()) is. A serialized object has been transported across 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

libdocdel@fr.com

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 the Toaster object 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 does. So if the client had the class definitions of all possible toasters, then it would never need to load a class and would not need a security manager that allows classes to be loaded across the network. This is not likely, but may perhaps be needed in a high-security environment.

Receiving the ServiceMatches Object

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 using the maxMatches parameter of the second lookup() method. The client gets back a ServiceMatches object that looks like this:

```
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. In that case, totalMatches will be set to 5 after a lookup. However, if you used maxMatches to limit the search to 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 serializable. No exception is thrown here, because although one match might be bad, the others might still be okay. So a value of null as the array element value is used to signify this. The following code shows how to properly handle the ServiceMatches object:

```
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++) {
```


libdocdel@fr.com

```
        Toaster toaster = (Toaster) matches.items[n].service;
    if (toaster != null) {
        toaster.setDarkness(1);
        toaster.startToasting();
        }
    }
```

This code will start up to ten toasters cooking at once!

Matching Services

As mentioned previously, a client attempts to find one or more services that satisfy its requirements 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;
```

If the client is repeating a request, then it may have recorded the `serviceID` from an earlier request. The `serviceID` is a globally unique identifier, so it can be used to identify a service unambiguously. This `serviceID` can be used by the service locator as a filter to quickly discard other services.

Alternatively, a client may want to find a service satisfying several 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). The client will fill the `serviceTypes` array with all of the interface classes that the service has to implement.

And finally, the client will specify a set of attributes in the `attrSetTemplates` field 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.

The more formal description that follows comes 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`

libdocdel@fr.com

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.

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 the `ServiceRegistrar` object by preparing a `ServiceTemplate` object and calling 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.

libdocdel@fr.com

CHAPTER 7

Leasing

In distributed applications, there may be partial failures of the network or of components on the network. Leasing is a way for components to register that they are alive, but to ensure that they are "timed out" if they fail or are unreachable. Leasing is the mechanism used between applications to give access to resources over a period of time in an agreed manner.

Leases are requested for periods of time, and these requests may be granted, modified, or denied. The most common example of a lease is when a service is registered with lookup services. A lookup service will not want to keep a service forever, because it may disappear. Keeping information about nonexistent services is a waste of resources on the lookup service and also may lead to clients wasting time trying to access services that aren't there. As a result, a lookup service will grant a lease saying that it will only keep information for a certain period of time, and the service can renew the lease later if it wants to.

Requesting and Receiving Leases

Leases are requested for a period of time. In Jini, a common use of leasing is for a service provider to request that a copy of the service be kept on a lookup service for a certain length of time, for delivery to clients on request. The service provider requests a time in the `ServiceRegistrar`'s `register()` method. Two special values of the time are

- `Lease.ANY`–the service lets the lookup service decide on the time
- `Lease.FOREVER`–the request is for a lease that never expires

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

libdocdel@fr.com

These objects are shown in Figure 7-1. The following are typical calls to register the service and then find the lease:

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

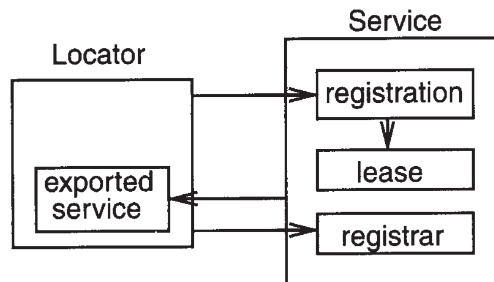


Figure 7-1. Objects in a leased system

The principal methods of the Lease object are these:

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

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


libdocdel@fr.com

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.

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. While it might seem nice to get warning of a lease expiring so that it can be renewed, this would have to be done in advance of the expiration ("I'm just about to expire; please renew me quickly!") but this would complicate the leasing system and not be completely reliable anyway (for example, how far in advance is soon enough?).

Instead, it is up to the service provider to call `renew()` before the lease expires if it wishes the lease to continue. The parameter for `renew()` is in milliseconds, and represents an extra duration from now. This is in contrast to the expiration time returned from `getExpiration()`, which is measured since the epoch.

Renewing Leases

Jini supplies a `LeaseRenewalManager` class that looks after the process of calling `renew()` at suitable times.

```
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);
```


libdocdel@fr.com

> **WARNING** *In Jini 1.0, this class was in package* `com.sun.jini`*; in Jini 1.1 it is now in package* `net.jini.lease`.

The `LeaseRenewalManager` manages a set of leases, which may be set by a constructor or added later by `renewFor()` or `renewUntil()`. The time requested in these methods is in milliseconds. The `expiration` time is measured since the epoch, whereas the `duration` time is measured 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, which will allow the application to take corrective action if its lease is denied. If the listener is `null`, then no notification will take place.

If you are using Jini 1.0, you have to be careful about setting the `duration` in `renewFor()` due to a bug that has since been fixed. If you want the service to be registered forever, it is tempting to use `Lease.FOREVER`. However, the Jini 1.0 implementation just adds this to `System.currentTimeMillis()`, which overflows to a negative value that is not checked. As a result, it never does any renewals. You need to check

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

before calling `renewFor()`. This is fixed in Jini 1.1. The `renewUntil()` method can use `Lease.FOREVER` with no problems.

Granting and Handling Leases

The preceding discussion looked at leases from the side of the 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 that you can skip for now if you want—it is not really needed until Chapter 14. An example of creating a lease is also given in Chapter 13.

A lease can be granted for almost any remote service—any one where one object wants to maintain information about another one that is not within the same virtual machine. As with other remote services, there are the added partial failure modes, such as network crash, remote service crash, timeouts, and so on. An object that keeps information on a remote service will hand out a lease to the service and will want the remote service to keep "pinging" it periodically to say that it is still alive and that it wants the information kept. Without this periodic assurance, the object might conclude that the remote service has vanished or is somehow unreachable, and that it should discard the information about it.

libdocdel@fr.com

Leases are a very general mechanism for allowing one service to have confidence in the existence of the other for a limited period. Because they are general, they allow for a great deal of flexibility in use. Because of the potential variety of services, some parts of the Jini lease mechanism cannot be completely defined and must be left as interfaces for applications to fill in. This generality means that all of the details are not filled in for you, as your own requirements cannot be completely predicted in advance.

A lease is given as an interface, and any agent that wishes to grant leases must implement this interface. Jini provides 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. A couple of simple possibilities are these:

- Always grant the requested time.
- Ignore the requested time and always grant a fixed time.

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

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

Abstract Lease

An abstract lease gives a basic implementation of a lease that can almost be used for simple leases.

```
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);
}
```


libdocdel@fr.com

> **WARNING** *This class, and those that depend on it, are still not fully specified and may change in future versions of Jini.*

This class supplies straightforward implementations of much of the `Lease` interface, with three provisos:

- The constructor is `protected`, so that constructing a lease with a specified duration is devolved to a subclass. This means that a lease duration policy must be set by this subclass.
- The `renew()` method calls into the abstract `doRenew()` method, again to force a subclass to implement a renewal policy.
- The `Lease` interface does not implement the `cancel()` method, so this must also be left to a subclass.

Thus, this class implements the easy things, and leaves all matters of policy to concrete subclasses.

Landlord Lease Package

The *landlord* is a package that allows 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. 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. Leases are identified 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 itself, as it can use a landlord lease factory to do this. (But, of course, it can create them, depending on how an implementation is done.) When a client wishes to cancel or renew a lease, it asks the lease to perform the renewal, and in turn the lease 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.

libdocdel@fr.com

The principal classes and interfaces in the `landlord` package are shown in Figure 7-2, where the interfaces are shown in italicized font and the classes in normal font.

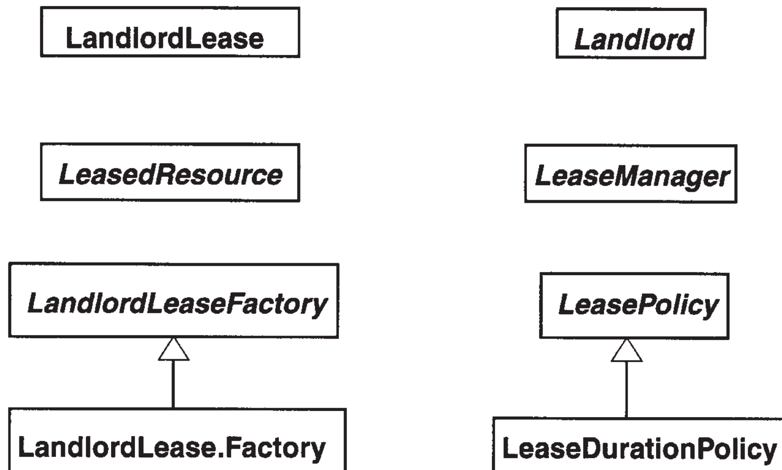


Figure 7-2. Class diagram of the landlord package

This fairly complex set of classes and interfaces is driven by a number of factors:

- The key object in a landlord system is the landlord itself. Because there are many ways that a landlord could manage a set of leases, the `Landlord` is an interface rather than a class, with many possible implementations.

- Because there are many possible landlords, there could be many possible lease-types created, which will all be subclasses of `Lease`. A common design pattern in such a circumstance is to use a *factory* object to create the leases. These factory objects will implement the `LandlordLeaseFactory` interface.

- A simple lease implementation was needed for a variety of situations, and this is the `LandlordLease` class. When a particular implementation is chosen, the factory pattern says that a new factory is needed to create new objects. So to create `LandlordLease` objects, the `LandlordLease.Factory` factory class is used. (Note the dot (`.`) in the `LandlordLease.Factory` class name, which distinguishes it from the `LandlordLeaseFactory` interface.) A lease (on the client) also requires the existence of some handler for its methods on the lease-granting side, which is the landlord.

- To handle all policy issues, such as initial granting of lease times, and requests for lease renewal, a policy object is used. There can be many possible policies

libdocdel@fr.com

implementing the LeasePolicy interface. Each lease policy needs to make decisions about leases, but it needs to make decisions on the lease-granting side, so a lease policy needs to keep enough information locally to make proper decisions. The information about leases on the granting side is kept in *leased resources*, which are implementations of the LeasedResource interface.

- For each lease on the client side, there will be a leased resource on the granting side. These must be stored and managed somehow. There may be only 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 that would be wrong half of the time, lease management is just left as an interface.

Java uses interfaces as specifications without implementation details. For individual classes this is often fine. However, using interfaces can be limiting when you are dealing with a set of classes that are expected to interact in certain ways. Interfaces do not show the interactions that may need to exist in order for an implementation of the set of classes to function together. This means the interface definitions are not complete as they stand, because they fail to show the links between classes that must exist in any implementation. To see what these links actually are, let us look at a simple implementation for the Foo landlord package.

If we have a landlord for a Foo resource, then we could end up with the class structure shown in Figure 7-3.

This diagram uses a UML class diagram annotated with arrows and multiplicities. An association with an arrow means that the object at the source of the arrow will know about the object at the other end of the arrow. For example, each LandlordLease knows about (has a reference to) a FooLandlord, but the landlord does not know about any leases. At each end of each association between classes, the multiplicity of that end of the link is also shown. A "*" is a wildcard pattern, meaning "zero to many." So for example, any number of LandlordLeases (from zero upwards) may know about a single FooLandlord.

Some comments are appropriate about the directions and multiplicities:

- A landlord can be managing many leases, but it doesn't know what the leases are—the leases know their landlord, and they call its methods using the lease cookie. So many LandlordLease objects contain a reference to a FooLandlord.

- Certain requests need to be forwarded through the system. For example, a renew() request from a lease will get passed to a landlord. The landlord cannot handle it directly, since the renewal is a matter requiring policy decisions. It must be passed to a lease policy object. One way of doing this (as shown in Figure 7-3) is for the landlord to have a reference to a lease

libdocdel@fr.com

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.

- A factory may be used by many lease policies, a policy may be used by many lease managers, and a lease manager may be used by many landlords.

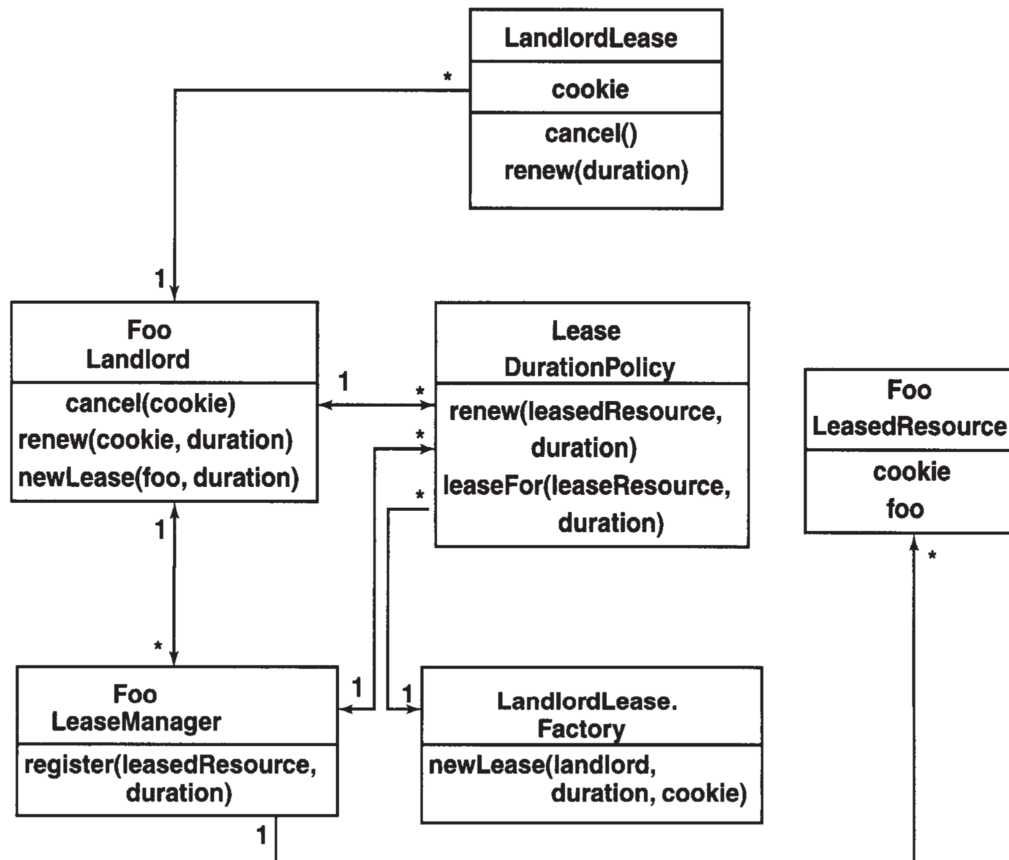


Figure 7-3. Class diagram of a landlord implementation

LandlordLease Class

The `LandlordLease` class extends `AbstractLease`. This class has the private fields `cookie` and `landlord`, as shown in Figure 7-4.

libdocdel@fr.com

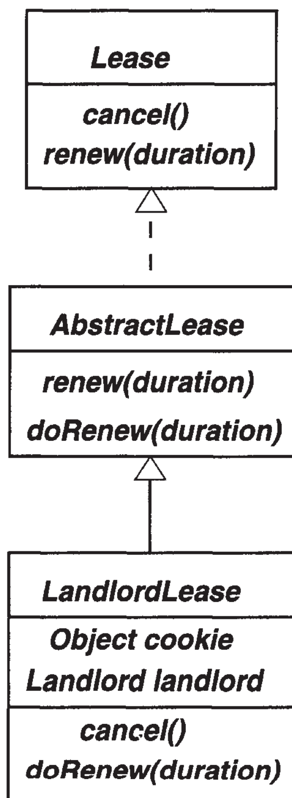


Figure 7-4. The class diagram for LandlordLease

Implementation of the methods `cancel()`and `doRenew()` in the `LandlordLease` is deferred to its landlord. The implementation of these methods in `LandlordLease` simply passes the requests on to the landlord:

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

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

The `LandlordLease` 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 the landlord system, on the lessor side, the cookie is used as an identifier for the lease.

libdocdel@fr.com

LeasedResource Interface

A LeasedResource is a convenience wrapper around a resource that includes extra information about a lease and methods for use by landlords. It defines an interface as follows:

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

This interface 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 that has been given out to a client.

An implementation of LeasedResource will typically include the resource that is leased, plus a method of setting the cookie. The following code shows an example:

```
/**
 * FooLeasedResource.java
 */
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;
    }
```


libdocdel@fr.com

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

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

LeasePolicy Interface

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 interface includes a factory method, `leaseFor()`, that returns a lease based on the policy and request.

LeaseDurationPolicy Class

An implementation of the `LeasePolicy` interface is given by `LeaseDurationPolicy` class. This class grants and renews leases based on constant values for maximum and default lease durations, as shown here:

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

public class LeaseDurationPolicy implements LeasePolicy {

    public LeaseDurationPolicy(long maximum, long defaultLength,
```


libdocdel@fr.com

```
        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 addition 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 duration is to set a hard upper limit (which could be, say, `Lease.FOREVER`), while the default is what is granted if the client asks for a duration of `Lease.ANY`.

LeaseManager Interface

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 `renew()` methods of the policy. These set or alter 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`.

The `LeaseManager` interface is defined as follows:

```
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 `LeaseManager` 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 `LeaseManager` will look after a set of leases (really, their resources) by adding a new lease resource to its set for each lease, 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, by the lease when the lease's `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

libdocdel@fr.com

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 the question 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 the `LeasePolicy` implementation, we are okay here.

A third question is 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, because it has knowledge of all the leases and their duration. Possible ways of doing this include the following:

- A thread per lease, which will sleep and time out when the lease should expire. This will need to sleep again if the lease has been renewed in the meantime.

- 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.

- A polling mechanism in which a thread sleeps for a fixed time and then cleans up all leases that have expired in the meantime.

- 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. (This lazy approach is taken by the JavaSpaces Outrigger service, which grants leases for Entry objects).

The `FooLeaseManager` implements this third polling mechanism method:

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


libdocdel@fr.com

```
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() {
```


libdocdel@fr.com

```
            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);

                        }
                    }
                }
            }
        }

    } // FooLeaseManager
```


libdocdel@fr.com

Landlord Interface

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 leases in batches, and will not be dealt with here. The Landlord interface is as follows:

```
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, and in the FooLandlord implementation, it gets this policy from the FooLeaseManager. The cancel() method needs to modify the list of leases, and in the FooLandlord implementation, it passes this on to 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, and 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 serializing the lease and sending it to this remote client. While serializing it, the landlord field will also be serialized and sent to the client.

libdocdel@fr.com

When the client methods such as `renew()` are called, the implementation of the `LandlordLease` will make a call to the landlord. The lease is on the client, which by then will be remote from its origin where the landlord lives. That means 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 us this:

```
/**
 * FooLandlord.java
 */

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) {
```


libdocdel@fr.com

```
            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
```

Building an implementation of the landlord package, such as the Foo package, means providing implementations of the `Landlord`, `LeasedResource`, and `LeaseManager` interfaces. This has been done using the `FooLandlord`, `FooLeasedResource`, and `FooLeaseManager` classes.

Summary

Leasing allows resources to be managed without complex garbage-collection mechanisms. Leases received from services can be dealt with easily, using `LeaseRenewalManager`. Entities that need to hand out leases can use a system, such as the landlord system, to handle these leases.

libdocdel@fr.com

CHAPTER 8

A Simple Example

THIS CHAPTER LOOKS AT A SIMPLE PROBLEM to give a complete example of a Jini service and client.

Before a Jini service can be built, common knowledge must be defined about the type of service that will be offered. This involves designing a set of "well-known" classes and interfaces. Based on a well-known interface, a client can be written to search for and use services implementing the interface.

The client can use either a unicast or multicast search to find services, but it will be uninterested in how any particular service is implemented. This chapter looks at building clients using both methods, and these clients will be heavily reused throughout the rest of the book.

The service, on the other hand, is implemented by each vendor in a different way. This chapter discusses a simple choice, with alternatives being dealt with in the next chapter. It is difficult to get a Jini service and client functioning correctly, as there are many configuration issues to be dealt with. These are discussed in some detail.

By the end of this chapter you should be able to build a client and a service, and configure your system so that they are able to run and communicate with each other.

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:

- By examining the file's name
- By examining 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 way of classifying files is into MIME types, such as `text/plain` and `image/gif`. There are tables of "official" MIME types (unofficial ones can be added

libdocdel@fr.com

on an ad hoc basis), and there are also tables of mappings from filename endings to corresponding MIME types. These tables have entries such as these:

```
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
```

These tables are stored in files for applications to access.

Storing these tables separately from the applications that would use them is considered bad from the object-oriented point of view, since each application would need to have code to interpret the tables. Also, 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 could define a MIME class as follows:

```
package standalone;

/**
 * MIMEType.java
 */

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;
```


libdocdel@fr.com

```
        this.subtype = subtype;
    }

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

We could then define a mapping class like this:

```
package standalone;

/**
 * FileClassifier.java
 */

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, because it just acts as a lookup table via its static method `getMIMEType()`.

Applications can make use of these classes as they stand, by simply compiling them and having the class files available at run time. This would still result in duplication throughout JVMs, possible multiple copies of the class files, and potentially severe maintenance problems if applications need to be recompiled, so it might be better to have the `FileClassifier` as a network service. Let's consider what would be involved in this.

libdocdel@fr.com

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. It will want an object that belongs to the `FileClassifier` class or one of its subclasses and will not usually care which of these it gets, as long as it contains the method `getMIMEType()`.

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

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

1. This is the silly option: push the entire implementation up to 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, because it has all the information needed! This doesn't lend itself to flexibility with new unknown services coming along, because the client already has to know the details. So this option is not feasible.

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 this changes over time, there is a maintenance issue with the possibility of version "skew." This can be used for Jini; it just isn't the best way.

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 an object for this interface, which implements the interface in some way or other. The client generally will not care how the object does this. 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 service provider deals in terms of a Java class that implements 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 it is enough to allow a service to be specified and to get on with building the client.

libdocdel@fr.com

One possibility for service implementation is looked at later in this chapter, and the next chapter is devoted to the full range of possibilities.

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.

MIMEType

The `MIMEType` class is known to the client and to any file-classifier service. The `MIMEType` class file can be expected to be known to the JVMs of all clients and services. That is, this class file needs to be in the `CLASSPATH` of every file-classifier service and of every client that wants to use a file-classifier service.

The `getMIMEType()` method will return an object from the file-classifier service. There are implementation possibilities that can affect this object:

- If the service runs in the client's JVM, then nothing special needs to be done.
- If the service is implemented remotely and runs in a separate JVM, then the `MIMEType` object must be serialized for transport to the client's JVM. For this to be possible, it must implement the `Serializable` interface. Note that while the class files are accessible to both client and service, the instance data of the `MIMEType` object needs to be serializable to move the object from one machine to the other.

There can be differences in the object depending on the implementation. If it implements `Serializable`, it can be used in both the remote and local cases, but if it doesn't, it can only be used in the local case.

Making decisions about interfaces based on future implementation concerns is traditionally rated as poor design. In particular, the philosophy behind remote procedure calls is that they should hide the network as much as possible and make the calls behave as though they were local calls. With this philosophy, there is no need to make a distinction between local and remote calls at design time. However, a document from Sun, "A Note on Distributed Computing" by Jim Waldo and others, argues that this is wrong, particularly in the case of distributed objects. The basis of their argument is that the network brings in a host of other factors, in particular that of partial failure. That is, part of the network, itself, may fail, or a component on the network may fail without all of the network or all of the components failing. If other components do not make allowance for this possible (or maybe

libdocdel@fr.com

even likely) behavior, then the system as a whole will not be robust and could be brought down by the failure of a single component.

According to this document, it is important to determine whether the objects could be running remotely and to adjust interfaces and classes accordingly at the design stage. This lets you to take into account possible extra failure modes of methods, and in this case, an extra requirement on the object. This important paper is reprinted in the Jini specification book from Sun (*The Jini Specification* by Ken Arnold and others) and is also at `http://www.sun.com/research/techrep/1994/abstract_29.html`.

These considerations lead to an interface that adds the `Serializable` interface to the original version of the `MIMEType` class, as objects of this class could be sent across the network.

```
package common;

import java.io.Serializable;

/**
 * MIMEType.java
 */

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
```


libdocdel@fr.com

FileClassifier Interface

Changes have to be made to the file-classifier interface, as well. First, interfaces cannot have static methods, so we will have to turn the `getMIMEType()` method into a public instance method.

In addition, all methods are defined to throw a `java.rmi.RemoteException`. This type of exception is used throughout Java (not just by 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 a class is not required to be `Remote` if its methods throw `RemoteExceptions`. If all the methods of a class throw `RemoteException`, it does not mean the class implements or extends `Remote`. It only means that an implementation may be implemented as a remote (distributed) object, and that an implementation might also use the RMI `Remote` interface.

There are some very fine points to this, which you can skip if you want. Basically, though, you can't go wrong if every method of a Jini interface throws `RemoteException` and the interface does not extend `Remote`. In fact, prior to JDK 1.2.2, making the interface extend `Remote` would force each implementation of the interface to actually be a remote object. At JDK 1.2.2, however, the semantics of `Remote` were changed a little, and this requirement was relaxed. From JDK 1.2.2 onwards, an interface can extend `Remote` without implementation consequences. At least, that is almost the case: "unusual" ways of implementing RMI, such as over IIOP (IIOP is the transport protocol for CORBA, and RMI can use this), have not yet caught up to this. So for maximum flexibility, just throw `RemoteException` from each method and don't extend `Remote`.

Doing so gives the following interface:

```
package common;

/**
 * FileClassifier.java
 */

public interface FileClassifier {

    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

libdocdel@fr.com

and should never change. The implementation discussed later in this chapter does not throw such an exception. However, other implementations in other sections use a `Remote` implementation, and this will require that the method throws a `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.

The Client

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

Unicast Client

If there is a known service locator that will know about the service, then there is no need to search for the service locator. This doesn't mean that the location of the service is known, only the location of the locator. For example, there might be a (fictitious) organization "All About Files" at `www.all_about_files.com` that would know about various file services, keeping track of them as they come online, 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
 */

public class TestUnicastFileClassifier {
```


libdocdel@fr.com

```
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 {
```


libdocdel@fr.com

```
            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 is illustrated in Figure 8-1. This shows a UML class diagram, surrounded by the JVM in which the objects exist.

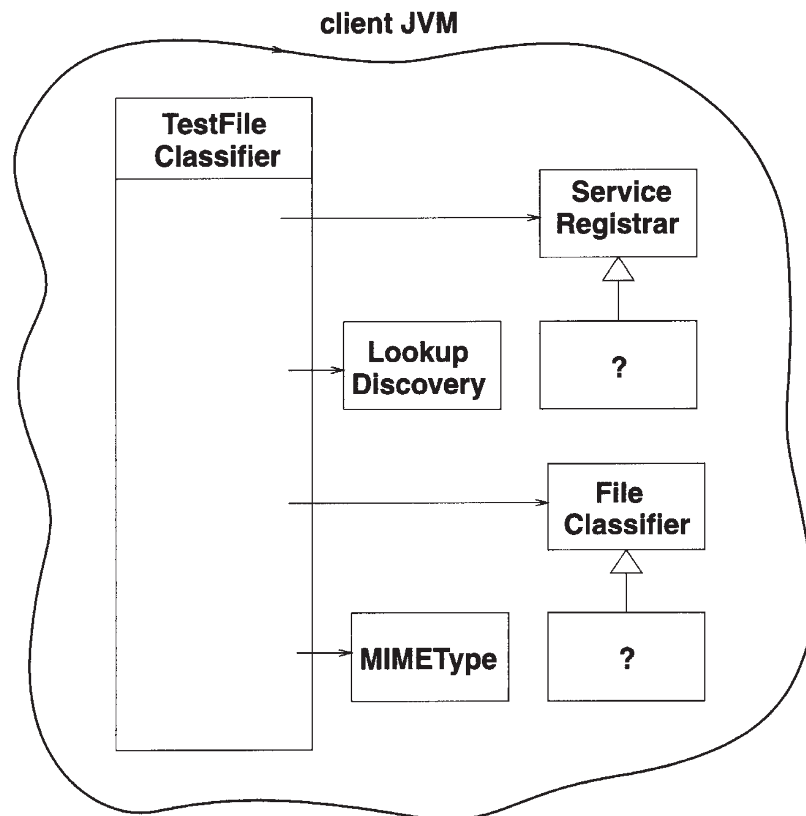


Figure 8-1. Objects in client JVM

The client has a main `TestFileClassifier` class, which has two objects of types `LookupDiscovery` and `MIMEType`. It also has objects that implement the interfaces `ServiceRegistrar` and `FileClassifier`, but it doesn't know, or need to know, what classes they are. These objects have come across the network as implementation objects of the two interfaces.

libdocdel@fr.com

Figure 8-2 shows the situation when the service locator's JVM is added in. The lookup service has an object implementing `ServiceRegistrar`, and this is the object exported to the client.

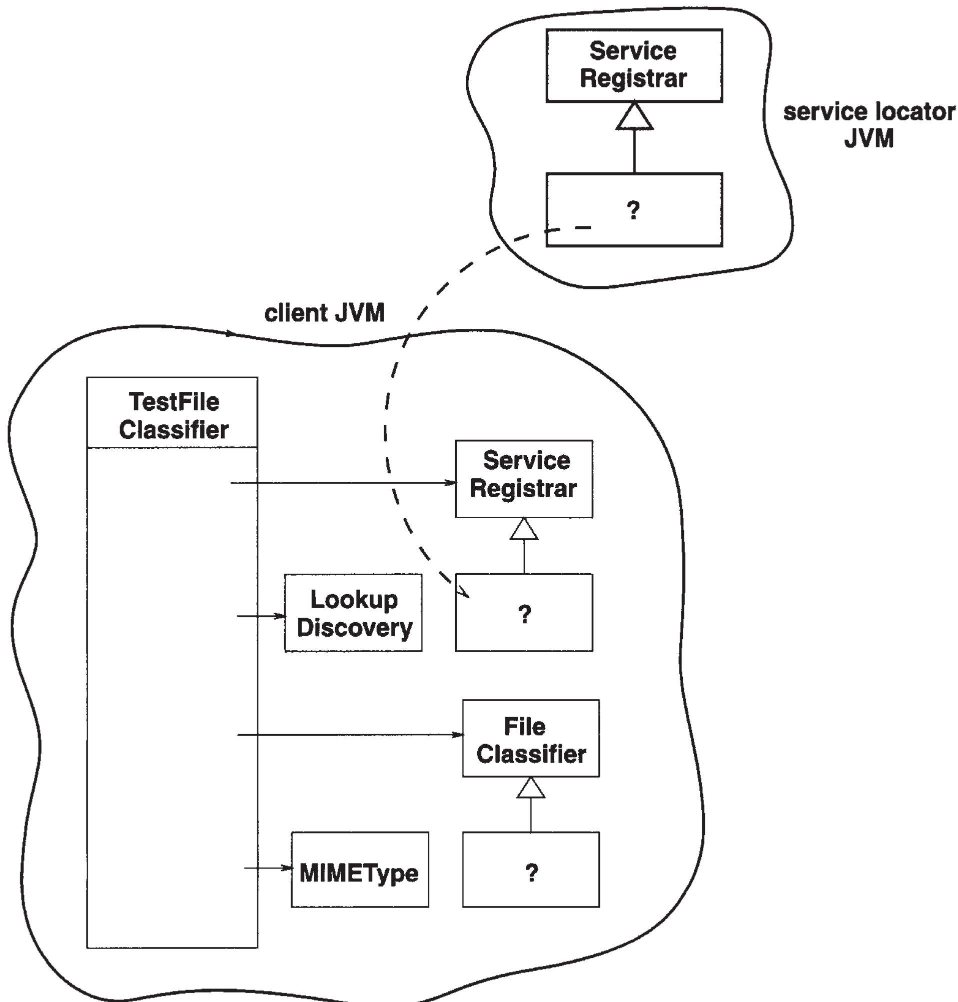


Figure 8-2. Objects in the client and service locator JVMs

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

The figure also shows a question mark for the object in the client implementing `FileClassifier`. The source of this object is not yet shown; it will get the object from a service, but we haven't yet discussed any of the possible implementations

libdocdel@fr.com

of a `FileClassifier` service. One implementation will be discussed in the "Uploading a Complete Service" section later in this chapter, and others will be discussed in Chapter 9.

Multicast Client

We have looked at the unicast client, where the location of the service locator is already known. However, it is more likely that a client will need to search for service locators until 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 exit after using the service. More complex behavior will be illustrated in later examples.

In this situation, 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. Therefore, in `main()` a user thread 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
 */

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) {
```


libdocdel@fr.com

```
            // 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();
                continue;
            }
            if (classifier == null) {
                System.out.println("Classifier null");
                continue;
            }

            // Use the service to classify a few file types
            MIMEType type;
            try {
```


libdocdel@fr.com

```
                    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());
                    continue;
                }
                // success
                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
```

Exception Handling

A Jini program can generate a huge number of exceptions, often related to the network nature of Jini. This is not accidental, but lies at the heart of the Jini approach to network programming. Services can disappear because the link to them has vanished, the server machine has crashed, or the service provider has died. Class files can disappear for similar problems with the HTTP server that delivers them.

libdocdel@fr.com

Timeouts can occur due to unpredictable network delays. Many of these exceptions have their own exception types, such as `LookupUnmarshalException`, which can occur when unmarshalling objects. Many others are simply wrapped in a `RemoteException`, which has a `detail` field for the wrapped exception.

Since many Jini calls can generate exceptions, these must be handled somehow. Many Java programs (or rather, their programmers!) adopt a somewhat cavalier attitude to exceptions: catch them, maybe put out an error message, and continue—Java makes it easy to handle errors! More seriously, whenever an exception occurs, the question has to be asked: Can the program continue, or has its state been corrupted but not so badly that it cannot recover, or has the program's state been damaged so much that the program must exit.

The multicast `TestFileClassifier` of the last section can throw exceptions at a number of places:

- The `LookupDiscovery` constructor can fail. This is indicative of some serious network error. The created `discover` object is needed to add a listener, and if this cannot be done, then the program really can't do anything. So it is appropriate to exit with an error value.

- The `ServiceRegistrar.lookup()` method can fail. This is indicative of some network error in the connection with a particular service locator. While this may have failed, it is possible that other network connections may succeed. The application can restore a consistent state by skipping the rest of the code in this iteration of the `for()` loop by using a `continue` statement.

- The `FileClassifier.getMIMEType()` method can fail. This can be caused by a network error, or perhaps the service has simply gone away. Regardless, consistent state can again be restored by skipping the rest of this loop iteration.

Finally, if one part of a program can exit with an abnormal (non-zero) error value, then a successful exit should signal its success with an exit value of `0`. If this is not done, then the exit value becomes indeterminate and is of no value to other processes that may wish to know whether the program exited successfully or not.

The Service Proxy

A service will be delivered from out of a service provider. That is, a server will be started to act as a service provider. It will create one or more objects, which between them will implement the service. Amongst these will be a distinguished object—the service object. The service provider will register the service object with service locators and then will wait for network requests to come in for the service.

libdocdel@fr.com

What the service provider will actually export as a service object is usually a proxy for the service. The proxy is an object that will eventually run in a client and will usually make calls back across the network to service backend objects. These backend objects running within the server actually complete the implementation of the service.

The proxy and the service backend objects are tightly integrated; they must communicate using a protocol known to them both, and they must exchange information in an agreed upon 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" (or "smart"), which means it does a lot of processing on the client side. Backend object(s) within the service provider are 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 "fat" backend objects, and most processing will be done by the backend objects running in the service provider.

As well as this choice of size, there is also a choice of communication mechanisms between the client and service provider objects. Client/server systems often have the choice of message-based or remote procedure call (RPC) communications. 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 need to use this, but many implementations of Jini proxies will do so because it is easy. RMI does force a particular choice of thin proxy to fat service backend, 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 implementation (the service backend is an empty set of objects). This is the simplest way of implementing the file-classifier service, but not always the most desirable. Chapter 9 will look in more detail at the other possibilities.

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 dependent, and does not make use of any files on the host side. 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.

libdocdel@fr.com

FileClassifier Implementation

The implementation of the `FileClassifier` is straightforward:

```
package complete;

import common.MIMEType;
import common.FileClassifier;

/**
 * FileClassifierImpl.java
 */

public class FileClassifierImpl implements FileClassifier, java.io.Serializable {

    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
```

FileClassifierServer Implementation

The service provider for the file-classifier service needs to create an instance of the exportable service object, register this, and keep the lease alive. In the `discovered()`

libdocdel@fr.com

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.

The following code uses an "unsatisfied wait" condition that will sleep forever until interrupted. Note that if the server does terminate, then the lease will fail to be renewed and the exported service object 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. This may be the first time this service is ever run, or at least the first time it is ever run with this particular implementation. Since service IDs are issued by lookup services, it must remain null until at least the first registration. Then the service ID can be extracted from the registration and reused for all further lookup services. In addition, the service ID can be saved in some permanent form so that if the server crashes and restarts, the service ID can be retrieved from permanent storage. The following server code saves and retrieves this value in a FileClassifier.id file. Note that we get the service ID from the registration, not from the registrar.

```
package complete;

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

import java.io.*;

/**
 * FileClassifierServer.java
 */

public class FileClassifierServer implements DiscoveryListener,
```


libdocdel@fr.com

```
                                         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
    Object keepAlive = new Object();
    synchronized(keepAlive) {
        try {
            keepAlive.wait();
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }
}

public FileClassifierServer() {
    // Try to load the service ID from file.
    // It isn't an error if we can't load it, because
    // maybe this is the first time this service has run
    DataInput din = null;
    try {
        din = new DataInputStream(new FileInputStream("FileClassifier.id"));
        serviceID = new ServiceID(din);
    } catch(Exception e) {
        // ignore
    }

    System.setSecurityManager(new RMISecurityManager());

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

    discover.addDiscoveryListener(this);
```


libdocdel@fr.com

```
}

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());
            continue;
        }
        System.out.println("Service registered with id " + reg.getServiceID());

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

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

            // try to save the service ID in a file
            DataOutputStream dout = null;
            try {
                dout = new DataOutputStream(new
                               FileOutputStream("FileClassifier.id"));
                serviceID.writeBytes(dout);
                dout.flush();
            } catch(Exception e) {
                // ignore
            }

        }
    }
}

public void discarded(DiscoveryEvent evt) {
```


libdocdel@fr.com

```
    }

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

} // FileClassifierServer
```

Figure 8-3 shows the server, by itself, running in its JVM.

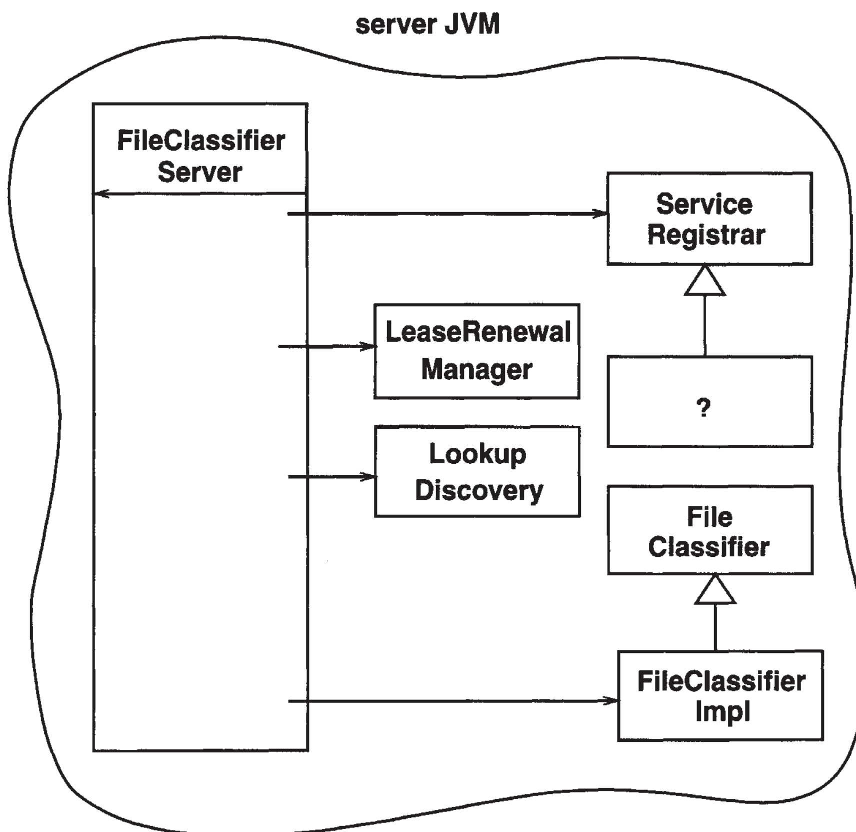


Figure 8-3. Objects in the server JVM

Figure 8-2 showed that the client receives an object implementing `ServiceRegistrar` from the service locator (such as `reggie`). When we add in the service locator and the client in their JVMs, the picture looks like what is shown in Figure 8-4.

libdocdel@fr.com

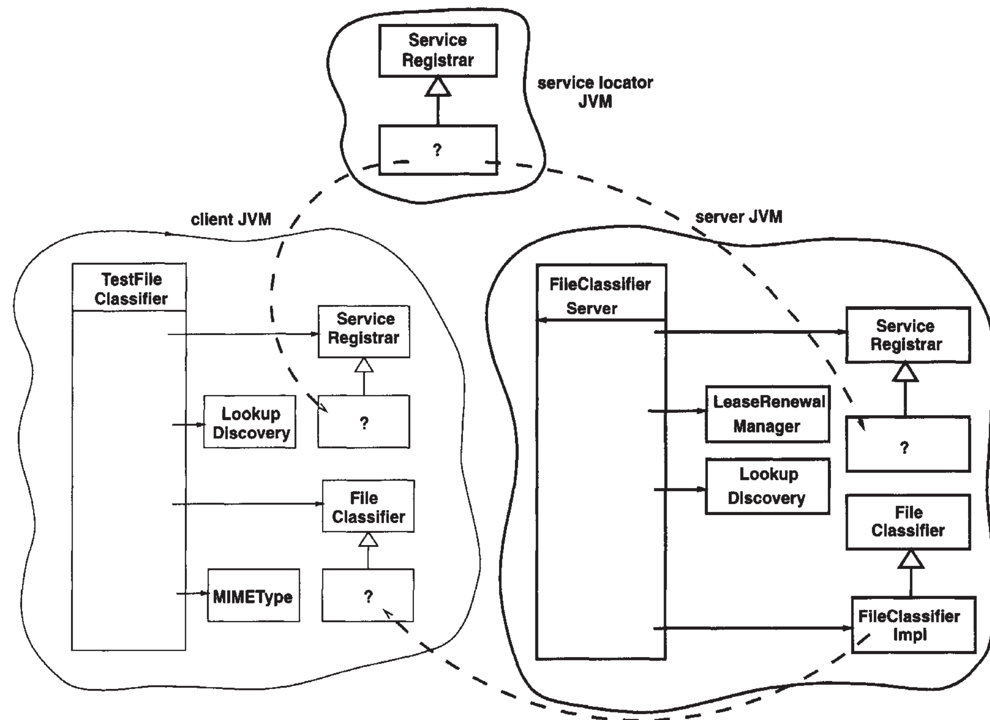


Figure 8-4. Objects in all the JVMs

The unknown `FileClassifier` object in the client is here supplied by the service object `FileClassifierImpl` (via the lookup service, where it is stored in passive form).

Client Implementation

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

What Classes Need to Be Where?

In this chapter we have defined the following classes:

- `common.MIMEType` (in the section "Common Classes")
- `common.FileClassifier` (in the section "Common Classes")
- `complete.FileClassifierImpl` (in the section "Uploading a Complete Service")

libdocdel@fr.com

- `complete.FileClassifierServer` (in the section "Uploading a Complete Service")
- `client.TestFileClassifier` (in the section "The Client")

These classes are all required to run a file-classifier application that consists of a file-classifier client and a file-classifier service.

Instance objects of these classes could be running on up to four different machines:

- The server machine for `FileClassifier`
- The machine for the lookup service
- The machine running the `TestFileClassifier` client
- 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 term "known" can refer to different things:

- The class may be in the `CLASSPATH` of a JVM.
- The class may be loadable across the network.
- The class may be accessible by an HTTP server.

Service Provider

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

- The `common.FileClassifier` interface
- The `common.MIMEType` class
- The `complete.FileClassifierServer` class
- The `complete.FileClassifierImpl` class

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

libdocdel@fr.com

HTTP Server

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

Lookup Service

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.

Client

The client needs to know the following:

- The common.FileClassifier interface
- The common.MIMEType class
- The client.TestFileClassifier class

These all need to be in the CLASSPATH of the client. In addition, the client will need to know the class files for complete.FileClassifierImpl. However, these will come across the network as part of the discovery process, and this will be invisible to the client's programmer.

Running the FileClassifier

We now have a FileClassifierServer service and a TestFileClassifier client 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 complete.FileClassifierImpl will be passed from the server to the locator and then to the client. Once on the client, the Jini classes will need to be able to restore the FileClassifierImpl object from the instance data and from the class file, and so will need to load the FileClassifierImpl 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/complete/FileClassifierImpl.class, then the service will be started by this command:

libdocdel@fr.com

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

In this command, `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 with the following statement:

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

Summary

The material of the previous chapters is put together in this chapter in a simple example. The requirements of class structures for a Jini system are discussed, along with details of what classes need to be available to each component of a Jini system.

libdocdel@fr.com

CHAPTER 9

Choices for Service Architecture

A CLIENT WILL ONLY BE LOOKING for an implementation of an interface, and the implementation can be done in many different ways, as discussed in this chapter. In the previous chapter we discussed the roles of service proxy and service backend and briefly talked about how different implementations could place different amounts of processing in the proxy or backend. This can lead to situations such as a thin proxy communicating to a fat backend using RMI, or at the other end of the scale, to a fat proxy and a thin backend. The last chapter showed one implementation—a fat proxy with a backend so thin that it did not exist. This chapter fills in some of the other possibilities.

Proxy Choices

A Jini service will be implemented using a proxy on the client side and a service backend on the service provider side. In RPC-like systems there is little choice: the proxy must be thin and the backend must be fat. Message-based client/server systems allow choices in the distribution of processing, so that one or other side can be fat or thin, or they can equally share. Jini allows a similar range of choices, but does so using the object-oriented paradigm supported by Java. The following sections discuss the choices in detail, giving alternative implementations of a file-classifier service.

Proxy Is the Service

One extreme proxy situation 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 given in Figure 9-1. This was the implementation discussed in the previous chapter.

libdocdel@fr.com

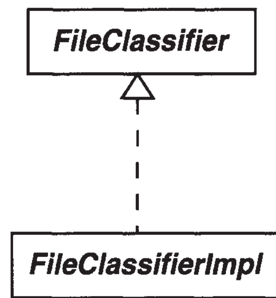


Figure 9-1. Class diagram for file classifier

We have already seen the full object diagram for the JVMs in Chapter 8, but just concentrating on these classes looks like Figure 9-2.

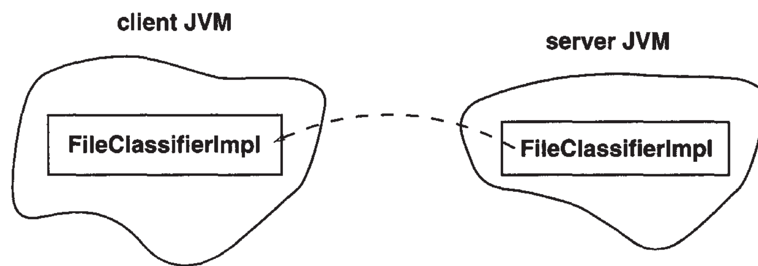


Figure 9-2. Objects in the JVMs

The client asks for a `FileClassifier`. What is uploaded to the service locators, and thus 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 if the service is purely a software one that does not need any link back to the server. It could be something like a calendar that is independent of location, or a diary that uses files on the client side rather than the server side.

RMI Proxy

The opposite proxy extreme 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

libdocdel@fr.com

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 shown in Figure 9.3.

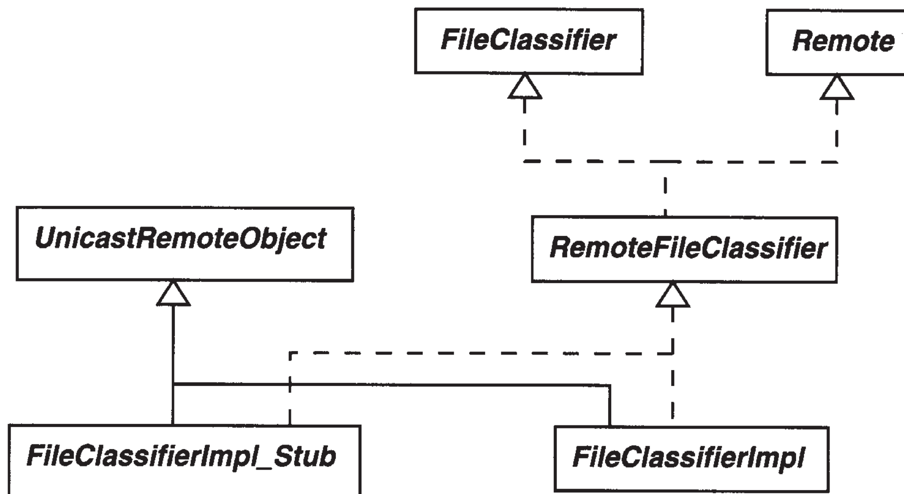


Figure 9-3. Class diagram for RMI proxy

The objects in the JVMs are shown in Figure 9-4.

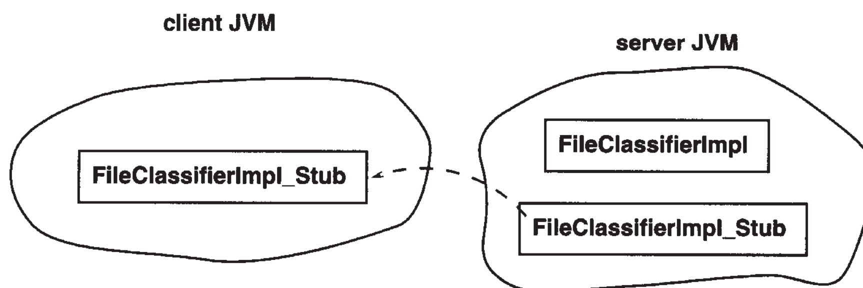


Figure 9-4. JVM objects for RMI proxy

The full code for this mechanism is given later in the chapter in the "RMI Proxy for FileClassifier" section.

The class structure for this mechanism is much more complex than the fat proxy because of RMI requirements. The `RemoteFileClassifier` interface has to be defined, and the implementation class has to implement (or call suitable methods

libdocdel@fr.com

from) the `UnicastRemoteObject` class. The `FileClassifierImpl_Stub` is generated from `FileClassifierImpl` by using 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 recognizes 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 class 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 give further examples.

Non-RMI Proxy

If RMI is not used, and the proxy and backend service want to share processing, then both the backend service and the proxy must be created explicitly on the service provider side. The proxy is explicitly exported by the service provider and must implement the interface, but on the server side this requirement does not hold, since the proxy and backend service are not tightly linked by a class structure any more. The class diagram for the file classifier with this organization is displayed in Figure 9-5.

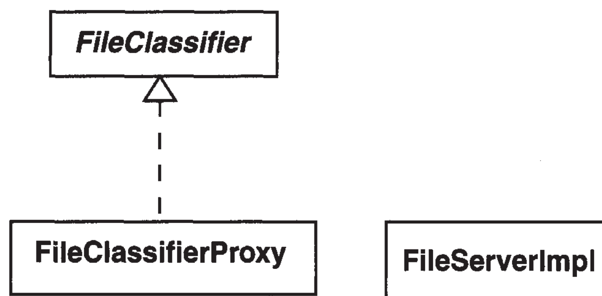


Figure 9-5. Class diagram for non-RMI proxy

The JVMs at runtime for this scenario are shown in Figure 9-6.

libdocdel@fr.com

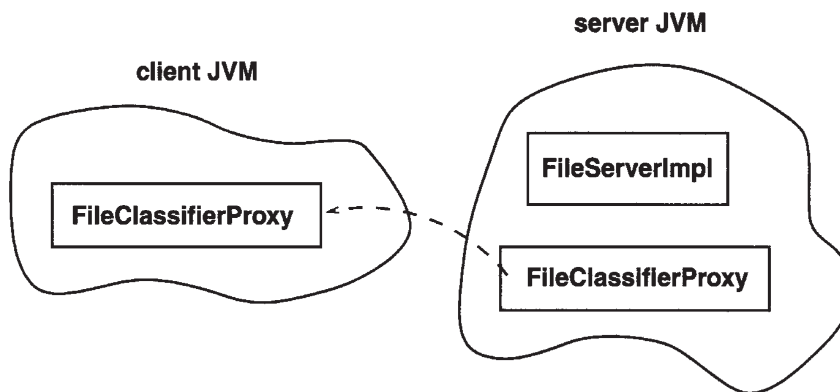


Figure 9-6. JVM objects for a non-RMI proxy

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 that 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 later in this chapter in the "Non-RMI Proxy for FileClassifier" section.

This model is good for bringing "legacy" client/server applications into the Jini world. Client/server applications often communicate using a specialized protocol between the client and server. Copies of the client have to be distributed to all machines, and 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 protocol, 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, that have varying degrees of support for HTML 3.2 and HTML 4.0 features, let alone new protocol extensions such as style sheets 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 Jini proxy is the real "heart" of the client, whereas the Jini backend service is the server part of the original client/server system. When changes occur, the backend 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.

libdocdel@fr.com

RMI and Non-RMI Proxies

The last variation is to have a backend 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 as an intermediary for RPC-like communication between the explicit proxy and the backend 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 simplify the task of the programmer.

Later in the chapter, in the "RMI and Non-RMI Proxies for FileClassifier" section, there is a non-RMI proxy, `FileClassifierProxy`, implementing the `FileClassifier` interface. This communicates with an object that 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 shown in Figure 9-7.

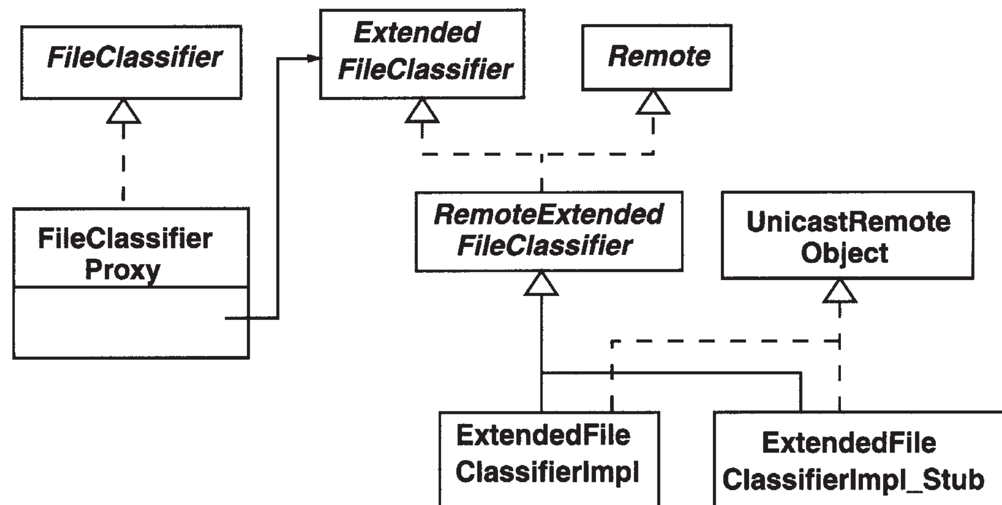


Figure 9-7. Class diagram for RMI and non-RMI proxies

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 JVMs are displayed in Figure 9-8.

libdocdel@fr.com

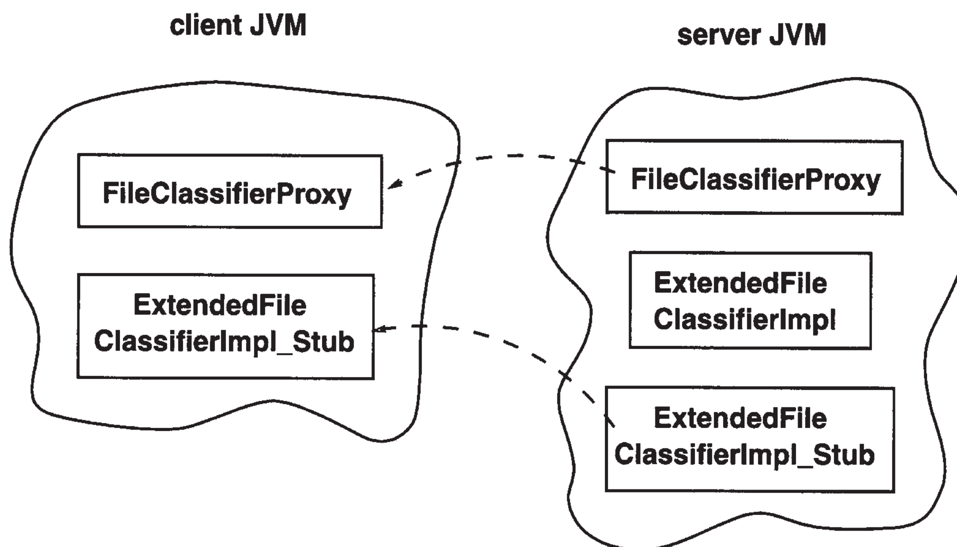


Figure 9-8. JVM objects for RMI and non-RMI proxies

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. In that case, 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 it's the backend service's constructor that 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.)

What Doesn't Change

In Chapter 8, we discussed a file-classifier application built from a client and a service, and in this chapter we have shown a different implementation of the service. A new file-classifier application can be built using this new implementation of the service. Clearly, some things must change in this new version, but because of the Jini architecture, the changes are basically localized to the service implementation. That

libdocdel@fr.com

is, most of the file-classifier application doesn't change at all, even if the service implementation changes.

The client is not concerned about the implementation of the service at all, and so the client 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. The `MIMEType` doesn't change either, since we have already declared it to implement `Serializable`–it is passed back across the network from the service to its proxy.

RemoteFileClassifier

An implementation of the service using an RMI proxy will need to implement both the `FileClassifier` and the `Remote` interfaces. It is convenient to define another interface, called `RemoteFileClassifier`, just to do this. This interface will be used fairly frequently in the rest of this book.

```
package rmi;

import common.FileClassifier;
import java.rmi.Remote;

/**
 * RemoteFileClassifier.java
 */

public interface RemoteFileClassifier extends FileClassifier, Remote {

} // RemoteFileClasssifier
```

FileClassifierImpl

The service provider will run the backend service. When the backend service exports an RMI proxy, it will look like this:

```
package rmi;

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


libdocdel@fr.com

```
/**
 * FileClassifierImpl.java
 */

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 constructor required by RMI
    }

} // FileClassifierImpl
```

FileClassifierServer

The service provider changes very little from the version in Chapter 8, which exported a complete service. Both this server and the earlier one export a service object with `register()`, but at this point the RMI runtime intervenes and substitutes an RMI stub object. The other major change is that the server no longer needs to explicitly stay alive. While the RMI system keeps a reference to the RMI stub object, it keeps alive the JVM that contains the stub object. This means that the daemon threads that are looking after the discovery process will continue to

libdocdel@fr.com

run, and in turn, since they have a reference to the service provider as listener, the service provider will continue to exist.

The following server creates and manages the RMI service:

```
package rmi;

import rmi.FileClassifierImpl;
import rmi.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
 */

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());
```


libdocdel@fr.com

```
            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);
```


libdocdel@fr.com

```
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }

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

} // FileClassifierServerRMI
```

What Classes Need to Be Where?

This chapter deals with a number of different implementations of the file-classifier service. Each implementation introduces some new classes, but also depends on some of the classes we have developed in earlier chapters. In deploying the service, we need to pay attention to this set of classes and determine which classes need to be known to the different parts of the Jini system. This "What Classes Need to Be Where?" section is repeated for each of the different service implementations, and it describes the configuration issues for each of these different implementation choices.

For the RMI proxy implementation, we need to consider these classes:

- `common.MIMEType`
- `common.FileClassifier`
- `rmi.RemoteFileClassifier`
- `rmi.FileClassifierImpl`
- `rmi.FileClassifierImpl_Stub`
- `rmi.FileClassifierServer`
- `client.TestFileClassifier`

(The `FileClassifierImpl_Stub` class is added to our classes by `rmic` as discussed in the next section.)

libdocdel@fr.com

These classes could be running on up to four different machines:

- The server machine for `FileClassifierServer`
- The HTTP server, which may be on a different machine
- The machine for the lookup service
- The machine running the `TestFileClassifier` client

So, which classes need to be known to which machines?

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

- The `common.FileClassifier` interface
- The `rmi.RemoteFileClassifier` interface
- The `common.MIMEType` class
- The `rmi.FileClassifierServer` class
- The `rmi.FileClassifierImpl` class

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 the following:

- The `common.FileClassifier` interface
- The `common.MIMEType` class

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

- The `rmi.FileClassifierImpl_Stub` interface
- The `rmi.RemoteFileClassifier` interface
- The `common.FileClassifier` interface
- The `common.MIMEType` class

libdocdel@fr.com

The reason for all of these is slightly complex. In the FileClassifierProxy constructor, the FileClassifierImpl class 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 MIMEType class, so this must also be available.

So, what does the phrase "available" mean in the last sentence? 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 references such as file://home/jan/index.html or HTTP references such as http://host/index.html. But for this case, clients running anywhere will use the URL, 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 preceding 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. In that case, these files must exist:

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

Running the RMI Proxy FileClassifier

As with the file classifier developed in Chapter 8, we again have a server and a client to run. The client does not depend on how the service is implemented, and it does not even find out about the service until it has been started and has performed a search for the service. That means the client is started in exactly the same way as it was started in Chapter 8:

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

The server in this situation is more complex than the one in Chapter 8, 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, because an RMISecurityManager is used.

libdocdel@fr.com

Although the `FileClassifierImpl` is created explicitly by the server, it is not this class file that is moved around. The `FileClassifierImpl` object continues to exist on the server machine. Rather, a stub object 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. The client machine must be able to access the class file for the stub. This class file has to be generated from the implementation class by the stub compiler `rmic` with the following command:

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

Here, 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). If the `-v1.2` option is omitted, `rmic` will also generate Java 1.1 skeleton files, which are not needed. In Java 1.3, it may not be necessary to even run `rmic`. 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 in and 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. For example, if the class files are stored on my Web server's pages under `classes/rmi/FileClassifierImpl_Stub.class`. the codebase would be specified as

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

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

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 `java.security.policy` property to point to a security policy file, such as `policy.all`.

Combining all these points leads to startups such as this:

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

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-upon 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.

libdocdel@fr.com

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 they do not have a shared class structure. The proxy and the service are written independently, using their own appropriate class hierarchies. However, 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—this is not the only choice, but it is the one used in this example. Thus, the service listens on an agreed-upon port, the client connects to this port, and they exchange messages.

The message format adopted for this solution is really simple:

- The proxy sends a message giving the file extension that it wants classified. This can be sent as a newline-terminated string (terminated by the `'\n'` character).
- The service will either succeed or fail in the classification. If it fails, it sends a single line of the empty string `""` 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.

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 to the service on an agreed-upon TCP port and exchanges messages on this port. It then returns a suitable result. The code looks like this:

```
package socket;

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

import java.io.Serializable;
import java.io.IOException;
```


libdocdel@fr.com

```
import java.rmi.Naming;

import java.io.*;

/**
 * FileClassifierProxy
 */

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) {
            return null;
        }

        String type = null;
        String subType = null;

        /*
         * protocol:
         * Write: file extension
         * Read: "null" + '\n'
         *       type + '\n' + subtype + '\n'
         */
```


libdocdel@fr.com

```
        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
```

FileServerImpl

The `FileServerImpl` service will be running on the server side. It 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 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 it is good practice for more complex situations.)

```
/**
 * FileServerImpl.java
 */

package socket;

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


libdocdel@fr.com

```
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);
```


libdocdel@fr.com

```
                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();
            }
        }
    }
```

Service Provider

The Jini service provider must start a `FileServerImpl` to listen for later connections. Then it can register a `FileClassifierProxy` proxy object with each lookup service, which will send them on to interested clients. The proxy object must know where the service backend object (the `FileServerImpl`) is listening in order to attempt a connection to it, and this information 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;
```


libdocdel@fr.com

```
// 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.LeaseRenewalManager;
import net.jini.lease.LeaseListener;
import net.jini.lease.LeaseRenewalEvent;
import java.rmi.RMISecurityManager;
import java.net.*;

/**
 * FileClassifierServer.java
 */

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();
            System.exit(1);
```


libdocdel@fr.com

```
        }

        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++) {
            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);
        }
    }
```


libdocdel@fr.com

```
    public void discarded(DiscoveryEvent evt) {

    }

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

} // FileClassifierServer
```

What Classes Need to Be Where?

This section has considered a non-RMI proxy implementation. An application that uses this service implementation will need to deal with these classes:

- common.MIMEType
- common.FileClassifier
- socket.FileClassifierProxy
- socket.FileServerImpl
- socket.FileClassifierServer
- client.TestFileClassifier

Objects in these classes could be running on up to four different machines:

- The server machine for FileClassifierServer
- The HTTP server, which may be on a different machine
- The machine for the lookup service
- The machine running the TestFileClassifier client

So, what classes need to be known to which machines?

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

- The common.FileClassifier interface

libdocdel@fr.com

- The `common.MIMEType` class
- The `socket.FileClassifierServer` class
- The `socket.FileClassifierProxy` class
- The `socket.FileServerImpl` class

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 the following:

- The `common.FileClassifier` interface
- The `common.MIMEType` class

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

- The `socket.FileClassifierProxy` interface
- The `common.FileClassifier` interface
- The `common.MIMEType` class

Running the RMI Proxy FileClassifier

A file classification application will have to run a server and a client, as in the earlier standalone implementation in Chapter 8 and the RMI implementation just a few pages earlier. The client is unchanged, as it does not care which server implementation is used:

```
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. For example, 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 server host).

libdocdel@fr.com

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 `java.security.policy` property to point to a security policy file, such as `policy.all`.

Combining all these points leads to startups such as this:

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

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 that makes use of an RMI proxy/stub when it needs to make calls back to the service. The RPC mechanism would most naturally be done using RMI in Java.

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

FileClassifierProxy

The proxy object will implement `FileClassifier` so 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. The `FileClassifierProxy` class is as follows:

```
/**
 * FileClassifierProxy.java
 */

package extended;

import common.FileClassifier;
import common.ExtendedFileClassifier;
import common.MIMEType;
```


libdocdel@fr.com

```
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
```

ExtendedFileClassifier

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

```
/**
 * ExtendedFileClassifier.java
 */
```


libdocdel@fr.com

```
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
 */

package extended;

import java.rmi.Remote;

interface RemoteExtendedFileClassifier extends common.ExtendedFileClassifier,
Remote {

} // RemoteExtendedFileClassifier
```

ExtendedFileClassifierImpl

The implementation of the `ExtendedFileClassifier` interface is done by an `ExtendedFileClassifierImpl` object. This will also need to extend `UnicastRemoteObject` so 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 using a hash table is given. This is more efficient for repeated searches:

```
/**
 * ExtendedFileClassifierImpl.java
 */

package extended;

import java.rmi.server.UnicastRemoteObject;
```


libdocdel@fr.com

```
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
         * optimizing 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
```


libdocdel@fr.com

FileClassifierServer

The final piece in this jigsaw puzzle 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
 */

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();
        // RMI keeps this alive
    }

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


libdocdel@fr.com

```
        }

        // 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++) {
            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();
                continue;
            }
            try {
                System.out.println("service registered at " +
                                   registrar.getLocator().getHost());
            } catch(Exception e) {
            }
```


libdocdel@fr.com

```
            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
```

What Classes Need to Be Where?

The implementation of the file classifier in this section uses both RMI and non-RMI proxies. As in other implementations, there is a set of classes involved that need to be known to different parts of an application. We have these classes:

- `common.MIMEType`
- `common.FileClassifier`
- `common.ExtendedFileClassifier`
- `extended.FileClassifierProxy`
- `extended.RemoteExtendedFileClassifier`
- `extended.ExtendedFileServerImpl`
- `extended.FileClassifierServer`
- `client.TestFileClassifier`

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

- The `common.FileClassifier` interface
- The `common.MIMEType` class
- The `common.ExtendedFileClassifier` class

libdocdel@fr.com

- The `extended.FileClassifierServer` class
- The `extended.FileClassifierProxy` class
- The `extended.RemoteExtendedFileClassifier` class
- The `extended.ExtendedFileServerImpl` class

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 the following:

- The `common.FileClassifier` interface
- The `common.MIMEType` class

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

- The `extended.FileClassifierProxy` interface
- The `extended.RemoteExtendedFileClassifier` class
- The `extended.ExtendedFileServerImpl_Stub` class
- The `common.FileClassifier` interface
- The `common.MIMEType` class

Using Other Services

In all the examples so far, a proxy has been created in a server and registered with a lookup service. Meanwhile, a service backend has usually been left behind in the server to handle calls from the proxy. However, there may be no need for the service to exist on the server, and the proxy could make use of other services elsewhere. This may be subject to security restrictions imposed by the client, which may disallow connections to some hosts.

In this section, we shall give an example of using a non-Jini service on another host. 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 small project to broadcast his heartbeat live to the Web; a heart monitor was attached to him, which talked via an RS232 link to a mobile phone he was carrying.

libdocdel@fr.com

This did a data transfer to a program running at `http://www.micromed.com.au` located at the Gold Coast, which forwarded the data to a machine at the Distributed Systems Technology Centre (DSTC) in Brisbane. This ran a Web server delivering an applet, and the applet talked back to a server on the DSTC machine, which sent 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. Create a service that 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. Have the service connect to an HTTP address encoded into the service by its constructor (or other means), and read from this and display the contents, assuming it is heart cardiograph data.

3. The information about whose trace it is can be given by a `Name` entry.

The client shows what you see in Figure 9-9. The break towards the right-hand side shows where the current trace is being written (it scans from left to right, overwriting as it goes). Cardiologists do not seem to be concerned about the lack of horizontal or vertical scales, as long as the trace is physically the right size!

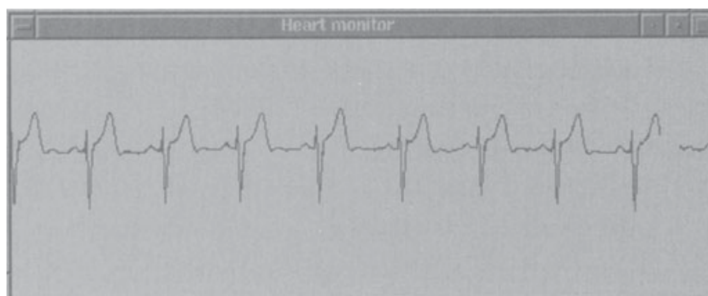


Figure 9-9. Heart monitor trace service

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

- It is a full-blown service uploaded to the client that just happens to use an external data source supplied from an HTTP server.

libdocdel@fr.com

- It is a "fat" proxy to the HTTP service, and it 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.

Heart Interface

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

```
/**
 * Heart.java
 */

package heart;

public interface Heart extends java.io.Serializable {

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

HeartServer

The HeartServer is similar to the method discussed in Chapter 8, of uploading a complete implementation of the service. This service, of type HeartImpl, is primed with a URL identifying where the heart data is stored. An HTTP server will later deliver this data.

This implementation is enough to locate the service. However, rather than just getting anyone's heart data, a client may wish to search for a particular person's data. This can be done by adding a Name entry as additional information about the service. A server that exports the complete service, plus the entry information, is as follows:

```
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;
```


libdocdel@fr.com

```
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
 */

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


libdocdel@fr.com

```
        }

        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("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());
                continue;
            }
            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
```


libdocdel@fr.com

HeartClient

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, the client just calls the `show()` method on the service 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
 */

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) {
```


libdocdel@fr.com

```
                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();
                    continue;
                }
                if (heart == null) {
                    System.out.println("Heart null");
                    continue;
                }
                heart.show();
                System.exit(0);
            }
        }

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


libdocdel@fr.com

Heart Implementation

The `HeartImpl` class opens a connection to an HTTP server and requests delivery of a file. Heart data needs to be displayed at a reasonable rate, so it reads, draws, and 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 customizes 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
 */

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 = "--";
```


libdocdel@fr.com

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

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();
```


libdocdel@fr.com

```
        } catch (Exception ex) {
            quit(ex, "fetching data from ECG server");
            return;
        }
        frame.setVisible(true);
    }
} // 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;
```


libdocdel@fr.com

```
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
            // convert 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;
```


libdocdel@fr.com

```
                // 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
        }
    }
  }
}
```


libdocdel@fr.com

Summary

Clients are built to make use of the services they find, but they do not need to be concerned with how the services are implemented. On the other hand, service implementers need to be aware of the choices they have in building services, and they need to choose the architecture that best suits the needs of the service. This chapter has looked at a number of possibilities and has used a simple running example to illustrate some of the possible design patterns.

libdocdel@fr.com

CHAPTER 10

Discovery Management

CLIENTS AND SERVICES BOTH NEED to find lookup services. In Chapter 3, we looked at the code that was common to both clients and services in both unicast and broadcast discovery. Parts of that code has been used in many examples since. This chapter discusses some utility classes that make it easier to deal with lookup services by encapsulating this type of code into common utility classes and providing a good interface to them. This chapter only applies to Jini 1.1, since these classes were only brought into Jini with version 1.1.

Finding Lookup Locators

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

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

Chapter 3 considered the cases of a single unicast lookup service and a set of multicast lookup services. This was all that was available in Jini 1.0. Jini 1.1 has been extended to handle a set of unicast lookup services *and* a set of multicast lookup services. The Jini 1.1 Helper Utilities document (part of the Jini 1.1 specification) defines three interfaces:

- `DiscoveryManagement`, which looks after discovery events
- `DiscoveryGroupManagement`, which looks after groups and multicast searches
- `DiscoveryLocatorManagement`, which looks after unicast discovery

Different classes may implement different combinations of these three interfaces. The `LookupDiscovery` class was changed in Jini 1.1 to use `DiscoveryGroupManagement` and `DiscoveryManagement`. The `LookupDiscovery` class performs multicast searches,

libdocdel@fr.com

informing its listeners when lookup services are discovered. The LookupLocatorDiscovery class is new in Jini 1.1 and is discussed later in this chapter. It performs a similar task for unicast discovery and implements the two interfaces DiscoveryLocatorManagement and DiscoveryManagement. Another class discussed later is LookupDiscoveryManager, which 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 as follows:

```
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 addDiscoveryListener() method is the most important method, as it allows a listener object to be informed whenever a new lookup service is discovered.

The DiscoveryGroupManagement interface is shown next:

```
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 methods 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 a search. Similarly, addGroups() will also initiate a search. (This is why they may throw remote exceptions.)

libdocdel@fr.com

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);
}
```

A client or service 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.

LookupLocatorDiscovery

In Chapter 3, 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 Chapter 3 into a loop. The LookupLocatorDiscovery class provides a more satisfactory 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 addresses, a listener object is informed.

The LookupLocatorDiscovery class is specified as follows:

```
package net.jini.discovery;

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

Rewriting the unicast example from Chapter 3 using this utility class makes it look much like the example on multicast discovery from the same chapter. The similarity is that it now uses the same event model for lookup service discovery; the difference is that it uses a set of LookupLocator objects rather than a set of groups.

```
package discoverymgt;
```


libdocdel@fr.com

```
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;

/**
 * UnicastRegister.java
 */

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();
            System.exit(1);
        }
        try {
            discover = new LookupLocatorDiscovery(locators);
        } catch(Exception e) {
            System.err.println(e.toString());
            e.printStackTrace();
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
    }
```


libdocdel@fr.com

```
    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) {

    }
} // UnicastRegister
```

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

```
package discoverymgt;

import net.jini.discovery.LookupDiscoveryManager;
import net.jini.discovery.DiscoveryGroupManagement;
```


libdocdel@fr.com

```
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
 */

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();
            System.exit(1);
        }
        try {
            discover = new _
LookupDiscoveryManager(DiscoveryGroupManagement.ALL_GROUPS,
                                                    locators,
                                                    this);
        } catch(IOException e) {
            System.err.println(e.toString());
            e.printStackTrace();
            System.exit(1);
        }
```


libdocdel@fr.com

```
    }

    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();
                continue;
            }
            // the code takes separate routes from here for client or service
          }
    }

    public void discarded(DiscoveryEvent evt) {

    }
} // AllcastRegister
```

Summary

The LookupLocatorDiscovery and LookupDiscoveryManager utility classes add to the LookupDiscovery class by making it easier to find lookup services using both unicast and broadcast searches.

libdocdel@fr.com

CHAPTER 11

Join Manager

FINDING A LOOKUP SERVICE INVOLVES a common series of steps, and convenience classes for encapsulating this were considered in the last chapter. Subsequent interaction with the discovered lookup services also involves common steps for services as they register with the lookup services. A join manager encapsulates these additional steps into one convenience class for services.

Jini 1.1 JoinManager

A service needs to locate lookup services and register the service with them. Locating services can be done using the utility classes from Chapter 10. As each lookup service is discovered, it needs to be registered, and the lease needs to be maintained. The `JoinManager` class 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 constructor is used when the service is new and does not have a service ID. A `ServiceIDListener` can be added to note and save the ID. The second constructor is used when the service already has an ID. The other parameters are for the service 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.

libdocdel@fr.com

The following example uses the `JoinManager` class to register a `FileClassifierImpl`. In the Chapter 8 example of "Uploading a Complete Service" (and other examples in Chapter 9) the server implemented the `DiscoveryListener` interface in order to be informed when new lookup locators were discovered so that the service could be registered with each of them. If you use a join manager, 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 rmi.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
 */

public class FileClassifierServer
    implements ServiceIDListener {

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

        // stay around forever
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(InterruptedException e) {
                // do nothing
            }
        }
    }
```


libdocdel@fr.com

```
    public FileClassifierServer() {

        JoinManager joinMgr = null;
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            joinMgr = new JoinManager(new FileClassifierImpl(), /* service */ 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` that allow you to modify the state of a service registration.

Jini 1.0 JoinManager

A version of `JoinManager` was present in Jini 1.0. At that time it was in the `com.sun` package and no formal specification was given. Classes in the `com.sun` packages may be changed in later versions of Jini, or may even disappear completely. In moving from Jini 1.0 to Jini 1.1, the `JoinManager` classes were specified and moved to a new package. This section describes the old version for those still using Jini 1.0. When possible, such users should switch to Jini 1.1

libdocdel@fr.com

There are a number of possible constructors for `JoinManager`. This is the simplest:

```
JoinManager(java.lang.Object obj,
           Entry[] attrSets,
           ServiceIDListener callback,
           LeaseRenewalManager leaseMgr)
```

This constructor specifies the service to be managed and its entry attributes. The `callback` is a listener object that will have its `serviceIDNotify()` method called when a new locator is discovered. This is usually used to find the value of the `ServiceID` assigned by a lookup locator to a service. The `callback` argument can be `null` if the programmer has no interest in saving the `ServiceID`. The `leaseMgr` can also be set to `null` and will then be created as needed.

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 is as follows:

```
JoinManager(java.lang.Object obj,
           Entry[] attrSets,
           java.lang.String[] groups,
           LookupLocator[] locators,
           ServiceIDListener callback,
           LeaseRenewalManager leaseMgr)
```

This constructor adds `groups` and `locators`, which allow multicast searches for locators belonging to certain groups, and also unicast lookups for known locators.

A multicast-only search for any groups would have both additional parameters set to `null`:

```
JoinManager joinMgr = new JoinManager(obj, null,
                                      LookupDiscovery.ALL_GROUPS,
                                      null, null, null);
```


libdocdel@fr.com

On the other hand, a unicast lookup for a single known site would be done like this:

```
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 ignores exception handling.)

For example, uploading the complete service of the `complete` package could be done as follows:

```
package joinmgr;

import complete.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
 */

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;
```


libdocdel@fr.com

```
        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
```

Getting Information from JoinManager

The JoinManager looks unhelpful in supplying information about the lookup locators it finds. However, this information is available 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 the serviceIDNotify() method we do know that a new service locator has been found, since that is the only occasion on which it is called.

The *complete set* of service locators can be found with the JoinManager's getJoinSet() method, which returns an array of ServiceRegistrar objects. We have met this class before: its getLocator() method will return a LookupLocator, which has information such as the host in getHost(). These classes can be put together as follows:

libdocdel@fr.com

```
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, 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.

Summary

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

libdocdel@fr.com

CHAPTER 12

Security

SECURITY PLAYS AN IMPORTANT ROLE in distributed systems. All parts of a Jini djinn, which consists of clients, services, and lookup services, can be subjected to attack by hostile agents. You could trust everyone, but the large number of attacks that are made on all sorts of systems by both skilled and unskilled people doesn't make this a reasonable approach. The Jini security model is based on the JDK 1.2 security system, and all components of a Jini system use the JDK 1.2 security mechanisms. This can be tricky to set up, and it is looked at in detail in this chapter.

Getting Going with No Security

Security for Jini is based on the JDK 1.2 security model, which makes use of a `SecurityManager` to grant or deny access to resources. All potentially dangerous requests, such as opening a file, starting a process, or establishing a network connection, are all passed to a `SecurityManager`. This manager will make decisions based on a security policy (which should have been established for that application) and either allow or deny the request.

A few of the examples given so far in this book may work fine without a security manager, but most will require an appropriate security manager to be in place. The major requirement in most examples is for the RMI runtime to be able to download class files to instantiate proxy objects. This can be enabled by installing an `RMISecurityManager`. Installing a security manager may be done by including this statement in your code:

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

This should be done before any network-related calls, and is often done in the `main()` method or in a constructor for the application class.

The security manager will need to make use of a security policy. This is typically given in policy files, which are kept 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 with this statement

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

will load the contents of the policy file.

libdocdel@fr.com

A totally permissive policy file can contain the following:

```
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 permissive policy file is that it gets you going on the rest of Jini without worrying about security issues while you are grappling with other problems!

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 it then executes that code. There are really no checks that the downloaded code is a genuine service: the downloaded code has to implement the requested interface and maybe satisfy conditions on associated Entry objects. If it passes these conditions, then it can do anything.

For example, a client asking for a simple file classifier could end up getting this hostile object:

```
package hostile;

import common.MIMEType;
import common.FileClassifier;

/**
 * HostileFileClassifier1.java
 */

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!
```


libdocdel@fr.com

```
            // 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
 */

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");
        }
    }
```


libdocdel@fr.com

```
    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 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, however, attacks based on hostile implementations of services may become more common.

Removing AllPermission

Setting the security access to `AllPermission` is easy and removes all possible security issues that may hinder development of a Jini application. However, it leaves your system open, so 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 open policy is that permissions are additive rather than subtractive. That is, you can't take permissions away from `AllPermission`; you have to start with an empty permission set and add to that.

Not giving enough permission can result in at least three situations when you try to access something:

- 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. Of course, this should be caught during testing, not when the application is deployed!

- A security-related exception can be thrown but caught by some library object, which attempts to handle it. 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 it will just suffer this time delay.

libdocdel@fr.com

- 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 that 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 in the `rmi.FileClassifierServer` of Chapter 9. The third occurs for the client `client.TestFileClassifier` of Chapter 8 if insufficient permissions are given.

There is a `java.security.debug` system property 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 such as the following:

```
...
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, because the client otherwise appears to just hang.

Jini with Protection

The safest way for a Jini client or service to be part of a Jini federation is through abstinence: that is, for it to refuse to take part. This doesn't get you very far in

libdocdel@fr.com

populating a federation, though. The JDK 1.2 security model offers a number of ways in which more permissive activity may take place:

- 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";
}
```

- Grant access only to particular hosts, subdomains, or domains.

```
grant codebase "http://sunshade.dstc.edu.au/classes/" {
    permission java.security.AllPermission "", "";
}
```

- Require digital signatures to be attached to code.

```
grant signedBy "sysadmin" {
    permission java.security.AllPermission "", "";
}
```

For any particular security access, you will need to decide which of these options is appropriate. This will depend on the overall security policy for your organization, and if your organization 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 organization, either)!

Service Requirements

In order to partake in a Jini federation, a service must become sufficiently visible. The service needs to find a service locator before it can advertise its services, and as explained in Chapter 3, 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.

For the multicast case, the service must have `DiscoveryPermission` for each group that it is trying to join. For all groups, the asterisk (*) wildcard can be used. So, to join all groups, the permission granted should be as follows:

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


libdocdel@fr.com

For example, to join the `printers` and `toasters` groups, the permission would be 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. This particular address is used by Jini for broadcasts and should be used in your policy files. Socket permission for these requests and announcements must be given as follows:

```
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`, which will need to communicate back to the server, and so the server will need to listen on a port for these remote object requests. Ports are numbered from 1 to 65,000, and the default constructor will assign a random port (greater than 1,024) for this. If desired, this port may be specified by other constructors. This will require further socket permissions, such as the following, to accept connections on any port above 1024 from the `localhost` or any computer in the `dstc.edu.au` domain:

```
permission java.net.SocketPermission "localhost:1024-", "connect,accept";
permission java.net.SocketPermission "*.dstc.edu.au:1024-", "connect,accept";
```

The reason Jini uses a port greater than 1024 is because the use of lower port numbers is restricted on Unix systems.

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 including code like the following in the policy file:

```
permission java.util.PropertyPermission "net.jini.discovery.*", "read";
```

A fairly minimal policy file suitable for a service exporting an RMI object could then be as follows:

```
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";
```


libdocdel@fr.com

```
    // 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";
};
```

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" or execute any of their methods; but the client brings an external object into its address space and runs it, giving it all of the permissions of a process running in an operating system. The object will run under the permissions of a particular user in a particular directory, with user access 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 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, 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 is shown in Figure 12-1. The client is the most likely candidate to require signed trust certificates since it has the highest trust requirement of the components of a Jini system.

libdocdel@fr.com

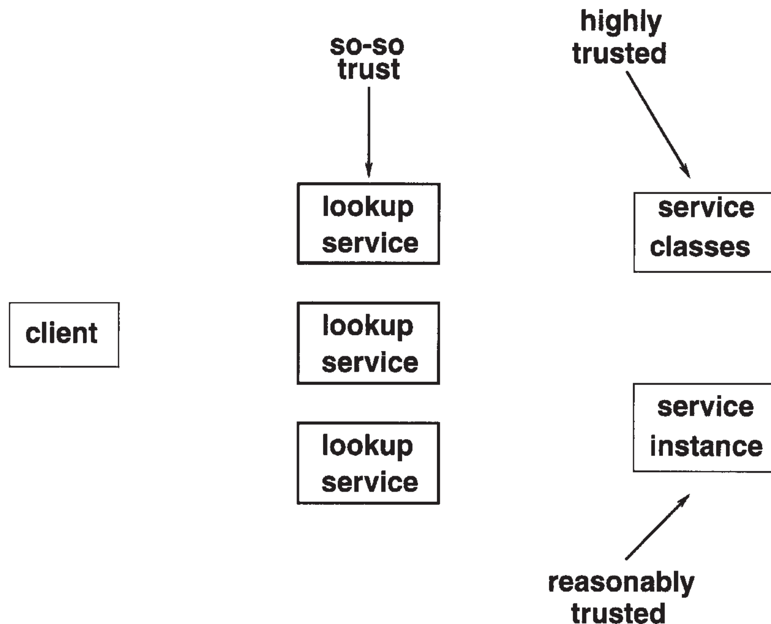


Figure 12-1. Trust levels of the client

Many services will just make use of whatever HTTP server is running on their system, and this will probably be on port 80. Permission to connect on this port can be granted with the following statements:

```
permission java.net.SocketPermission "127.0.0.1:80", "connect,accept";
permission java.net.SocketPermission "*.dstc.edu.au:80", "connect,accept";
```

However, while this will allow code to be downloaded on port 80, it may not block some malicious attempts. Any user can start an HTTP server on any port (Windows) or above 1024 (Unix). A service can then set its codebase to whatever port the HTTP server is using. Perhaps these other ports should be blocked, but unfortunately, RMI uses random ports, so these ports need to be open.

So, it is probably not possible to close all holes for hostile code to be downloaded to a client. What you need is a second stage defense: given that hostile code may reach you, set the JDK security so that hostile (or just buggy) code cannot perform harmful actions in the client.

A fairly minimal policy file suitable for a client could then be as follows:

```
grant {
    permission net.jini.discovery.DiscoveryPermission "*";

    // multicast request address
    permission java.net.SocketPermission "224.0.1.85", "connect,accept";
    // multicast announcement address
```


libdocdel@fr.com

```
    permission java.net.SocketPermission "224.0.1.84", "connect,accept";

    // RMI connections
    // DANGER
    // HTTP connections - this is where external code may come in - careful!!!
    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";

};
```

RMI Parameters

A service is specified by an interface. In many cases, an RMI proxy will be delivered to the client that implements this interface. Depending on the interface, this can be used by the client to attack the service. The `FileClassifier` interface is safe, but in Chapter 14 we will look at how a client can upload a new MIME type to a service, and this extended interface exposes a service to attack.

This is the relevant method from the `MutableFileClassifier` interface of Chapter 14:

```
public void addType(String suffix, MIMEType type)
       throws java.rmi.RemoteException;
```

This method allows a client to pass an object of type `MIMEType` up to the service, where it will presumably try to add it to a list of existing MIME types. The `MIMEType` class is an ordinary class, not an interface. Nevertheless, it can be subclassed, and this subclass can make an attack as described in the second section of this chapter.

This particular attack can be avoided by ensuring that the parameters to any method call in an interface are all `final` classes. If the `MIMEType` class was defined by

```
public final class MIMEType {...}
```


libdocdel@fr.com

then it would not be possible to subclass it. No attack could be made by a subclass, since no subclass could be made! There aren't enough Jini services defined yet to know whether making all parameters `final` is a good enough solution.

ServiceRegistrar

Services will transfer objects to be run within clients. This chapter has so far been concerned with the security policies that will allow this and the restrictions that may need to be in place. The major protection for clients at the moment is that there are no standardized service interfaces, so attackers do not yet know what hostile objects to write.

A lookup service, on the other hand, exports an object that implements `ServiceRegistrar`. It does not use the same mechanism as a service would to get its code into a client. Instead, the lookup service replies directly to unicast connections with a registrar object, or responds to multicast requests by establishing a unicast connection to the requester and again sending a registrar. The mechanism is different, but it is clearly documented in the Jini specifications and it is quite easy to write an application that performs at least this much of the discovery protocols.

The end result of lookup discovery is that the lookup service will have downloaded registrar objects. The registrar objects run in both clients and services—they both need to find lookup services. The `ServiceRegistrar` interface is standardized by the Jini specification, so it is fairly easy to write a hostile lookup service that can attack both clients and services.

While it is unlikely that anyone will knowingly make a unicast connection to a hostile lookup service, someone might get tricked into it. There are already some quite unscrupulous Web sites that will offer "free" services on producing a credit card (to the user's later cost). There is every probability that such sites will try to entice Jini clients if they see a profit in doing so. Also, anyone with access to the network and within broadcast range of clients and services (i.e., on your local network) can start lookup services that will be found by multicast discovery.

The only real counter to this attack is to require that all connections that can result in downloaded code should be covered by digital certificates, so that all downloaded code must be signed. This covers all possible ports, since an HTTP server can be started on any port on a Windows machine. The objects that are downloaded in the Sun implementation of the lookup service, `reggie`, are all in `reggie-dl.jar`. This is not signed by any certificates. If you are worried about an attack through this route, you should sign this file, as well as the jar files of any services you wish to use.

libdocdel@fr.com

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. The location of this jar file is specified in the first command-line argument. For example, to access it from the HTTP server on my laptop, jannote, I would issue the following command:

```
java -jar mahalo.jar http://jannote.dstc.edu.au/mahalo-dl.jar
```

The transaction manager is a Jini service, and so should set a security policy. This security policy should allow the transaction manager to register with a lookup service, and allow client access to it. In addition, the transaction manager needs to maintain state about transactions in permanent storage. To do this, it needs access to the file system, and since it has a security manager installed, this access needs to be granted explicitly. This is done using the normal java.security.policy property:

```
java -Djava.security.policy=policy.txn \
     -jar mahalo.jar http://jannote.dstc.edu/au/mahalo-dl.jar
```

This will allow the service to be registered and uploaded, and also will allow access to the file system.

A suitable policy to set up the permissions discussed earlier and grant file system access could be as follows:

```
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 to transaction manager log files
    permission java.io.FilePermission "/tmp/mahalo_log", "read,write";
    permission java.io.FilePermission "/tmp/mahalo_log/-", "read,write,delete";
```


libdocdel@fr.com

```
    // 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";
};
```

The new activation system of JDK 1.2 takes a little getting used to and causes confusion to Jini newcomers, because Sun implementations of major Jini services (such as `mahalo`) use activation. An *activatable service* like `mahalo` hands over responsibility for execution to a third party, an activation service. This activation service is usually `rmid`, and is used by `reggie` as well as `mahalo`.

A service (e.g., `mahalo`) starts, registers itself with this third-party service (e.g., `rmid`), and then exits. The third-party service (`rmid`) is responsible for fielding calls to the service (`mahalo`), and either awakening it or restoring it from scratch to handle the call. There is a subtlety here: the service (`mahalo`) begins execution in one JVM, but promptly delegates its execution to this third-party service (`rmid`) running in a different JVM! Thus, there are two JVMs involved in running an activatable service, and so there are two security policies—one for each of the JVMs.

The first security policy is used when the service is first started (say by a user). This uses the command line argument `-Djava.security.policy=...` and is used to register the service (`mahalo`) with the activation service (`rmid`). This startup service then exits. Some time later, the activation service will try to restart the registered service (`mahalo`) and will need to know the security policy to apply to it. This second security policy must be passed from the original startup through to the activation service, and this is specified in an additional command-line argument, `policy.actvn`.

```
java -Djava.security.policy=policy.txn -jar mahalo.jar \
      http://jannote.dstc.edu.au/mahalo-dl.jar \
      policy.actvn
```

The policy file just discussed is suitable for starting the `mahalo` service. A suitable activation policy for actually *running* the `mahalo` service from the activation server could be as follows:

```
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";
```


libdocdel@fr.com

```
    // access the 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 downloading mahalo-dl.jar from HTTP server
    permission java.net.SocketPermission "*:8080", "connect,accept,resolve";
    permission net.jini.discovery.DiscoveryPermission "*";
};
```

rmid

An activatable service runs within a JVM started by `rmid`. It does so with the same user identity as `rmid`, so if `rmid` is run by, say, the superuser `root` on a Unix system, then all activatable services will run with that same user ID, `root`. This is a security flaw, as any user on that system can write and start an activatable service, and then write and run a client that makes calls on the service. This is a way to run programs with superuser privileges from an arbitrary user.

My own machine has only a few users, and all of them I trust not to write deliberately malicious programs (and right now, I am the only one who can write Jini services). However, most people may not be in such a fortunate position. Consequently, `rmid` should be run in such a way that even if it is attacked, it will not be able to do any damage.

On Unix, there are two ways of reducing the risk:

- Run `rmid` as a harmless user, such as user `nobody`. This can be done by changing `rmid` to be `setuid` to this user. Note that the program `rmid` in the Java `bin` directory is actually a shell script that eventually calls a program such as `bin/i386/green_threads/rmid`, and it is this program that needs to have the `setuid` bit set.

libdocdel@fr.com

- Use the `chroot` mechanism to run `rmid` in a subdirectory that appears to be the top-level directory '/'. This will make all other files invisible to `rmid`.

> **NOTE** `setuid` *is the description of the Unix mechanism for changing the apparent user of a program. Unix performs this change when the* `setuid` *bit is set in the file permissions of the program. This can be added by the Unix command* `chmod +s program`.

Since the attack can only come from someone who already has an account on the machine, the `setuid` method is probably good enough, and it is certainly simpler to set up than `chroot`.

On an NT system, `rmid` should be set up so that it only runs under general user access rights.

rmid and JDK 1.3

The security problems of the last section have been partly addressed by a tighter security mechanism introduced in JDK 1.3. These restrict what activatable services can do by using a security mechanism that is under the control of whoever starts `rmid`. This means that there has to be cooperation between the person who starts `rmid` and the person who starts an activatable service that will use `rmid`.

The simplest mechanism is to just turn the new security system off. This was discussed briefly in Chapter 3, and it means running `rmid` with an additional argument:

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

All that `rmid` then checks is that any activatable service that registers with it is started on the same machine as `rmid`. This is the weak security mechanism in the JDK 1.2 version of `rmid`, which assumes that users on the same machine pose no security risks.

The default new mechanism can also be set explicitly:

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

This requires an additional security policy file that will be used by `rmid`, and the location of this policy file is also given on the command line for `rmid`. For example,

libdocdel@fr.com

the following command will start rmid using the new default mechanism with the policy file set to /usr/local/jini1_1/rmid.policy:

```
rmid -J-Djava.security.policy=/usr/local/jini1_1/rmid.policy
```

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

For example, reggie is an activatable service. To run this on my system, I use this command:

```
java -jar /usr/local/jini1_1/lib/reggie.jar \
        http://jannote.dstc.edu.au:8080/reggie-dl.jar \
        /usr/local/jini1_1/example/lookup/policy \
        /tmp/reggie_log public
```

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

```
grant {
    permission com.sun.rmi.rmid.ExecOptionPermission
        "/usr/local/jini1_1/lib/reggie.jar";
    permission com.sun.rmi.rmid.ExecOptionPermission
       "-Djava.rmi.server.codebase=http://jannote.dstc.edu.au:8080/reggie-dl.jar";
    permission com.sun.rmi.rmid.ExecOptionPermission
        "-Djava.security.policy=/usr/local/jini1_1/example/lookup/policy";
    permission com.sun.rmi.rmid.ExecOptionPermission "-cp";
};
```

The permissions granted are, in turn:

1. The jar file that contains the main application class. This distinguishes reggie from other activation services.

2. The HTTP address of the class files used in the implementation.

3. The security policy file used by reggie.

Unless all three match, rmid will not run reggie. Wildcards can be used, but this will reduce the amount of security that rmid has over the activatable services it looks after.

You may have noticed that there is a mismatch between the command I type to get reggie running and the contents of the policy file. Not all of the command

libdocdel@fr.com

line arguments I type are in the policy file. For example, what has happened to the `/tmp/reggie_log` argument?

Well, arguments like `/usr/local/jini1_1/example/lookup/policy` are property overrides that are defined to the JVM in the form `-D=`, as shown here:

```
-Djava.security.policy=/usr/local/jini1_1/example/lookup/policy
```

On the other hand, the argument `/tmp/reggie_log` is just a simple command line argument and not a property override at all. The property overrides need to go in the policy file, but the ordinary command line arguments do not.

So do you have to go through each argument in turn, to decide if it is a property? No, that would be too tedious. Instead, you start with an empty policy file, start `rmid`, and then start an activatable service such as `reggie`. Generally, this will fail with an exception message such as this:

```
Unable to invoke by reflection, the method:
com.sun.jini.reggie.CreateLookup.create.
An exception was thrown by the invoked method.
java.lang.reflect.InvocationTargetException: java.rmi.activation.ActivateFailedEx-
ception: failed to activate object; nested exception is:
        java.security.AccessControlException: access denied (com.sun.rmi.rmid.Exec-
OptionPermission -Djava.security.policy=/usr/local/jini1_1/example/lookup/policy)
java.security.AccessControlException: access denied (com.sun.rmi.rmid.ExecOption-
Permission -Djava.security.policy=/usr/local/jini1_1/example/lookup/policy)
```

In this exception message is the phrase "com.sun.rmi.rmid.ExecOptionPermission -Djava.security.policy=/usr/local/jini1_1/example/lookup/policy." The person who wants to run `reggie` must communicate this information to the person who controls `rmid` so that they can place this information in the `rmid` policy file.

You'll need to go through this process a few times to build up the complete set of permissions for `reggie`. That's tedious too ... but there isn't any other way. The document `http://developer.java.sun.com/developer/products/jini/execpolicy.html` gives policy files for the Jini services `reggie`, `mahalo`, and `FrontEndSpace`.

There is a third choice in mechanisms, and that is to specify an object that will be used to establish the security access, but that is beyond the scope of this chapter. It is discussed in the JDK 1.3 documentation for `rmid`.

libdocdel@fr.com

Being Paranoiac

Jini applications download and execute code from other sources:

- Both clients and services download `ServiceRegistrar` objects from lookup services. They then call methods such as `lookup()` and `register()`.
- A client will download services and execute whatever methods are defined in the interface.
- A remote listener will call the `notify()` method of foreign code.

In a safe environment where all code can be trusted, no safeguards need to be employed. However, most environments carry some kind of risk from hostile agents. An attack will consist of a hostile agent implementing one of the known interfaces (of `ServiceRegistrar`, of a well-known service such as the transaction manager, or of `RemoteEventListener`) with code that does not implement the implied "contract" of the interface but instead tries to perform malicious acts. These acts may not even be deliberately hostile; most programmers make at least some errors, and these errors may result in risky behavior.

There are all sorts of malicious acts that can be performed. Hostile code can simply terminate the application, but the code can perform actions such as read sensitive files, alter sensitive files, forge messages to other applications, perform denial of service attacks such as filling the screen with useless windows, and so on.

It doesn't take much reading about security issues to instill a strong sense of paranoia, and possible overreaction to security threats. If you can trust everyone on your local network (which you are already doing if you run a number of common network services such as NFS), then the techniques discussed in this section are probably overkill. If you can't, then paranoia may be a good frame of mind to be in!

Protection Domains

The Java 1.2 security model is based on the traditional idea of "protection domains." In Java, a protection domain is associated with classes based on their `CodeSource`, which consists of the URL from which the class file was loaded (the codebase), plus a set of digital certificates used to sign the class files. For example, the class files for the `LookupLocator` class are in the file `jini-core.jar` (in the `lib` directory of the Jini distribution). This class has a protection domain associated with the `CodeSource` for `jini-core.jar`. (All of the classes in `jini-core.jar` will belong to this same protection domain.)

libdocdel@fr.com

Information about protection domains and code sources can be found by code such as this, which can be placed anywhere after the registrar object is found:

```
java.security.ProtectionDomain domain = registrar.
                                        getClass().getProtectionDomain();
java.security.CodeSource codeSource = domain.getCodeSource();
```

Information about the digital signatures attached to code can be found by code like this, which can also be placed anywhere after the registrar object is found:

```
Object [] signers = registrar.getClass().getSigners();
if (signers == null) {
    System.out.println("No signers");
} else {
    System.out.println("Signers");
    for (int m = 0; m < signers.length; m++)
        System.out.println(signers[m].toString());
}
```

By default, no class files or jar files have digital signatures attached. Digital signatures can be created using keytool (part of the standard Java distribution). These signatures are stored in a keystore. From there, they can be used to sign classes and jar files using jarsigner, exported to other keystores, and generally be spread around. Certificates don't mean anything unless you believe that they really do guarantee that they refer to the "real" person, and certificate authorities, such as VeriSign, provide validation techniques for this.

This description has been horribly brief and is mainly intended as a reminder for those who already understand this stuff. If you want to experiment, you can do as I did and just create certificates as needed, using keytool, although there was no independent authority to verify them. A good explanation of this topic is given by Bill Venners at http://www.artima.com/insidejvm/ed2/ch03Security1.html.

Signing Standard Files

None of the Java files in the standard distribution are signed. None of the files in the Jini distribution are signed either. For most of these it probably won't matter, since they are local files.

However, all of the Jini jar files ending in -dl.jar are downloaded to clients and services across the network and are Sun implementations of "well-known" interfaces. For example, the ServiceRegistrar object that you get from the discovery process (described in Chapter 3) has its class files defined in reggie-dl.jar, as a com.sun.jini.reggie.RegistrarImpl_Stub object. Hostile code implementing the

libdocdel@fr.com

ServiceRegistrar interface can be written quite easily. If there is the possibility that hostile versions of lookup services (or other Sun-supplied services) may be set running on your network, then you should only accept implementations of ServiceRegistrar if they are signed by an authority you trust.

Signing Other Services

Interfaces to services such as printers will eventually be decided upon, and will become "well known." There should be no need to sign these interface files for security reasons, but an authority may wish to sign them for, say, copyright reasons. Any implementations of these interfaces are a different matter. Just like the cases above, these implementation class files will come to client machines from other machines on the local or even remote networks. These are the files that can have malicious implementations. If this is a possibility, you should only accept implementations of the interfaces if they are signed by an authority you trust.

Permissions

Permissions are granted to protection domains based on their codesource, which consists of the codebase and a set of digital signatures. In the Sun implementation, this is done in the policy files, by grant blocks:

```
grant codeBase "url" signedBy "signer" {
    ...
}
```

When code executes, it belongs to the protection domains of all classes on the call stack above it. So, for example, when the ServiceRegistration object in the complete.FileClassifierServer is executing the register() method, the following classes are on the call stack:

- The com.sun.jini.reggie.RegistrarImpl_Stub class from reggie-dl.jar
- The complete.FileClassifierServer class, from the call discovered()
- Core Jini classes that have called the discovered() method
- Classes from the Java system core that are running the application

The permissions for executing code are generally the intersection of all the permissions of the protection domains it is running in. Classes in the Java system

libdocdel@fr.com

core grant all permissions, but if you restrict the permissions granted to your own application code to core Jini classes, or to code that comes across the network, you restrict what an executing method can do.

For example, if multicast request permission is not granted to the Jini core classes, then discovery cannot take place. This permission needs to be granted to the application code and also to the Jini core classes.

It may not be immediately apparent which protection domains are active at any point. For example, in the earlier call of

```
registrar.getClass().getProtectionDomain()
```

I fell into the assumption that the `reggie-dl.jar` domain was active because the method was called on the `registrar` object. But it wasn't. While the `getClass()` call is made on the `registrar`, this completes and returns a `Class` object so that the call is made on this object, which by then is just running in the three domains: the system, the application, and the core Jini classes domains, but not the `reggie-dl.jar` domain.

There are two exceptions to the intersection rule. The first is that the RMI security manager grants `SocketPermission` to connect back to the codebase host for remote classes. The second is that methods may call the `AccessController.doPrivileged()` method. This essentially prunes the class call stack, discarding all classes below this one for the duration of the call, and it is done to allow permissions based on this class's methods, even though the permissions may not be granted by classes earlier in the call chain. This allows some methods to continue to work even though the application has not granted the permission, and it means that the application does not have to generally grant permissions required only by a small subset of code.

For example, the `Socket` class needs access to file permissions in order to allow methods such as `getOutputStream()` to function. By using `doPrivileged()`, the class can limit the "security breakout" to particular methods in a controlled manner. If you are running with security access debugging turned on, this explains how a large number of accesses are granted even though the application has not given many of the permissions.

Putting It Together

Adding all the bits of information presented in this chapter together leads to security policy files that restrict possible attacks:

1. Grant permissions to application code based on the codesource. If you suspect these classes could be tampered with, sign them as well.

libdocdel@fr.com

2. Grant permission to Jini core classes based on the codesource. These may be signed if need be.

3. Grant permission to downloaded code only if it is signed by an authority you trust. Even then, grant only the minimum permission needed to perform the service's task.

4. Don't grant any other permissions to other code.

A policy file based on these principles might look like this:

```
keystore "file:/home/jan/.keystore", "JKS";

// Permissions for downloaded classes
grant signedBy "Jan" {
    permission java.net.SocketPermission "137.92.11.117:1024-",
                                         "connect,accept,resolve";
};

// Permissions for the Jini classes
grant codeBase "file:/home/jan/tmpdir/jini1_1/lib/-" signedBy "Jini" {
    // The Jini classes shouldn't require more than these

    permission java.util.PropertyPermission "net.jini.discovery.*", "read";
    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 and HTTP
    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 "137.92.11.*:1024-",
                                         "connect,accept,resolve";
    permission java.net.SocketPermission "130.102.176.*:1024-",
                                         "connect,accept,resolve";
    permission java.net.SocketPermission "130.102.176.249:1024-",
                                         "connect,accept,resolve";
    // permission java.net.SocketPermission "137.92.11.117:1024-",
                                             "connect,accept,resolve";

    // debugging
```


libdocdel@fr.com

```
    permission java.lang.RuntimePermission "getProtectionDomain";
};

// Permissions for the application classes
grant codeBase "file:/home/jan/projects/jini/doc/-" {
    permission java.util.PropertyPermission "net.jini.discovery.*", "read";
    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 and HTTP
    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 "137.92.11.*:1024-",
                                         "connect,accept,resolve";
    permission java.net.SocketPermission "130.102.176.*:1024-",
                                         "connect,accept,resolve";
    permission java.net.SocketPermission "130.102.176.249:1024-",
                                         "connect,accept,resolve";
    // permission java.net.SocketPermission "137.92.11.117:1024-",
                                            "connect,accept,resolve";

    // debugging
    permission java.lang.RuntimePermission "getProtectionDomain";

    // Add in any file, etc, permissions needed by the application classes
};
```

Summary

You have to pay attention to security when running in a distributed environment, and Jini enforces security by using the JDK 1.2 security model. This chapter has considered the range of security mechanisms possible, from turning security off through to paranoiac mode. It should be noted that this does not cover issues such as encryption or non-repudiation. These are still under development for later versions of Jini.

libdocdel@fr.com

CHAPTER 13

More Complex Examples

CHAPTER 8 LOOKED AT A SIMPLE JINI APPLICATION. In Chapter 9, some of the architectural choices for services were explored. There are, however, many other issues involved in building Jini services and clients.

This chapter delves into some of the more complex things that can happen with Jini applications. It covers issues such as the location of class files, multithreading, extending the matching algorithm used by Jini service locators, finding a service once only, and lease management. These are issues that can arise using the Jini components discussed so far. There are also further aspects to Jini that are explored in later chapters.

Where Are the Class Files?

Clients, servers, and service locators can use class files from a variety of sources. Which source they use can depend on the structure of the client and the server. This section looks at some of the variations that can occur.

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 discussed in earlier chapters, 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, as shown here:

```
public class Client {
    String getName() {
        ...
        service.setName(...);
        ...
    };
}

class Service {
    void setName(String name) {
```


libdocdel@fr.com

```
        ...
    };
}
```

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. In any case, the service proxy may perform some sort of input processing on the client side before communicating with the real service. The service proxy may need to find extra classes in order to perform this processing.

A standalone application that gets a user name might use a GUI interface as shown in Figure 13-1.

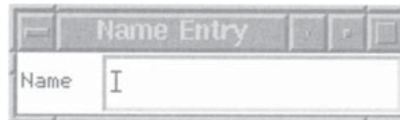


Figure 13-1. User interface for name entry

The implementation for this name entry user interface might look like this:

```
package standalone;

import java.awt.*;
import java.awt.event.*;

/**
 * NameEntry.java
 */

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);
```


libdocdel@fr.com

```
        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 implementation are these:

- A set of standard classes: `Frame`, `Label`, `TextField`, `ActionListener`, `ActionEvent`, `BorderLayout`, `WindowEvent`, and `System`
- A couple of new classes: `NameEntry` and `NameHandler`

At compile time, and at runtime, these will need to be accessible.

NameEntry Interface

A standalone application needs to have all the class files available to it. In a Jini system, we have already seen that different components may only need access to a subset of the total set of classes. The simple application just shown used a large set of classes. If this is used to form part of a Jini system then some parts of this application will end up in Jini clients, and some will end up in Jini services. Each of them will have requirements about which classes they have access to, and this will depend on how the components are distributed.

We don't want to be overly concerned about the program logic of what is done with the user name once it has been entered—the interesting part is the location

libdocdel@fr.com

of the classes. All possible ways of distributing this application into services and clients will need an interface definition, which we can make as follows:

```
package common;

/**
 * NameEntry.java
 */

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 topic is taken up in Chapter 19.*

Naive Implementation

A simple implementation of this `NameEntry` interface is as follows:

```
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
 */
```


libdocdel@fr.com

```
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 implementation of the user interface 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.

Figure 13-2 shows which instances are running in which JVM.

libdocdel@fr.com

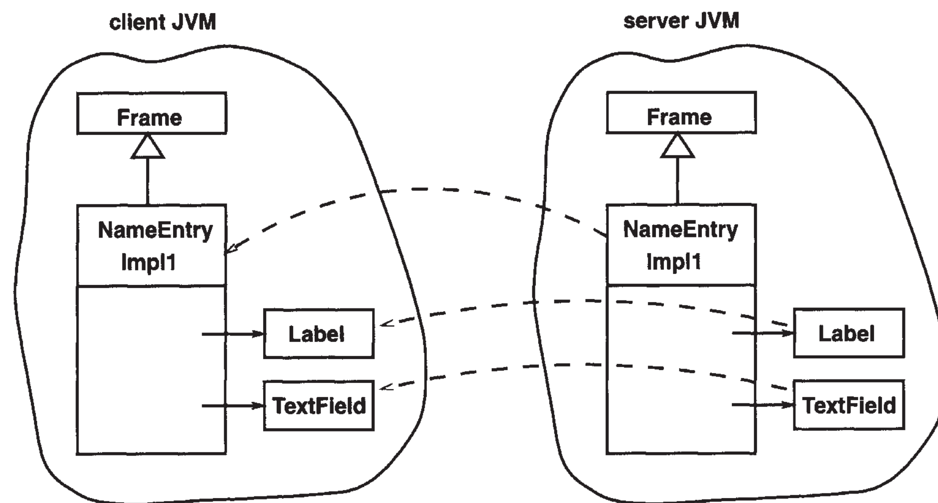


Figure 13-2. JVM objects for the naive implementation of the user interface

Another problem with this code is that it first creates an object on the server that has heavy reliance on environmental factors on the server. It then removes itself from that environment and has to reestablish itself on the target client environment.

On my current system, this dependence on environments shows up as a _TextField complaining that it cannot find a whole bunch of fonts on my server. Of course, that doesn't matter because it gets moved to the client machine. (As it happens, the fonts aren't available on my client machine either, so I end up 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 pieces on the server side that exist on the client.

What Files Need to Be Where?

The client needs to know the `NameEntry` interface class. This must be in its `CLASSPATH`.

The server needs to know the class files for

- `NameEntry`
- `Server1`
- `NameEntryImpl1`

These must be in its `CLASSPATH`.

The HTTP server needs to know the class files for `NameEntryImpl1`. This must be in the directory of documents for this server.

libdocdel@fr.com

Factory Implementation

The second implementation minimizes the amount of serialized 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`, because that class also exists on the client side. The client calls the `show()` interface method, 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
 */

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) {}});
```


libdocdel@fr.com

```
        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
```

Figure 13-3 shows which instances are running in which JVM.

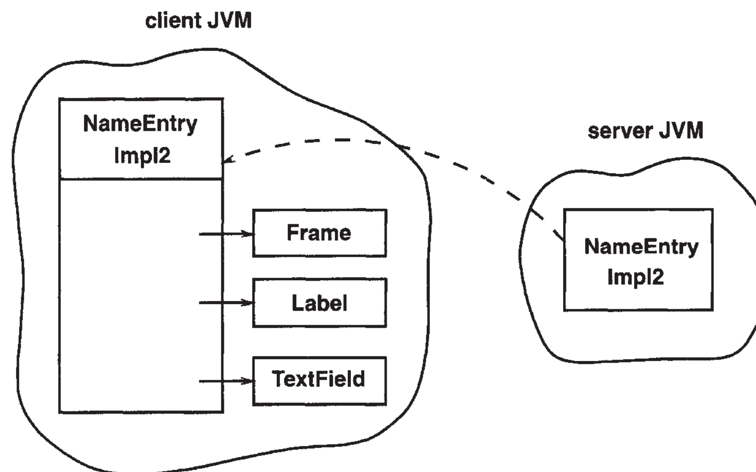


Figure 13-3. JVM objects for the factory implementation of the user interface

There are some standard classes that cannot be serialized: one example is the Swing `JTextArea` class (as of Swing 1.1). This has frequently been 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.

> **NOTE** *Swing is the set of user interface classes introduced as part of the Java Foundation Classes in JDK 1.2*

libdocdel@fr.com

What Files Need to Be Where?

For this implementation, the client needs to know the `NameEntry` interface class.

The server needs to know the class files for

- `NameEntry`
- `Server2`
- `NameEntryImpl2`
- `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 the naive implementation earlier in the chapter, this part of the code was commented out for simplicity.

The HTTP server needs to know the class files for

- `NameEntryImpl2`
- `NameEntryImpl2$1`

Using Multiple Class Files

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. That is, the server might upload an object from a single class to the lookup service and from there to a client. However, when the object runs, it needs to create *other* objects using class files that are not known to the client.

For example, the listener object of the last implementation could belong to a separate `NameHandler` class. The code looks like this:

```
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;
```


libdocdel@fr.com

```
import com.sun.jini.lease.LeaseRenewalManager;

/**
 * NameEntryImpl3.java
 */

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 of the user interface implementation uses a `NameHandler` class that only exists on the server machine. When the client attempts to deserialize the

libdocdel@fr.com

NameEntryImpl3 instance, it will fail to find this class and be unable to complete deserialization. How is this resolved? Well, in the same way as before, by making it available through the HTTP server.

Figure 13-4 shows which instances are running in which JVM.

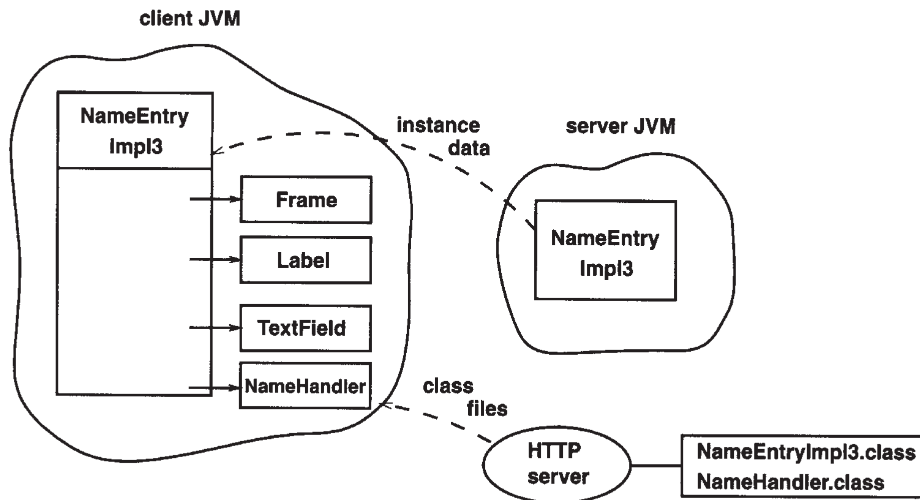


Figure 13-4. JVM objects for multiple class files implementation

What Files Need to Be Where?

The client needs to know the NameEntry interface class.

The server needs to know the class files for

- NameEntry
- Server3
- NameEntryImpl3
- NameEntryImpl3$1
- NameHandler

The NameHandler class file is another one produced by the compiler.

The HTTP server needs to know the class files for

- NameEntryImpl3
- NameEntryImpl3$1
- NameHandler

libdocdel@fr.com

Running Threads from Discovery

The previous section looked at issues involving the location of classes in order to reduce network traffic and to improve the speed and responses of clients and services. Within a client or service, other techniques, such as multithreading, can also be used to improve responsiveness.

In all of the examples using explicit registration (such as those in Chapters 8 and 9), a single thread was used. That is, as a service locator was discovered, the registration process commenced in the same thread. This registration may take some time, and during this time, new lookup services may be discovered. To avoid the possibility of these new services timing out and being missed, all registration processing should be carried out in a separate thread, rather than possibly holding up the discovery thread.

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 file classifier server, which is based on the server in Chapter 3 that uploads a complete service. In this version, the registration code is moved to a separate thread, which is implemented using an inner class:

```
package complex;

import complete.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;
```


libdocdel@fr.com

```
/**
 * FileClassifierServer.java
 */

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
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(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];
```


libdocdel@fr.com

```
                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());
                    return;
                }
                System.out.println("service registered");

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


libdocdel@fr.com

Join Manager Threads

If you use a JoinManager to handle lookup and registration, then it essentially does this for you: it creates a new thread to handle registration. Thus, the examples in Chapter 11 do not need any modification, as the JoinManager already uses the concepts of this section.

Client Threads

It is probably more important to use threads in the client than in the server, because the client will actually perform some computation (which may be lengthy) based on the service it discovers. Again, this is a simple matter of moving code into a new class that implements Thread. Doing this to the multicast client TestFileClassifier of Chapter 3 results in the following code:

```
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
 */

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
```


libdocdel@fr.com

```
    }
}

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();
    }
}

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

class LookupThread extends Thread {

    ServiceRegistrar registrar;

    LookupThread(ServiceRegistrar registrar) {
        this.registrar = registrar;
    }


    public void run() {
```


libdocdel@fr.com

```
            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();
                return;
            }
            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
```

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 simple printer interface that will allow us to print documents and also allow us to access the printer speed as follows:

```
package common;

import java.io.Serializable;
```


libdocdel@fr.com

```
/**
 * Printer.java
 */

public interface Printer extends Serializable {

    public void print(String str);
    public int getSpeed();

} // Printer
```

I don't want to delve here into the complexities of building a real printer service. A "fake" printer implementation that takes its speed from a parameter in the constructor can be written as a complete uploadable service (see Chapter 3) as follows:

```
package printer;

/**
 * PrinterImpl.java
 */

public class PrinterImpl implements common.Printer, java.io.Serializable {

    protected int speed;

    public PrinterImpl(int sp) {
        speed = sp;
    }

    public void print(String str) {
       // fake stuff:
    System.out.println("I'm the " + speed + " pages/min printer");
    System.out.println(str);
    }

    public int getSpeed() {
        return speed;
    }

} // PrinterImpl
```

Printer implementations can be created and made available using server implementations of earlier chapters.

libdocdel@fr.com

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 satisfies other types of Boolean conditions.

The following program shows how you can 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
 */

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);

    }
```


libdocdel@fr.com

```
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) {
```


libdocdel@fr.com

```
        TestPrinterSpeed f = new TestPrinterSpeed();

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

} // TestPrinterSpeed
```

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, it may know both, or it may know none! When the client starts up, it will be discovering these services in an arbitrary order, and the code must be structured to deal with this.

These are some of the cases that may arise:

- A printer may be discovered before any distance service has been found. In this case, the printer must be stored for later distance checking.

- A printer may be discovered after a distance service has been found. It can be checked immediately.

libdocdel@fr.com

- A distance service is found after some printers have been found. This saved set of printers 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 save 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. Therefore, instead of just storing printers, we need to store `ServiceItem` objects, 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! This could be done, for example, by popping up a dialog box asking the user to locate themselves.

A client satisfying these requirements is given in the following program. (The location of the client is hard-coded into the `getMyLocation()` method for simplicity.)

```
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
 */

public class TestPrinterDistance implements DiscoveryListener {

    protected Distance distance = null;
    protected Object distanceLock = new Object();
```


libdocdel@fr.com

```
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();
    }
}

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


libdocdel@fr.com

```
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]);
                        }
                    }
                }
            }
        }
    }
```


libdocdel@fr.com

```
/*
 * 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
```


libdocdel@fr.com

```
         */
        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
 */

public interface Distance extends java.io.Serializable {

    int getDistance(Location loc1, Location loc2);

} // Distance
```


libdocdel@fr.com

The following is an example implementation of a distance service:

```
package complex;

import net.jini.lookup.entry.Location;

/**
 * DistanceImpl.java
 */

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
```

Earlier in this chapter we gave the code for `PrinterImpl`. A simple program to start up a distance service and two printers is as follows:

```
package complex;

import printer.PrinterImpl;
import printer.PrinterImpl;
import complex.DistanceImpl;

// import com.sun.jini.lookup.JoinManager;
import net.jini.lookup.JoinManager;
```


libdocdel@fr.com

```
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
 */

public class PrinterServerLocation implements ServiceIDListener {

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

        // run forever
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(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());
```


libdocdel@fr.com

```
            // slow printer in room 120
            joinMgr = new JoinManager(new PrinterImpl(20),
                                      new Entry[] {new Location("1", "120",
                                                                "Building 1")},
                                      this,
                                      mgr,
                                      new LeaseRenewalManager());

            // fast printer in room 130
            joinMgr = new JoinManager(new PrinterImpl(30),
                                      new Entry[] {new Location("1", "130",
                                                                "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
```

Finding a Service Once Only

There may be many lookup services on the local network, perhaps specializing in certain groups of services. There could be many lookup services running on the Internet, which could act as global repositories of information. For example, there is a lookup service running at `http://www.jini.canberra.edu.au` that acts as a publicly available lookup service for those who wish to experiment with Jini. One may expect to find lookup services acting in a "portal" role, listing all of the public clock services, the real estate services, and so on.

A 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

libdocdel@fr.com

find all service implementations. This second case can cause some uniqueness problems: if a client finds every service that has been registered with multiple locators, then it will probably find the same service more than once.

Why is this a problem? Well, suppose the client wants to find all power drills in the factory and ask them to drill exactly one hole each. Or suppose it finds all backup services for the system, and wants each one to perform a single backup. In that case, 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, each drill might make six holes because the client got a copy of each drill from six service locators, or you might get 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.

Jini has a concept of a service being a "good citizen." This concept includes having a single identity across all lookup services, which allows clients to tell whether they have come across multiple copies of the same service or have encountered a different implementation of the service. The behavior on the part of services is contained in the Jini "Lookup Service" specification, and it 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 the `ServiceID` 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 `ServiceID` for two different services, and two different locators cannot generate the same `ServiceID`. This provides a unique identifier that can be used to identify duplicates.

The procedure for a service to follow when it is registering itself with service locators is as follows:

1. The very first time a service is registered, use `null` as the `serviceID` value of the `ServiceItem` in `ServiceRegistrar.register()`.

2. The returned `ServiceRegistration` has a `getServiceID()` method for retrieving the `ServiceID`. This `ServiceID` should then be used in any future registrations both with this service locator and with any others. This ensures that the service has a unique identity across all lookup services. It should be noted that `JoinManager` already does this, although this is not stated in its documentation. We have done this in earlier examples, such as the server in Chapter 8.

3. The client has a choice of two `lookup()` methods to use with its `ServiceRegistrar` object: the first just returns a single object, and the second returns an array of `ServiceMatches` objects. This second one is more

libdocdel@fr.com

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 IDs that it has seen, and compare any new ones against it—then it can check whether a service is a new one or a previously seen one.

5. The service should save its ID in persistent storage so that if it dies and restarts, it can use the same ID—after all, it is the same service. (This may involve subtle considerations: it should only use the same ServiceID if it really is the same service. For example, if the service maintains state that is lost in a crash, then it isn't the same service!)

In Chapter 8 we gave the code for a multicast client that looked for a service, used it, and exited. A modified client that looks for all *unique* services and uses each one is as follows:

```
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
 */

public class TestFileClassifier implements DiscoveryListener {

    protected Vector serviceIDs = new Vector();

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


libdocdel@fr.com

```
        // 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();
                continue;
            }

            ServiceItem[] items = matches.items;
```


libdocdel@fr.com

```
            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
```

Leasing Changes to a Service

Sometimes a service may allow changes to be made to its state 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 that can handle a particular MIME type starts: it can register the filename mapping with a file classifier service. However, the file classifier service will just time out the mapping unless the new service keeps it renewed.

The example in this section follows the "Granting and Handling Leases" section of Chapter 7. It gives a concrete illustration of that section, now that there is enough background to do so.

libdocdel@fr.com

Leased FileClassifier

A dynamically extensible version of a file classifier will have methods to add and remove MIME mappings:

```
package common;

import java.io.Serializable;

/**
 * LeaseFileClassifier.java
 */

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.

     */
    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 Chapter 7. The client and the service will be in different Java VMs, probably on different machines. Figure 13-5 gives the object structure on the service side.

libdocdel@fr.com

This should be compared to Figure 7-3 where we considered the "foo" implementation of landlord leasing.

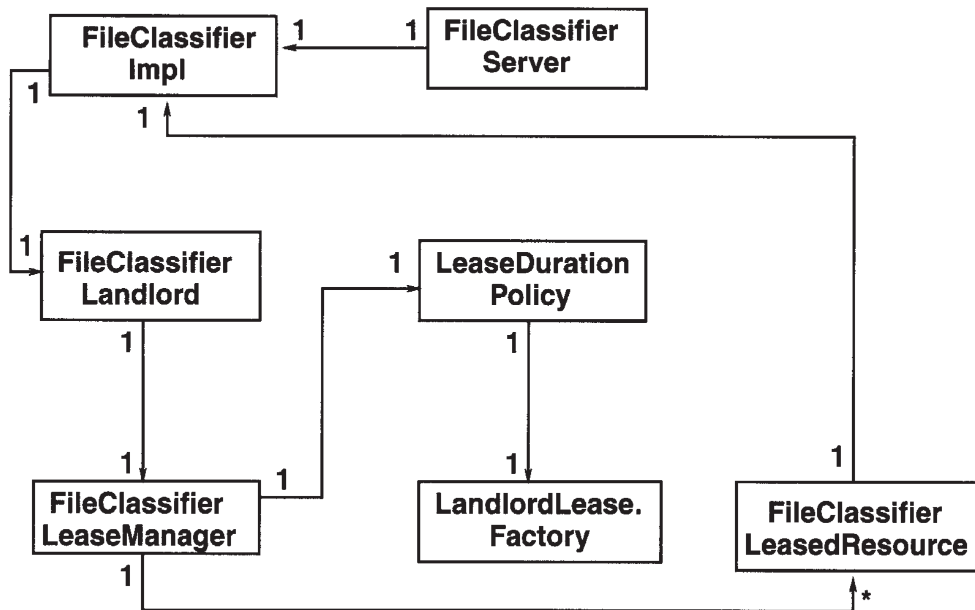


Figure 13-5. Class diagram for leasing on the server

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. This is shown in Figure 13-6.

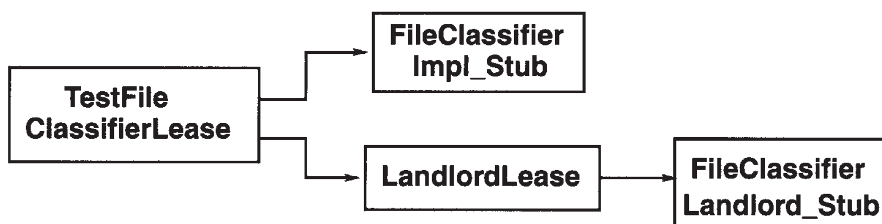


Figure 13-6. Class diagram for leasing on the client

libdocdel@fr.com

The FileClassifierLeasedResource Class

The `FileClassifierLeasedResource` class acts as a wrapper around the actual resource, adding cookie and time expiration fields around the resource. It adds a unique cookie mechanism, in addition to making the wrapped resource visible.

```
/**
 * FileClassifierLeasedResource.java
 */
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;
    }
```


libdocdel@fr.com

```
    public String getSuffix() {
        return suffix;
    }
} // FileClassifierLeasedResource
```

The FileClassifierLeaseManager Class

The `FileClassifierLeaseManager` class is very similar to the code given for the `FooLeaseManager` in Chapter 7:

```
/**
 * FileClassifierLeaseManager.java
 */
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) {
```


libdocdel@fr.com

```
        fileClassifierResources.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 = 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;
    }
```


libdocdel@fr.com

```
    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
```

The FileClassifierLandlord Class

The `FileClassifierLandlord` class is very similar to the `FooLandlord` in Chapter 7:

```
/**
 * FileClassifierLandlord.java
 */

package lease;

import common.LeaseFileClassifier;
```


libdocdel@fr.com

```
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 Land-
lord, 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) {
        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 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,
```


libdocdel@fr.com

```
                                            long[] extension) {
        return null;
    }
} // FileClassifierLandlord
```

Summary

Jini provides a framework for building distributed applications. Nevertheless, there is still room for variation in how services and clients are written, and some of these are better than others. This chapter has looked at some of the variations that can occur and how to deal with them.

libdocdel@fr.com

CHAPTER 14

Remote Events

COMPONENTS OF A SYSTEM CAN CHANGE STATE and may need to inform other components that this change has happened. Java Beans and user-interface elements such as AWT or Swing objects use events to signal these changes. Jini also has an event mechanism, and 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. Event listeners may disappear, and so the Jini event mechanism uses leases to manage listener lists.

This chapter also looks at how leases are managed by event sources. Finally, we'll look at how events can be used by applications to monitor when services are registered or discarded from service locators.

Event Models

Java has a number of event models, differing in various subtle ways. All of these involve an object (an *event source*) generating an event in response to some change of state, either in the object itself (for example, if someone has changed a field), or in the external environment (such as when a user moves the mouse). At some earlier stage, a listener (or set of listeners) will have registered interest in this event. When the event source generates an event, it will call suitable methods on the listeners with the event as parameter. The event models all have their origin in the `Observer` pattern from *Design Patterns*, by Eric Gamma et al., but this is modified by other pressures, such as Java Beans.

There are low-level input events, which are generated by user actions when they control an application with a graphical user interface. These events—of type `KeyEvent` and `MouseEvent`–are placed in an event queue. They 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 dispatching it to listeners, and the 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 losing 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 them 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

libdocdel@fr.com

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 involved in this, specified 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 by the event source calling the listener's `PropertyChangeListener`'s `propertyChange()` method or the `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 the listeners either for information or for vetoing.

Jini objects may also be interested in changes in other Jini objects, and might like to be listeners for such changes. The networked nature of Jini has led to a particular event model that differs slightly from the other models already in Java. The differences are caused by several factors:

- Network delivery is unreliable—messages may be lost. Synchronous methods requiring a reply may not work here.
- Network delivery is time-dependent—messages may arrive at different times to different listeners. As a result, the state of an object as perceived by a listener at any time may be inconsistent with the state of that object as perceived by others. Passing complex object state across the network may be more complex to manage than passing simpler information.
- A remote listener may have disappeared by the time the event occurs. Listeners have to be allowed to time out, like services do.
- Java Beans can require method names and event types that vary and can use many classes. This requires a large number of classes to be available across the network, which is more complex than a single class with a single method with a single event type as parameter (the original `Observer` pattern used a single class with only one method, for simplicity).

Remote Events

Unlike the large number of event classes in the AWT and Swing, for example, Jini uses events of one type, the `RemoteEvent`, or a small number of subclasses of `RemoteEvent`. The `RemoteEvent` class has these public methods (and some inherited methods):

```
package net.jini.core.event;

public class RemoteEvent implements java.io.Serializable {
    public long getID();
    public long getSequenceNumber();
```


libdocdel@fr.com

```
    public java.rmi.MarshalledObject getRegistrationObject();
}
```

Events in Beans and AWT convey complex object state information, and this is enough for the listeners to act with full knowledge of the changes that have caused the event to be generated. Jini events avoid this, and convey just enough information to allow state information to be found if needed. A remote event is serializable and is moved around the network to its listeners. The listeners then have to decide whether or not they need more detailed information than the simple information in each remote event. If they do need more information, they will have to contact the event source to get it.

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 the `id` 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 and order of events. In a network system this is not so easy. Jini makes no assumptions about guarantees of delivery and does not even assume that events are delivered in order. The Jini event mechanism does not specify how events get from producer to listener—it could be by RMI calls, but it may be through an unreliable third party. The event source supplies a sequence number that could be used to construct state and ordering information if needed, and this generalizes 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." Another event with `id` of `REMOVE_TYPE` and sequence number of `11` would be taken as a later event, even if it arrived earlier. The listener will receive the event with id and sequence number only. Either this will be meaningful to the listener, or it will need to contact the event source and ask for more information about that sequence number. The event source should be able to supply state information upon request, given the sequence number.

An idea borrowed from systems such as the Xt Intrinsics and Motif is called *handback* data. This is a piece of data that is given by the listener to the event source at the time it registers itself for events. The event source records this handback and then returns it to the listener with each event. This handback can be a reminder of listener state at the time of registration.

This can be a little difficult to understand at first. The listener is basically saying to the event source that it wants to be told whenever something interesting happens, but when that does happen, the listener may have forgotten why it was

libdocdel@fr.com

interested in the first place, or what it intended to do with the information. So the listener also the gives the event source some extra information that it wants returned as a "reminder."

For example, a Jini taxi-driver might register interest in taxi-booking events from the base station while passing through a geographical area. It registers itself as a listener for booking events, and as part of its registration, it could include its current location. Then, when it receives a booking event, it is told 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 could register a different object for events, so your stockbroker could register you for events about stock movements, and when you receive an event, you would also get a reminder about who registered your interest (plus a request for commission...).

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, like this

```
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 is the convenience class `EventRegistration`:

```
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 values 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.

libdocdel@fr.com

This means that as the programmer of a event producer, you have to define (and implement) methods such as these:

```
public EventRegistration addRemoteEventListener(RemoteEventListener listener);
```

There is no standard interface for this.

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`, 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. There is no public API for dealing with event-listener lists, and so the programmer has to reinvent (or copy) code to pass events to listeners. There are basically two cases:

- Only one listener can be in the list.
- Any number of listeners can be in the list.

Single Listener

The case where there is only one listener allowed in the list can be implemented by using a single-valued variable, as shown in Figure 14-1.

This is the simplest case of event registration:

```
protected RemoteEventListener listener = null;

public EventRegistration addRemoteListener(RemoteEventListener listener)
       throws java.util.TooManyListenersException {
```


libdocdel@fr.com

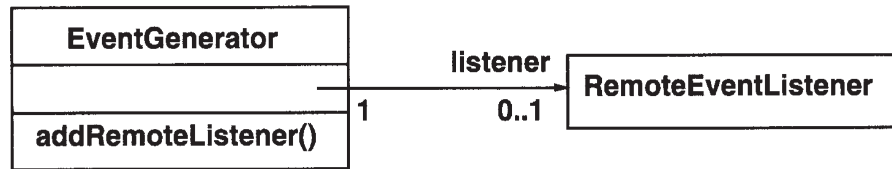


Figure 14-1. A single listener

```
    if (this.listener == null {
        this.listener = listener;
    } else {
        throw new java.util.TooManyListenersException();
    }
    return new EventRegistration(0L, this, null, 0L);
}
```

This is close 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 you 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 object in the registration and notify methods. Far more complex is adding a non-null lease. Firstly, the event source has to decide on a lease policy, that is, for what periods of time it will grant leases. Then it has to implement a timeout 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.

libdocdel@fr.com

Multiple Listeners

For the case where there can be any number of listeners, the convenience class `javax.swing.event.EventListenerList` can be used. The object delegates some of the list handling to the convenience class, as shown in Figure 14-2.

Figure 14-2. Multiple listeners

A version of event registration suitable for ordinary events is as follows:

```
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
```


libdocdel@fr.com

```
    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();
            }
        }
    }
}
```

In this case, 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.

Listener Source

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. Jini is a networked federation of objects, and in many cases one is dealing with proxy objects, not the real objects.

This is the same with remote events, except that in this case we often have the direction of proxies reversed. To see what I 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, and it will use this proxy to register itself as a listener. However, 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.

libdocdel@fr.com

Consider the file classification problems we looked at in earlier chapters. The file classifier had a hard-coded set of filename extensions built in. However, it would be possible to extend these, if applications come along that know how to define (and maybe handle) such extensions. For example, 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 (and is probably sleeping) on an unknown filename extension. When the classifier installs a new file type, it can send an event saying so. The blocked application could then try again to see if the extension is now known. If so, it uses it, and if not, it blocks again. Note that we don't bother with identifying the actual state change, since it is just as easy to make another query once you know that the state has changed. More complex situations may require more information to be maintained. However, 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 the listener.

How this gets resolved is for the client to first find the service in the same way as we discussed in Chapter 6. The client ends up with a proxy object for the service in the client's address space. One of the methods on the proxy will add an event listener, and this method will be called by the client.

For simplicity, assume that the client is being added as a listener to the service. The client will call the add listener method of the proxy, with the client as parameter. 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. When the service makes notification calls to the proxy listeners, the client's proxy can make a remote call back to the client itself. These proxies are shown in Figure 14-3.

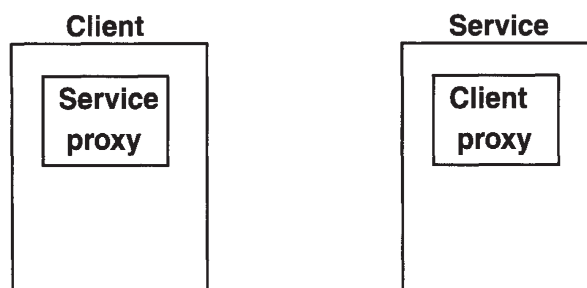


Figure 14-3. Proxies for services and listeners

libdocdel@fr.com

File Classifier with Events

Let's make this discussion more concrete by looking at a new file classifier application that can have its set of mappings dynamically updated.

The first thing to be modified is the `FileClassifier` interface. This needs to be extended to a `MutableFileClassifier` interface, known to all objects. This new interface adds methods that will add and remove types, and that will also register listeners for events. The event types are labeled with two constants. The listener model is simple, and does not include handbacks or leases. The sequence identifier must be increasing, so we just add 1 on each event generation, although we don't really need it here: it is easy for a listener to just make MIME type queries again.

```
package common;

import java.io.Serializable;

/**
 * MutableFileClassifier.java
 */

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;
```


libdocdel@fr.com

```
    public EventRegistration addRemoteListener(RemoteEventListener listener)
        throws java.rmi.RemoteException;


} // MutableFileClasssifier
```

The RemoteFileClassifier interface is known only to services, and it just changes its package and inheritance for any service implementation:

```
package mutable;

import common.MutableFileClassifier;
import java.rmi.Remote;

/**
 * RemoteFileClassifier.java
 */

public interface RemoteFileClassifier extends MutableFileClassifier, Remote {

} // RemoteFileClasssifier
```

Previous implementations of file classifier services (such as in Chapter 8) use a static list of if...then statements because they deal with a fixed set of types. For this implementation, where the set of mappings can change, we change the implementation to a dynamic map keyed on file suffixes. It manages the event listener list for multiple listeners in the simple way discussed earlier in this chapter, and it generates events whenever a new suffix/type is added or successfully removed. The following code is an implementation of the file classifier service with this alternative implementation and an event list:

```
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;
```


libdocdel@fr.com

```
import common.MutableFileClassifier;
import java.util.Map;
import java.util.HashMap;

/**
 * FileClassifierImpl.java
 */

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();

    protected long seqNum = 0L;

    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 {
```


libdocdel@fr.com

```
    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,
                                              seqNum++, null);
            }
            try {
                listener.notify(remoteEvent);
            } catch(UnknownEventException e) {
                e.printStackTrace();
            } catch(RemoteException e) {
                e.printStackTrace();
```


libdocdel@fr.com

```
                }
            }
        }
    }

    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. The following class is for the proxy:

```
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
 */

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)
```


libdocdel@fr.com

```
        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
```

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. However, there are a lot of times when other services or clients will want to know when services start or are removed. For example, an editor may want to know if a disk service has started so that it can save its file; a graphics display program may want to know when printer services start up; the user interface for a camera may want 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 `ServiceEvent` type events, which subclass from `RemoteEvent`. These events are generated in response to changes in the state of services that match (or fail to match) a template pattern for services. This event type has three categories from the `ServiceEvent.getTransition()` method:

- `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 or to spot wanted changes in the attributes of an existing registered service; for example, an offline printer can change attributes to being online, which now makes it a useful service.

libdocdel@fr.com

- 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. This transition can also be used to spot changes in the attributes of an existing registered service that are not wanted; for example, an online printer can change attributes to being offline.

- TRANSITION_MATCH_MATCH: A service has changed state, but it matched both before and after. This typically happens when an Entry value changes, and it is used to monitor changes of state, such as a printer running out of paper, or a piece of hardware signaling 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 services it is interested in. A client that wants to monitor all changes could prepare a template such as this:

```
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. The following is a program to monitor all changes.

```
/**
 * RegistrarObserver.java
 */

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;
```


libdocdel@fr.com

```
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("notified 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");
```


libdocdel@fr.com

```
                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
```

The following is a suitable driver for the preceding observer class:

```
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
 */

public class ReggieMonitor implements DiscoveryListener {
```


libdocdel@fr.com

```
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 lookup found");
        ServiceRegistrar registrar = registrars[n];
        if (registrar == null) {
            System.out.println("registrar null");
            continue;
        }
        try {
            System.out.println("Lookup service at " +
                               registrar.getLocator().getHost());
        } catch(RemoteException e) {
```


libdocdel@fr.com

```
                System.out.println("Lookup service info unavailable");
            }

            try {
                observers.add(new RegistrarObserver(registrar));
            } catch(RemoteException e) {
                System.out.println("adding observer failed");
            }

            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");
            }

            for (int m = 0; m < matches.items.length; m++) {
                if (matches.items[m] != null && matches.items[m].service != null) {
                    System.out.println("Reg knows about " + matches.items[m].ser-
vice.toString() +
                                       " with id " + matches.items[m].serviceID);
                }
            }

        }
    }

    public void discarded(DiscoveryEvent evt) {
        // remove observer
    }
} // ReggieMonitor
```

Summary

This chapter has looked at how the remote event differs from the other event models in Java and at how to create and use them. Jini events allow distributed components to inform other components when they change state and to supply enough support information for listeners to determine the nature of the change. This adds an asynchronous state-change mechanism to Jini, which can allow more flexible systems to be built.

libdocdel@fr.com

CHAPTER 15

ServiceDiscoveryManager

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.

The `ServiceDiscoveryManager` class performs client-side functions similar to that of `JoinManager` for services, and simplifies the task of finding services. The `ServiceDiscoveryManager` class is only available in Jini 1.1.

ServiceDiscoveryManager Interface

The `ServiceDiscoveryManager` class is a utility class designed to help in the various client-side lookup cases that can occur:

- A client may wish to use a service immediately or later.
- A client may want to use multiple services.
- A client will want to find services by their interfaces, but may also want to apply additional criteria, such as being a "fast enough" printer.
- A client may just wish to use a service if it is available at the time of the request, but alternatively may want to be informed of new services becoming available and to respond to this new availability (for example, a service browser).

Due to the variety of possible cases, the `ServiceDiscoveryManager` class is more complex than `JoinManager`. Its interface includes the following:

```
package net.jini.lookup;

public class ServiceDiscoveryManager {
    public ServiceDiscoveryManager(DiscoveryManagement discoveryMgr,
                                   LeaseRenewalManager leaseMgr)
           throws IOException;
```


libdocdel@fr.com

```
    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();
}
```

ServiceItemFilter Interface

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 way for a client to find a service is to ask for an instance of an interface, possibly with additional entry attributes. This matching is performed on the lookup service, and it only involves a form of exact pattern matching. It allows the client to ask for a toaster that will handle two slices of toast exactly, but not for one that will toast two or more.

Performing arbitrary Boolean matching on the lookup service raises a security issue as it would involve running some code from the client or service in the lookup service, and it also raises a possible performance issue for the lookup service. This means that enhancing the matching process in the lookup service is unlikely to ever occur, so any more sophisticated matching must be done by the client. 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 ServiceItemFilter interface is as follows:

```
package net.jini.lookup;

public interface ServiceItemFilter {
    boolean check(ServiceItem item);
}
```


libdocdel@fr.com

A client filter will implement this interface to perform additional checking.

Client-side filtering will not solve all of the problems of locating the "best" service. Some situations will still require other services that know "local" information, such as distances in a building.

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. The client will be prepared to wait a certain amount of time before giving up. All issues of discovery can be given to the `ServiceDiscoveryManager`, and the task of 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.ServiceDiscoveryManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;

/**
 * ImmediateClientLookup.java
 */

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(2*WAITFOR);
```


libdocdel@fr.com

```
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public ImmediateClientLookup() {
        ServiceDiscoveryManager clientMgr = null;

        System.setSecurityManager(new RMISecurityManager());

        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            clientMgr = new ServiceDiscoveryManager(mgr,
                                                new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        Class [] classes = new Class[] {FileClassifier.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        ServiceItem item = null;
        // Try to find the service, blocking till timeout if necessary
        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);
        }

        // Get the service
        FileClassifier classifier = (FileClassifier) item.service;
```


libdocdel@fr.com

```
        if (classifier == null) {
            System.out.println("Classifier null");
            System.exit(1);
        }

        // Now we have a suitable service, use it
        MIMEType type;
        try {
            String fileName;

            // Try several file types: .txt, .rtf, .abc
            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
```

Using a Filter

An example in Chapter 13 discussed how to select a printer with a speed greater than a certain value. This type of problem is well suited to the `ServiceDiscoveryManager`

libdocdel@fr.com

using a `ServiceItemFilter`. The `ServiceItemFilter` interface has a `check()` method, which is called on the client side to perform additional filtering of services. This method can accept or reject a service based on criteria supplied by the client.

The following program illustrates how this `check()` method can be used to select only printer services with a speed greater than 24 pages per minute:

```
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.ServiceDiscoveryManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.ServiceItemFilter;
/**
 * TestPrinterSpeedFilter.java
 */

public class TestPrinterSpeedFilter implements ServiceItemFilter {
    private static final long WAITFOR = 100000L;

    public TestPrinterSpeedFilter() {
        ServiceDiscoveryManager clientMgr = null;

        System.setSecurityManager(new RMISecurityManager());

        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            clientMgr = new ServiceDiscoveryManager(mgr,
                                                new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

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


libdocdel@fr.com

```
        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(2*WAITFOR);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

} // TestPrinterSpeed
```


libdocdel@fr.com

Building a Cache of Services

A client may wish to make use of a service multiple times. If the client simply found a suitable reference to a service, then before each use it would have to check whether the reference was still valid, and if not, it would need to find another one. The client may also want to use minor variants of a service, such as a fast printer one time and a slow one the next. While this management can be done easily enough in each case, the ServiceDiscoveryManager can supply a cache of services that will do this work for you. This cache will monitor lookup services to keep the cache as up-to-date as possible.

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();
}
```

A suitable implementation object can be created by the ServiceDiscoveryManager method:

```
LookupCache createLookupCache(ServiceTemplate tmpl,
                                  ServiceItemFilter filter,
                                  ServiceDiscoveryListener listener);
```

We will ignore the ServiceDiscoveryListener until the next section of this chapter. 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 references to services matching the template and passing the filter. These references are all local to the client and consist of the service proxies and their attributes downloaded to the client. Searching for a service can then be done by local methods: LookupCache.lookup(). These can take an additional filter that can be used to further refine the set of services returned to the client.

The search in the cache is done directly on the proxy services and attributes already found by the client, and does not involve querying lookup services.

libdocdel@fr.com

Essentially, this involves a tradeoff of lookup service activity while the client is idle to produce fast local response when the client is active.

There are versions of `ServiceDiscoveryManager.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 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.

Here is a version of the file classifier client that creates and examines the cache for a suitable service:

```
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.ServiceDiscoveryManager;
import net.jini.lookup.LookupCache;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;

/**
 * CachedClientLookup.java
 */

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
        }
    }
```


libdocdel@fr.com

```
public CachedClientLookup() {
    ServiceDiscoveryManager clientMgr = null;
    LookupCache cache = null;

    System.setSecurityManager(new RMISecurityManager());

    try {
        LookupDiscoveryManager mgr =
            new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                       null /* unicast locators */,
                                       null /* DiscoveryListener */);
        clientMgr = new ServiceDiscoveryManager(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);
    }

    // loop until we find a service
    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;
```


libdocdel@fr.com

```
        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
```

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 because 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 (as explained in Chapter 14).

libdocdel@fr.com

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 `ServiceDiscoveryManager` uses a nested class, `ServiceDiscoveryManager.LookupCacheImpl.LookupListener`, which has an RMI stub.

In order for the cache to actually work, 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/ServiceDiscoveryManager$LookupCacheImpl$LookupListener_Stub.class` is accessible from this codebase. The stub file may be found in the `lib/jini-ext.jar` library in the Jini 1.1 distribution. It has to be extracted from there and placed in the codebase using a command such as this:

```
unzip jini-ext.jar 'net/jini/lookup/ServiceDiscoveryManager$LookupCache-
Impl$LookupListener_Stub.class' -d /home/WWW/htdocs/classes
```

Note that the specification just says that 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 `ServiceDiscoveryManager` as implemented by Sun. If another implementation is made of the Jini classes, then it would probably use a different remote class.

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 to 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);
}
```

The `ServiceDiscoveryListener` methods take a parameter of type `ServiceDiscoveryEvent`. This class has methods:

```
package net.jini.lookup;

class ServiceDiscoveryEvent extends EventObject {
    ServiceItem getPostEventServiceItem();
    ServiceItem getPreEventServiceItem();
}
```


libdocdel@fr.com

Clients are not likely to be interested in all events generated by lookup services, even for the 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, because 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 Chapter 14, an example was given on monitoring changes to services from a lookup service viewpoint, reporting each change to lookup services. A client-oriented view just monitors changes in services themselves, which can be done easily using `ServiceDiscoveryEvent` objects:

```
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.ServiceDiscoveryManager;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.LookupCache;

/**
 * ServiceMonitor.java
 */

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
        }
    }
```


libdocdel@fr.com

```
public ServiceMonitor() {
    ServiceDiscoveryManager clientMgr = null;
    LookupCache cache = null;

    System.setSecurityManager(new RMISecurityManager());

    try {
        LookupDiscoveryManager mgr =
            new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                       null /* unicast locators */,
                                       null /* DiscoveryListener */);
        clientMgr = new ServiceDiscoveryManager(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());
```


libdocdel@fr.com

```
    }
    public void serviceRemoved(ServiceDiscoveryEvent evt) {
        // evt.getPostEventServiceItem() == null
        ServiceItem preItem = evt.getPreEventServiceItem();
        System.out.println("Service disappeared: " +
                           preItem.service.getClass().toString());
    }

} // ServiceMonitor
```

Summary

The client lookup manager can handle a variety of common situations that arise as clients need to find services under different situations.

libdocdel@fr.com

CHAPTER 16

Transactions

TRANSACTIONS ARE A NECESSARY 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 the property having either no owners or two owners.

The theory of transactions often includes mention of 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 are asked to "vote" on a transaction. If all participants agree to go ahead, then the transaction "commits," which is binding on all the participants. If any "abort" during this voting stage, this forces abortion of the transaction for 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 choose to do so. Jini essentially supplies the mechanism of two-phase commit and leaves the policy to the participants in a transaction.

Transaction Identifiers

Restricting Jini transactions to a two-phase commit model without associating a particular semantics to it means that a transaction can be represented in a simple way, as a `long` identifier. This identifier is obtained from a transaction manager and

libdocdel@fr.com

will uniquely label the transaction to that manager. (It is not guaranteed to be unique between managers, though—unlike service IDs.) All participants in the transaction communicate with the transaction manager using this identifier to label which transaction they belong to.

The participants in a transaction may disappear, or the transaction manager may disappear. As a result, 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, which contains 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, and one of them will need to look after lease renewals. All the participants will use the transaction identifier in communication with the transaction manager.

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 with its `create()` method. Any of the participants can force the transaction to abort by calling `abort()`, or they 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

libdocdel@fr.com

commit, then the manager moves all of these listeners to the commit stage. Alternatively, any of the participants can call abort(), which forces all of the listeners to abort.

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 methods are triggered by calls made upon the transaction manager. For example, if one client calls the transaction manager to abort, then the transaction manager 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 first prepare() and then to either commit() or abort().

Mahalo

Mahalo is a transaction manager supplied by Sun as part of the Jini distribution. It can be used without any changes. It runs as a Jini service, like reggie, and 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, an HTTP server, and log files.

Mahalo can be started using a command line like this:

```
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 &
```


libdocdel@fr.com

A Transaction Example

The classic use of transactions is to handle money transfers between accounts. In this scenario there are two accounts, one of which is debited and the other credited.

This is not a very exciting example, so we shall try a more complex situation. Suppose a service decides 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 one in which the service gets some sort of customer ID from the client, and passes its own ID and the customer ID to the Accounts service, which manages both accounts. This is simple, it is prone to all sorts of e-commerce issues that we will not go into, and it is similar to the way credit cards work!

Figure 16-1 shows the messages in a normal sequence diagram. The client makes a `getCost()` call to the service and receives the `cost` in return. It then makes a `credit()` call on the service, which makes a `creditDebit()` call on the Accounts service before returning. The client then makes a final `requestService()` call on the service and gets back a `result`.

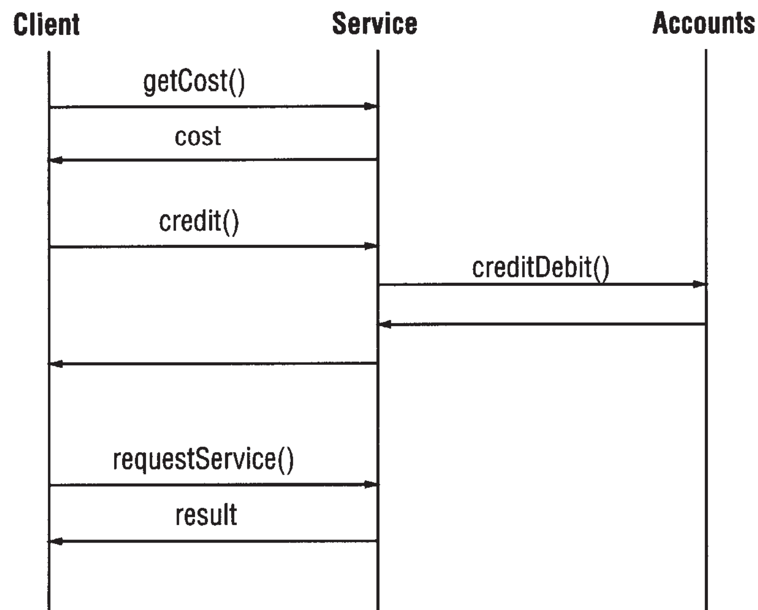


Figure 16-1. Sequence diagram for credit/debit example

There are a number of problems with the sequence of steps that can benefit by using 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

libdocdel@fr.com

issue of the quality of the service—suppose the client is not happy with the results from the service and would like to reclaim its money, or better yet, not spend it in the first case! If we include the delivery of the service in the transaction, then there is the opportunity for the client to abort the transaction before it is committed.

Figure 16-2 shows the larger set of messages in the sequence diagram for normal execution. As before, the client requests the cost from the service, and after getting this, it asks the transaction manager to create a transaction and receives back the transaction ID. It then joins the transaction itself. When it asks the service to credit an amount, the service also joins the transaction. The service then asks the account to `creditDebit()` the amount, and as part of this, the account also joins the transaction. The client then requests the service and gets the result. If all is fine, it then asks the transaction manager to `commit()`, which triggers the prepare-and-commit phase. The transaction manager asks each participant to `prepare()`, and if it gets satisfactory replies from each, it then asks each one to `commit()`.

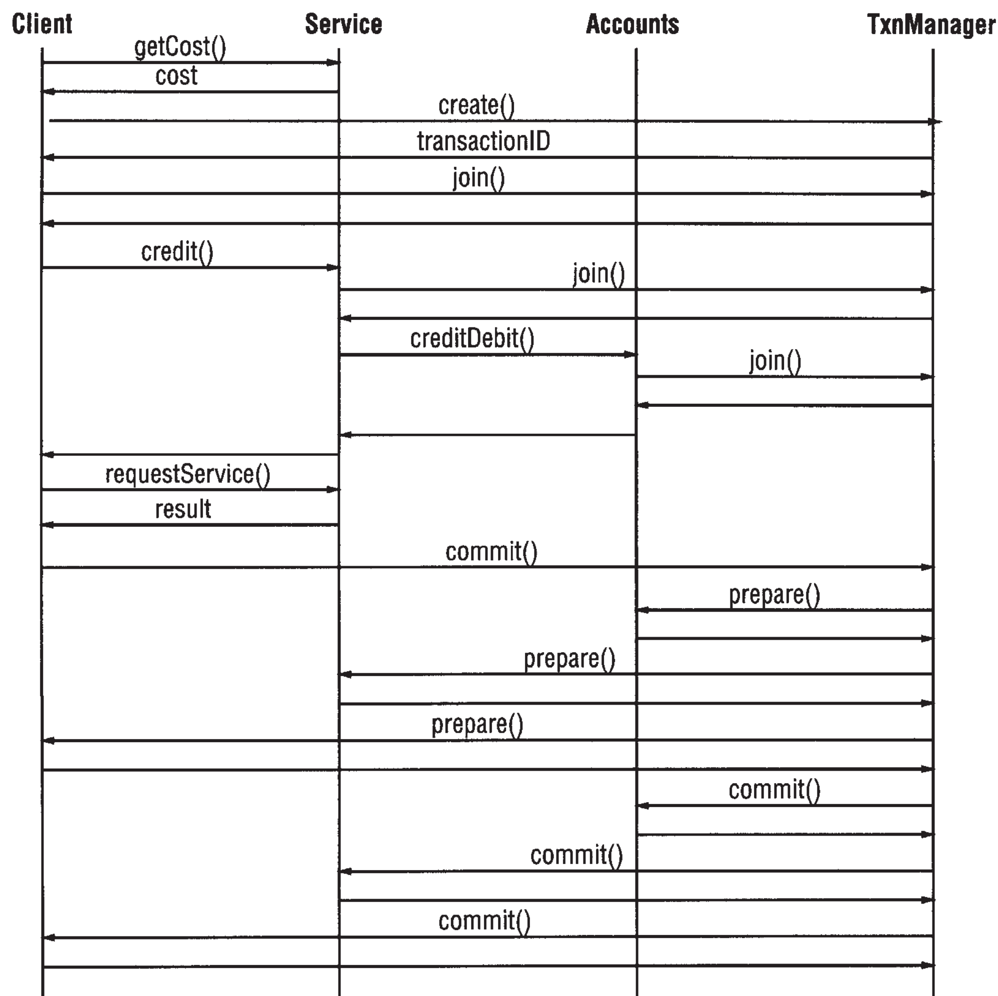


Figure 16-2. Sequence diagram for credit/debit example with transactions

libdocdel@fr.com

There are several points of failure in this transaction:

- The cost may be too high for the client. However, at this stage the client has not created or joined a transaction, so this doesn't matter.
- 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 roll back the transaction. (Of course, it could instead throw a `NotEnoughPayment` exception—joining and aborting is used for illustrating transaction possibilities.)
- 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 roll back.
- 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 roll back.
- The Accounts service may abort the transaction if sufficient client funds are unavailable.

PayableFileClassifierImpl

The service we will use here is a version of the familiar file classifier that requires a payment before it will divulge the MIME type for a filename. A bit unrealistic, perhaps, but that doesn't matter for our purposes here.

There will be a `PayableFileClassifier` interface, 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 with a `PayableFileClassifierImpl`.

The `PayableFileClassifierImpl` can use the implementation of the `rmi.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 inheritance diagram shown in Figure 16-3, which isn't really as complex as it looks.

The first new element in this hierarchy is the interface `Payable`:

```
package common;

import java.io.Serializable;
```


libdocdel@fr.com

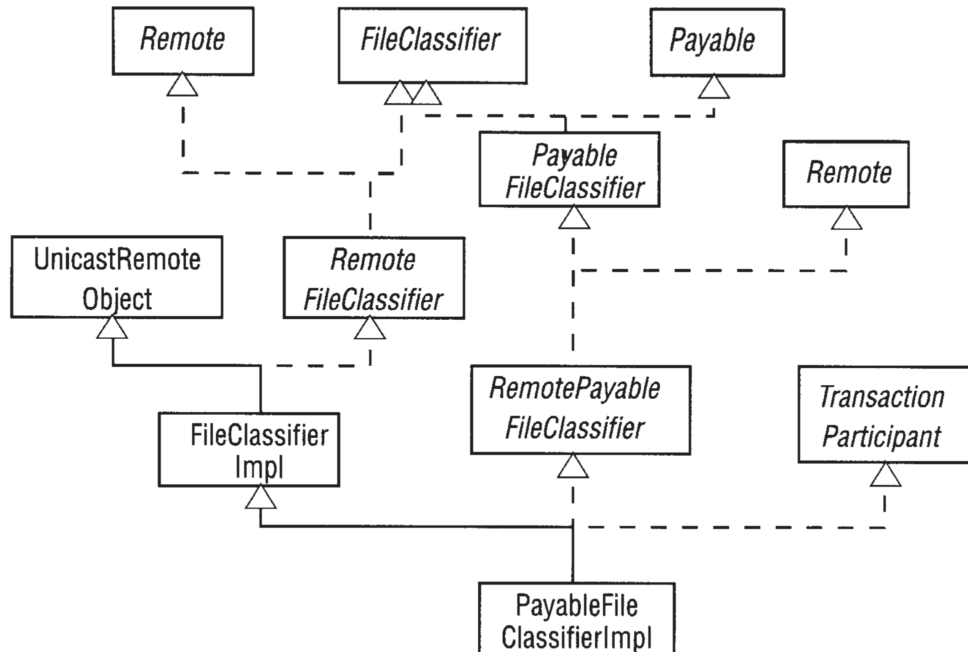


Figure 16-3. Class diagram for transaction participant

```
import net.jini.core.transaction.server.TransactionManager;

/**
 * Payable.java
 */

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 `Payable` is the `PayableFileClassifier` interface:

```
package common;

/**
 * PayableFileClassifier.java
 */
```


libdocdel@fr.com

```
public interface PayableFileClassifier extends FileClassifier, Payable {

} // PayableFileClassifier
```

PayableFileClassifier will be used by the client to search for the service. The service will use a RemotePayableFileClassifier, which is a simple extension to this:

```
package txn;

import common.PayableFileClassifier;
import java.rmi.Remote;

/**
 * RemotePayableFileClassifier.java
 */

public interface RemotePayableFileClassifier extends PayableFileClassifier, Remote
{

} // RemotePayableFileClasssifier
```

The implementation of this service joins the transaction, finds an Accounts service from a known location (using unicast lookup), registers the money transfer, and then performs the service. This implementation doesn't keep any state information that can be 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. The prepareAndCommit() method can be called by a transaction manager as an optimization, and the version given in this example follows the specification given in the "Jini Transaction" chapter of *The Jini Specification* by Ken Arnold et al. The following program gives this service implementation:

```
package txn;

import common.MIMEType;
import common.Accounts;
import rmi.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;
```


libdocdel@fr.com

```
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
 */

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) {
```


libdocdel@fr.com

```
            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());
```


libdocdel@fr.com

```
            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");

    }
```


libdocdel@fr.com

```
    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
```

AccountsImpl

We shall assume that all accounts in this example are managed by a single `Accounts` service that knows about all accounts by using a `long` identifier. These should be stored in permanent form, and there should be proper crash-recovery mechanisms, etc. For simplicity, we shall just use a hash table of accounts, with uncommitted transactions kept in a "pending" list. When commitment occurs, the pending transaction takes place.

Figure 16-4 shows the `Accounts` class diagram.

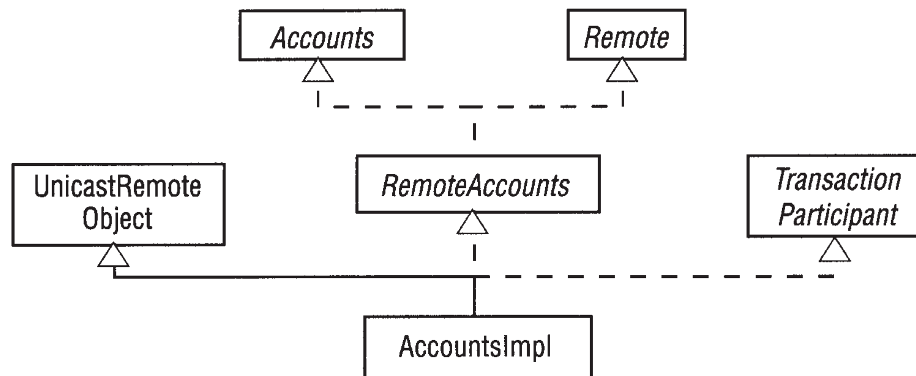


Figure 16-4. Class diagram for Accounts

libdocdel@fr.com

The `Accounts` interface looks like this:

```
/**
 * Accounts.java
 */

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 this is the implementation:

```
/**
 * AccountsImpl.java
 */

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??
```


libdocdel@fr.com

```
    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) {

        // Ensure stub class is loaded by getting its class object.
        // It has to be loaded from the same place as this object
        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) {
```


libdocdel@fr.com

```
        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;
}
```


libdocdel@fr.com

```
    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
```


libdocdel@fr.com

Client

The final component in this application is the client that starts the transaction. The simplest code for this would just use the blocking `lookup()` method of `ClientLookupManager` to find first the service and then the transaction manager. We will use the longer way to show various ways of doing things.

This implementation uses a nested class that extends `Thread`. Because of this, it cannot extend `UnicastRemoteObject` and so is not automatically exported. In order to export itself, it has to call the `UnicastRemoteObject.exportObject()` method. 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
 */

public class TestTxn implements DiscoveryListener {

    PayableFileClassifier classifier = null;
    TransactionManager mgr = null;

    long myClientID; // my account id
```


libdocdel@fr.com

```
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
}
```


libdocdel@fr.com

```
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) {
```


libdocdel@fr.com

```
                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);
```


libdocdel@fr.com

```
} 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());
}
```


libdocdel@fr.com

```
            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");
    }
```


libdocdel@fr.com

```
        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
```


libdocdel@fr.com

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 being left to the clients and services. The Jini distribution supplies a transaction manager, called `Mahalo`, that can be used.

libdocdel@fr.com

CHAPTER 17

LEGO MINDSTORMS

LEGO MINDSTORMS IS A "ROBOTICS INVENTION SYSTEM" that allows you to build LEGO toys with a programmable computer. This chapter looks at the issues involved in interfacing with a specialized hardware device, using MINDSTORMS as an example.

Making Hardware into Jini Services

Hardware devices and preexisting software applications can equally be turned into Jini services. A legacy piece of software can have a "wrapper" placed around it, and this wrapper can act as a Jini service. Remote method calls into this service can then make calls into the application. 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.

There are two major categories of hardware services: those that can run a Java virtual machine, and those that do not have enough memory or an adequate processor. For example, an 8086 with 20-bit addressing and only 1 MB of addressable memory would not be an adequate processor, while the owner of a Palm handheld might not wish to squander too many of its limited resources running a JVM. 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. At the time of writing, the lightweight JVM under development by Sun Microsystems called KVM does not support the features of JDK 1.2 required to run Jini.

Jini does not require all the core Java classes to run a service. For example, for a service that engages in discovery and registration does not require the AWT. However, it does require support for the newer RMI features found in JDK 1.2, and it does require enough of the standard language features. Again, this is not inclusive of all parts of Java; for example, floating point numbers are not required. Because many of the current embedded or small JVMs have removed features and standard core libraries, at present none of them have enough support for JDK 1.2 features to run Jini.

The current developments for embedded or small JVMs start with a minimal set of features and classes and incrementally allow more to be added, up to the

libdocdel@fr.com

level of a full JDK 1.2 with Jini. In any case, a device capable of running Jini will have 8 MB 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 32 MB of 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.

MINDSTORMS

LEGO MINDSTORMS (`http://www.LEGOMINDSTORMS.com`) is a Robotics Invention System that consists of a number of LEGO parts, a microcomputer called the RCX, an infrared transmitter (connected to the serial port of an ordinary computer), and various sensors and motors. Using this system, one can build an almost infinite variety of LEGO robots that can be controlled by the RCX. This RCX 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 system 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 32 K 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.

MINDSTORMS as a Jini Service

As previously mentioned, 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 that a robot is controlled by a single computer, and that it always stays in range of this computer.

There must be a way of communicating with any hardware device. For a MINDSTORMS robot, this is done via the serial port, but other devices may have different mechanisms. Communication may be by Java code or by the native code of the device. 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

libdocdel@fr.com

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 package includes platform-independent Java code, and also platform-specific native code libraries supplied as DLLs for Windows and Solaris. I am running Linux on my laptop, so I am using a Linux version of the DLL. This has been made by Trent Jarvi (trentjarvi@yahoo.com) and can be found at http://www.frii.com/~jarvi/rxtx/. The native code part of communicating with 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 that start with standard headers, and so on. A Java package that makes generating messages in the correct format easier has been created by Dario Laverde and is available at http://www.escape.com/~dario/java/rcx. There are other packages that will do the same thing—see the "LEGO MINDSTORMS Internals" Web 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 constructing an RCX program on a client and sending this program back to the server where it can be sent on to the RCX via the serial port. This program will then allow a client to control a MINDSTORMS robot remotely.

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 low-level code (probably using the Java Native Interface, JNI) and Java code to drive it.

RCXPort

Version 1.1 of the rcx 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();
```


libdocdel@fr.com

```
}
```

The `RCXOpcode` class has a useful static method for creating byte code:

```
package rcx;

public class RCXOpcode {
    public static byte[] parseString(String str);
}
```

The relevant methods for this project are the following:

- The constructor `RCXPort()`. This takes the name of a port as parameter, which should be something like `COM1` for Windows and `/dev/ttyS0` for Linux.

- The `write()` method is used to send an array of opcodes and their arguments to the RCX. This is machine code, and you can only read it with a disassembler or a Unix tool like octal dump (`od -t xC`).

- The static `parseString()` method of `RCXOpcode` can be used to translate a string of instructions in readable form to an array of bytes for sending to the RCX. It isn't as good as an assembler, because you have to give strings such as `"21 81"` to start the A motor. To use this method for Jini, we will have to use a non-static method in our interface, because static methods are not allowed.

- To handle responses from the RCX, a listener may be added with `addRCXListener()`. The listener must implement this 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);
}
```


libdocdel@fr.com

RCX Programs

At the lowest level, the RCX is controlled by machine-code programs sent via the infrared link. It will respond to these programs by stopping and starting motors, changing speed, and so on. 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, and it is available at `http://graphics.stanford.edu/~kekoa/rcx/`. Using these and the `rcx` package from Dario Laverde, we can control the RCX from a computer by standalone programs such as this:

```
/**
 * TestRCX.java
 */

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);
```


libdocdel@fr.com

```
    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
```


libdocdel@fr.com

```
     */
    public void receivedError(String error) {
        System.err.println("Error: " + error);
    }

    public static void main(String[] args) {
        new TestRCX();
    }

} // TestRCX
```

Jini Classes

A simple Jini service can use an 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 involves constructing the hierarchy of classes shown in Figure 17-1.

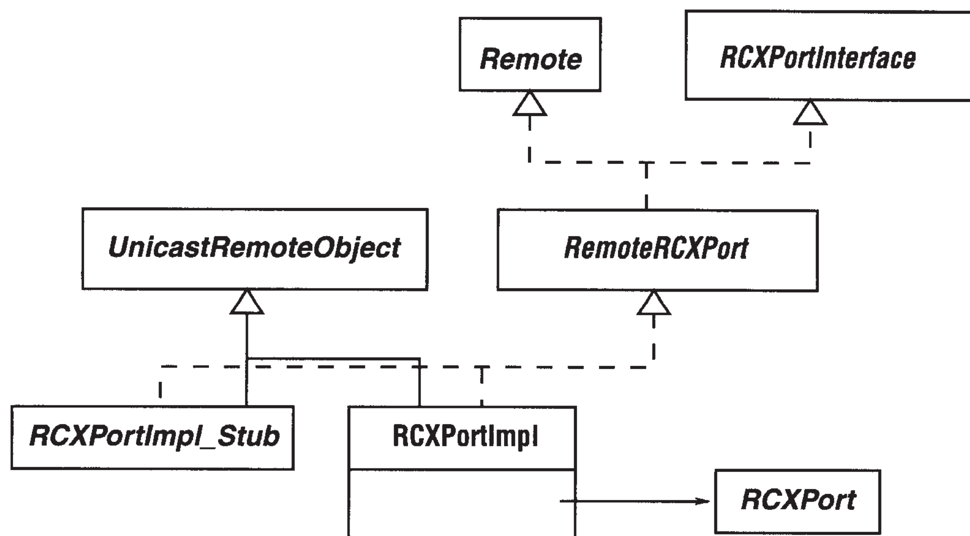


Figure 17-1. Class diagram for MINDSTORMS with RMI proxies

The RCXPortInterface just defines the methods we will 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 follows:

```
/**
 * RCXPortInterface.java
```


libdocdel@fr.com

```
 */

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
```


libdocdel@fr.com

We have chosen to make a subpackage of the rcx package and to place the preceding class in this package to make its role clearer. Note that the RCXPortInterface 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 used to handle messages sent from the RCX (addListener(), getMessage(), and getError()). Any listener that is added 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, as it only contains 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 with getMessage() and getError().

The RemoteRCXPort interface just adds the Remote interface:

```
/**
 * RemoteRCXPort.java
 */

package rcx.jini;

import java.rmi.Remote;

public interface RemoteRCXPort extends RCXPortInterface, Remote {

} // RemoteRCXPort
```

The RCXPortImpl constructs its own RCXPort object and feeds methods, such as write(), through to it. 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 provides a GUI interface for driving the RCX.) It looks for this port property in the parameters.txt file, which should have lines such as this:

```
port=/dev/ttyS0
```

Note that the parameters file exists on the server side—no client would know this information!

The RCXPortImpl also acts as a listener for "ordinary" RCX events signaling 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 there is one) by calling its notify() method.

libdocdel@fr.com

The implementation looks like this:

```
/**
 * RCXPortImpl.java
 */

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 {
```


libdocdel@fr.com

```
            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();
        }
    }
```


libdocdel@fr.com

```
/**
 * 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
 * sequence 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
 * sequence 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;
}
```


libdocdel@fr.com

```
    /**
     * 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
```

Getting It Running

To make use of these classes, we need to provide a server to get the service put onto the network, and we need some clients to make use of the service. This section will just look at a simple way of doing this, and later sections in this chapter will put in more structure.

The following is a simple server that follows the earlier examples of servers using RMI proxies (such as in Chapter 9), just substituting `RCXPort` for `FileClassifier` and using a `JoinManager`. It creates an `RCXPortImpl` object and registers it (or 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
 */

public class RCXServer implements ServiceIDListener {

    protected RCXPortImpl impl;
```


libdocdel@fr.com

```
    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 this example 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!

libdocdel@fr.com

An equally simplistic client that makes the RCX perform a few actions is given below. In addition to sending a set of commands to the RCX, the client must also listen for replies from the RCX. I have separated 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 I have subclassed it from `UnicastRemoteObject`.

This particular client is designed to drive a particular robot: the "RoverBot," described in the LEGO MINDSTORMS "Constructopedia" (the instruction manual that comes with each MINDSTORMS set), is pictured in Figure 17-2.

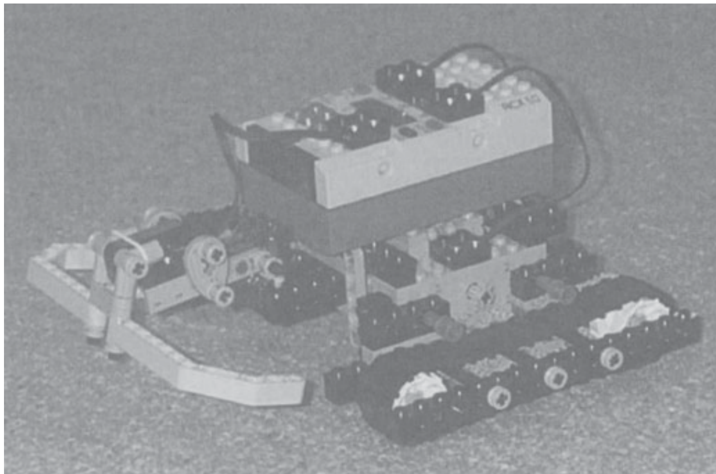


Figure 17-2. RoverBot MINDSTORMS robot

The RoverBot has motors to drive tracks or wheels on either side. The client can send instructions to make the RoverBot move forward or backward, 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.

Here is the client code:

```
package client;

import rcx.jini.*;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import java.rmi.RMISecurityManager;
```


libdocdel@fr.com

```
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
 */

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);
        }
```


libdocdel@fr.com

```
        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;
```


libdocdel@fr.com

```
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.ps"));
    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) {
```


libdocdel@fr.com

```
        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();
```


libdocdel@fr.com

```
        } else  if (name.equals("Back")) {
            backwards();
        } else  if (name.equals("Left")) {
            sendCommand("e1 84");
            sendCommand("21 84");
            sendCommand("21 41");
            try {
                Thread.sleep(100);
            } catch(InterruptedException e) {
            }
            restoreState();

        } else  if (name.equals("Right")) {
            sendCommand("e1 81");
            sendCommand("21 81");
            sendCommand("21 44");
            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 {
```


libdocdel@fr.com

```
                throw new UnknownEventException("Unknown message " + evt.getID());
            }
        }
    }

} // TestRCX
```

Why is this a simplistic *client*? It tries to find all robots on the local network, and creates a top-level window for each of them. If a robot has registered with, say, half-a-dozen service locators, and the client finds all of these, then it will create six top-level windows, one for each copy of the same robot. Some smarts are needed here, such as using the `ClientLookupManager` of Chapter 15.

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 RCXs by the infrared transmitter—then one RCX may have to decide whether it is the master, which it does by setting a local variable to "master" if it broadcasts before it receives, and the other RCXs will set the variable to "slave" if they receive before broadcasting. Then each waits for a random amount of time 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, because the source computers tend to assume that there is only one target at a time, and they can get confused by responses from multiple RCXs. The solution 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, and so on. Not very elegant, but it works.

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 RCXs in order to drive the correct ones. An `Entry` class for distinguishing them should contain information such as this:

- 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 identified. The RCX will have no knowledge of its identifier—it must be externally supplied.

libdocdel@fr.com

- The RCX can be driven by direct commands or by executing a program already downloaded (there may be up to five of these). An identifier for each downloaded program should be available.

- The RCX will have some sort of location, although it may move around to a limited extent. This location 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 to be resolved about how the information could be stored and accessed from an RCX. However, they stray beyond the bounds of this chapter.

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. Just as `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 that encapsulates client-side behavior may as well look as much as possible like the local `RCXPort` class. We define its (public) methods as follows:

```
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 possible constructors, all ending up calling a constructor that looks like this:

```
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 they match the search criteria. Therefore, this class maintains a list of RCXs and does not add to the list if it has already

libdocdel@fr.com

seen the RCX from another service locator. This requires that a single RCX should be registered using the same ServiceID with all locators, which will be the case because the RCX server uses JoinManager.

Higher-Level Mechanisms: Not Quite C

"Not Quite C" (nqc) is a language and a compiler from David Baum, designed for the RCX. It defines a language with C-like syntax that defines tasks that can be executed concurrently. The RCX API also defines a number of constants, functions, and macros targeted 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.

The following is a trivial nqc program to turn motor A on for 1 second (units are 1/100th of a second):

```
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 it is a typical example of a standalone, 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. Precompiled 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 OS and shared library configuration). It is software, not hardware like the MINDSTORMS robots, but it 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 and is shown in Figure 17-3.

libdocdel@fr.com

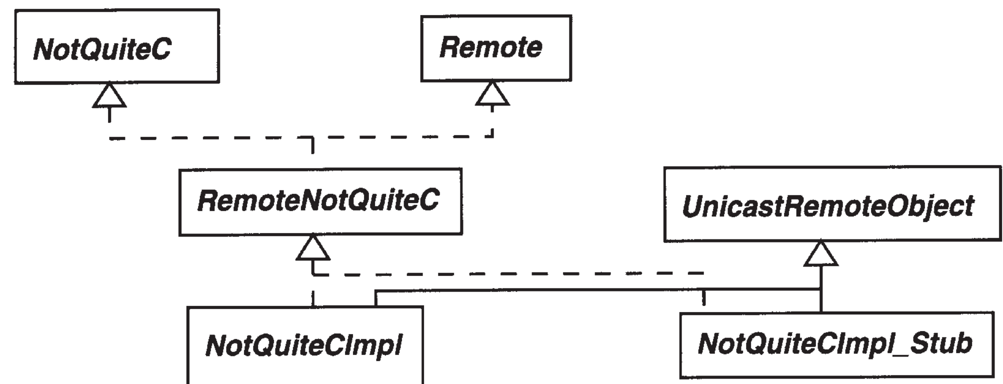


Figure 17-3. Class diagram for `nqc` *with RMI proxy*

The `NotQuiteC` and `RemoteNotQuiteC` interfaces are defined by

```
/**
 * NotQuiteC.java
 */

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 by

```
/**
 * RemoteNotQuiteC.java
 */

package rcx.jini;

import java.rmi.Remote;

public interface RemoteNotQuiteC extends NotQuiteC, Remote {
```


libdocdel@fr.com

```
} // RemoteNotQuiteC
```

The compile exception is thrown when things go wrong:

```
/**
 * CompileException.java
 */

package rcx.jini;

public class CompileException extends Exception {

    protected String error;

    public CompileException(String err) {
        error = err;
    }

    public String toString() {
        return error;
    }
} // CompileException
```

An implementation of the `RemoteNotQuiteC` interface 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 known as an exit code, which by convention is 0 for success and some other integer value 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 with `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—in this example, we will do it via an exception constructor.

libdocdel@fr.com

The hard part in this example is plowing 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 (the current version I have of nqc has a bug with pipelines) and the standard error stream for compile errors.

```
/**
 * NotQuiteCImpl.java
 */

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" +
```


libdocdel@fr.com

```
                                                  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 {
        // clean up 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;
```


libdocdel@fr.com

```
        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. 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.

Summary

This chapter has considered some of the issues involved in using 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 much work involved in making it a suitable Jini service, and these have been discussed. This work is not yet complete, and more remains to be done for LEGO MINDSTORMS.

libdocdel@fr.com

CHAPTER 18

CORBA and Jini

THERE ARE MANY DIFFERENT DISTRIBUTED SYSTEM ARCHITECTURES in addition to Jini. Many have only limited use, but some such as DCOM and CORBA are widely used, and there are many systems that have been built using these other distributed frameworks. This chapter looks at the similarities and differences between Jini and CORBA and shows how services built using one architecture can be used by another.

CORBA

Like Jini, CORBA is an infrastructure for distributed systems. CORBA was designed out of a different background than Jini, and there are some minor and major differences between the two.

- CORBA allows for specification of objects that can be distributed. The concentration is on distributed objects rather than on distributed services.
- CORBA is language-independent, using an Interface Definition Language (IDL) for specifying interfaces.
- CORBA objects can be implemented in a number of languages, including C, C++, SmallTalk, and Java
- 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 to 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 the declaration of attributes and operations (methods). The complete IDL specification can be found on the Object Management Group (OMG) Web site (`http://www.omg.org/`).

libdocdel@fr.com

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 MeetingFactory factory to produce them, 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, because the mapping from IDL to Java has changed slightly for elements nested within interfaces since that book was written. The following is the modified IDL for the room-booking service:

```
module corba {
module RoomBooking {

    interface Meeting {

        // A meeting has two read-only attributes that describe
        // 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 simplicity 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 MaxSlots constant is defined.

    const short MaxSlots = 8;
```


libdocdel@fr.com

```
    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);
    };
};
};
```

CORBA to Java Mapping

CORBA has bindings to a number of languages. That is, there is a translation from IDL to each language, and there is a runtime environment that supports objects written in these languages. A recent addition is Java, and this binding is still under active development (that is, the core is basically settled, but some parts are still

libdocdel@fr.com

changing). This binding must cover all elements of IDL. Here is a horribly brief summary of the CORBA translations:

- **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 that are "global" have no direct equivalent in Java, and so are mapped to Java interfaces with a single field that 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 methods that I write, I will use the more accepted naming forms.

Jini Proxies

A Jini service exports a proxy object that acts within the client on behalf of the service. On the service provider side, there may be service backend objects, completing the service implementation. The proxy may be fat or thin, depending on circumstances.

In Chapter 17 the proxy had to be thin: all it does is pass on requests to the service backend, which is linked to the hardware device, and the service cannot move, because it has to talk to a particular serial port. (The proxy may have an extensive user interface, but the Jini community seems to feel that any user

libdocdel@fr.com

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 backend which is implemented as remote objects.

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, but 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, and the object 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 talks a different protocol (hardware or software) will require a bridge between itself and Jini clients.

Most CORBA implementations use a protocol called IIOP (Internet Inter-ORB Protocol), which is based on TCP. 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 due to this confluence.

A Jini service has a lot of flexibility in implementation and can choose to place logic in the proxy, in the backend, or anywhere else for that matter. The combination of Jini flexibility and IIOP allows a larger variety of implementation possibilities than is possible with fixed pieces of hardware such as MINDSTORMS. Here are a couple of examples:

- The Jini proxy could invoke the CORBA naming service lookup to locate the CORBA service, and then make calls directly on the CORBA service from the client. This is a fat proxy model in which the proxy contains all of the service implementation. There is no need for a service backend, and the service provider just exports the service object as proxy and then keeps the leases for the lookup services alive.

- The Jini proxy could be an RMI stub, passing on all method calls to a backend service running as an RMI remote object in the service provider. This is a thin proxy with fat backend, where all service implementation is done on the backend. The backend uses the CORBA naming service lookup to find the CORBA service and then makes calls on this CORBA service from the backend.

libdocdel@fr.com

A Simple CORBA Example

The standard introductory example to any new system is "hello world." and it seems to get more complex with every advance that is made in computing technology! A CORBA version can be defined by the following IDL:

```
module corba {
    module HelloModule {
        interface Hello {
            string getHello();
        };
    };
};
```

This code can be compiled into Java using a compiler such as Sun's `idltojava` (or another CORBA 2.2 compliant compiler). This results in a `corba.HelloModule` package containing a number of classes and interfaces. `Hello` is an interface that is used by a CORBA client (in Java).

```
package corba.HelloModule;
public interface Hello
    extends org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity {
    String getHello();
}
```

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 will 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—this is the CORBA *backplane*, which acts as the runtime link between different objects in a CORBA system). The *servant* is the CORBA term for what we have been calling the "backend service" in Jini, and this object is created and run by the server. The server must also find a name server and register the name and the servant implementation. The servant implements the `Hello` interface. The server can just sleep to continue existence after registering the servant.

```
/**
 * CorbaHelloServer.java
```


libdocdel@fr.com

```
 */
package corba;

import corba.HelloModule.*;
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);
            HelloImpl hello = new HelloImpl();
            orb.connect(hello);

            // get the name server
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext namingContext = NamingContextHelper.narrow(objRef);

            // bind the Hello service to the name server
            NameComponent nameComponent = new NameComponent("Hello", "");
            NameComponent path[] = {nameComponent};
            namingContext.rebind(path, hello);

            // sleep
            java.lang.Object sleep = new java.lang.Object();
            synchronized(sleep) {
                sleep.wait();
            }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

} // CorbaHelloServer

class HelloImpl extends _HelloImplBase {
```


libdocdel@fr.com

```
    public String getHello() {
        return("hello world");
    }
}
```

CORBA Client in Java

A standalone client finds a proxy implementing the `Hello` interface with methods such as one that looks 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
 */
package corba;

import corba.HelloModule.*;
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);

            // get the name server
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext namingContext = NamingContextHelper.narrow(objRef);

             // get the Hello proxy
            NameComponent nameComponent = new NameComponent("Hello", "");
            NameComponent path[] = {nameComponent};
            org.omg.CORBA.Object obj = namingContext.resolve(path);
            Hello hello = HelloHelper.narrow(obj);
```


libdocdel@fr.com

```
            // now invoke methods on the CORBA proxy
            String hello = hello.getHello();
            System.out.println(hello);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

} // CorbaHelloClient
```

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 can 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. The Jini service implementation is moved from the Jini server to a Jini client as the service object. Once in the client, the service implementation is responsible for locating the CORBA service by using the CORBA naming service, and it then translates client calls on the Jini service directly into calls on the CORBA service. The processes that run in this, with their associated Jini and CORBA objects, are shown in Figure 18-1.

The Java interface for this service is quite simple and basically just copies the interface for the CORBA service:

```
/**
 * JiniHello.java
 */
package corba;

import java.io.Serializable;

public interface JiniHello extends Serializable {

    public String getHello();
} // JiniHello
```

The `getHello()` method 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 (in the "Room-Booking Example" section) will show a more

libdocdel@fr.com

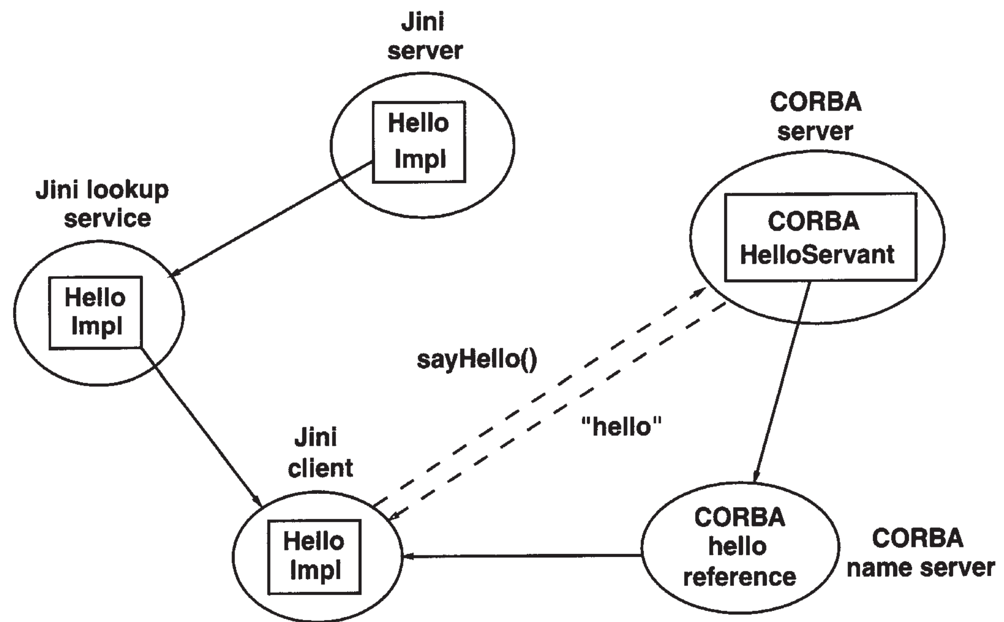


Figure 18-1. CORBA and Jini services

complex case where CORBA objects may be returned. Note that because this is 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 `getHello()` method will contact the CORBA naming service, find a reference to the CORBA `Hello` object, and call its `getHello()` method. The Jini service can just return the string it gets from the CORBA service.

```
/**
 * JiniHelloImpl.java
 */
package corba;

import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import corba.HelloModule.*;

public class JiniHelloImpl implements JiniHello {

    protected Hello hello = null;
    protected String[] argv;
```


libdocdel@fr.com

```
    public JiniHelloImpl(String[] argv) {
        this.argv = argv;
    }

    public String getHello() {

        if (hello == null) {
            hello = getHello();
        }
        // now invoke methods on the CORBA proxy
        String hello = hello.getHello();
        return hello;
    }

    protected Hello getHello() {
        ORB orb = null;
        // Act like a CORBA client
        try {
            orb = ORB.init(argv, null);

            // get the CORBA name server
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext namingContext = NamingContextHelper.narrow(objRef);

             // get the CORBA Hello proxy
            NameComponent nameComponent = new NameComponent("Hello", "");
            NameComponent path[] = {nameComponent};
            org.omg.CORBA.Object obj = namingContext.resolve(path);
            Hello hello = HelloHelper.narrow(obj);
            return hello;
        } catch(Exception e) {
            e.printStackTrace();
            return null;
        }
    }
} // JiniHelloImpl
```


libdocdel@fr.com

Jini Server and Client

The Jini server that exports the service doesn't contain anything new compared to the other service providers we have discussed. It creates a new `JiniHelloImpl` object and exports it using a `JoinManager`:

```
joinMgr = new JoinManager(new JiniHelloImpl(argv), ...)
```

Similarly, the Jini client doesn't contain anything new, except that it catches CORBA exceptions. After lookup discovery, the code is as follows:

```
try {
    hello = (JiniHello) 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.getHello();
    System.out.println(msg);
} catch(Exception e) {
   // a CORBA runtime error may occur
    System.err.println(e.toString());
}
```

Building the Simple CORBA Example

Compared to the Jini-only examples that have been looked at so far, the major additional step in this CORBA example 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 `corba.HelloModule` package. The files that are produced are standard Java files, and they 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 they can be compiled like earlier Jini files, with the `CLASSPATH` set to include the Jini libraries.

libdocdel@fr.com

Running the Simple CORBA Example

There are a large number of elements and processes that must be set running to get this example working 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 it can be started during boot time. An ordinary user will need to use the option `-ORBInitialPort` port to run it on a port above 1024:

   ```
   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 with this command:

   ```
   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 with this command:

   ```
   java corba.JiniHelloServer -ORBInitialPort 1055
   ```

5. Finally, the Jini client can be run with this command:

   ```
   java client.TestCorbaHello -ORBInitialPort 1055
   ```

CORBA Implementations

There are interesting considerations about what is needed in Java to support CORBA. The example discussed previously uses the CORBA APIs that are part of the standard OMG binding of CORBA to Java. The packages rooted in `org.omg` are in major distributions of JDK 1.2, such as the Sun SDK. This example should compile properly with most Java 1.2 compilers using these OMG classes.

Sun's JDK 1.2 runtime includes a CORBA ORB, and the example will run as is, using this ORB. However, there are many implementations of CORBA ORBs, and they do not always behave in quite the same way. This can affect compilation and

libdocdel@fr.com

runtime results. Which CORBA ORB is used is determined at runtime, based on properties. If a particular ORB is not specified, then it defaults to the Sun-supplied ORB (using Sun's SDK). To use another ORB, such as the Orbacus ORB, the following code needs to be inserted before the call to `ORB.init()`:

```
java.util.Properties props = System.getProperties();
props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
props.put("org.omg.CORBA.ORBSingletonClass",
          "com.ooc.CORBA.ORBSingleton");
System.setProperties(props);
```

Similar code is required for the ORBS from IONA and other vendors.

Variations in CORBA implementations could affect the runtime behavior of the client: if the proxy expects to use a particular ORB other than the default, then the class files for that ORB must be available to the client or be downloadable across the network. Alternatively, the proxy could be written to use the default Sun ORB, and then may need to make inter-ORB calls between the Sun ORB and the actual ORB used by the CORBA service. Such issues take us beyond the scope of this chapter, though. Vendor documentation for each CORBA implementation should give more information on any additional requirements.

Room-Booking Example

The IDL for a room-booking problem was briefly discussed in the introductory "CORBA" section in this chapter. This room-booking example has a few more complexities than the previous example. The problem here is to have a set of rooms, and for each room have a set of bookings that can be made for that room. The bookings may be made on the hour, from 9 a.m. until 4 p.m. (this only covers the bookings for one day). Bookings may be cancelled after they are made. 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 non-null including the details of the participants and the purpose of the meeting.

There are other things to consider in this example:

- Each room is implemented as a separate CORBA object. There is also a "meeting factory" that produces more objects. This is a system with multiple CORBA objects residing on many CORBA servers. There are several possibilities for implementing a system with multiple objects.

- Some of the methods return CORBA objects, and 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.

libdocdel@fr.com

- Some of the methods throw user-defined exceptions, in addition to CORBA-defined exceptions. Both of these need to be handled appropriately.

CORBA Objects

CORBA defines a set of "primitive" types in the IDL, such as integers of various sizes, chars, etc. The language bindings specify the primitive types in each language that they are converted into. For example, the CORBA wide character (`wchar`) becomes a Java Unicode `char`. Things are different for non-primitive objects, which depend on the target language. For example, an IDL *object* turns into a Java *interface*.

The room-booking IDL defines CORBA interfaces for `Meeting`, `MeetingFactory`, and `Room`. These can be implemented 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 objects will essentially be references to remote CORBA objects. Two things are certain about these references:

- CORBA interfaces generate Java interfaces, such as `Hello`. These inherit from `org.omg.CORBA.portable.IDLEntity`, which implements `Serializable`. As a result, 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. Therefore, CORBA references cannot be usefully moved around. At present, the best way to move them around is to convert them to "stringified" form and move that around, though 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, whereas the Java method serializes the object's instance data.

- The references 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 following Java interface:

```
/*
 * File: ./corba/RoomBooking/Meeting.java
 * From: RoomBooking.idl
```


libdocdel@fr.com

```
 * 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 like this:

```
/**
 * JavaMeeting.java
 */
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 in the preceding interface allow information about a meeting to be accessible to applications that do not want to contact the CORBA service. The third allows a CORBA object reference to be reconstructed within a new ORB. A suitable implementation is as follows:

```
/**
 * JavaMeetingImpl.java
 */
package corba.RoomBookingImpl;

import corba.RoomBooking.*;
import org.omg.CORBA.*;
```


libdocdel@fr.com

```
import corba.common.*;

/**
 * A portable Java object representing a CORBA object.
 */
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);
    }
```


libdocdel@fr.com

```
} // JavaMeetingImpl
```

Multiple Objects

The implementation of the room-booking problem in the Vogel and Duddy book (*Java Programming with CORBA*, `http://www.wiley.com/compbooks/vogel`) runs each room as a separate CORBA object, each with its own server. A meeting factory creates meeting objects that 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:

1. A Jini server may exist for each CORBA server.
    - Each Jini server may export fat proxies, which build CORBA references in the same Jini client.
    - 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 all the CORBA objects.
    - The single Jini server exports a fat proxy, which builds CORBA references in the Jini client.
    - The single Jini server exports a thin proxy, with all CORBA references within this single server.

The first of these pairs of options essentially isolates each CORBA service into its own Jini service. This may be appropriate in an open-ended system where there may be a large set of CORBA services, only some of which are needed by any application.

The second pair of options deals with the case where services come logically grouped together, such that one cannot exist without the other, even though they may be distributed geographically.

Intermediate schemes exist, where some CORBA services have their own Jini service, while others are grouped into a single Jini service. For example, rooms may be grouped into buildings and cannot exist without these buildings, whereas a client may only want to know about a subset of buildings, say those in New York.

libdocdel@fr.com

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 from either Jini or CORBA for this to be so. On the other hand, if a client is running as an applet, then applet security restrictions may force all the Jini servers to run 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 the case 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. This situation is shown in Figure 18-2.

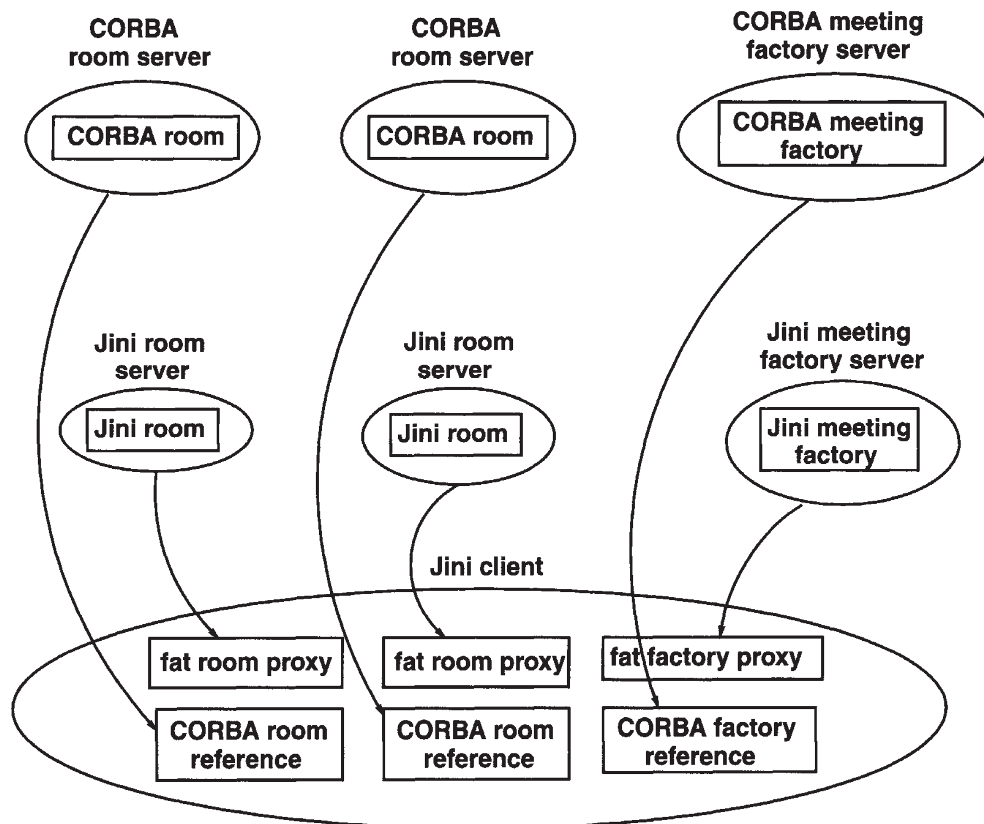


Figure 18-2. CORBA and Jini services for fat proxies

libdocdel@fr.com

The CORBA servers are all accessed from within the Jini client. This arrangement may be ruled out if the client is an applet and the servers are on different machines.

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. This situation is shown in Figure 18-3.

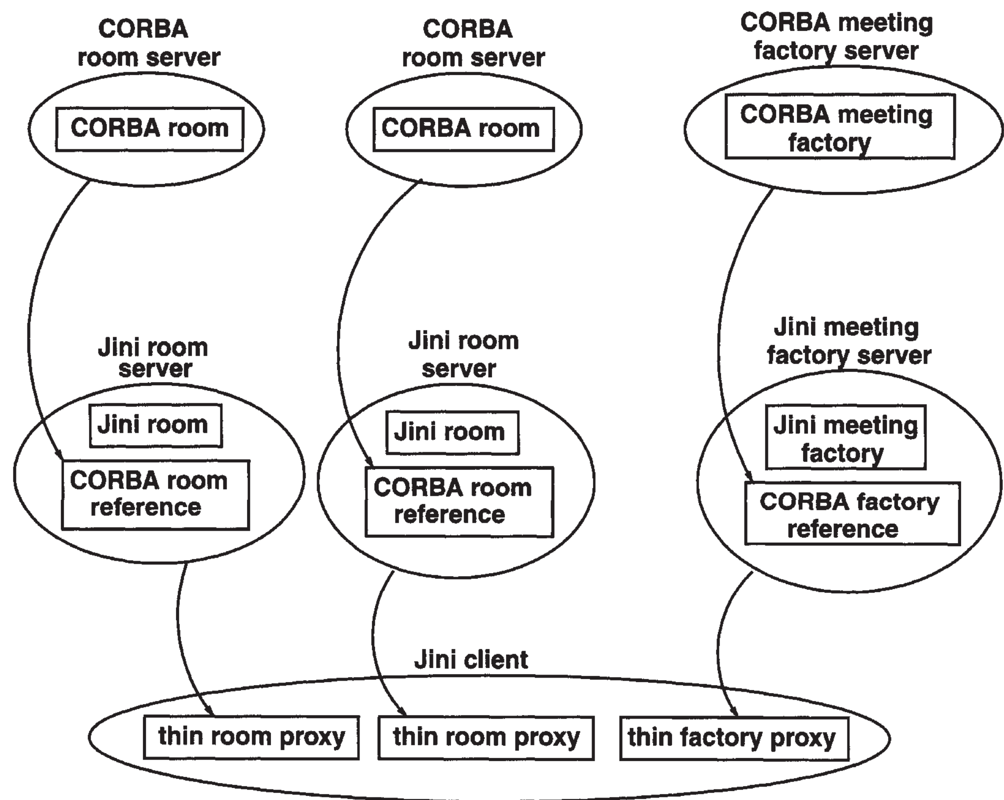


Figure 18-3. CORBA and Jini services for thin proxies

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 previously. The Jini client will also need to get information about the CORBA objects, which can be gained from these portable Jini objects.

libdocdel@fr.com

Single Fat Proxy

An alternative to Jini servers for each CORBA server is to have a single Jini bridge 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. Then you have 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 is shown in Figure 18-4.

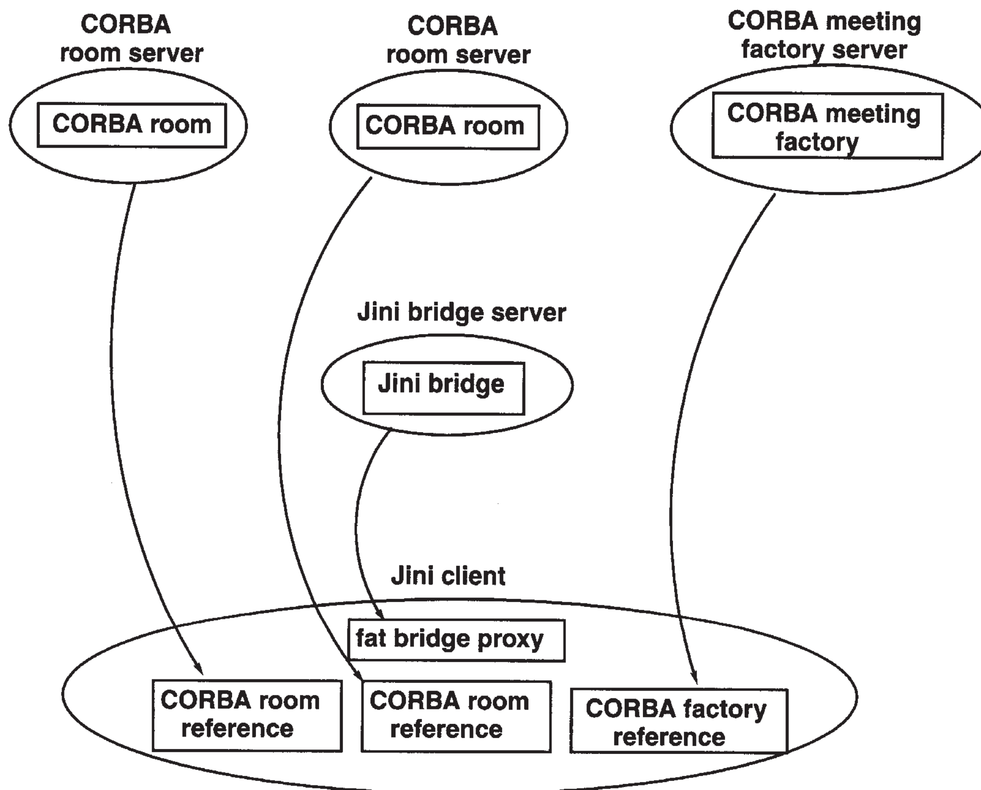


Figure 18-4. CORBA and Jini services for single fat proxy

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 scenario is shown in Figure 18-5. This is probably the best option from a security viewpoint of a Jini client.

libdocdel@fr.com

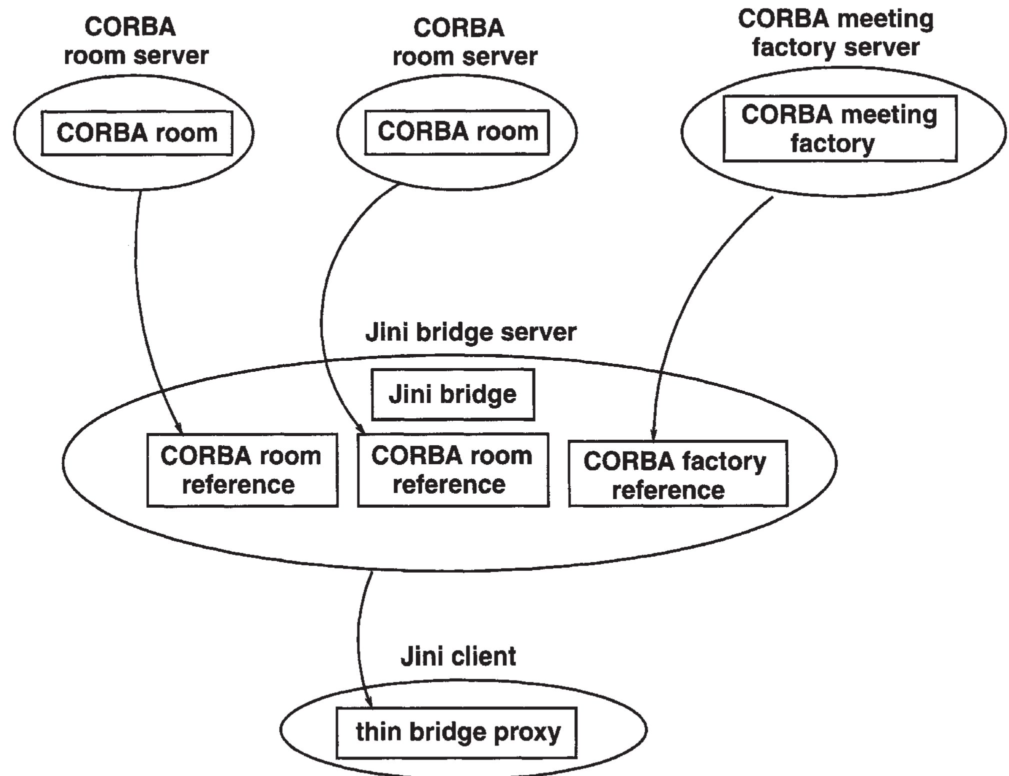


Figure 18-5. CORBA and Jini services for single thin proxy

Exceptions

CORBA methods can throw exceptions of two types: system exceptions and user exceptions. System exceptions subclass from `RuntimeException` and so are unchecked. They 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 will 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

libdocdel@fr.com

remote issues. User exceptions should be treated as they are, without wrapping them in `Remote` exceptions.

Interfaces for Single Thin Proxy

This and the following sections build 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 will be made available to Jini clients only by means of portable Jini versions of the CORBA objects. These portable objects are defined by two interfaces, the `JavaRoom` interface

```
/**
 * JavaRoom.java
 */
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 the `JavaMeeting` interface

```
/**
 * JavaMeeting.java
 */
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();
```


libdocdel@fr.com

```
    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 for obtaining 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, getting a list of rooms, and getting a list of bookings for a room.

```
/**
 * RoomBookingBridge.java
 */

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 interface that comes from the original "mono-block" CORBA client by Vogel and Duddy. In that client, 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 identified simply by their index in the appropriate array. In splitting the client apart into multiple (and remote) classes, this is not really a good idea anymore because it assumes a commonality of implementation across objects, which may not occur. It doesn't seem worthwhile being too fussy about that here, though.

libdocdel@fr.com

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 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
 */
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 Remote-
RoomBookingBridge {

    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 {
            // initialize the ORB
            orb = ORB.init(args, null);
```


libdocdel@fr.com

```
        }
        catch(SystemException system_exception ) {
            throw new RemoteException("constructor RoomBookingBridge: ",
                                      system_exception);
        }
        init_from_ns();
        update();
    }

    public void init_from_ns()
        throws RemoteException, UserException {

        // initialize from Naming Service
        try {
            // get room context
            String str_name = "/BuildingApplications/Rooms/";
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext namingContext = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent(str_name,  " ");
            NameComponent path[] = {nc};

            org.omg.CORBA.Object roomRef = namingContext.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(namingContext.resolve(path));
            if( meeting_factory == null ) {
                System.err.println(
                    "No Meeting Factory registered at Naming Service" );
                System.err.println( "exiting ..." );
                System.exit( 1 );
            }
        }
        catch(SystemException system_exception ) {
```


libdocdel@fr.com

```
            throw new RemoteException("Initialize ORB", system_exception);
        }
    }

    public void update()
        throws RemoteException, UserException {

        try {
            // list rooms
            // initialize 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 );
```


libdocdel@fr.com

```
            // be friendly 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
     */
```


libdocdel@fr.com

```
    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
```

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 in their book with a later one with the slightly more complex standard mechanism for creating contexts and placing names within this context. This mechanism can use the `tnameserv` CORBA naming server, for example.

I have modified the Vogel and Duddy room-booking client a little bit, but its essential structure remains unchanged. The GUI elements, for example, were not altered. All CORBA-related code was removed from the client and placed into the bridge classes.

The Vogel and Duddy code samples can all be downloaded from a public Web site (`http://www.wiley.com/compbooks/vogel`) 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 that can be found at `http://www.apress.com`.

libdocdel@fr.com

Building the Room-Booking Example

The `RoomBooking.idl` IDL interface needs to be compiled to Java by a suitable IDL-to-Java compiler, such as Sun's `idltojava`. This produces classes in the `corba.RoomBooking` package. These can then 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.

Running the Room-Booking Example

There are a large number of elements and processes that must be set running to get this example working satisfactorily:

1. A CORBA name server must be set running, as in the earlier example. For example, you could use the following command:

   ```
   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:

   ```
   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 with this command:

   ```
   java corba.RoomBookingImpl.RoomBookingBridgeServer -ORBInitialPort 1055
   ```

6. Finally, the Jini client can be run with this command:

   ```
   java corba.RoomBookingImpl.RoomBookingClientApplication -ORBInitialPort
   1055
   ```


libdocdel@fr.com

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 in a series of steps to Jini clients of a Jini service acting as a front-end to CORBA objects. 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 client, mixing GUI elements, CORBA calls, and glue to hold it all together. This had a number of objects playing different roles all together, without a clear distinction about roles in some cases. 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 (okay, 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: 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 interfaces such as `JavaRoom` and implementations such as `JavaRoomImpl`. The GUI code in the client had no need to directly modify fields in these objects, so they ended up as read-only 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 objects 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.

libdocdel@fr.com

6. Finally, the `RoomBookingBridgeImpl` was turned into a `UnicastRemoteObject` and placed into a Jini server. The client was changed to look up 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 could be exported as part of the service.

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. Therefore, a Jini service interface can be written as a CORBA interface. A Jini client could then be written as the implementation of a CORBA server to this IDL.

At present, with a paucity of Jini services, it does not seem worthwhile to explore this in detail. This may change in the future, though.

Summary

CORBA is a separate distributed system from Jini. However, it is quite straightforward to build bridges between the two systems, and there are a number of different possible architectures. This makes it possible for CORBA services to be used by Jini clients.

libdocdel@fr.com

CHAPTER 19

User Interfaces for Jini Services

SOME EARLIER CHAPTERS HAVE USED CLIENTS with graphical user interfaces to services. Clients may not always know which is the most appropriate user interface, and sometimes may not even know of any suitable user interface. Services should be able to define their own user interfaces, and the question of how they should best do this is explored in this chapter. We'll also look at how clients can discover, download, and use these user interfaces.

User Interfaces as Entries

Interaction with a service is specified by its interface, and the interaction will be the same across all implementations of the interface. This consistency doesn't allow any flexibility in using the service, since a client will only know about the methods defined in the interface. The interface is the defining level for using this type of service.

However, services can be implemented in many different ways, and service implementations do in fact differ. For example, one service may be offered on a "take it or leave it" basis, while another might have a warranty attached. This does not affect how the client calls a service, but it may affect whether or not the client wants to use one service implementation or another. There is a need to allow for this, and 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 type interface and these additional entry items primarily in the selection of a service. But once clients have the service, are they just constrained to use it via the type interface? The type interface is designed to allow a client application to use the service in a programmatic way by calling methods. However, many services could probably benefit from some sort of user interface (UI). 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 option by supplying a user-interface object with a special component for the number of copies.

libdocdel@fr.com

> **NOTE** *In this chapter I talk about interfaces we have been using throughout the book, and also about user interfaces. To avoid possible confusion, in this chapter I will use the term "type interface" to refer to a Java interface as used in the rest of this book, and "user interface" for any sort of interaction with the user.*

A client can only be expected to know about the type interface of a service. If it uses this to build a user interface, then at best it could only manage a fairly generic one that will work for all service implementations. A vendor will know much more detail about any particular implementation of a service, and so the vendor is best placed to supply the user interface. In some cases, the service vendor may be unwilling or incapable of supplying user interfaces for a service, and a third party may supply it.

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 are not just restricted to providing static data; as Java objects, they are perfectly capable of running as user-interface objects.

User interfaces are not yet part of the Jini standard, but the Jini community (with a semi-formal organization as the "Jini Community") is moving toward a standard way of specifying many things, including user-interface standards and guidelines. Guideline number one from the serviceUI group is this: user interfaces for a service should be given in `Entry` objects.

User Interfaces from Factory Objects

In Chapter 13, some discussion was given to the location of code, using user-interface components as examples. That chapter suggested that user interfaces should not be created on the server side but on the client side—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 approach:

- 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).

- There may be many potential user interfaces for any particular service: Palm handhelds (many with small grayscale screens) require a different interface than a high-end workstation with a huge screen and enormous numbers of colors. It is not reasonable to expect the service to create every one of these user interfaces, but it could export factories capable of doing so.

libdocdel@fr.com

- Localization of internationalized services cannot be done on the service side, only on the client side.

The service should export zero or more user-interface factories, with methods to create the interface, such as `getJFrame()`. The service and its user-interface factory 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 beforehand; if the factory was given one during its construction (on the service side), the factory would end up with a service-side copy of the service instead of a client-side copy. Therefore, when the factory is asked for a user-interface (on the client side), it should be passed the service. In fact, the factory should probably be passed all of the information about the service, as retrieved in the `ServiceItem` from a lookup service.

A typical factory is the one that returns a `JFrame`. This is defined by the type interface as follows:

```
package net.jini.lookup.ui.factory;

import javax.swing.JFrame;

public interface JFrameFactory {
    String TOOLKIT = "javax.swing";
    String TYPE_NAME = "net.jini.lookup.ui.factory.JFrameFactory";

    JFrame getJFrame(Object roleObject);
}
```

The factory imports the minimum number of classes needed to compile the type interface. The `JFrameFactory` above needs to import `javax.swing.JFrame` because the `getJFrame()` method returns a `JFrame`. An implementation of this type interface will probably use many more classes. The `roleObject` passes any necessary information to the UI constructor. This is usually the `ServiceItem`, as it contains all the information (including the service) that was retrieved from a lookup service. The factory can then create an object that acts as a user interface to the service, and can use any additional information in the `ServiceItem`, such as entries for `ServiceInfo` or `ServiceType`, which could be shown, say, in an "About" box.

A factory that returns a visual component, such as a `JFrame`, should not make the component visible. This will allow the client to set the `JFrame`'s size and placement before showing it. Similarly, a "playable" user interface, such as an audio file, should not be in a "playing" state.

libdocdel@fr.com

Current Factories

A service may supply lots of these user interface factories, each capable of creating a different user interface object. This allows for the differing capabilities of viewing devices, or even for different user preferences. One user may always like a Web-style interface, another may be content with an AWT interface, a third may want the accessibility mechanisms possible with a Swing interface, and so on.

The set of proposed factories currently includes the following:

- `DialogFactory`, which returns an instance of `java.awt.Dialog` (or one of its subclasses)
- `FrameFactory`, which returns an instance of `java.awt.Frame` (or one of its subclasses)
- `JComponentFactory`, which returns an instance of `javax.swing.JComponent` (or one of its subclasses, such as a `JList`)
- `JDialogFactory`, which returns an instance of `javax.swing.JDialog` (or one of its subclasses)
- `JFrameFactory`, which returns an instance of `javax.swing.JFrame` (or one of its subclasses)
- `PanelFactory`, which returns an instance of `java.awt.Panel` (or one of its subclasses)

These factories are all defined as interfaces. An implementation will define a `getXXX()` method that will return a user interface object. The current set of factories returns objects that belong to the Swing or AWT classes. Factories added in later iterations of the specification may return objects belonging to other user interface styles, such as speech objects. Although an interface may specify that a method, such as `getJFrame()`, will return a `JFrame`, an implementation will in fact return a subclass of this, which also implements a role interface.

Marshalling Factories

There may be many factories for a service, and each of them will generate a different user interface. These factories and their user interfaces will be different for each service. The standard factory interfaces will probably be known to both clients and services, but the actual implementations of these will only be known to services (or maybe to third-party vendors who add a user interface to a service).

libdocdel@fr.com

If a client receives a `ServiceItem` containing entries with many factory implementation objects, it will need to download the class files for all of these as it instantiates the `Entry` objects. There is a strong chance that each factory will be bundled into a `jar` file that also contains the user interface objects themselves, so if the entries directly contain the factories, then the client will need to download a set of class files before it even goes about the business of deciding which of the possible user interfaces it wants to select.

This downloading may take time on a slow connection, such as a wireless or home network link. It may also cost memory, which may be scarce in small devices such as PDAs. Therefore, it is advantageous to hide the actual factory classes until the client has decided that it does in fact want a particular class. Then, of course, it will have to download all of the class files needed by that factory.

In order to hide the factories, they are wrapped in a `MarshalledObject`. This keeps a representation of the factory and also a reference to its codebase, so that when it is unwrapped, the necessary classes can be located and downloaded. Clients should have the class files for `MarshalledObject`, because this class is part of the Java core. By putting a factory object into entries in this form, no attempt is made to download the actual classes required by the factory until it is unmarshalled.

The decision as to whether or not to unmarshall a class can be made on a separate piece of information, such as a set of `Strings` that hold the names of the factory class (and all of its superclasses and interfaces). This level of indirection is a bit of a nuisance, but not too bad:

```
if (typeNames.contains("net.jini.lookup.ui.factory.JFrameFactory") {
    factory = (JFrameFactory) marshalledObject.get();
    ....
}
```

A client that does not want to use a `JFrameFactory` will just not perform the preceding Boolean test. It will not call the unmarshalling `get()` method and will not attempt the coercion to `JFrameFactory`. This will avoid downloading classes that are not wanted. This indirection does place a responsibility on service-side programmers to ensure that the coercion will be correct. In effect, this is a maneuver to circumvent the type-safe model of Java purely for optimization purposes.

There is one final wrinkle when loading the class files for a factory: a running JVM may have many class loaders. When loading the files for a factory, you want to make sure that the class loader is one that will actually download the class files across the network as required. The class loader associated with the service itself will be the most appropriate loader for this.

libdocdel@fr.com

UIDescriptor

An entry for a factory must contain the factory, itself, hidden in a MarshalledObject and some string representation of the factory's class(es). It may also need other descriptive information about the factory. The UIDescriptor captures all this:

```
package net.jini.lookup.entry;

public class UIDescriptor extends AbstractEntry {

    public String role;
    public String toolkit;
    public Set attributes;
    public MarshalledObject factory;

    public UIDescriptor();
    public UIDescriptor(String role, String toolkit,
                        Set attributes, MarshalledObject factory);

    public final Object getUIFactory(ClassLoader parentLoader)
          throws IOException, ClassNotFoundException;
}
```

There are several features in the UIDescriptor that we haven't mentioned yet, and the factory type appears to be missing (it is one of the attributes).

Toolkit

A user interface will typically require a particular package to be present or it will not function. For example, a factory that creates a JFrame will require the javax.swing package. These requirements can provide a quick filter for whether or not to accept a factory—if it is based on a package the client doesn't have, then it can just reject this factory.

This isn't a bulletproof means of selection. For example, the Java Media Framework is a fixed-size package designed to handle lots of different media types, so if your user interface is a QuickTime movie, you might specify the JMF package. However, the media types handled by the JMF package are not fixed, and they can depend on native code libraries. For example, the current Solaris version of the JMF package has a native code library to handle MPEG movies, which is not present in the Linux version. Having the package specified by the toolkit does not guarantee that the class files for this user interface will be present. It is primarily intended to narrow lookups based on the UIs offered.

libdocdel@fr.com

Role

There are many possible roles for a user interface. For example, a typical user may be using the service, in which case the UI plays the "main" role. Alternatively, a system administrator may be managing the service, and he or she might require a different user interface, in which case the UI then plays the "admin" role.

The `role` field in a `UIDescriptor` is intended to describe these possible variations in the use of a user interface. The value of this field is a string, and to reduce the possibility of spelling errors that are not discovered until runtime, the value should be one of several constant string values. These string constants are defined in a set of type interfaces known as *role* interfaces. There are currently three role interfaces:

- The `net.jini.lookup.ui.MainUI` role is for the standard user interface used by ordinary clients of the service:

```
package net.jini.lookup.ui;
public interface MainUI {
    String ROLE = "net.jini.lookup.ui.MainUI";
}
```

- The `net.jini.lookup.ui.AdminUI` role is for use by the service's administrator:

```
package net.jini.lookup.ui;
public interface AdminUI {
    String ROLE = "net.jini.lookup.ui.AdminUI";
}
```

- The `net.jini.lookup.ui.AboutUI` role is for information about the service, which can be presented by a user interface object:

```
package net.jini.lookup.ui;
public interface AboutUI {
    String ROLE = "net.jini.lookup.ui.AboutUI";
}
```

A service will specify a role for each of the user interfaces it supplies. This role is given in a number of ways for different objects:

- The `role` field in the `UIDescriptor` must be set to the `String ROLE` of one of these three role interfaces.

libdocdel@fr.com

- The user interface indicates that it acts a role by implementing the particular role specified.

- The factory does not explicitly know about the role, but the factory contained in a `UIDescriptor` must produce a user interface implementing the role.

The service must ensure that the `UIDescriptors` it produces follows these rules. How it actually does so is not specified. There are several possibilities, including these:

- When a factory is created, the role is passed in through a constructor. It can then use this role to cast the `roleObject` in the `getXXX()` method to the expected class (currently this is always a `ServiceItem`).

- There could be different factories for different roles, and the `UIDescriptor` should have the right factory for that role.

The factory could perform some sanity checking if desired; since all `roleObjects` are (presently) the service items, it could search through these items for the `UIDescriptor`, and then check that its `role` matches what the factory expects.

There has been much discussion about "flavors" of roles, such as an "expert" role or a "learner" role. This has been deferred because it is too complicated, at least for the first version of the specification.

Attributes

The `attributes` section of a `UIDescriptor` can carry any other information about the user interface object that the service thinks might be useful to clients trying to decide which user interface to choose. Currently this includes the following:

- A `UIFactoryTypes` object, which contains a set of `Strings` for the fully qualified class names of the factory that this entry contains. The current factory hierarchy is very shallow, so this may be just a singleton set, like this:

  ```
  Set attribs = new HashSet();
  Set typeNames = new HashSet();
  typeNames.add(JFrameFactory.TYPE_NAME);
  attribs.add(new UIFactoryTypes(typeNames));
  ```

  Note that a client is not usually interested in the actual type of the factory, but rather in the interface it implements. This is just like Jini services themselves, where we only need to know the methods that can be called and are not concerned with the implementation details.

libdocdel@fr.com

- An `AccessibleUI` object. Inclusion of this object indicates that the user interface implements `javax.accessibility.Accessible` and that the user interface would work well with assistive technologies.

- A `Locales` object, which specifies the locales supported by the user interface.

- A `RequiredPackages` object, which contains information about all of the packages that the user interface needs to run. This is not a guarantee that the user interface will actually run, nor a guarantee that it will be a usable interface, but it may help a client decide whether or not to use a particular user interface.

File Classifier UI Example

The file classifier has been used throughout this book as a simple example of a service to illustrate various features of Jini. We can use it here too, by supplying simple user interfaces to the service. Such a user interface would consist of a text field for entering a filename, and a display to show the MIME type of the filename. There is only a "main" role for this service, as no administration needs to be performed.

Figure 19-1 shows what a user interface for a file classifier could look like.

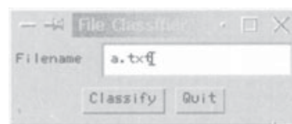


Figure 19-1. FileClassifier user interface

After the service has been invoked, it could pop up a dialog box, as shown in Figure 19-2.

Figure 19-2. FileClassifier return dialog box

libdocdel@fr.com

A factory for the "main" role that will produce an AWT `Frame` is shown next:

```
/**
 * FileClassifierFrameFactory.java
 */

package ui;

import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.lookup.entry.UIDescriptor;
import java.awt.Frame;
import net.jini.core.entry.Entry;
import net.jini.core.lookup.ServiceItem;

public class FileClassifierFrameFactory implements FrameFactory {

    /**
     * Return a new FileClassifierFrame that implements the
     * MainUI role
     */
    public Frame getFrame(Object roleObject) {
        // we should check to see what role we have to return
        if (! (roleObject instanceof ServiceItem)) {
            // unknown role type object
            // can we return null?
            return null;
        }
        ServiceItem item = (ServiceItem) roleObject;

        // Do sanity checking that the UIDescriptor has a MainUI role
        Entry[] entries = item.attributeSets;
        for (int n = 0; n < entries.length; n++) {
            if (entries[n] instanceof UIDescriptor) {
                UIDescriptor desc = (UIDescriptor) entries[n];
                if (desc.role.equals(net.jini.lookup.ui.MainUI.ROLE)) {
                    // Ok, we are in the MainUI role, so return a UI for that
                    Frame frame = new FileClassifierFrame(item, "File Classifier");
                    return frame;
                }
            }
        }
        // couldn't find a role the factory can create
        return null;
    }
```


libdocdel@fr.com

```
} // FileClassifierFrameFactory
```

The user interface object that performs this role is as follows:

```
/**
 * FileClassifierFrame.java
 */

package ui;

import java.awt.*;
import java.awt.event.*;
import net.jini.lookup.ui.MainUI;
import net.jini.core.lookup.ServiceItem;
import common.MIMEType;
import common.FileClassifier;
import java.rmi.RemoteException;

/**
 * Object implementing MainUI for FileClassifier.
 */
public class FileClassifierFrame extends Frame implements MainUI {

    ServiceItem item;
    TextField text;

    public FileClassifierFrame(ServiceItem item, String name) {
        super(name);

        Panel top = new Panel();
        Panel bottom = new Panel();
        add(top, BorderLayout.CENTER);
        add(bottom, BorderLayout.SOUTH);

        top.setLayout(new BorderLayout());
        top.add(new Label("Filename"), BorderLayout.WEST);
        text = new TextField(20);
        top.add(text, BorderLayout.CENTER);

        bottom.setLayout(new FlowLayout());
        Button classify = new Button("Classify");
        Button quit = new Button("Quit");
        bottom.add(classify);
        bottom.add(quit);
```


libdocdel@fr.com

```
        // listeners
        quit.addActionListener(new QuitListener());
        classify.addActionListener(new ClassifyListener());

        // We pack, but don't make it visible
        pack();
    }

    class QuitListener implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            System.exit(0);
        }
    }

    class ClassifyListener implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            String fileName = text.getText();
            final Dialog dlg = new Dialog((Frame) text.getParent().getParent());
            dlg.setLayout(new BorderLayout());
            TextArea response = new TextArea(3, 20);

            // invoke service
            FileClassifier classifier = (FileClassifier) item.service;
            MIMEType type = null;
            try {
                type = classifier.getMIMEType(fileName);
                if (type == null) {
                    response.setText("The type of file " + fileName +
                                     " is unknown");
                } else {
                    response.setText("The type of file " + fileName +
                                     " is " + type.toString());
                }
            } catch(RemoteException e) {
                response.setText(e.toString());
            }

            Button ok = new Button("ok");
            ok.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    dlg.setVisible(false);
                }
            });
```


libdocdel@fr.com

```
            dlg.add(response, BorderLayout.CENTER);
            dlg.add(ok, BorderLayout.SOUTH);
            dlg.setSize(300, 100);
            dlg.setVisible(true);
        }
    }

} // FileClassifierFrame
```

The server that delivers both the service and the user interface has to prepare a `UIDescriptor`. In this case, it only creates one such object for a single user interface, but if the server exported more interfaces, it would simply create more descriptors. Here is the server code:

```
/**
 * FileClassifierServer.java
 */

package ui;

import complete.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;
import net.jini.core.entry.Entry;

import net.jini.lookup.ui.MainUI;
import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.lookup.entry.UIDescriptor;
import net.jini.lookup.ui.attribute.UIFactoryTypes;

import java.rmi.MarshalledObject;
import java.io.IOException;
import java.util.Set;
import java.util.HashSet;
```


libdocdel@fr.com

```
public class FileClassifierServer
    implements ServiceIDListener {

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

        // stay around forever
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(InterruptedException e) {
                // do nothing
            }
        }
    }

    public FileClassifierServer() {

        JoinManager joinMgr = null;

        // The typenames for the factory
        Set typeNames = new HashSet();
        typeNames.add(FrameFactory.TYPE_NAME);

        // The attributes set
        Set attribs = new HashSet();
        attribs.add(new UIFactoryTypes(typeNames));

        // The factory
        MarshalledObject factory = null;
        try {
            factory = new MarshalledObject(new FileClassifierFrameFactory());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(2);
        }
        UIDescriptor desc = new UIDescriptor(MainUI.ROLE,
                                             FileClassifierFrameFactory.TOOLKIT,
                                             attribs,
                                             factory);

        Entry[] entries = {desc};
```


libdocdel@fr.com

```
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            joinMgr = new JoinManager(new FileClassifierImpl(), /* service */
                                      entries /* 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
```

Finally, a client needs to look for and use this user interface. The client finds a service as usual and then does a search through the `Entry` objects, looking for a `UIDescriptor`. Once it has a descriptor, it can check whether the descriptor meets the requirements of the client. Here we shall check whether it plays a `MainUI` role and can generate an AWT `Frame`:

```
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;
import net.jini.core.entry.Entry;
```


libdocdel@fr.com

```
import net.jini.lookup.ui.MainUI;
import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.lookup.entry.UIDescriptor;
import net.jini.lookup.ui.attribute.UIFactoryTypes;

import java.awt.*;
import javax.swing.*;

import java.util.Set;
import java.util.Iterator;
import java.net.URL;

/**
 * TestFrameUI.java
 */

public class TestFrameUI {

    private static final long WAITFOR = 100000L;

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

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(2*WAITFOR);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestFrameUI() {
        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());
```


libdocdel@fr.com

```
    } catch(Exception e) {
        e.printStackTrace();
        System.exit(1);
    }

    Class [] classes = new Class[] {FileClassifier.class};
    UIDescriptor desc = new UIDescriptor(MainUI.ROLE,
                                         FrameFactory.TOOLKIT,
                                         null, null);
    Entry [] entries = {desc};

    ServiceTemplate template = new ServiceTemplate(null, classes,
                                                   entries);

    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);
    }

    // We now have a service that plays the MainUI role and
    // uses the FrameFactory toolkit of "java.awt".
    // We now have to find if there is a UIDescriptor
    // with a Factory generating an AWT Frame
    checkUI(item);
}

private void checkUI(ServiceItem item) {
    // Find and check the UIDescriptors
    Entry[] attributes = item.attributeSets;
    for (int m = 0; m < attributes.length; m++) {
        Entry attr = attributes[m];
        if (attr instanceof UIDescriptor) {
            // does it deliver an AWT Frame?
```


libdocdel@fr.com

```
                checkForAWTFrame(item, (UIDescriptor) attr);
            }
        }
    }

    private void checkForAWTFrame(ServiceItem item, UIDescriptor desc) {
        Set attributes = desc.attributes;
        Iterator iter = attributes.iterator();
        while (iter.hasNext()) {
            // search through the attributes, to find a UIFactoryTypes
            Object obj = iter.next();
            if (obj instanceof UIFactoryTypes) {
                UIFactoryTypes types = (UIFactoryTypes) obj;
                // see if it produces an AWT Frame Factory
                if (types.isAssignableTo(FrameFactory.class)) {
                    FrameFactory factory = null;
                    try {
                        factory = (FrameFactory) desc.getUIFactory(this.getClass().
                                                          getClassLoader());
                    } catch(Exception e) {
                        e.printStackTrace();
                        continue;
                    }

                    Frame frame = factory.getFrame(item);
                    frame.setVisible(true);
                }
            }
        }
    }


} // TestFrameUI
```

Images

User interfaces often contain images. They may be used as icons in toolbars, as general images on the screen, or as 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.

libdocdel@fr.com

The Swing package contains a convenience class called ImageIcon. This class 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, the ImageIcon constructor can use this version directly. There may be failures in this approach: the URL may be incorrect or malformed or the image may not exist on the HTTP server. Suitable code to create an image from a URL is as follows:

```
ImageIcon icon = null;
        try {
            icon = new ImageIcon(new URL("http://localhost/images/MINDSTORMS.ps"));
            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
```

ServiceType

A user interface may use code like that in the previous section directly to include images. The service may also supply useful images and other human-oriented information in a ServiceType entry object. The ServiceType class is defined as follows:

```
package net.jini.lookup.entry;

public class ServiceType {
    public String getDisplayName();     // Return the localized display
                                        // name of this service.
    public Image getIcon(int iconKind)  // Get an icon for this service.
    public String getShortDescription() // Return a localized short
                                        // description of this service.
}
```


libdocdel@fr.com

The class is supplied with empty implementations, returning `null` for each method. A service will need to supply a subclass with useful implementations of the methods. This is a useful class that could be used to supply images and information that may be common to a number of different user interfaces for a service, such as a minimized image.

MINDSTORMS UI Example

In Chapter 17, an example was given, in the "Getting It Running" section, of a client supplying a user interface to a MINDSTORMS service. This client not only knew that the service was a MINDSTORMS robot, but that it was a particular robot for which it could use a customized UI. In this section, we'll give two user interfaces for the MINDSTORMS "RoverBot," one of which is fairly general and could be used for any robot, and another that is customized to the RoverBot. The service is responsible for creating and exporting both of these user interfaces to a client.

RCXLoaderFrame

A MINDSTORMS robot is primarily defined by the `RCXPort` interface. The Jini version is defined by the `RCXPortImplementation` interface:

```
/**
 * RCXPortInterface.java
 */

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.
     */
```


libdocdel@fr.com

```
    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 type interface 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 these:

- Enter machine code (somehow) and download that.
- Enter RCX assembler code in the form of strings, and then assemble and download them.
- Enter NQC (Not Quite C) code, and then compile and download it.

The set of RCX classes by Laverde at `http://www.escape.com/~dario/java/rcx` includes a standalone application called `RCXLoader`, which does the second of these

libdocdel@fr.com

options. We can steal code from RCXLoader and some of his other classes to define an RCXLoaderFrame class:

```
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);
```


libdocdel@fr.com

```
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 truncated to save space in listing
}

// 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");
```


libdocdel@fr.com

```
        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);
    }

    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
```


libdocdel@fr.com

```
                    "+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());
```


libdocdel@fr.com

```
        } 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);
    }
}
```

RCXLoaderFrameFactory

The factory object for the RCX is now easy to define—it just returns a `RCXLoaderFrame` in the `getUI()` method:

```
/**
 * RCXLoaderFrameFactory.java
 */
```


libdocdel@fr.com

```
package rcx.jini;

import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.core.lookup.ServiceItem;
import java.awt.Frame;

public class RCXLoaderFrameFactory implements FrameFactory {

    public Frame getFrame(Object roleObj) {
        ServiceItem item= (ServiceItem) roleObj;
        RCXPortInterface port = (RCXPortInterface) item.service;
        return new RCXLoaderFrame(port);
    }

} // RCXLoaderFrameFactory
```

Exporting the FrameFactory

The factory object is exported by making it a part of a `UIDescriptor` entry object with a role, toolkit, and attributes:

```
Set typeNames = new HashSet();
typeNames.add(FrameFactory.TYPE_NAME);

Set attribs = new HashSet();
attribs.add(new UIFactoryTypes(typeNames));
// add other attributes as desired

MarshalledObject factory = null;
try {
    factory = new MarshalledObject(new
                                  RCXLoaderFrameFactory());
} catch(Exception e) {
    e.printStackTrace();
    System.exit(2);
}

UIDescriptor desc = new UIDescriptor(MainUI.ROLE,
                                     FrameFactory.TOOLKIT,
                                     attribs,
                                     factory);
Entry[] entries = {desc};
```


libdocdel@fr.com

```
JoinManager joinMgr = new JoinManager(impl,
                                      entries,
                                      this,
                                      new LeaseRenewalManager());
```

Customized 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 international support, the appearance, etc. All the variations, however, will just use the standard RCXPortInterface, because 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 customized user interface for their brand of blender.

I have been using an RCX car. While it can do lots of things, it has been convenient to use five commands for demonstrations: forward, stop, back, left, and right, with a user interface as shown in Figure 19-3.

Figure 19-3. This is a control panel taken from the LEGO® MINDSTORMS™ Robotics Invention System RCX programming system.

In Chapter 17, this appearance was hard-coded into the client. Since the client was just searching for *any* MINDSTORMS robot, it really shouldn't know about this sort of detail and should get this user interface from the robot service.

libdocdel@fr.com

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;
import java.rmi.RemoteException;

class CarJFrame extends JFrame
    implements RemoteEventListener, ActionListener {

    public static final int STOPPED = 1;
    public static final int FORWARDS = 2;
    public static final int BACKWARDS = 4;

    protected int state = STOPPED;

    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://www.LEGOMINDSTORMS.com/images/home_logo.ps"));
            switch (icon.getImageLoadStatus()) {
            case MediaTracker.ABORTED:
            case MediaTracker.ERRORED:
```


libdocdel@fr.com

```
                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);
```


libdocdel@fr.com

```
        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();
```


libdocdel@fr.com

```
}

public void actionPerformed(ActionEvent evt) {
    String name = evt.getActionCommand();
    byte[] command;

    if (name.equals("Forward")) {
        forwards();
    } else  if (name.equals("Stop")) {
        stop();
    } else  if (name.equals("Back")) {
        backwards();
    } else  if (name.equals("Left")) {
        sendCommand("e1 84");
        sendCommand("21 84");
        sendCommand("21 41");
        try {
            Thread.sleep(100);
        } catch(InterruptedException e) {
        }
        restoreState();

    } else  if (name.equals("Right")) {
        sendCommand("e1 81");
        sendCommand("21 81");
        sendCommand("21 44");
        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++) {
```


libdocdel@fr.com

```
                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());
        }
    }
}
```

CarJFrameFactory

The factory generates a `CarJFrame` object, like this:

```
/**
 * CarJFrameFactory.java
 */

package rcx.jini;

import net.jini.lookup.ui.factory.JFrameFactory;
import net.jini.core.lookup.ServiceItem;
import javax.swing.JFrame;

public class CarJFrameFactory implements JFrameFactory {

    public JFrame getJFrame(Object roleObj) {
        ServiceItem item  = (ServiceItem) roleObj;
        RCXPortInterface port = (RCXPortInterface) item.service;
        return new CarJFrame(port);
    }

} // CarJFrameFactory
```


libdocdel@fr.com

Exporting the FrameFactory

Both of the user interfaces discussed—the `RCXLoaderFrame` and the `CarJFrame`—can be exported by expanding the set of `Entry` objects.

```
// generic UI
Set genericAttribs = new HashSet();
Set typeNames = new HashSet();
typeNames.add(FrameFactory.TYPE_NAME);
genericAttribs.add(new UIFactoryTypes(typeNames));
MarshalledObject genericFactory = null;
try {
    genericFactory = new MarshalledObject(new
                              RCXLoaderFrameFactory());
} catch(Exception e) {
    e.printStackTrace();
            System.exit(2);
}
UIDescriptor genericDesc = new UIDescriptor(MainUI.ROLE,
                                            FrameFactory.TOOLKIT,
                                               genericAttribs,
                                            genericFactory);

// car UI
Set carAttribs = new HashSet();
typeNames = new HashSet();
typeNames.add(JFrameFactory.TYPE_NAME);
carAttribs.add(new UIFactoryTypes(typeNames));
MarshalledObject carFactory = null;
try {
    carFactory = new MarshalledObject(new CarJFrameFactory());
} catch(Exception e) {
    e.printStackTrace();
    System.exit(2);
}
UIDescriptor carDesc = new UIDescriptor(MainUI.ROLE,
                                        JFrameFactory.TOOLKIT,
                                        carAttribs,
                                        carFactory);

Entry[] entries = {genericDesc, carDesc};

JoinManager joinMgr = new JoinManager(impl,
                                      entries,
```


libdocdel@fr.com

```
                                        this,
                                        new LeaseRenewalManager());
```

The RCX Client

The following client will start up all user interfaces that implement the main UI role and that use a `Frame` or `JFrame`:

```
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 net.jini.lookup.entry.UIDescriptor;
import net.jini.lookup.ui.MainUI;
import net.jini.lookup.ui.attribute.UIFactoryTypes;
import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.lookup.ui.factory.JFrameFactory;

import java.util.Set;
import java.util.Iterator;

/**
 * TestRCX2.java
 */
```


libdocdel@fr.com

```
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;

        UIDescriptor desc = new UIDescriptor(MainUI.ROLE, null, null, null);
        Entry[] entries = {desc};
        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;
```


libdocdel@fr.com

```
        try {
            matches = registrar.lookup(template, 10);
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
            System.exit(2);
        }
        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;
            for (int m = 0; m < attributes.length; m++) {
                Entry attr = attributes[m];
                if (attr instanceof UIDescriptor) {
                    showUI(port, item, (UIDescriptor) attr);
                }
            }
        }



    }
}

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

private void showUI(RCXPortInterface port,
                    ServiceItem item,
                    UIDescriptor desc) {
    Set attribs = desc.attributes;
    Iterator iter = attribs.iterator();
    while (iter.hasNext()) {
        Object obj = iter.next();
        if (obj instanceof UIFactoryTypes) {
            UIFactoryTypes types = (UIFactoryTypes) obj;
            Set typeNames = types.getTypeNames();
            if (typeNames.contains(FrameFactory.TYPE_NAME)) {
                FrameFactory factory = null;
```


libdocdel@fr.com

```
                    try {
                       factory = (FrameFactory) desc.getUIFactory(this.getClass().
                                                          getClassLoader());
                    } catch(Exception e) {
                        e.printStackTrace();
                        continue;
                    }
                    Frame frame = factory.getFrame(item);
                    frame.setVisible(true);
                } else if (typeNames.contains(JFrameFactory.TYPE_NAME)) {
                    JFrameFactory factory = null;
                    try {
                      factory = (JFrameFactory) desc.getUIFactory(this.getClass().
                                                          getClassLoader());
                    } catch(Exception e) {
                        e.printStackTrace();
                        continue;
                    }
                    JFrame frame = factory.getJFrame(item);
                    frame.setVisible(true);
                }
            } else {
                System.out.println("non-gui entry");
            }
        }
    }
} // TestRCX
```

Summary

The serviceUI group is evolving a standard mechanism for services to distribute user interfaces for Jini services. The preference is to do this by `Entry` objects that contain factories for producing user interfaces.

libdocdel@fr.com

CHAPTER 20

Activation

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 that will be idle most of the time, probably swapped out to disk, but still using virtual memory. In JDK 1.2, the memory requirements on the server side can be enormous (hopefully this will be 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 because 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 were the only concern, there are a variety of other systems, such as echidna, that run multiple applications within a single JVM. These may be adequate to solve the memory issues. However, RMI Activation is also designed to work with distributed objects and allows JVMs to hold remote references to objects that are no longer active. Instead of throwing a remote exception when 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!

libdocdel@fr.com

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 cooperating objects.

The next sections show how to create a service as an activatable object that starts life in a server that sets up the activation group. Issues related to activation, such as security and state maintenance, will also be discussed.

The Service

An activable object subclasses from `Activatable` and uses a special two-argument constructor that will be called when the object needs to be reconstructed. There is a standard implementation of this constructor that just calls the superclass constructor:

```
public ActivatableImpl(ActivationID id, MarshalledObject data)
    throws RemoteException {
    super(id, 0);
}
```

(The use of the marshalled object parameter is discussed later in the "Maintaining State" section). Adding this constructor is all that is normally needed to change a remote service (that implements `UnicastRemoteObject`) into an activatable service. For example, an activatable version of the remote file classifier described in Chapter 9 in the "RMI Proxy for FileClassifier" section is as follows:

```
package activation;

import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;

import common.MIMEType;
import common.FileClassifier;
import rmi.RemoteFileClassifier;

/**
```


libdocdel@fr.com

```
 * FileClassifierImpl.java
 */

public class FileClassifierImpl extends Activatable
                                implements RemoteFileClassifier {

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.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
            // 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 default no-args constructor to initialize itself, since this new constructor is required for the object to be constructed by the activation system.

The Server

The server needs to create an activation group for the objects to run in. The main issue involved here 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 sets a policy file for running methods of the service object. The policy file for the server is set using

libdocdel@fr.com

the normal `-Djava.security.policy=...` argument to start the server. After setting various parameters, the activation group is set for the JVM by `ActivationGroup.createGroup()`.

Remote objects that subclass `UnicastRemoteObject` are created in the normal way using a constructor on the server. Activatable objects are not constructed in the server but are instead registered with `rmid`, which will look after construction on an as-needed basis.

In order to create activatable objects, `rmid` needs to know the class name and the location of the class files. The server wraps these up in an `ActivationDesc`, and registers this with `rmid` by using `Activatable.register()`. This returns an RMI stub object that can be registered with lookup services using the `ServiceRegistrar.register()` methods. This is also a little different from subclasses of `UnicastRemoteObject`, which pass an object that is converted to a stub by the RMI runtime. The required actions, in point form, are as follows:

- A service creates a subclass of `UnicastRemoteObject` using its constructor.
- A subclass of `Activatable` is created by `rmid` using a special constructor.
- For a `UnicastRemoteObject` object, the server needs to know the class files for the class in its `CLASSPATH` and the client needs to know the class files for the stub from an HTTP server.
- For an `Activatable` object, `rmid` needs to know the class files from an HTTP server, the server must be able to find the stub files from its `CLASSPATH`, and the client must be able to get the stub files from an HTTP server.
- A server hands a `UnicastRemoteObject` object to the `ServiceRegistrar.register()`. This is converted to the stub by the RMI runtime.
- A server gets a stub for an `Activatable` object from `Activatable.register()`. This stub is given directly to `ServiceRegistrar.register()`.

Changes need to be made to servers that export activatable objects instead of unicast remote objects. The server in Chapter 9, in the "RMI Proxy for FileClassifier" section, creates a unicast remote object and exports its RMI proxy to lookup services by passing the remote object to the `ServiceRegistrar.register()` method. The changes for such servers to export activatable objects are as follows:

- An activation group has to be created with a security policy file.
- The service is not created explicitly but is instead registered with `rmid`.

libdocdel@fr.com

- The return object from the registration is a stub that can be registered with lookup services.

- Leasing vanishes—the server just exits. The service will just expire after a while. See the “LeaseRenewalService” section later in the chapter for more details on how to keep the service alive.

The file classifier server using an activatable service would look like this:

```
package activation;

import rmi.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
 */

public class FileClassifierServer implements DiscoveryListener {

    static final protected String SECURITY_POLICY_FILE =
        "/home/jan/projects/jini/doc/policy.all";
    // Don't forget the trailing '/'!
```


libdocdel@fr.com

```
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);
    }
```


libdocdel@fr.com

```
    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);
    }

    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];
```


libdocdel@fr.com

```
            // 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
```

Running the Service

The service backend and the server must be compiled as usual, and in addition, an RMI stub object must be created for the service backend using the `rmic` compiler (in JDK 1.2, at least). The class files for the stub must be copied to somewhere where 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 backend (the client must be able to reconstruct a copy of the service's stub). This means that `rmid` must have access to the class files of the service backend, either from an HTTP server or from the file system. In the previous server, the `codebase` property in the `ActivationDesc` is set to an HTTP URL, so the class files for the service backend must be accessible to an HTTP server. Note that `rmid` does not get the class files for a service backend from the `CLASSPATH`, but from the codebase of the service. The HTTP server need not be on the same machine as the service backend.

libdocdel@fr.com

Before starting the service provider, an `rmid` process must be set running on the same machine as the service provider. An HTTP server must be running on a machine specified by the `codebase` property on the service. The service provider can then be started. This will register the service with `rmid` and will copy a stub object to any lookup services that are found. The server can then terminate. (As mentioned earlier, this will cause the service's lease to expire, but techniques to handle this are described later).

In summary, there are typically three processes involved in getting an activatable service running:

- The service provider, which specifies the location of class files in its codebase.
- `rmid`, which must be running on the same machine as the service provider and must be started before the service provider. It gets class files using the codebase of the service.
- An HTTP server, which can be on a different machine and is pointed to by the codebase.

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 its own log files.

Security

The JVM for the service will be created by `rmid` and will be running in the same environment as `rmid`. 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 that 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.

Some of the security issues with `rmid` have been addressed in JDK 1.3. These were discussed in Chapter 12, and they allow a security policy to be associated with each activatable service.

libdocdel@fr.com

Non-Lazy Services

The types of services discussed in this chapter so far 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 logging mechanism of `rmid`. If `rmid` crashes and is restarted, or the machine is rebooted and `rmid` restarts, then the server is able to use its log files to restart any "active" services registered with it, as well as to restore "lazy" services on demand. By making services non-lazy and ensuring that `rmid` is started on reboot, you can avoid messing around with boot configuration files.

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 saving and restoring state using a disk file typically does this.

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 example was discussed in Chapter 14—it could be sent `addType()` and `removeType()` messages. It begins with a given set of MIME type/file extension mappings. State here is very simple; it is just a matter of storing all the file extensions and their corresponding MIME types in a `Map`. If we turn this into an activatable object, we store the state by just storing the map. This map can be saved to disk using `ObjectOutputStream.writeObject()`, and it can be 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.

libdocdel@fr.com

The “normal” way of instantiating an object—through a constructor—doesn’t work very well with activatable objects. If a constructor for a class doesn’t start by calling another constructor with `this(...)` or `super(...)`, then the no-argument superclass constructor `super()` is called. However, 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 problem by not inheriting from `Activatable` and registering explicitly with the activation system, like this:

```
public FileClassifierMutable(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException {
        Activatable.exportObject(this, id, 0);
        // continue with instantiation
```

Nevertheless, this is a bit clumsy: you create an object solely to build up initial state, and then discard it because the activation system will recreate it on demand.

The technique we’ll use here is to create initial state if the attempt to restore state from the state file fails for any reason when the object is activated. This is done in the `restoreMap()` method 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;
```


libdocdel@fr.com

```
/**
 * FileClassifierMutable.java
 */

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 {
```


libdocdel@fr.com

```
        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();
```


libdocdel@fr.com

```
            } 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) {
```


libdocdel@fr.com

```
            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 it with the activation system. Here the filename is hard-coded, but it could be given as a command line argument (as 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;
```


libdocdel@fr.com

```
import java.rmi.activation.ActivationException;
import java.rmi.RemoteException;

/**
 * FileClassifierServerMutable.java
 */

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) {
```


libdocdel@fr.com

```
        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);
    }

    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);
    }
```


libdocdel@fr.com

```
    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) {

}
```


libdocdel@fr.com

```
} // FileClassifierServerMutable
```

This example used a simple way of storing state. Sun uses a far more complex system in many of its services, such as reggie—a "reliable log" in the package com.sun.jini.reliableLog. However, this package is not a part of standard Jini, so it may change or even be removed in later versions of Jini. There is nothing to stop you from using it, though, if you need a robust storage mechanism.

LeaseRenewalService

Activatable objects are an 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 occasion. These services raise a number of problems, and one was skirted around in the last section: How do you renew leases when the object is not alive?

Activatable objects are brought back to life when methods are invoked on them, and the expiration of a lease does not cause any methods to be invoked. There is no "lease-expiring event" generated that could cause a listener method to be invoked, either. It is true that a ServiceRegistrar such as reggie will generate an event when a lease changes status, but this is a "service removed" event rather than a "service about to be removed" event—at that point it is too late.

If a server is alive, then it can use a LeaseRenewalManager to keep leases alive, but there are two problems with this: first the renewal manager works by sleeping and waking up just in time to renew the leases, and second, if the server exits, then no LeaseRenewalManager will continue to run.

Jini 1.1 supplies a lease renewal service that partly solves 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, renewing them as needed. In general, it can keep leases alive without waking the service itself, which can slumber until it is activated by clients calling methods.

There is a small hiccup in this system, though: 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 running in the same JVM, 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, as specified by a lease. The service will still have to renew its lease, but with a LeaseRenewalService

libdocdel@fr.com

instead of a bunch of lookup services. The lease granted by this service will need to 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 in advance 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.

The Norm Service

Jini 1.1 supplies an implementation of `LeaseRenewalService` called `norm`. This is a non-lazy Activatable service that requires `rmid` to be running. This is run with the following command

```
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 the following

```
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 `norm.jar` file 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 the 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.

libdocdel@fr.com

The norm service will maintain a set of leases for a period of up to two hours. The reggie lookup service only grants leases for five minutes, so using this service increases the amount of time between renewing leases by a factor of over 20.

Using the LeaseRenewalService

The norm service exports an object of type LeaseRenewalService, which is defined by the following interface:

```
package net.jini.lease;

public interface LeaseRenewalService {
    LeaseRenewalSet createLeaseRenewalSet(long leaseDuration)
        throws java.rmi.RemoteException;
}
```

A server that wants to use the lease renewal service will first find this service and then call the createLeaseRenewal() method. The server requests a leaseDuration value, measured in milliseconds, for the lease service to manage a set of leases. The lease service creates a lease for this request, but the lease time may be less than the requested time (for norm, it is a maximum of two hours). In order for the lease service to continue to manage the set beyond the lease's expiry, the lease must be renewed before expiration. Since the service may be inactive at 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 the listener 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 an interface including the following:

```
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;
```


libdocdel@fr.com

```
    ....
}
```

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 for the whole LeaseRenewalSet expires (or until an exception happens, like a lease being refused).

Setting an expiration warning listener means that the notify() method of the listener will be called at least minWarning milliseconds before the lease for the set expires. The event argument 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. Here is a simple activatable class that can renew the lease:

```
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
```


libdocdel@fr.com

```
            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 `FileClassifierServer` example of the first section of this chapter. In addition, it will need to do a few other things:

- It will need to register the lease listener with the activation system as an activatable object.
- It will need to find a `LeaseRenewalService` from a lookup service.
- It will need to register all leases from lookup services with the `LeaseRenewalService`. Since it may find lookup services before it finds the renewal service, it will need to keep a list of lookup services found before finding the service, in order to register them with it.

Adding these additional requirements to the `FileClassifierServer` of the first section results in this server:

```
package activation;

import rmi.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;
```


libdocdel@fr.com

```
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;

/**
 * FileClassifierServerLease.java
 */

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);
```


libdocdel@fr.com

```
    } 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);
```


libdocdel@fr.com

```
        } 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 {
```


libdocdel@fr.com

```
            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");
```


libdocdel@fr.com

```
                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
```

LookupDiscoveryService

It is easy enough for a server to discover all of the lookup services within reach at the time it is started, by 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 any new lookup services will probably never be found. This will result in the service not being registered with them, which could mean 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 them 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 server that would have done registered the

libdocdel@fr.com

service 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 needs 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 sitting around waiting for such notification, so the listener object will probably also be an activatable object.

The LookupDiscoveryService interface has the following specification:

```
package net.jini.discovery;
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 expiring (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 and call the notify() method of the listener. The event interface includes the following:

```
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 whether the lookup service 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().

libdocdel@fr.com

The Fiddler Service

The Jini 1.1 release includes an implementation of the lookup discovery service called fiddler. It is a non-lazy activatable service and is started much like other services, such as reggie:

```
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]
```

For example,

```
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
```

Using the LookupDiscoveryService

An activatable service can make use of a lease renewal service to look after the leases for discovered lookup services. It can find these lookup services by means of a lookup discovery service. The logic that manages these two services is a little tricky.

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 occasion. The lease renewal service can call an activatable RenewLease object to do this, as shown in the preceding section of this chapter.

The lookup discovery service is also a leased service—it will only report changes to lookup services while its own lease is current. Therefore, 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. We define a DiscoveryChange object with a notify() method to handle changes in lookup services. If a lookup service has disappeared, we don't worry about it. If a lookup service has been discovered, we want to register the service with it and then manage the resultant lease. This means that the DiscoveryChange object must know both the service to be registered and the lease renewal service. This is static data, so these two objects can be passed in an array of two objects as the MarshalledObject to the activation constructor.

libdocdel@fr.com

The class itself can be implemented as shown here:

```
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 rmi.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());
```


libdocdel@fr.com

```
            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 the lease renewal object. After this, it can use a `ClientLookupManager` to find the lease service and register the lease renewal object with it. Now that it has a proxy for the service object, and also a lease renewal service, it can create the marshalled data for the lookup discovery service and register this with `rmid`. Then 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 initialization.

This all leads to the following server:

```
package activation;

import rmi.RemoteFileClassifier;

import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.LookupDiscoveryService;
```


libdocdel@fr.com

```
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
 */

public class FileClassifierServerDiscovery
    /* implements DiscoveryListener */ {
```


libdocdel@fr.com

```
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);
```


libdocdel@fr.com

```
    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 {
```


libdocdel@fr.com

```
            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());
```


libdocdel@fr.com

```
        } 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();
        }
```


libdocdel@fr.com

```
        // 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;
```


libdocdel@fr.com

```
        }
        return item.service;
    }
} // FileClassifierServerDiscovery
```

Summary

Some objects may not always be available, either because of mobility issues or because they are activatable objects. This chapter has dealt with activatable objects, and also with some of the special services that are needed to properly support these transient objects.

libdocdel@fr.com

Index

Symbols

* (asterisk) wildcard
  using to grant permissions, 174–176

A

AbstractEntry class
  implementation of Entry interface with, 46–47
abstract lease, 67–68
AbstractLease
  LandlordLease as subclass of, 67
AccessibleUI object
  of UIDescriptor, 363
Accounts
  class diagram for, *282*
Accounts interface
  code for, 283
  implementation of, 283–286
ACID properties, 271
Activatable class, 15
activatable class
  simple for renewing a lease, 414–415
Activatable object
  creating a service from, 396–397
activatable service
  maintaining state, 402–411
  processes involved in getting it running, 403
  security issues, 403
  security policies for, 181–182
Activation, 395–433
  using to create a service, 394–411
activation group
  creation of by server for objects to run in, 397–402
activation system
  registering explicitly with to build up initial state, 403
Address class
  subclassed out of AbstractEntry, 46–47
AllPermission
  removing, 172–173
  why this is bad security policy, 170–172
A Note on Distributed Computing
  from Sun Microsystems, 87–88
application
  as a collection of services, 12–13
  partitioning, 11–13
Atomicity
  in transactions, 271
attribute matching mechanism
  implementation of, 45–46
attributeSetTemplates
  elements used to match attributes, 58
attributes section
  of UIDescriptor, 362–363
attrSets parameter
  ServiceItem object, 50
attrSetTemplates field
  ServiceTemplate object, 61
AWT event dispatch system
  use of id field by, 237
AWT Frame
  produced by a factory for the "main" role, 364–372

B

backend service
  with RMI and non-RMI proxies, 114–115
basic directory
  Java packages, 17–18
Baum, David
  Not Quite C (nqc) from, 317–322
broadcast discovery, 32–39

libdocdel@fr.com

broadcast range
LookupDiscovery class, 39

C

cache
monitoring changes to, 266–267
CachedClientLookup
running, 265–266
cancel( )
using to cancel a lease, 65
cancelling
leases, 65
CarJFrame
exporting, 387–388
CarJFrame class
RCX code for, 383–387
CarJFrameFactory
generation of CarJFrame object by, 387
CarJFrame object
generated by CarJFrameFactory, 387
check( ) method
using in ServiceItemFilter interface, 259–261
class diagram
for leasing on the client, *227*
for leasing on the server, *227*
classes
defined in simple example, 104–105
needed for client and service implementation, 87–88
class files
sources of, 193–203
using multiple, 201–203
client
class diagram for leasing on, *227*
components needed in CLASSPATH, 106
implementation of, 104
implementation of transactions started by, 287–294
in a Jini system, 2–3
options for locating a suitable service, 86–87
uploading file-classifier service to, 98–107
client JVM
objects in, *92*, *93*
client leasing
class diagram for, *227*
client lookup
querying for a service locator, 5–6
ClientLookupManager class
in Jini version 1.1, 18
client requirements
security permissions, 176–178
clients
class file sources, 193–203
example of building, 83–107
client search, 57–62
client-side RCX class
defining public methods for, 316–317
client structure, 8–9
client.TestFileClassifier class, 105
client threads
moving code into a new class, 207–209
Comment class
subclassed out of AbstractEntry, 46
common.FileClassifier class, 104
common.MIMEType class, 104
complete.FileClassifierImpl class, 104
complete.FileClassifierServer class, 105
com.sun package
Jini 1.0 JoinManager class in, 163–166
configuration problems
troubleshooting in Jini system, 17–22
consistency
in transactions, 271
convenience classes, 46–47
subclassed out of AbstractEntry, 46–47
cookie
field in LandlordLease, 71–74
CORBA
building a simple example, 334
differences from Jini, 323
as distributed system architecture, 2–3
IDL used for specifying interfaces, 323

libdocdel@fr.com

implementations, 335–336
and Jini, 323–354
running the simple example, 335
a simple hello world IDL example, 328
CORBA and Jini services, *332*
CORBA backplane, 328
CORBA basic types
translation to a Java package, 326
CORBA client
implementation of Jini interface to act as, 332–333
migrating to Jini, 353–354
CORBA client in Java
proxy object for calling methods in CORBA server, 330–331
CORBA constant
translation to a Java package, 326
CORBA enumerated types
translation to a Java package, 326
CORBA exception
translation to a Java package, 326
CORBA interface
translation to a Java package, 326
CORBA meeting factory interface
CORBA and Jini services for fat proxies, *341*
CORBA and Jini services for single fat proxy, *343*
CORBA and Jini services for single thin proxy, *344*
CORBA and Jini services for thin proxies, *342*
exceptions, 344–345
making objects accessible to a Jini client, 340–345
many fat proxies exported, 341–342
many thin proxies exported, 342
multiple objects in, 340–345
single fat proxy for, 343
single thin proxy for, 343–344
CORBA meeting interface
making available as a mobile Jini object, 338
in room-booking example, 337–340
CORBA module
translation to a Java package, 326
CORBA object reference
reconstructing within a new ORB, 338–340
CORBA objects
languages for implementation of, 323
making accessible to the Jini world, 330–331
possibility of making into Jini service, 354
CORBA server in Java
for hello IDL, 328–330
CORBA service
copying the Java interface for, 331–332
CORBA structure
translation to a Java package, 326
CORBA to Java mapping, 325–326
CORBA translations
brief summary of, 326
credit/debit example
sequence diagram for, *274*
credit/debit example with transactions
sequence diagram for, *275*

D

DCOM
as distributed system architecture, 2–3
debugging
a Jini application, 22
Design Patterns (Eric Gamma et al.)
origin of event models from, 235
DialogFactory, 358
digital signatures
creating, 187
for interfaces to other services, 188
signing standard files, 187–188
Web site address for information about, 187
discarded( ) method, 34
discovered( ) method, 34
discovery
running threads from, 204–206
DiscoveryEvent object, 35–37
DiscoveryGroupManagement interface, 154
DiscoveryListener, 34
DiscoveryLocatorManagement interface, 155

libdocdel@fr.com

discovery management, 153–159
DiscoveryManagement interface, 154
discovery permission
  granting, 176–178
DiscoveryPermission
  granting, 174–176
distance interface
  implementing, 218–219
distance service
  finding after a printer is found, 213–221
  program for starting up with two printers, 219–221
distributed systems
  building with Jini, 1
distributed systems architectures
  CORBA and Jini, 323–354
  Jini as one of, 2–3
djinn
  security risks, 169–191
durability
  of transactions, 271

E

Editor class
  diagram, *43*
  searching for suitable editors in, 44–45
Enterprise Java Beans (EJBs)
  function of, 2
Entry class, 43–46
  information needed for distinguishing them, 315–316
Entry objects, 43–48
  further uses of entries in, 47–48
  passing into the ServiceItem object, 55
  restrictions on entries, 46–47
entry objects
  for robots, 315–316
equals( ) method
  implementation of, 46
errors
  Java packages, 17–18
  typical Jini package, 19–20
  typical lookup service, 20
  typical RMI stubs, 20–22
event models
  in Java, 235–236
  naming conventions specified by Java Beans, 236
  origin of, 235
event registration
  a version suitable for ordinary events, 241–242
EventRegistration
  convenience class, 238–239
  single listener, 239–240
example
  of a Jini service and client, 83–107
  problem description, 83–85
exception handling
  in Jini programs, 96–97
expiration
  of leases, 65
ExpirationWarningEvent, 412
ExtendedFileClassifierImpl object, 135–139
ExtendedFileClassifier interface, 134–135

F

factories
  marshalling, 358–359
  set of proposed, 358
factory implementation
  for creating user interface, 199–200
  HTTP server class files needed for, 201
  what files are needed where, 201
factory objects
  user interfaces from, 356–357
fiddler service
  implementation of, 422
file classifier
  with events, 244–249
FileClassifier
  implementation of, 99
  making available as network service, 85–87

libdocdel@fr.com

non-RMI proxy for, 123–124
RMI and non-RMI proxies for, 133–140
RMI proxy for, 115–123
running, 106–107
FileClassifier class
implementation of, 21–22
file classifier client
that creates and examines cache for suitable services, 263–265
FileClassifierImpl
for running the backend service, 116–117
FileClassifierImp object
exporting an implementation of a file classifier service as, 10–11
FileClassifier interface
for Jini service and client example, 89–90
modifying to dynamically update mappings, 244–249
FileClassifierLandlord class
code for, 231–233
FileClassifierLeasedResource class
code for, 228–229
FileClassifierLeaseManager class
code for, 229–231
file classifier problem
class diagram for, *110*
FileClassifierProxy
code for, 124–126
FileClassifierProxy class
code for, 133–134
FileClassifier return dialog box, *363*
file classifier server
code for using an activatable service, 399–402
FileClassifierServer, 117–120
adding additional requirements to, 415–420
implementation of, 99–104
file-classifier service
uploading to client, 98–107
what classes need to be where, 120–122, 139–140
FileClassifier service
code for finding, 9
file classifier UI example, 363–372
FileClassifier user interface, *363*
file command utility (Unix)
for determining file type, 83–85
file editors
as services, 43–45
FileServerImpl
thread creation for, 126–128
file type
methods for determining, 83–85
FrameFactory, 358
exporting, 387–388
FrameFactory object
exporting, 380–381

G

getExpiration( ) method
expiration value returned from, 64
for lease object, 54
getGroups( ) method
ServiceRegistrar, 41
getLease( ) method
of ServiceRegistration object, 63–66
getLocator( ) method
ServiceRegistrar, 41
getMIMEType()
calling, 9
getRegistrar( ) method
search and lookup performed by, 30–32
getServiceID( ) method
ServiceRegistrar, 41
.GIF file format, 11
grant blocks
for granting permissions to protection domains, 188–189
groups
broadcast discovery for, 33

libdocdel@fr.com

H

handback data
defined, 237
hardware devices
making into Jini services, 295–296
Heart client
code for, 145–146
HeartImpl class
code for, 147–151
Heart interface
method for, 142
heart monitor
making data into a Jini service, 140–151
HeartServer
code for, 142–144
hostile object
from bad security policy, 170–172
HTTP server
class needed to be accessable to, 106
for delivery of stub class files to clients, 24
as Jini support services, 13–15
required by Reggie, 24

I

IDL (Interface Definition Language)
used by CORBA for specifying interfaces, 323
IDL (Interface Definition Language) specification
Web site address for, 323
idltojava compiler
for converting CORBA IDL files to Java files, 334
IIOP (Inter-ORB Protocol)
used by CORBA implementations, 327
images
code for creating from a URL, 373
supplying in ServiceType entry objects, 373–374
in user interfaces, 372–373
initial state
creating if restoration from state file fails, 402–405
input events
generation of, 235–236
Interface Definition Language (IDL). *See* IDL (Interface Definition Language)
interfaces
for exported stubs, 21
InvalidLookupLocator.java
program in Java basic directory, 17–18
InvalidLookupLocator program, 26–27
running, 29
sample for building and running a simple Jini program, 27–28
isolation
in transactions, 271

J

.jar files
for Jini class files, 19–20
jarsigner
for signing classes and jar files, 187
Jarvi, Trent
Web site address for Linux version of DLL by, 297
Java
distributed computing environment for, 1
rmid support service as part of, 24
Java Media Framework (JMF) package, 360
Java packages
typical error, 17–18
Java Programming with CORBA (Vogel and Duddy)
room-booking service example in, 324
java.rmi.RemoteException
defining all methods to throw, 89–90
java.rmi.server.codebase property
specifiying protocol and class files location with, 14
JavaRoom interface
for single thin proxy, 345–346
java.security.debug
setting, 173
Java Virtual Machine. *See* JVM (Java Virtual Machine)
JComponentFactory, 358

libdocdel@fr.com

jdb debugger
debugging Jini applications with, 22
JDialogFactory, 358
JDK 1.3
and rmid security issues, 183–185
starting rmid in, 26
JFrame
returned by a typical factory, 357
JFrameFactory, 358
Jini
class files, 19–20
components, 2–3
and CORBA, 323–354
differences from CORBA, 323
example of a service and client, 83–107
migrating a CORBA client to, 353–354
new classes in version 1.1, 18
overview of, 1–16
with protection, 173–174
support services, 13
troubleshooting configuration problems, 17–22
use of SecurityManager, 169–170
uses for, 1–2
versions available, 18–19
Jini 1.0
JoinManager class, 163–166
Jini 1.1
JoinManager class, 161–163
ServiceDiscoveryManager class in, 255–256
Jini 1.2
security model options, 174
Jini applications
debugging, 22
more complex examples, 193–233
problem domain in, 193–195
Jini classes
for MINDSTORMS with RMI proxies, 301–307
jini-core.jar jar file
major packages of Jini in, 19
Jini Discovery Utilities Specification, 34
Jini djinn. *See* djinn
Jini environments
client trust certificates required in, 176–178
trust levels of the client, *177*
jini-ext.jar jar file
use of by Jini, 19–20
Jini federation
security, 173–176
Jini HTTP server
default port for, 24
Jini packages
typical package errors, 19–20
Jini programs
exception handling in, 96–97
Jini proxies, 326–327
examples of, 327
Jini server and client, 334
Jini services
making hardware into, 295–296
possibility of making CORBA objects into, 354
user interfaces for, 355–393
join manager, 161–167
JoinManager
changes in Jini version 1.1, 18
JoinManager class
getting information from, 166–167
in Jini 1.0, 163–166
in Jini 1.1, 161–163
versus ServiceDiscoveryManager class, 255–256
JoinManager threads, 207
JVM (Java Virtual Machine), 3
JVM objects
for factory implementation of user interface, *200*
for multiple class files implementation, *203*
for naive implementation of user interface, *198*
for a non-RMI proxy, *131*
JVMs
objects in, *110*
objects in all, *104*

libdocdel@fr.com

K

Kekoa Proudfoot
list of opcodes understood by the RCX by, 299
keystore
storing digital signatures in, 187
keytool
creating digital signatures with, 187

L

landlord
class diagram of an implementation, *71*
field in LandlordLease, 71–74
Landlord interface, 79–81
LandlordLease
class diagram for, 72
as subclass of AbstractLease, 67
LandlordLease class
AbstractLease extended by, 71–72
private fields, 71
landlord lease package, 68–81
class diagram of, *69*
factors that drive it, 69–70
Lease.ANY value
register( ) method, 63
leased FileClassifier
for adding and removing MIME mapping, 226–227
LeasedResource interface
sample code, 73–74
leased system
objects in, 64
leaseDuration method
ServiceRegistrar, 49–55
LeaseDurationPolicy class
LeasePolicy interface, 74–75
Lease.FOREVER value
register( ) method, 63
leaseFor( ) factory method
in LeasePolicy interface, 74
LeaseManager interface, 74–79
lease object
getExpiration( ) method, 54
Lease object
principal methods of, 64
LeasePolicy interface, 74
implementation of, 69–70
LeaseRenewalManager
changes in Jini version 1.1, 18
LeaseRenewalManager class
renewing leases with, 65–66
LeaseRenewalService
for activatable objects, 411–420
norm service implementation of, 410–412
using, 413–420
LeaseRenewalSet
interface, 413–420
leases
cancelling, 65
expiration of, 65
granting and handling, 66–81
renewing, 65–66
requesting and receiving, 63–66
leasing, 63–81
LEGO MINDSTORMS
for building LEGO toys with programmable computers, 295–322
as a Jini service, 296–297
user interface example, 374–393
Web site address, 296
LEGO MINDSTORMS Internals Web page
Web site address, 297
LEGO robots
building with LEGO MINDSTORMS, 296
listener lists
Jini management of, 235–254
listener source
proxies for services and listeners, 242–243
Locales object
of UIDescriptor, 363
local services
matching using, 213–221

libdocdel@fr.com

Location class
subclassed out of AbstractEntry, 47
long identifier
representation of transactions, 271–272
lookup-client-codebase parameter
for reggie, 25
LookupDirectoryService
interface specification, 421
for monitoring state of lookup services, 418–420
LookupDiscovery class
broadcast range, 39
use of for broadcast discovery, 33
LookupDiscoveryManager class
in Jini version 1.1, 18
LookupDiscoveryManager utility, 157–159
finding lookup services with, 255–256
LookupDiscoveryService
using, 420–429
LookupLocator class
constructors, 27
methods, 29
unicast discovery, 26–27
LookupLocatorDiscovery class
in Jini version 1.1, 18
specification, 155–157
lookup locators
components involved in finding, 153
finding, 153–155
interfaces for, 153–155
lookup-policy-file parameter
for reggie, 25
lookup-server-jarfile parameter
for reggie, 25
lookup service. *See also* service locator
discovering in Jini, 23–42
finding immediately, 257–259
finding once only, 221–225
finding to register a service object, 4–5
in a Jini system, 2–3
leases, 63–81
public, 24
registrar for, 4–5
running, 23–26
storage of service object on, 5
using network for connecting to, 26–27
lookup service errors
typical, 20
lookup-service-group parameter
for reggie, 25

M

mahalo
starting, 273
as a transaction manager, 14–15
mahalo.jar class files
using, 14–15
main( ) method
DiscoveryListener, 34
marshalled object
preparing for state file and registering with activation system, 407–411
MarshalledObject
wrapping factories in, 359
marshalling factories, 358–359
matching services, 61–62
methods
of LookupLocator class, 29
Mime class
defining, 84–85
MIMEType class
adding the Serializable interface to, 88
MimeType class
for all clients and file-classifier services, 87–88
MimeType object
serialization of, 87–88
MIME types, 83–85
searching for, 135–139
MINDSTORMS. *See also* LEGO MINDSTORMS
MINDSTORMS car
control panel from Robotics Invention System RCX programming system, *382*

libdocdel@fr.com

MINDSTORMS RoverBot. *See also* RoverBot
RCXPortImplementation interface for, 374–381
MINDSTORMS UI example, 374–393
MINDSTORMS with RMI proxies
class diagram for, *301*
multicast announcement permission
granting, 176–178
multicast client
example, 94–96
possible sources of exceptions, 97
MulticastRegister program
running, 38–39
that implements multicast searches, 36–37
multiple class files
using, 201–203
multiple class files implementation
JVM objects for, *203*
multiple listeners
diagram of, *241*
multithreading
improving responsiveness with, 204–206
MutableFileClassifier interface, 244–249
method for uploading new MIME type to a service, 178–179

N

Name class
subclassed out of AbstractEntry, 47
name entry
implementation of user interface for, 194–195
user interface for, *194*
NameEntry interface, 195–196
files needed for simple implementation, 198
naive implementation of, 196–198
NameEntry interface class
what files need to be where, 203
NameHandler class
using, 201–203
Nelson, Russell
LEGO MINDSTORMS Internals Web page BY, 297
net.jini.lookup.ui.AboutUI role, 361
net.jini.lookup.ui.AdminUI role, 361
net.jini.lookup.ui.MainUI role, 361
network plug and play
using Jini to distribute services with, 2
network plug and work
Jini advertised as, 17
non-Jini services
using on another host, 140–151
non-lazy services, 404
non-RMI and RMI proxies
class diagram for, *114*
non-RMI proxy
class diagram for, *121*
classes needed to deal with implementation, 131–132
communication with the server, 113
for FileClassifier, 123–124
JVM objects for, *131*
non-static entry
example of, 48
norm service
of LeaseRenewalService, 414–415
notify( ) method
keeping MulticastRegister program alive with, 38
Not Quite C (nqc)
language and compiler designed for RCX, 317–322
limitations of version 2.0.2, 319
NotQuiteC interface
code for, 318–319
nqc. *See* Not Quite C (nqc)
nqc compiler
limitations of, 317
turning into a Jini service, 317
nqc with RMI proxy
class diagram for, 318

libdocdel@fr.com

O

Object Management Group (OMG) Web site
IDL specification on, 323
Orbacus ORB
code needed to use ORB other than CORBA, 336
output-log-dir parameter
for reggie, 25

P

PanelFactory, 358
parameters
compulsory for running reggie, 25–26
in LookupDiscovery constructor, 33
register( ) method, 50–51
ServiceItem object, 50
parseString( ) method
of RCXOpcode, 298
PayableFileClassifierImpl interface
first element in, 276–277
PayableFileClassifier interface
extending Payable with, 277–278
permissions
using (*) wildcard when granting, 174–176
policy.all
restricting use of, 169–170
prepareAndComit( ) method
implementation of, 278–282
printer interface
designing, 209–213
finding a fast printer, 211–213
PropertyChangeListener
propertyChange( ) method, 236
propertyChange( ) method
PropertyChangeListener, 236
protection domains
in Java 1.2 security model, 186–191
permissions granted to, 188–189
proxies
how they are primed, 7–8
service objects as, 7–8
proxy choices, 109–115
proxy service, 7–8

R

RCX
language for programming, 317–322
listener for handling responses from, 298
RCX car
RCX code for, 383–387
RCX class
client-side, 316–317
RCX client
for starting up all user interfaces, 389–392
RCXLoaderFrame
defining, 374–381
exporting, 387–388
RCXLoaderFrameFactory
defining, 381
RCX microcomputer
LEGO MINDSTORMS, 296
RCXOpcode class, 298
RCXPort
code for, 297–298
relevent methods for, 298
RCXPort( ) constructor, 298
RCXPortImpl
implementation, 303–307
RCXPortImplementation interface
for MINDSTORMS robot, 374–381
RCX programs
controlling from a computer by standalone programs, 299–301
simple client for getting RCX to perform actions, 309–315
simple server for getting them running, 307–308
RCX robot
building customized user interfaces for, 381-382
general purpose user interfaces for, 375
reggie
compulsory parameters for running, 25–26
lookup service as part of Jini, 23–26

libdocdel@fr.com

support services required by, 24
reggie service locator
exporting registrar objects with, 15
register( ) method
parameters, 50–51
ServiceRegistrar, 49–55
values, 63
registrar
as proxy to the lookup service, 4
returned to client, 6
registrar objects
exporting with reggie service locator, 15
registration code
moving to a seperate thread, 204–206
RemoteDiscoveryEvent
interface, 421
RemoteEvent class
public methods for, 236–238
RemoteEventListener interface
implementation of, 239
remote events, 235–254, 236–238
RemoteExtendedFileClassifier subinterface
adding remote interface with, 134–135
RemoteFileClassifier, 116
Remote Method Invocation (RMI)
in JDK 1.2, 15
use of by Jini, 1
RemoteNotQuiteC interface
code for implementation of, 318–319
RemoteRCXPort interface
adding remote interface with, 303
renew( )
parameter for, 65
renewFor( )
setting lease duration with, 66
renew( ) method
LeasePolicy interface, 74–79
RequiredPackages object
of UIDescriptor, 363
Rich Text Format (.RTF), 11
RMI Activation. *See* Activation
running the service, 402–403
RMI and non-RMI proxies, 114–115
class diagram for, *114*
for FileClassifier, 133–140
rmid
and JDK 1.3, 26
security issues in JDK 1.3, 183–185
security issues on activatable service, 182–183
securtiy issue on multiuser systems, 25
RMI daemon
as Jini support service, 13, 15
required by Reggie, 24
rmid support service
default TCP port used, 25
options, 24–25
as part of standard Java distribution, 24
RMI parameters
for system security, 178–179
RMI proxy, 110–112
class diagram for, *111*
configuration issues for implementation, 120–122
JVM objects for, *111*
RMI proxy FileClassifier
running, 122–123, 132–133
RMI proxy for
for FileClassifier, 115–123
RMISecurityManager
installing, 169
RMI stubs
typical error, 20–22
robot
entry objects for, 315–316
Robotics Intervention System
LEGO MINDSTORMS as, 295–322
role
played by user interface, 361–362
ways given for different objects, 361–362
role interfaces
number of, 361

libdocdel@fr.com

RoomBookingBridge interface
implementation, 347–351
for single thin proxy, 346
room-booking example, 336–352
building, 352
considerations, 336–337
CORBA objects, 337–340
other classes in, 351
running, 352
room-booking service
modified IDL for, 324–325
RoverBot
client code, 309–315
MINDSTORMS robot, *309*
user interface examples for, 374–393
.RTF file format, 11
RXCPortInterface
defines methods made available from Jini service, 301–302

S

security
being paranoid about, 186
of Jini distributed systems, 169–191
in Jini federation, 173–176
for joining groups, 175
reducing risks on Unix, 182–183
risks of not giving enough permission, 172–173
of ServiceRegistrar interface, 179
service requirements, 174–176
socket permission, 175
SecurityManager
security decisions made by, 169–170
statement for including in code, 169
security model
options in Jini 1.2, 174
security policy
for activatable service, 181–182
example of minimal, 175–176
to restrict possible attacks, 189–191
setting for transaction manager, 180–182
Serializable interface
implementation of, 87–88
server
class diagram for leasing on, *227*
classes needed in the CLASSPATH of, 105
class files needed for factory implementation, 201
communication with the non-RMI proxy, 113
creation of activation group by, 397–402
server JVM
objects in, *103*
server leasing
class diagram for, *227*
servers
class file sources, 193–203
server structure, 10–11
server threads
moving registration code to a separate, 204–206
service
creation of in a Jini system, 3–5
in a Jini system, 2–3
service architecture
choices for, 109–152
service backend objects, 7–8
ServiceDiscoveryEvent objects
monitoring changes in services with, 267–269
ServiceDiscoveryListener interface
monitoring changes to the cache, 266–267
ServiceDiscoveryManager, 255–269
ServiceDiscoveryManager interface, 255–256
building a cache of services in, 262–265
ServiceEvent.getTransition( ) method
categories from, 249–250
ServiceID, 54
using when registering with a service locator, 222–225
serviceID field
ServiceTemplate object, 61
ServiceIDListener
changes in Jini version 1.1, 18

libdocdel@fr.com

serviceID parameter
ServiceItem object, 49–55
ServiceInfo class
subclassed out of AbstractEntry, 47
ServiceItemFilter interface
client-side filter for finding services, 256–257
using check( ) method in, 259–261
ServiceItem object
creating, 49–50
parameters, 50
service locator. *See also* lookup service
class file sources, 193–203
discovering in Jini, 23–42
leasing changes to, 225–233
procedure to use when registering with, 222–225
querying for, *4*
service locator JVM
objects in, *93*
ServiceMatches object
receiving, 60–61
service matching
inexact, 209–213
using local services, 213–221
service object
registration of with lookup services, 3–5
storing on the lookup service, 5
service parameter
ServiceItem object, 50
service( ) parameter
ServiceItem object, 49–55
service provider
choices for, 128–131
classes and interfaces needed to be known by, 105
creation of a service by, 3–5
role of, 3–5
service proxy
integration with service backend objects, 97–98
ServiceRegistrar
implementation of, 39–40
information from, 41
methods for, 40
methods used with, 57–58
searching for services with, 57–60
service registration with, 49
ServiceRegistrar interface
security issues, 179
service registration, 3–5
with lookup locators, 49–55
with ServiceRegistrar, 49
ServiceRegistration class
public methods for, 51
ServiceRegistration object
creating, 51–52
getting information from, 54
objects in, *51*
services
building a cache of, 262–265
matching, 61–62
monitoring changes in, 249–254
program for monitoring all changes in, 250–254
searching for with ServiceRegistrar, 57–60
template for monitoring all changes, 250
service specification
for building a client, 86–87
ServiceTemplate class
for searching for services, 57–58
ServiceTemplate object
fields, 61
ServiceType class
defined, 373
subclassed out of AbstractEntry, 47
ServiceType entry object
supplying images and other information in, 373–374
serviceTypes field
ServiceTemplate object, 61
setuid method
settings for security, 183–185
SimpleService program
running, 53

libdocdel@fr.com

for unicast server, 52–53
single listener
diagram of, *240*
event registration, 239–240
single thin proxy
building for a federation of CORBA objects, 345–346
JavaRoom interface, 345–346
RoomBookingBridge interface, 346
sleep( ) method
keeping MulticastRegister program alive with, 37–38
smart file view application, 11–13
SmartViewer application
use of, 12–13
software applications
making into Jini services, 295–296
Status class
subclassed out of AbstractEntry, 47
Sun extension package
for talking to serial and parallel ports, 297
sun-util.jar jar file
contents of, 20
superclasses
for exported stubs, 21
support services
use of by Jini, 13
system exceptions
CORBA methods, 344

T

TCP/IP
implementation of Jini on, 2
TCP port
rmid default, 25
The Jini Specification (Ken Arnold, et al.), 88
Thread
creating for FileServerImpl, 126–128
toolkit, 360
tools.jar file
Jini HTTP server stored in, 24
toString( ) method
implementation of, 46
transaction example
to handle money transfers, 274–294
transaction identifiers, 271–272
transaction manager
security issues, 180–182
TransactionManager, 272–273
TransactionManager.Created object
transaction identifier and lease contained in, 272
transaction manager proxy
reconstituting at the client, 14–15
transaction participant
class diagram for, *277*
TransactionParticipant, 273
transactions, 271–294
TRANSITION_MATCH_MATCH category
ServiceEvent.getTransition( ) method, 250
TRANSITION_MATCH_NOMATCH category
ServiceEvent.getTransition( ) method, 250
TRANSITION_NOMATCH_MATCH category
ServiceEvent.getTransition( ) method, 249
troubleshooting
Jini configuration problems, 17–22
trust certificates
required for clients in Jini environments, 176–178
trust levels
of client in Jini environment, *177*
two-phase commit protocol
used by transactions, 271
type interface
as reference to Java interface, 356

U

UDP multicast requests
finding lookup services with, 4–5
UI (user interface). *See* user interface; user interfaces

libdocdel@fr.com

UIDescriptor, 360
  attributes section, 362–363
  role field in, 361–362
  rules for, 362
UIFactoryTypes object
  of UIDescriptor, 362
UML class diagram
  for smart file viewer application, 11–12
  surrounded by the JVM in which objects exist, *92*
UML sequence diagram
  for discovery, 35
  for lookup, *30*
unicast client
  lookup techniques for service, 90–94
unicast discovery, 26–32
UnicastRegister program
  running, 32
UnicastRemoteObject class, 15
unicast TCP
  connecting to a lookup service with, 4
Unix
  reducing security risks on, 182–183
user exceptions
  CORBA methods, 344–345
user interface
  factory implementation, 199–200
  for implementation of name entry, 194–195
  roles, 361–362
  toolkit, 360
user interfaces
  as entries, 355–356
  exporting as factory objects, 356–357
  from factory objects, 356–357
  images in, 372–373
  for Jini services, 355–393
  RCX client for starting, 389–392
  suitable code to create an image from a URL, 373

V

VetoableChangeListener
  vetoableChange( ) method, 236
vetoableChange( ) method
  VetoableChangeListener, 236

W

Waldo, Jim
  A Note on Distributed Computing by, 87–88
Web site address
  for digital signature information, 187
  for Java package by Dario Laverde, 297
  for The Jini Specification, 88
  LEGO MINDSTORMS, 296
  for LEGO MINDSTORMS Internals Web page, 297
  for Linux version of DLL by Trent Jarvi, 297
  for list of opcodes understood by the RCX, 299
  for Sun extension package, 297
write( ) method
  for RCXPort, 298

libdocdel@fr.com

The Story Behind Apress

APRESS IS AN INNOVATIVE PUBLISHING COMPANY devoted to meeting the needs of existing and potential programming professionals. Simply put, the "A" in Apress stands for the "author's press™." Our unique author-centric approach to publishing grew from conversations between Dan Appleman and Gary Cornell, authors of best-selling, highly regarded computer books. They wanted to create a publishing company that emphasized quality above all—a company whose books would be considered the best in their market.

To accomplish this goal, they knew it was necessary to attract the very best authors—established authors whose work is already highly regarded, and new authors who have real-world practical experience that professional software developers want in the books they buy. Dan and Gary's vision of an author-centric press has already attracted many leading software professionals—just look at the list of Apress titles on the following pages.

libdocdel@fr.com

Would You Like to Write for Apress?

APRESS IS RAPIDLY EXPANDING its publishing program. If you can write and refuse to compromise on the quality of your work, if you believe in doing more then rehashing existing documentation, and if you are looking for opportunities and rewards that go far beyond those offered by traditional publishing houses, we want to hear from you!

Consider these innovations that we offer every one of our authors:

- Top royalties with *no* hidden switch statements. For example, authors typically only receive half of their normal royalty rate on foreign sales. In contrast, Apress' royalty rate remains the same for both foreign and domestic sales.

- A mechanism for authors to obtain equity in Apress. Unlike the software industry, where stock options are essential to motivate and retain software professionals, the publishing industry has stuck to an outdated compensation model based on royalties alone. In the spirit of most software companies, Apress reserves a significant portion of its equity for authors.

- Serious treatment of the technical review process. Each Apress book has a technical reviewing team whose remuneration depends in part on the success of the book since they, too, receive a royalty.

Moreover, through a partnership with Springer-Verlag, one of the world's major publishing houses, Apress has significant venture capital behind it. Thus, Apress has the resources both to produce the highest quality books *and* to market them aggressively.

If you fit the model of the Apress author who can write a book that gives the "professional what he or she needs to know™," then please contact any one of our editorial directors, Gary Cornell (gary_cornell@apress.com), Dan Appleman (dan_appleman@apress.com), or Karen Watterson (karen_watterson@apress.com), for more information on how to become an Apress author.

libdocdel@fr.com

ISBN	LIST PRICE	AVAILABLE	AUTHOR	TITLE
1-893115-83-6	$44.95	Winter 2000	Wells	Code Centric: T-SQL Programming with Stored Procedures and Triggers
1-893115-05-4	$39.95	Winter 2000	Williamson	Writing Cross-Browser Dynamic HTML
1-893115-02-X	$49.95	Now	Zukowski	John Zukowski's Definitive Guide to Swing for Java 2
1-893115-78-X	$49.95	Now	Zukowski	Definitive Guide to Swing for Java 2, Second Edition

To order, call (800) 777-4643 or email `sales@apress.com`.

libdocdel@fr.com

Apress Titles

ISBN	LIST PRICE	AVAILABLE	AUTHOR	TITLE
1-893115-01-1	$39.95	Now	Appleman	Dan Appleman's Win32 API Puzzle Book and Tutorial for Visual Basic Programmers
1-893115-23-2	$29.95	Now	Appleman	How Computer Programming Works
1-893115-09-7	$24.95	Now	Baum	Dave Baum's Definitive Guide to LEGO MINDSTORMS
1-893115-84-4	$29.95	Now	Baum, Gasperi, Hempel, Villa	Extreme MINDSTORMS
1-893115-82-8	$59.95	Now	Ben-Gan/Moreau	Advanced Transact-SQL for SQL Server 2000
1-893115-14-3	$39.95	Winter 2000	Cornell/Jezak	Visual Basic Add-Ins and Wizards: Increasing Software Productivity
1-893115-85-2	$34.95	Winter 2000	Gilmore	A Programmer's Introduction to PHP 4.0
1-893115-17-8	$59.95	Now	Gross	A Programmer's Introduction to Windows DNA
1-893115-86-0	$34.95	Now	Gunnerson	A Programmer's Introduction to C#
1-893115-10-0	$34.95	Now	Holub	Taming Java Threads
1-893115-04-6	$34.95	Now	Hyman/Vaddadi	Mike and Phani's Essential C++ Techniques
1-893115-79-8	$49.95	Now	Kofler	Definitive Guide to Excel VBA
1-893115-75-5	$44.95	Now	Kurniawan	Internet Programming with VB
1-893115-19-4	$49.95	Now	Macdonald	Serious ADO: Universal Data Access with Visual Basic
1-893115-06-2	$39.95	Now	Marquis/Smith	A Visual Basic 6.0 Programmer's Toolkit
1-893115-22-4	$27.95	Now	McCarter	David McCarter's VB Tips and Techniques
1-893115-76-3	$49.95	Now	Morrison	C++ For VB Programmers
1-893115-80-1	$39.95	Now	Newmarch	A Programmer's Guide to Jini Technology
1-893115-81-X	$39.95	Now	Pike	SQL Server: Common Problems, Tested Solutions
1-893115-20-8	$34.95	Now	Rischpater	Wireless Web Development
1-893115-24-0	$49.95	Now	Sinclair	From Access to SQL Server
1-893115-16-X	$49.95	Now	Vaughn	ADO Examples and Best Practices

libdocdel@fr.com