The Jini™ Specification

The Jini™ Technology Series

Lisa Friendly, Series Editor
Ken Arnold, Technical Editor
For more information see: http://java.sun.com/docs/books/jini/

This series, written by those who design, implement, and document the Jini™ technology, shows how to use, deploy, and create Jini applications. Jini technology aims to erase the hardware/software distinction, to foster spontaneous networking among devices, and to make pervasive a service-based architecture. In doing so, the Jini architecture is radically changing the way we think about computing. Books in **The Jini Technology Series** are aimed at serious developers looking for accurate, insightful, thorough, and practical material on Jini technology.

The Jini Technology Series web site contains detailed information on the Series, including existing and upcoming titles, updates, errata, sources, sample code, and other Series-related resources.

Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, Ann Wollrath, *The Jini™ Specification*
ISBN 0-201-61634-3

Eric Freeman, Susanne Hupfer, and Ken Arnold, *JavaSpaces™ Principles, Patterns, and Practice*
ISBN 0-201-30955-6

The Jini™ Specification

Ken Arnold
Bryan O'Sullivan
Robert W. Scheifler
Jim Waldo
Ann Wollrath

ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

This book is dedicated to the Jini team
without whom this book
would not have been necessary

Contents

Foreword

THE emergence of the Internet has led computing into a new era. It is no longer what your computer can do that matters. Instead, your computer can have access to the power of everything that is connected to the network: The Network is the Computer™. This network of devices and services is the computing environment of the future.

The Java™ programming language brought reliable object-oriented programs to the net. The power of the Java platform is its simplicity, which allows programmers to be fully fluent in the language. This simplicity allows debugged Java programs to be written in about a quarter the time it takes to write programs in C++. We believe that use of the Java platform is the key to the emergence of a "best practices" discipline in software construction to give us the reliability we need in our software systems as they become more and more widely used.

The Jini™ architecture is designed to bring reliability and simplicity to the construction of networked devices and services. The philosophy behind Jini is language-based systems: a Jini system is a collection of interacting Java programs, and you can understand the behavior of this Jini system completely by understanding the semantics of the Java programming language and the nature of the network, namely that networks have limited bandwidth, inherent latency, and partial failure.

Because the Jini architecture focuses on a few simple principles, we can teach Java language programmers the full power of the Jini technology in a few days. To do this, we introduce remote objects (they just throw a `RemoteException`), leasing (commitments in a Jini system are of limited duration), distributed events (in the network events aren't as predictable on a single machine), and the need for two-phase commit (because the network is a world of partial failures). This small set of additional concepts allows distributed applications to be written, and we can illustrate this with the JavaSpaces™ service, which is also specified here.

For me, the Jini architecture represents the results of almost 20 years of yearning for a new substrate for distributed computing. Ever since I shipped the first

widely used implementation of TCP/IP with the Berkeley UNIX system, I have wanted to raise the level of discourse on the network from the bits and bytes of TCP/IP to the level of objects. Objects have the enormous advantage of combining the data with the code, greatly improving the reliability and integrity of systems. For me, the Jini architecture represents the culmination of this dream.

I would like to thank the entire Jini team for their continuing hard work and commitment. I would especially like to thank my longtime collaborator Mike Clary for helping to get the Jini project started and for directing the project; the Jini architects Jim Waldo, Ken Arnold, Bob Scheiffler, and Ann Wollrath for designing and implementing such a simple and elegant system; Mark Hodapp for his excellent engineering management; and Samir Mitra for committing early to the Jini project, helping us understand how to explain it and what problems it would solve, and for driving the key business development that helped give Jini technology the momentum it has in the marketplace today. I would also like to thank Mark Tolliver, the head of the Consumer and Embedded Division, which the Jini project became part of, for his support.

Finally, I would like to thank Scott McNealy, with me a founder of Sun Microsystems™, Inc., and its longtime CEO. It is his continuing support for breakthrough technologies such as Java and Jini that makes them possible. As Machiavelli noted, it is hard to introduce new ideas, and support like Scott's is essential to our continuing success.

BILL JOY
ASPEN, COLORADO
APRIL, 1999

Preface

Perfection is reached, not when there is no longer anything to add,
but when there is no longer anything to take away.
—Antoine de Saint-Exupery

THE Jini architecture is designed for deploying and using services in a network. Networks are by nature dynamic: new things are added, old things are removed, existing things are changed, and parts of the network fail and are repaired. There are therefore problems unlike any that will appear in a single process or even multiple processes in a single machine.

These differences require an approach that takes them into account, makes changes apparent, and allows older parts to work with newer parts that are added. A distributed system must adapt as the network changes since the network *will* change. The Jini architecture is designed to be adaptable.

This book contains three parts. The first part gives an overview of the Jini architecture, its design philosophy, and its application. This overview sets up the following sections, which contain examples of programming in a Jini system. The first section of the introduction is also usable as a high-level overview for technical managers.

The sections of the introduction that contain examples are designed to orient you within the Jini technology and architecture. They are not a full tutorial: Think of them as a tour through the process of design and implementation in a Jini system. As with any tour, you can get the flavor of how things work and where you can start your own investigation.

The second part of the book is the specification itself. Each chapter of the specification has a brief introduction describing its place in the overall architecture.

The third part of the book contains supplementary material: a glossary that defines terms used in the specifications and in talking about Jini architecture, design, and technology, and two appendices. Appendix A is a reprint of "A Note

on Distributed Computing," which describes critical differences between local and remote programming. Appendix B contains the full source code for the examples in the introductory material.

HISTORY

The Jini architecture is the result of a rather extraordinary string of events. But then almost everything is. The capriciousness of life—and to the fortunate, its occasional serendipity—is always extraordinary. It is only in retrospect that we examine the causes and antecedents of something interesting and decide that, because they shaped that interesting result, we will call them "extraordinary." Other events, however remarkable, go unremarked because they are unexamined. Those of us who wrote the Jini architecture, along with the many who contributed to its growth, are lucky to have a reason to examine our particular history to notice its pleasures.

This is not the proper place for a long history of the project, but it seems appropriate to give a brief summary of the highlights. The project had its origins in Sun Microsystems Laboratories, where Jim Waldo ran the Large Scale Distribution research project. Jim Waldo and Ken Arnold had previously been involved with the Object Management Group's first CORBA specification while working for Hewlett-Packard. Jim brought that experience and a long-term background in distributed computing with him to Sun Labs.

Soon after joining the Labs, Jim made Ann Wollrath part of the team. Soon after, observations about many common approaches in the field of distributed computing led Jim, Ann, and the other authors to write "A Note on Distributed Computing," which outlined core distinctions between local and distributed design. Many people had been trying to hide those differences under the general rubric of "local/remote transparency." The "Note" argued that this was not possible. It has become the most cited Sun Laboratories technical report, and the lessons it distills are at the core of the design approach taken by the project.

At this time the project was using Modula 3 Network Objects for experiments in distributed computing. As Modula 3 ceased to be developed, the team looked around for a replacement language. At that time Oak, the language an internal Sun project, seemed a viable replacement with some interesting new properties. To a research project, the fact that Oak was commercially insignificant was irrelevant. It was at this time that Ken rejoined Jim on his new team.

Soon after, Oak was renamed "Java."

When it was still Oak, it once had a remote method invocation mechanism, but that was removed when the mechanism failed—it, too, had fallen into the local/remote transparency trap. When Bill Joy and James Gosling wanted to create a working distributed computing mechanism, they asked Jim to lead the effort,

which switched our team from the laboratories into the JavaSoft product group. As the first result of this effort, Ann, as the Java RMI architect, steered the team on an exploration of what could be done with a language-centric approach to distributed computing (most distributed computing systems are built on language-neutral approaches).

After RMI became part of the Java platform, Bill Joy asked the team to expand its horizons to include a platform for easier distributed computing, coining the name "Jini."[1] He convinced Sun management to put the RMI and Jini project into a separate unit. This new unit started with Jim, Ann, Ken, and Peter Jones, and was soon joined by Bob Scheiffler who had extensive distributed computing experience from the X Windows project that he ran. This put together the original core architectural team: Jim, Ann, Ken, and Bob.

As the team grew, many people had a hand in the direction of various parts of the architecture, including Bryan O'Sullivan who took over the design of the lookup discovery protocol. Mike Clary took the project under his wing to give it time to grow. Mark Hodapp joined the team to manage its software development and run it in partnership with its technical leadership. Gary Holness, Zane Pan, Brian Murphy, John McClain, and Bob Resendes all reviewed the primary architecture documents and had responsibility for various parts of the tool design, implementation design, and the implementations themselves. Laird Dornin and Adrian Colley joined the RMI sub-team to continue and expand its development. Charlie Lamb joined the architectural team to oversee work with outside companies, starting with printing and storage service standards. Jen McGinn joined the team to document what we had done, later with the help of Susan Snyder on production support. Jimmy Torres started out as our release engineer and has changed to working on helping build our public developer community. Frank Barnaby took over the release engineering duties. Helen Leary joined early and kept our infrastructure humming along.

Our QA team was Mark Schuldenfrei and Anand Dhingra, managed by Brendan Daly. Alan Mortensen wrote the conformance tests and their infrastructure. Emily Suter and Theresa Lanowitz started out our marketing team, with Franc Romano, Donna Michael, Joan MacEachern, and Paula Kozak joining later. Jim Hurley started setting up our support organization, and Keith Thompson and Peter Marks joined to work on sales engineering. Samir Mitra led a marketing and business development team that included Jon Bostrom, Jaclyn Dahlby, Mike McNerny, Miko Matsamura, Darryl Mocek, Sharam Moradpour, and Vince Vasquez. Many others, too numerous to mention, did important work that made the Jini architecture possible and real.

---

[1] Jini is not an acronym. To remember this, think of it as standing for "Jini Is Not Initials." It is pronounced the same as "genie."

ACKNOWLEDGMENTS

As the specifications were written, almost every member of the team made important contributions. Their names are listed above; we note the fact here to express our gratitude. A good idea and a dollar will buy a bad cup of espresso—you need people who will make that idea live, sand off any rough edges, and help you rework any bad parts of the idea into good ones. We had those people—some of the best we've ever worked with. Without them the Jini architecture would be some rather nice ideas on paper. Because of their commitment to adopt the vision as their own, to make it better, and to make it real, there are people (like you, the reader) who care about these ideas and can do something with them. We thank the entire team for what they have done to improve the Jini architecture and to help us write and release the Jini technology.

Bill Joy created the environment in which the Jini architecture could be developed and nurtured, and fed the architecture with his own reviews and ideas. His vision and support inside and outside of Sun made the project possible. This book itself is also his idea.

Bob Sproull gave the Large Scale Distribution project scope and support that has continued to this day, through all its many twists and turns, even after we were no longer were part of his Sun Labs organization. Mike Clary's protection and guidance was critical to fostering the creative atmosphere around the Jini project.

Jen McGinn and Susan Snyder did a lot of work to make this book possible, including hours in front of a screen converting the specification documents from their original form into that of the book. Jen also worked hard to improve the content of the specifications and introductory material during their creation, making them clearer and their English more correct. Dick Gabriel contributed to the content and organization of the *Jini Architecture Specification,* making it clearer and easier to use.

Many people reviewed the introductory material, making comments that improved it tremendously: Liz Blair, Charlie Lamb, John McClain, Bob Resendes, and Bob Sproull. Lisa Friendly has applied her experience as series editor with the Java Series to help us create this sibling Jini Series. We would also like to thank the people at Addison-Wesley's Professional Computing group who worked with us on this book and the series: Mike Hendrickson, Julie DeBaggis, Sarah Weaver, Marina Lang, and Diane Freed. And without Susan Stambaugh's help, communicating with Bill (and sometimes Mike) is not merely difficult, but probably theoretically impossible.

To these and many others too numerous to mention we give our thanks and appreciation for what they did to make these ideas and this book possible.

PART 1

Overview and Examples

The Jini Architecture: An Introduction

1 Overview

The man who sets out to carry a cat by its tail
learns something that will always be useful
and which never will grow dim or doubtful.
—Mark Twain

JINI technology is a simple infrastructure for providing services in a network, and for creating spontaneous interactions between programs that use these services. Services can join or leave the network in a robust fashion, and clients can rely upon the availability of visible services, or at least upon clear failure conditions. When you interact with a service, you do so through a Java object provided by that service. This object is downloaded into your program so that you can talk to the service even if you have never seen its kind before—the downloaded object knows how to do the talking.

That's the whole system in a nutshell. It's not very much to say (although you will learn a lot more about the details). But like many ideas that are relatively simple to explain, there is a lot of power in those few ideas. Together, they allow you to build systems that are dynamic, flexible, and robust, and to build them out of many parts, created independently by many providers.

This book contains the formal specifications for the Jini technology, preceded by this introductory part that gives you an overview of the design and basic usage. The specifications that follow give you the details that make this flexibility possible. Each specification has a brief introduction that places it in context.

In this section you will find discussion of several examples. Some of these will come from standard office environments and talk about printers, fax

machines, and desktop systems. But others will come from less traditional networking environments: home entertainment systems, cars, and houses. These environments are quickly becoming networked, and Jini systems, with their relatively small size, are ideal for such use.

1.1 Goals

The Jini architecture is designed to allow a service on a network be available to anyone who can reach it, and to do so in a type-safe and robust way. The goals of the architecture are:

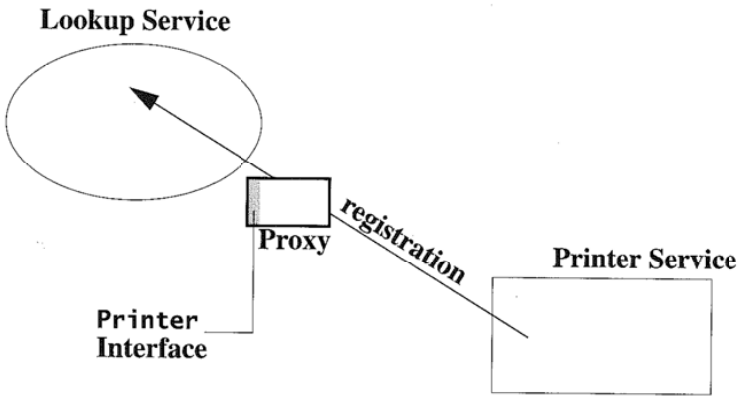
- **Network plug-and-work:** You should be able to plug a service into the network and have it be visible and available to those who want to use it. Plugging something into a network should be all or almost all you need to do to deploy the service.
- **Erase the hardware/software distinction:** You want a service. You don't particularly care what part of it is software and what part is hardware as long as it does what you need. A service on the network should be available in the same way under the same rules whether it is implemented in hardware, software, or a combination of the two.
- **Enable spontaneous networking:** When services plug into the network and are available, they can be discovered and used by clients and by other services. When clients and services work in a flexible network of services, they can organize themselves in the most appropriate way for the set of services that are actually available in the environment.
- **Promote service-based architecture:** With a simple mechanism for deploying services in a network, more products can be designed as services instead of stand-alone applications. Inside almost every application is a service or two struggling to get out. An application lets people who are in particular places (such as in front of a keyboard and monitor) use its underlying service. The easier it is to make the service itself available on the network, the more services you will find on the network.
- **Simplicity:** We are aesthetically driven to make things simple because simple systems please us. Much of our design time is spent trying to throw things out of a design. We try to throw out everything we can, and where we can't throw something out, we try to invent reusable pieces so that one idea can do duty in many places. You benefit because the resulting system is easier to learn to use and easier to provide systems in. Being a well-behaved Jini service is relatively simple, and much of what you need to do can be auto-

mated by other tools, leaving you with a few necessary pieces of work to do. Equally important, a large system built on simple principles is going to be more robust than a large complicated system.

1.2 Architecture

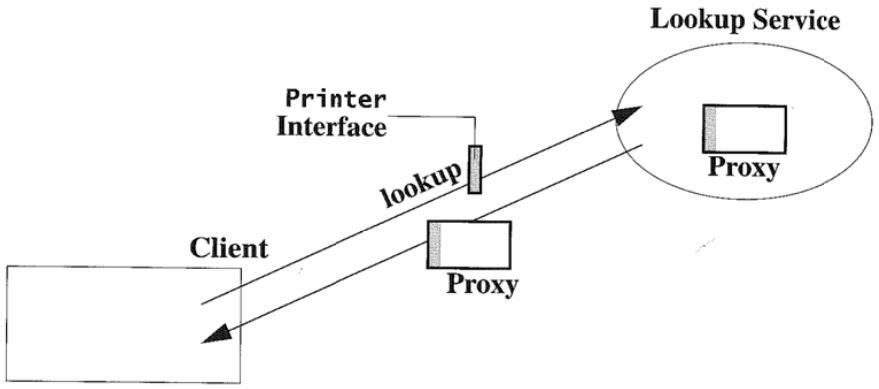
Each Jini system is built around one or more *lookup* services. The lookup service is where services advertise their availability so that you can find them. There may be one or more lookup services running in a network.

When a service is booted on the network, it uses a process called *discovery* to find the local lookup services. The service then registers its *proxy* object with each lookup service. The proxy object is a Java object, and its types—the interfaces it implements and its superclasses—define the service it is providing. For example, a proxy object for a printer will implement a `Printer` interface. If the printer is also capable of receiving faxes, the proxy object will also implement the `FaxReceiver` interface.



A client program asks for services by the Java language type the client will use. A client wanting a printer will ask the lookup service for a service that implements the `Printer` interface. When the lookup service returns the printer's proxy

object, the client will automatically download the code for that object if it doesn't have it already.



The client issues printer requests by invoking methods on the proxy object. The proxy communicates with the printer as it needs to in order to execute the requests. The Jini system does not define what the protocol between the proxy and its service should be; that is defined by the printer and its proxy object.

In fact, the proxy may talk to any number of remote systems to implement a single method, including zero. Whoever writes the proxy object determines when it talks to whom to get what, constrained, of course, by the security environment in which it executes. As long as the proxy object provides the services advertised by its interfaces and/or classes, the client will be satisfied. This encapsulation is one of the basic powers of object-oriented programming. The invoker of a method cares only that the method implementation does what is expected, not how it does it. The proxy object in a Jini system extends the benefits of this encapsulation to services on the network.

In effect, the proxy object is a driver for the printer that is downloaded on demand. This allows a client to speak to a kind of printer it has never before encountered without any human having to install the printer's driver on the client's computer. When the driver is needed, it is downloaded. When it is no longer needed, it can be disposed of automatically.

1.3 What the Jini Architecture Depends Upon

The Jini architecture relies upon several properties of the Java virtual machine:

- **Homogeneity:** The Java virtual machine provides a homogeneous platform—a single execution environment that allows downloaded code to behave the same everywhere.
- **A Single Type System:** This homogeneity results in types that mean the same thing on all platforms. The same typing system can be used for local and remote objects and the objects passed between them.
- **Serialization:** Java objects typically can be serialized into a transportable form that can later be deserialized.
- **Code Downloading:** Serialization can mark an object with a codebase: the place or places from which the object's code can be downloaded. Deserialization can then download the code for an object when needed.
- **Safety and Security:** The Java virtual machine protects the client machine from viruses that could otherwise come with downloaded code. Downloaded code is restricted to operations that the virtual machine's security allows.

Taken together, these properties mean that objects can be moved around the network in a consistent and trustable manner. These properties enable a system built on dynamic service proxies moving object state and implementation to the most useful parts of a system when they are needed. Such proxies are part of the foundation on which the Jini architecture is built.

1.4 The Value of a Proxy

The proxy object is central to the benefit of using a Jini system. The proxy defines a service type by being of a particular Java type. It implements that type in whatever way is appropriate for the service implementation that registered it. This is basic object-oriented philosophy: You know *what* the object does because you know its Java language type, but you don't know *how* it implements the methods

defined by that type. The proxy is the part of the service that runs in the client's virtual machine.

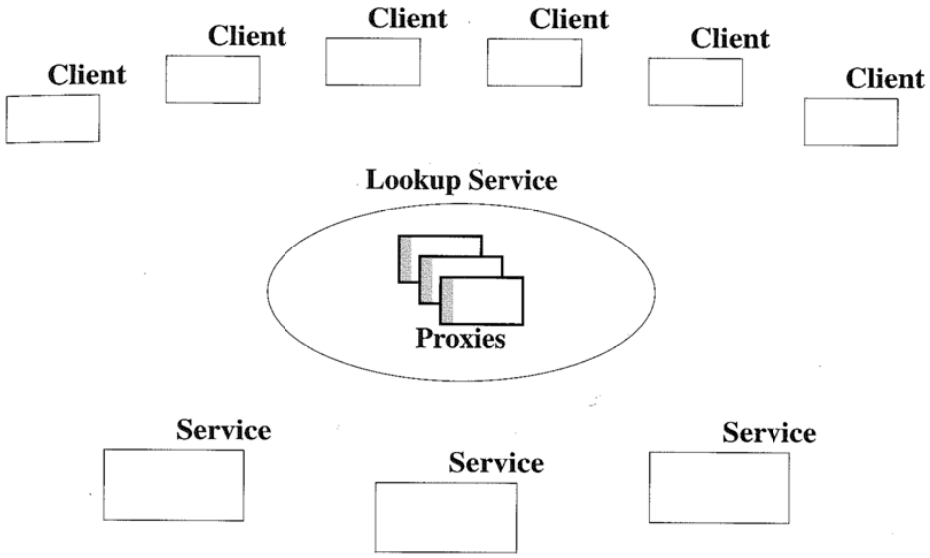
This encapsulation allows the `Printer` interface to be designed as a good client API without requiring it to be a good network protocol for talking to a remote printer. The `Printer` interface should be designed at the abstraction level appropriate for client code. Each proxy object that implements the `Printer` interface does so in the right way for the particular printer, using that printer's network protocol. While it is very useful for everyone to agree on the design of the `Printer` interface, nobody needs to agree on the network protocol. The `Printer` interface's `printText` method would be implemented differently for a PostScript printer than for one that had a different printer language. The proxy object encapsulates such differences so the client can simply invoke the method.

And anyone can write a proxy object. If the printer manufacturer does not provide a Jini service proxy, you can write your own or buy one from someone else. As long as the proxy correctly implements the appropriate interface it is a valid proxy for the printer. If your use of a Jini system relies upon, say, a video camera, and the camera's manufacturer hasn't yet provided a proxy implementation you need, you can write it yourself or find someone else who has already done so. This works for integration of legacy services of any kind, not just devices. An existing database server can be made available through a Jini service's proxy, usually without modifying the server.

The service defines where the proxy code is loaded from. This allows the service to be its own HTTP server for its classes or to rely on an HTTP server somewhere else in the network. The service can, in fact, be unrelated to the hardware and software on which it is based. A service might, for example, be built from a server that monitors the network for some legacy hardware and when the hardware is present, registers a proxy on that hardware's behalf, unregistering the service when the hardware is disconnected. In such a model the service is completely uncoupled from the hardware on which it relies.

1.5 The Lookup Service

Each lookup service provides a list of available services, the proxy objects that know how to talk to the service, and attributes defined by either the local administrator or the service itself.



When a service is first booted up, it uses a *discovery* protocol to find local lookup services. This protocol will vary depending upon the kind of network, but its basic outline is:

- The service sends a "looking for lookup services" message to the local network. This is repeated for some period of time after initial startup.
- Each lookup service on the network responds with a proxy for itself.
- The service registers with each lookup service using its proxy by providing the service's proxy object and any desired initial attributes.

A client that wants a service goes through a matching protocol:

- The client sends a "looking for lookup services" message to the local network.
- Each lookup service in the network responds with a proxy for itself.

- The client searches for types of services it needs using the proxies of one or more lookup services. The lookup service returns one or more matching proxy objects, whose code is downloaded to the client if necessary.

The discovery protocol is how services and clients find nearby lookup services. A client or service can also be configured to locate specific lookup services as well as (or instead of) ones discovered on the local network. For example, when you plug in your laptop in a hotel, you might want not only to find the lookup service for your hotel room, but also to contact the lookup service in your home so you can interact with services there (such as programming the "Call Me" button on your home's telephone to call your hotel and ask for your room). Once a lookup service is located, rather than discovered, the registration and lookup steps are the same for service and client.

Matching in the lookup service is performed using standard Java language typing rules. If you ask for `Printer` objects, you will get only objects that implement the `Printer` interface. The actual object you get may also implement other interfaces, including subinterfaces of `Printer`, such as `ColorPrinter`. As with any other object you can check to see what types it supports. For example, you could check to see whether the `Printer` proxy implements the `ColorPrinter` interface, printing in color if it does, and otherwise printing in black and white.

Sometimes a service will be attached to a network when no lookup service can be found, for example in a broken network. The service's "looking for lookup services" message will therefore not reach the lookup service, and so the service cannot register. When the network is repaired, the service will be available but invisible. In order that this invisibility be temporary, each lookup service intermittently sends a "here I am" message to the network. When a service gets such a message, it registers with that lookup service if it isn't currently registered.

1.5.1 Attributes

When you look up an object by type you will get an object with the capabilities you need, but it might not be the one you want. If you have two television sets in your house connected on one network, you will want to connect your VCR to the one you are about to watch. Both televisions will be `VideoDisplay` objects, so how do you distinguish between them?

Each proxy object in the lookup service can have *attributes*. These are objects that describe features relevant to distinguish one service from another in ways that are not reflected by the interfaces supported by the service. These often reflect ways to choose among services of the same type but are different in some way that is important to a human. In a home entertainment service, naming each television set by its location is probably enough—you can set the VCR to send its output to

the `VideoDisplay` object with the `Name` attribute `"living room"`. In an office environment you might use `Location` attributes to help you choose the printer that is near your office, not at the other end of the hallway.

The Jini architecture does not define which attributes a service should have. The local administrator will decide which attributes are helpful in the local environment, and the service designer will decide which ones help users and clients find the right service. The Jini architecture does define a few example attributes in the package `net.jini.lookup.entry` as suggestions, but whether to use these, or others, or none, is up to service designers and local administrative policies.

An attribute is an object that is an *entry,* that is, it must implement the interface `net.jini.core.entry.Entry`, and have the associated semantics, which are:

- All non-static, non-transient, non-final fields must be public.
- Each field must be of an object type, not a primitive type (`int`, `char`, ...).
- The class must be public and have a public no-arg constructor.

An entry may have other kinds of fields, but they will not be saved when an attribute (entry) is stamped on a proxy or considered when matching attributes in lookup requests.

Attribute matching is done with simple expressions that use exact matching. You can say one of two things about an attribute: You require an attribute of that class (including a subclass) to be stamped on the proxy, or you don't care. Within each attribute you require, you can say a similar thing about each field: You require the field to have exactly some value or you don't care about its value. If you specify more than one attribute, the lookup service will return only proxies that match all the attributes you specify.

Attributes are properties of the service, not of its proxy in each individual lookup service. A service will have the same attributes in all lookup services in which it is registered (although network delays may allow you to see inconsistent sets of attributes in different lookup services while the service is updating its registrations).

1.5.2 Membership Management

When a service registers with a lookup service, it gets back (among other things) a *lease* on its presence in the lookup service. Leases are a programming model within the Jini architecture designed to allow providers of resources to clean up when the resource is no longer needed. In the lookup service case, for example, the lease keeps the list of available services fresh—as long as a service is up and

running, it will renew its lease. If the service crashes or the network between the service and the lookup service breaks, the service will fail to renew its lease and thus be evicted from the lookup service.

This means that the list of services you find in a lookup service is a list of services that are available to you, modulo the time allowed by the lease. For example, if the lease time given to services by the lookup service (both initially and upon renewal) is five minutes, each service you see in the lookup service spoke to the lookup service within the last five minutes. Most lookup service implementations will let you tune this time to your required tolerances.

When combined with discovery of lookup services, the leased membership gives a powerful result: The list of services is current, self-healing, and self-replicating:

- It is current (modulo the lease times) because the leases make it so. Any network or host failure will force the removal of unreachable services.
- It is self-healing because if a network failure isolates a service from a lookup service, when the network is fixed, the service will receive a "here I am" message from the lookup service and rejoin.
- It is self-replicating because a service joins each lookup service it belongs to. If you want replication to increase robustness, just start another lookup service. All the services will simply register with both lookup services. If the only host running your lookup service crashes, just start a new one on a new host, and all the services will register with the new lookup service.

These features work together. If you run two lookup services on different hosts and the network between them fails, after the leases expire each will have the available services on its part of the network. When the network is fixed, each lookup service's "here I am" message will reconnect it with the services that were lost.

1.5.3 Lookup Groups

The discovery request may encounter many lookup services, but you might want a service to be visible in only a few of them. For example, if you have a lookup service that represents those services available to users of a conference room (fax machine, printer, projector, telephone, web server), you do not want those services available as default resources for the people who sit in offices next to the conference room. Nor do you want the people in the conference room to accidentally use a printer down the hall.

To limit a lookup service's scope, you place the lookup service in the conference room in its own *group* and configure each of the room's services to join only lookups in that group. The lookup discovery messages include the groups of the parties involved. Lookup services ignore discovery messages that are for groups they are not in, and services ignore "here I am" messages of lookup services in groups they are not configured to join. So when new services are added to the neighborhood, they will not be registered in the conference room's lookup service unless they are explicitly configured to join lookups in the right group.

1.5.4 Lookup Service Compared to Naming/Directory Services

A lookup service in a Jini system is the nexus where clients locate network services. In this sense its role is analogous to what are called naming or directory services in other distributed systems. The analogy is real, but it fails at some crucial junctures. In discussing the failures of the analogy we will use the term "naming services" to mean both naming and directory services, which are equivalent for this discussion.

In a directory system, services are stored by name, a human-readable string. The string is split up by conventional symbols that separate the components. For example, all printers may be stored under the directory `"/devices/printers"`. If you want to see the printers that are available in the directory service, you ask it for all the references to remote objects in this directory. Each installed printer will be placed in the directory when it is installed.

This system starts becoming unwieldy as you increase the number of services and their types. Color printers, for example, might be placed in the printers' directory, or possibly in a separate `"/devices/printers/color"` directory, or both so that people finding regular printers can find color printers, which after all can also be used as printers. Printers that are also fax machines would certainly be placed in at least two directories, since nobody would think to look for a fax machine in the printers' directory.

Also, note that the correlation between `"/devices/printers"` and print services is purely conventional. Should someone mistakenly place a fax service in the directory, clients will get very confused when the remote reference they get back is not actually a printer.

To find a service in a directory-based system, your client does the following:

1. Takes a string that is bound by convention to printers.
2. Asks the directory service what it has bound under that string.

3. Takes what it gets back and tries to use it as a `Printer` object (in the Java programming language this would be by casting it to the type `Printer` after checking, if you want a robust program, to be sure that it *is* a `Printer`).

Because the strings in a directory service are related only by convention to the type you need, failures to follow convention lead to errors for the client. The human-readable strings are actually of no value to the client except as a (risky) means to an end. The Jini Lookup service architecture gives your client a way to get at that end directly:

1. Asks the lookup service for a `Printer` object.
2. Takes the `Printer` object it gets back and uses it.

This directness also provides the benefits of object-oriented polymorphism: The object you get back will be at least a `Printer`, but it may in addition be something more: a `ColorPrinter`, possibly, or a `FaxSender`, `FaxReceiver`, or `Scanner`. You can use it as a `Printer` without regard to these extra capabilities, or you can test for their presence using the `instanceof` operator in the language.

People want to name things, of course. Most computers, printers, and other major systems in network are named. In a Jini system those names are attributes on the service that help humans distinguish between services. As attributes, names can be used to distinguish between services of identical type, but the primary mechanism a program uses to find services is the thing the program most cares about: the type of the service it will use.

1.6 Conclusion

The Jini architecture provides a platform for deploying services in a network. This platform is robust at many levels:

- It is robust in the face of network failures. The set of services automatically adapts the actual state of the network and service topology.
- It is robust in the face of changes in the implementation of services. As long as the service interface is implemented correctly, the details of the service implementation can change as you buy new equipment and as equipment generally becomes more capable.
- It is robust in the face of old services. It is relatively easy to incorporate old devices and servers seamlessly instead of leaving them as an impediment to progress.

- It is robust in the face of competition. The minimum standards necessary for cooperation are defined in the architecture—the definition of what defines a service (a Java language type) and how you find a service (in a lookup service)—and lets variation exist where it needs to. An industry can standardize on common ground (such as the basic `Printer` interface) and individual companies can add specific features in company-specific interfaces (such as `MyCompanysPrinter`) for clients that want to use them, without breaking generic clients that want only the common `Printer` functionality.
- It is robust in the face of scale. Jini services can be very large or very small, and can work with small devices via a supporting virtual machine.

The Jini architecture is not only robust, it is also flexible. Here are sketches of a few ways in which it can be used.

- You could design a kiosk that allowed the user to download information. For example, I might plug my PDA (personal digital assistant) into the kiosk and ask for directions to someplace. The kiosk can publish the information as a simple `TextPublisher` service which I would use to download the directions onto a text device such as a pager, as well as an `HTMLPublisher` service which I would use to download them onto a more capable device, such as a laptop computer.
- You could have expense sources (such as a taxi meter or credit card scanner) provide an `ExpenseSource` service that my PDA could use to download travel expense details. When I return to my office, my PDA could be its own `ExpenseSource` service that my spreadsheet or company expense report software could use as a source for expense report information.
- You could make sensors in a water supply system be Jini services and have several monitoring and report-generating applications adapt automatically to new sensors that are added to the network. Adding a new sensor would then be as simple as plugging it into the network: The monitoring applications would find the new service and incorporate it into the data flow. New "sensors" could be software services that aggregate and analyze information from sensors into higher-level data. The clients will be blissfully unaware of this hardware-software distinction.

These examples suggest the flavor of the benefits you can find using Jini technology. The example code that follows introduces you to the design of Jini clients and services. The specification that comes afterwards give you the details.

1.7 Notes on the Example Code

In the following two sections you will see an example service, an example client that uses that service, and two example implementations of that service. There are a few things you should know before we get started.

First, we have kept the examples as simple as possible. This means, for example, that we are using command line programs instead of graphical user interfaces. Graphical user interfaces require a good deal of programming, and explaining that part of the code would teach you nothing about using the Jini technology. We have also used very simple error-checking and handling except where more sophisticated techniques help us explain how you should use the Jini architecture.

We have also not shown some parts of the code that do not explain anything about programming in a Jini system—file system manipulation, string parsing, and so on. The full code for all the examples is in Appendix B.

1.7.1 Package Structure

The Jini technology is expressed in Java language interfaces and classes that live in three major package categories:

- `net.jini.core`: Standard interfaces and classes that are central ("core") to the Jini architecture live in subpackages of `net.jini.core`.
- `net.jini`: Interfaces and classes that are standards in the Jini architecture are in subpackages of `net.jini` (except the `net.jini.core` subpackage).
- `com.sun.jini`: Some interfaces and classes that are non-standard but potentially useful live in the subpackages of `com.sun.jini`. These packages may contain utility classes that help you write clients and services, example implementations of standard services, or utility classes used inside the example implementations.

As an example, there are actually three separate `lookup` packages:

- `net.jini.core.lookup`: The interfaces and class that comprise the lookup service that is at the heart of the Jini architecture.
- `net.jini.lookup`: An interface (`DiscoveryAdmin`) that lookup services can support to allow administrators to configure which lookup groups the service will be a member of. This interface is advisory but standard: you need not use it, but it is a common, traditional way to enable such changes.

- `com.sun.jini.lookup`: A utility class (`JoinManager`) that helps service implementations to manage registration with appropriate lookup services.

These packages progress from the core (the lookup service itself) to the standard (defined, though optional, ways to administer a lookup service) to the extended (useful utilities you may choose to use). Broken out these ways, the packages are:

- `net.jini.core.discovery`: A class (`LookupLocator`) that connects to a single lookup service
- `net.jini.core.entry`: The `Entry` interface that defines attributes
- `net.jini.core.event`: The interfaces and classes for distributed events
- `net.jini.core.lease`: The interfaces and classes for distributed leases
- `net.jini.core.lookup`: The interfaces and classes for the lookup service
- `net.jini.core.transaction`: The interfaces and classes for the clients of the transaction service
- `net.jini.core.transaction.server`: The interfaces and classes for the manager and participants in the transaction service
- `net.jini.admin`: Some standard administrative interfaces for services
- `net.jini.discovery`: Some standard utility classes that help clients and service implementations with the discovery protocol
- `net.jini.entry`: A useful base utility class (`AbstractEntry`) for entry (attribute) classes
- `net.jini.lookup`: A standard administrative interface (`DiscoveryAdmin`) for lookup services
- `net.jini.lookup.entry`: Some standard attribute interfaces and classes you can use
- `net.jini.space`: The interfaces and classes that define the JavaSpaces technology
- `com.sun.jini.admin`: Interfaces for administering some common service necessities
- `com.sun.jini.discovery`: A utility class (`LookupLocatorDiscovery`) that helps you contact specific lookup services
- `com.sun.jini.lease`: Some utility classes that may help your client manage the leases that it gets from services (such as a lookup service)
- `com.sun.jini.lease.landlord`: Some utility classes that may help your service implement and manage the leases it exports to its clients

- `com.sun.jini.lookup`: A utility class (`JoinManager`) to help your service implementation discover and join lookup services in the network, and manage its attributes in those lookup services
- `com.sun.jini.lookup.entry`: Some utility classes for working with lookup service attributes.

Other `com.sun.jini` classes exist. We have listed here the ones that you are most likely to find valuable in implementing your own clients and services.

As you will notice, we have taken a fine-grained approach to package structure—we make each package contain only related interfaces and classes. This leads to many well-focused packages instead of a few packages with many loosely related interfaces and classes. As the Jini architecture evolves, other packages will be added to this list. The notions of "core," "standard," and "extended" are currently mapped directly to package names. Future additions might not be able to follow this. For example, if a standard evolves that becomes core to the Jini architecture it could be viewed as "core" without renaming the package with a `net.jini.core` name. Such decisions are still in the future, and we cannot yet define a fixed policy until we have examples to consider.

You will see code from many of these packages in our example code. We will name the package of each Jini architecture interface or class when it first appears. The packages of the example classes themselves will be described at the beginning of the example. To keep the code to a reasonable size for the text, we will not show the import statements in the chapters. The full source (including import statements) is in Appendix B.

2 Writing a Client

A successful [software] tool is one that was used to do something undreamed of by its author.
—S.C. Johnson

LET'S make this architecture more concrete, first by showing how you would write a client that uses the Jini architecture. The next section will show how you would write two corresponding service implementations that are usable by this client. We will first describe the service being performed.

2.1 The `MessageStream` Interface

The example interface `MessageStream` provides an iterator through a stream of messages. It provides one method that returns the next message in the stream:

```
package message;

public interface MessageStream {
    Object nextMessage()
        throws EOFException, RemoteException;
}
```

The `nextMessage` method returns the next message as an object whose `toString` method prints out its default printed form. An `EOFException` signals the end of the stream. A `RemoteException` reflects failures in network messaging.

This simple interface could be used for many situations; in the next section we will show two: a "fortune cookie" service that returns a random saying, and a chat service whose messages are the utterances of the speakers in the discussion. Because the stream interface is general, the client that reads it can work with any type of message stream. The implementations of each stream will vary, but the client can do the same thing.

Our example client will simply find a user-specified stream and print out the requested number of messages. Other general clients could be fancier in many ways. In fact, many design features of our example client and service implementa-

tions are optimized for simplicity to keep the focus on the relevant Jini architecture and technology. You will see command line applications instead of graphical user interfaces, basic choices available rather than rich ones, and simple error handling. These simplifying choices help teaching by keeping the focus on the relevant parts of the code, even if they are sometimes unrealistic for product design (although simple choices for products are very often correct ones, too). The complete code for all examples is in Appendix B.

2.2 The Client

Now let's look at how you would write a client that finds and uses a message stream. Your users will need to give you enough information to pick the correct stream from among the available streams. Our example client allows the user to specify:

- Lookup groups that will be used in discovery or a specific lookup service
- The type of the service
- Attributes to use in selecting the service

The client bundles the service type and attribute information into a search template, queries the appropriate lookup services to find a matching service, and prints out one or more messages.

We will examine the client from the top down. Parts of the code that have little to do with learning the Jini architecture have been left out of the code presented here. The complete source to all examples is in Appendix B.

The command line syntax looks like this:

```
java [java-options] client.StreamReader [-c count]
     [groups|lookup-url] [stream-type|attributes ...]
```

The *java-options* will typically include setting a security policy file. The name of our client class is `client.StreamReader` (the `StreamReader` class in the `client` package). The `-c` option lets the user specify a count of messages to read; the default is one message. The user must choose from the set of lookup services by providing either a group specification for lookup discovery or an explicit lookup *locator*, which specifies a particular lookup service by its URL, which has the form `jini://host[:port]`. The user may also specify a type of stream, which must be a subtype of `MessageStream`, and/or a list of attributes. To simplify parsing, attributes are specified by either their type name, or their type name and a `String` parameter for the constructor. This means that only attributes with

no-arg constructors or with single-argument `String` constructors can be used with `StreamReader` (a fancier client could let the user specify a richer set of attributes.)

A typical invocation might look like this:

```
java -Djava.security.policy=/policies/policy
     client.StreamReader "" fortune.FortuneStream
     fortune.FortuneTheme:General
```

In this invocation the group will be the empty string, which is the name of the public group; the type of the stream must be at least `fortune.FortuneStream`; and the registration in the lookup service must at least have an attribute of the type `fortune.FortuneTheme` that matches an attribute created with the string `"General"`. We will discuss the `fortune` package types when we show how the service is written.

When a user invokes the client command line, the `main` method of the class `client.StreamReader` will be invoked:

```
package client;

public class StreamReader implements DiscoveryListener {
    private int count;
    private String[] groups = new String[0];
    private String lookupURL;
    private String[] typeArgs;

    public static void main(String[] args) throws Exception
    {
        StreamReader reader = new StreamReader(args);
        reader.execute();
    }

    //...
}
```

The `main` method simply creates a `StreamReader` object with the command line arguments and then invokes the object's `execute` method. The `StreamReader` constructor parses the command line to set the fields `count`, `groups`, `lookupURL`, and `typeArgs`. This parsing is shown only in the full source.

The execute method starts discovering lookup services:

```
public void execute() throws Exception {
    if (System.getSecurityManager() == null)
        System.setSecurityManager(new RMISecurityManager());

    // Create lookup discovery object and have it notify us
    LookupDiscovery ld = new LookupDiscovery(groups);
    ld.addDiscoveryListener(this);

    searchDiscovered(); // search discovered lookup services
}
```

First we set a security manager to protect the client against misbehaving downloaded code. RMI requires a security manager to be in place during calls to ensure that you have thought about the security aspects of the code it will download. This code uses the RMISecurityManager, which is quite conservative about what it permits.

LookupDiscovery is a utility class that you can use to help you perform the lookup discovery protocol. It lives in the net.jini.discovery package. Each LookupDiscovery object starts a thread that notifies listeners when new lookup services are discovered or when known ones have gone away. We create a LookupDiscovery object and tell it that this StreamReader object is a listener. Once this is set up, we will have two threads of control running in parallel: the main thread in which execute was invoked and a separate thread in which LookupDiscovery will invoke callback methods. Our implementation uses a simple model to coordinate these threads—the registrars field contains a list of known net.jini.lookup.ServiceRegistrar objects (the main interface for the lookup service).

LookupDiscovery does its callbacks via the DiscoveryListener interface (also in the net.jini.discovery package), which declares the methods discovered and discarded. We use these methods to maintain the registrars list:

```
public synchronized void discovered(DiscoveryEvent ev) {
    ServiceRegistrar[] regs = ev.getRegistrars();
    for (int i = 0; i < regs.length; i++)
        registrars.add(regs[i]);
    notifyAll(); // notify waiters that the list has changed
}

public synchronized void discarded(DiscoveryEvent ev) {
```

```
        ServiceRegistrar[] regs = ev.getRegistrars();
        for (int i = 0; i < regs.length; i++)
            registrars.remove(regs[i]);
        notifyAll(); // notify waiters that the list has changed
    }
```

Each invocation of `discovered` represents one or more newly discovered lookup services. Our implementation gets the array of `ServiceRegistrar` objects (the lookup service's primary interface) and adds each to the list of known registrars. When it is complete, it invokes `notifyAll` in case `searchDiscovered` is blocked waiting for the list to have some elements. Our `discarded` implementation removes elements from the list.

The `searchDiscovered` method invoked by `execute` loops checking out members of that list until it finds a matching service or until `MAX_WAIT` milliseconds have passed:

```
    private List registrars = new LinkedList();

    private final static int MAX_WAIT = 5000;   // five seconds

    private synchronized void searchDiscovered()
        throws Exception
    {
        ServiceTemplate serviceTmpl = buildTmpl(typeArgs);

        // Loop searching in discovered lookup services
        long end = System.currentTimeMillis() + MAX_WAIT;
        for (;;) {
            // wait until a lookup is discovered or time expires
            long timeLeft = end - System.currentTimeMillis();
            while (timeLeft > 0 && registrars.isEmpty()) {
                wait(timeLeft);
                timeLeft = end - System.currentTimeMillis();
            }
            if (timeLeft <= 0)
                break;

            // Check out the next lookup service
            ServiceRegistrar reg =
                (ServiceRegistrar)registrars.remove(0);
            try {
                MessageStream stream =
```

```
                (MessageStream)reg.lookup(serviceTmpl);
            if (stream != null) {
                readStream(stream);
                return;
            }
        } catch (RemoteException e) {
            continue;               // skip on to next
        }
    }
    System.err.println("No service found");
    System.exit(1);          // nothing happened in time
}
```

First the method uses the command line arguments to build up a template. It then starts looping. Each time through the loop the list of registrars is checked. If it is empty, we wait until either the remaining time expires or the list ceases to be empty. During the invocation of `wait` the `discovered` method can be invoked by `LookupDiscovery` in its thread, adding registrars to the list. When registrars are added, the `notifyAll` in the `discovered` method will allow the `wait` in `searchDiscovered` to return. The code in `searchDiscovered` then takes the first element from the list and asks it to look up a service that matches our template. If it finds one, it asks `readStream` to try and read messages from the stream (you will see `readStream` shortly).

If `readStream` executes successfully, `searchDiscovered` will return, which signals successful execution. If `searchDiscovered` does not find a readable stream within the allotted time, it prints out an error message and exits with a non-zero status, indicating failure of the command.

The `buildTmpl` method creates the `net.jini.lookup.ServiceTemplate` object that is passed to the lookup service's `lookup` method. Let's look at how the template is built:

```
private ServiceTemplate buildTmpl(String[] typeNames)
    throws ClassNotFoundException, IllegalAccessException,
           InstantiationException, NoSuchMethodException,
           InvocationTargetException
{
    Set typeSet = new HashSet();     // service types
    Set attrSet = new HashSet();     // attribute objects

    // MessageStream class is always required
    typeSet.add(MessageStream.class);
```

```
        for (int i = 0; i < typeNames.length; i++) {
            // break the type name up into name and argument
            StringTokenizer tokens =    // breaks up string
                new StringTokenizer(typeNames[i], ":");
            String typeName = tokens.nextToken();
            String arg = null;          // string argument
            if (tokens.hasMoreTokens())
                arg = tokens.nextToken();
            Class cl = Class.forName(typeName);

            // test if it is a type of Entry (an attribute)
            if (Entry.class.isAssignableFrom(cl))
                attrSet.add(attribute(cl, arg));
            else
                typeSet.add(cl);
        }

        // create the arrays from the sets
        Entry[] attrs = (Entry[])
            attrSet.toArray(new Entry[attrSet.size()]);
        Class[] types = (Class[])
            typeSet.toArray(new Class[typeSet.size()]);

        return new ServiceTemplate(null, types, attrs);
    }
```

The `buildTmpl` method loops through the type arguments given on the command line. The arguments can be either a type name or, in the case of attributes, a type name followed by a `String` argument to pass to the constructor, of the form *`type(arg)`*. The first part of the loop takes the name and checks to see whether it has an open parenthesis. If it does, it strips any closing parenthesis and remembers the argument in the variable `arg`, which is otherwise `null`. Once any argument has been stripped off from the class name in `cName`, we translate the name into a `Class` object for the type. If the type is assignable to `Entry` it is an attribute, and so an object is created of that attribute type, using `arg` if it was present—the method `attribute` (not shown) does this work. If it is not assignable to `Entry`, it must be a service type, and so we add its type to the types the service must support. When the loop is finished, `typeSet` contains all the required service types and `attrSet` contains all the required attribute templates. We then create appropriate arrays from the contents of these sets and pass the arrays to the

ServiceTemplate constructor (the first null argument would be the service ID if we needed to match on a specific one).

As you have seen, when searchDiscovered finds a matching service, it tries to read the stream by invoking the readStream method:

```
private final static int MAX_RETRIES = 5;

public void readStream(MessageStream stream)
    throws RemoteException
{
    int errorCount = 0;      // # of errors seen this message
    int msgNum = 0;          // # of messages
    while (msgNum < count) {
        try {
            Object msg = stream.nextMessage();
            printMessage(msgNum, msg);
            msgNum++;                // successful read
            errorCount = 0;          // clear error count
        } catch (EOFException e) {
            System.out.println("---EOF---");
            break;
        } catch (RemoteException e) {
            e.printStackTrace();
            if (++errorCount > MAX_RETRIES) {
                if (msgNum == 0)     // got no messages
                    throw e;
                else {
                    System.err.println("too many errors");
                    System.exit(1);
                }
            }
            try {
                Thread.sleep(1000); // wait 1 second, retry
            } catch (InterruptedException ie) {
                System.err.println("---Interrupted---");
                System.exit(1);
            }
        }
    }
}
```

```
public void printMessage(int msgNum, Object msg) {
    if (msgNum > 0) // print separator
        System.out.println("---");
    System.out.println(msg);
}
```

The `readStream` method will try to read the number of messages desired. If `readStream` gets a `RemoteException`, it retries up to `MAX_RETRIES` times, waiting one second (1,000 milliseconds) between each try. If it fails to read even a single message it throws `RemoteException`, letting the loop in `searchDiscovered` continue looking for a usable stream. If it reads at least one message, it prints out its failure and exits, so that the user will not see some messages from one stream and a few more from the next one should a failure occur before the desired number of messages are read.

2.3 In Conclusion

Let us revisit the example execution of `StreamReader` from page 21. If you use that command line, the client will look for a `fortune.FortuneStream` service (an interface that we will define in the next section) with an attribute that is of type `fortune.FortuneTheme` created with the string `"General"`. This search will be conducted in lookup services that manage the public group. If any such lookups are found, the `LookupDiscovery` utility object we created in `execute` will invoke our `discovered` method, which adds it to the list of known lookup services. The `searchDiscovered` method looks in each discovered lookup service for a matching stream, and invokes `readStream` to read one message from a stream and print it out. When all this is complete, you should (assuming there is an available matching fortune cookie service) have a fortune cookie message on your screen.

Again, notice that this client can work with any `MessageStream` service. The user specifies which particular service to use by the service's type and any desired attributes. Each message stream service implementation provides a proxy that works properly for the service's needs. The `StreamReader` client you have seen will print messages from any implementation of a message stream, using the proxy as an adaptor from the service definition (`MessageStream`) to the particular service that was matched (`FortuneStream`, `ChatStream`, or whatever). You will next see how to write two different message stream services that can be used by `StreamReader` or any other `MessageStream` client.

3 Writing a Service

Dare to be naïve.
—R. Buckminster Fuller

THE `MessageStream` interface is designed to work for many purposes. We will now show you two example implementations of a message stream service. The first will be a `FortuneStream` subinterface that returns randomly selected "fortune cookie" messages. The second will provide a chat stream that records a history of a conversation among several speakers. First, though, we must talk about what it means to be a Jini service.

A service differs from a client in that a service registers a proxy object with a lookup service, thereby advertising its services—the interfaces and classes that make up its type. A client finds one or more services in a lookup service that it wants to use. Of course, a service might rely on other services and therefore be both a service and a client of those other services.

3.1 Good Lookup Citizenship

To be a usable service, the service implementation must register with appropriate lookup services. In other words, it must be a good *lookup citizen,* which means:

- When starting, discovering lookup services of appropriate groups and registering with any that reply
- When running, listening for lookup service "here I am" messages and, after filtering by group, registering with any new ones
- Remembering its join configuration—the list of groups it should join and the lookup locators for specific lookup services
- Remembering all attributes stamped on it and informing all lookups of changes in those attributes
- Maintaining all leases in lookup services for as long as the service is available

- Remembering the service ID assigned to the service by the first lookup service, so that all registrations of the same service, no matter when made, will be under the same service ID

3.1.1 The JoinManager Utility

Although the work for these tasks is not a vast amount of labor, it is also more than trivial. Services may provide these behaviors in a number of ways. The utility class `com.sun.jini.lookup.JoinManager` (part of the first release of the Jini Technology Software Kit) handles most of these tasks on a service's behalf, except for the management of storage for attributes and service IDs which the service implementation must provide.

Our example service implementations use `JoinManager` to manage lookup membership. You are not required to do so—you might find other mechanisms more to your liking, or you might want or need to invent your own.

3.2 The FortuneStream Service

Our first example service will extend `MessageStream` to provide a "fortune cookie" service, which returns a randomly selected message from a set of messages. Typically, such messages are intended to be amusing, informative, or inspiring. The collections are often broken up into various themes. The most general theme is to be amusing, but collections drawn from particular television shows, movie types, comic strips, or inspirational speakers also exist. Our `FortuneStream` interface looks like this:

```
package fortune;

interface FortuneStream extends MessageStream, Remote {
    String getTheme() throws RemoteException;
}
```

As with all the classes defined in this example, this interface is in the `fortune` package. The `FortuneStream` interface extends the `MessageStream` interface because it is a particular kind of message stream. `FortuneStream` extends the interface `Remote`, which indicates to RMI that objects implementing the `FortuneStream` interface are accessible remotely using RMI.

The `getTheme` method returns the theme of the particular stream. As you will see, the theme is primarily reflected as an attribute on the service so that a user can

select a FortuneStream with a theme to their liking. The getTheme method is added here to allow queries after a stream has been selected.

Each fortune stream's theme is represented both in the interface via the getTheme method and as an attribute in the lookup service to help users find a stream that gives the types of fortunes they want:

```
public class FortuneTheme extends AbstractEntry
    implements ServiceControlled
{
    public String theme;

    public FortuneTheme() { }

    public FortuneTheme(String theme) {
        this.theme = theme;
    }
}
```

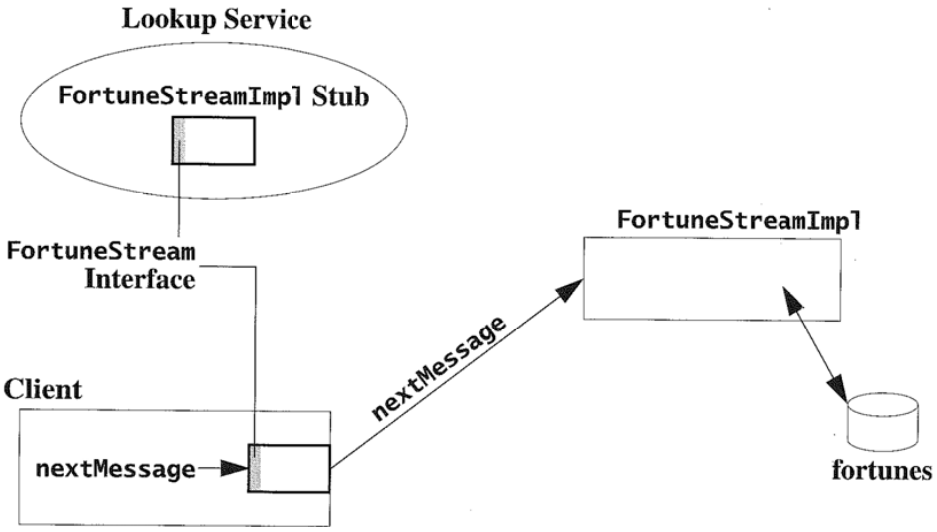
The FortuneTheme attribute is part of the service definition, and is independent of our particular implementation of FortuneStream—a different implementation of FortuneStream would use the same attribute type.

The FortuneTheme attribute fits the requirements for all entries: It has public object-typed fields and a public no-arg constructor. It adds another constructor for convenience. Each FortuneStream service expresses its theme as both a FortuneTheme attribute and a value returned by the FortuneStream class's getTheme method. This redundancy has a purpose—it allows a client of a fortune stream to be written independently of the code that finds the service. For example, it would be possible for a fortune stream client to display the theme of a stream it obtained without using a FortuneTheme attribute.

FortuneTheme extends net.jini.entry.AbstractEntry, which implements Entry and provides useful semantics for entry classes, specifically in defining semantics for the equals, hashCode, and toString methods. Using AbstractEntry is optional—we use it for convenience. FortuneTheme also implements ServiceControlled, which marks the attribute as one that is controlled by the service itself, as opposed to one placed on the service by an administrator. Any tools that let administrators modify attributes should not let ServiceControlled attributes be changed. Only attributes that are exclusively controlled by the service itself should be marked with this interface.

3.2.1 The Implementation Design

The overall fortune service implementation looks like this:



The running service is composed of three parts:

- A database of fortunes, consisting of the collection of fortunes and position offsets for the start of each fortune. The position information is built by reading the fortune collection.
- A server that runs on the same system that contains the database. This server reads the database, choosing a fortune at random each time it needs to return the next message.
- A proxy for the service. This proxy is the object installed in the lookup service to represent the fortune stream service in the Jini system. In this particular case, the proxy is simply a Java RMI stub that passes method invocations directly to the remote server.

3.2.2 Creating the Service

Our `FortuneStream` implementation is provided by the `FortuneStreamImpl` class, which is a Java RMI remote object. Requests for the next message in the

stream will be sent directly to this remote object that will return a random fortune selected from its database.

The fortune database lives in a particular directory, which is set up by a separate `FortuneAdmin` program that creates the database of fortunes from the raw data. The `FortuneAdmin` program is run before the service is created to set up the database a running `FortuneStream` service will use. When the database is ready, you will run `FortuneStreamImpl` to get the service going.

The `FortuneStreamAdmin` command line looks like this:

```
java [java-options] fortune.FortuneAdmin database-dir
```

The *`database-dir`* parameter is the directory in which the database lives. This directory must initially contain a file named `fortunes`, which contains fortunes separated by lines that start with %%, as in:

```
"As an adolescent I aspired to lasting fame, I craved
factual certainty, and I thirsted for a meaningful vision
of human life -- so I became a scientist.  This is like
becoming an archbishop so you can meet girls."
                -- Matt Cartmill
%%
As far as the laws of mathematics refer to reality, they
are not certain, and as far as they are certain, they do
not refer to reality.
                -- Albert Einstein
%%
As far as we know, our computer has never had an undetected
error.
```

The `FortuneAdmin` program creates the position database in that directory if it does not already exist or if it is older than the fortune database file. The position database is stored in a file named `pos`. A typical invocation might look like this:

```
java fortune.FortuneAdmin /files/fortunes/general
```

`FortuneAdmin` will look in the directory `/files/fortunes/general` for a `fortunes` file and will read it to create a `/files/fortunes/general/pos` file.[1] The source to `FortuneAdmin` just manipulates files, so we will not describe it here.

[1] On a Windows system it would be something like `C:\files\fortunes\general`; on a MacOS system it would be more like `Hard Disk:fortunes:general`. We use POSIX-style paths in this book.

3.2.3 The Running Service

The fortune service is started by the `main` method of `FortuneStreamImpl`. The command line looks like this:

```
java [java-options] fortune.FortuneStreamImpl database-dir
     groups|lookup-url theme
```

The *java-options* must include a security policy file and the RMI server codebase URL. The *database-dir* should be the directory given to `FortuneAdmin`. The running service will join lookup services with the given groups or the specified lookup service, with a `FortuneTheme` attribute with the given name. A typical invocation might look like this:

```
java -Djava.security.policy=/file/policies/policy
     -Djava.rmi.server.codebase=http://server/fortune-dl.jar
     fortune.FortuneStreamImpl /files/fortunes/general ""
     General
```

Our implementation of the fortune stream service executes in the virtual machine this command creates, and therefore lives only as long as that virtual machine is running. Later you will see how to write services that live longer than the life of a single virtual machine.

Here is the code that starts the service running:

```
public class FortuneStreamImpl implements FortuneStream {
    private String[] groups = new String[0];
    private String lookupURL;
    private String dir;
    private String theme;
    private Random random = new Random();
    private long[] positions;
    private RandomAccessFile fortunes;
    private JoinManager joinMgr;

    public static void main(String[] args) throws Exception
    {
        FortuneStreamImpl f = new FortuneStreamImpl(args);
        f.execute();
    }

    // ...
}
```

The `main` method creates a `FortuneStreamImpl` object, whose constructor initializes the `groups`, `lookupURL`, `dir`, `theme`, and `initialAttrs` fields from the command line arguments. The rest of the work is done in the object's `execute` method:

```
private void execute() throws IOException {
    System.setSecurityManager(new RMISecurityManager());
    UnicastRemoteObject.exportObject(this);

    // Set up the fortune database
    setupFortunes();

    // set our FortuneTheme attribute
    FortuneTheme themeAttr = new FortuneTheme(theme);
    Entry[] initialAttrs = new Entry[] { themeAttr };

    LookupLocator[] locators = null;
    if (lookupURL != null) {
        LookupLocator loc = new LookupLocator(lookupURL);
        locators = new LookupLocator[] { loc };
    }
    joinMgr = new JoinManager(this, initialAttrs,
        groups, locators, null, null);
}
```

First `execute` sets a security manager, as you saw done in the client. Next we export the `FortuneStreamImpl` object as an RMI object. Specifically, we export the object as a `UnicastRemoteObject`, which means that as long as this virtual machine is running, the object will be usable remotely. When the virtual machine dies, the remote object that it represents dies too. RMI provides a mechanism for activatable servers that will be restarted when necessary; most Jini software services are actually best written as activatable services. You will see an activatable service in the next example.

We then call `setupFortunes` to initialize this server's use of its fortune database. We do not show the code for that here because it is not relevant to the example; `setupFortunes` sets the `positions` and `fortunes` fields that are used by the implementation of `nextMessage`.

The next two lines create the service-owned `FortuneTheme` attribute that will identify the theme of this fortune stream in the lookup service. Then we create the `JoinManager`, which manages all the interactions with lookup services in the net-

work. To do so, you must tell the JoinManager several things. The constructor used by execute (there are others) takes the following parameters:

- The proxy object for the service. We use this because RMI will convert this to the remote stub for the FortuneStreamImpl object, which is what we want in this case. (FortuneStreamImpl implements a Remote interface—FortuneStream extends Remote—so when a FortuneStreamImpl object is marshalled, it gets replaced by its stub.)
- An Entry array that is the initial set of attributes to be associated with the service. Here we provide an array that contains only our FortuneTheme.
- A String array that is the initial set of lookup groups. In our case this will be taken from the command line and be either an array of the groups specified or an empty array if a URL was specified instead.
- A net.jini.discovery.LookupLocator array. LookupLocator is a class that locates lookup services by URL. The array has a LookupLocator for the URL specified, or null if groups were specified instead.
- A com.sun.jini.lookup.ServiceIDListener object. The interface ServiceIDListener provides a method to be called when the service's ID is assigned. This is a hook that lets the service store its ID persistently if it needs to. Since our particular service does not outlive its virtual machine there is no need to store the ID. We therefore pass null, meaning the service will not be notified. (The next example will show this feature in action.)
- A com.sun.jini.lease.LeaseRenewalManager object to manage renewing the leases returned by lookup services. We use null, which tells the JoinManager to create and use its own LeaseRenewalManager. In another situation (for example, exporting multiple services in the same virtual machine) you might want to specify this parameter (in our example, by using the same object in each service's JoinManager to reduce the number of lease manager objects).

When execute is finished we have a service ready to receive messages and, by virtue of its JoinManager, the service registers with all appropriate lookup services and will continue to register appropriately so as long as the service is running. In other words, at this point we have a running Jini service. When execute returns, so does main. RMI will keep the virtual machine running in another thread, waiting to receive requests.

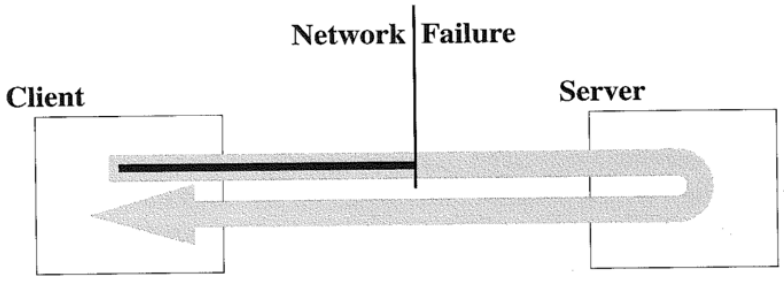
The rest of the code implements nextMessage by picking a random fortune and getTheme by returning the theme field. Again, since these parts show no Jini service code, we leave them to Appendix B.

3.3 The `ChatStream` Service

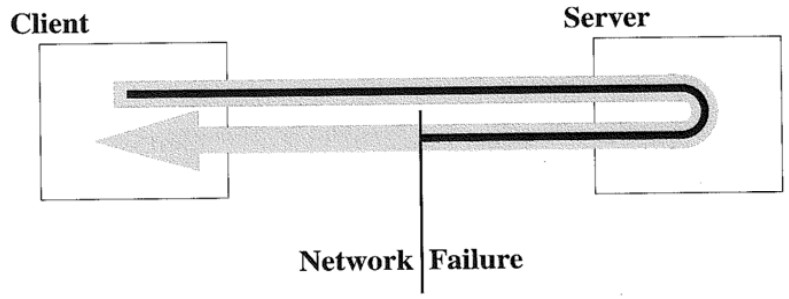
For a more involved example, we provide a message stream whose messages are the utterances of people in a conversation, such as in a chat room. In this case there must be an order to the messages. The fortune stream was picking a message at random, so any message was as good as any other. For a conversation clients will want the messages in the order in which they were spoken.

Consider what happens when `nextMessage` is invoked and a network failure occurs. Either of two interesting situations may have occurred:

- The network failure prevented the request from getting to the remove server:



- The request made it to the remote server, but the network failure blocked the response:



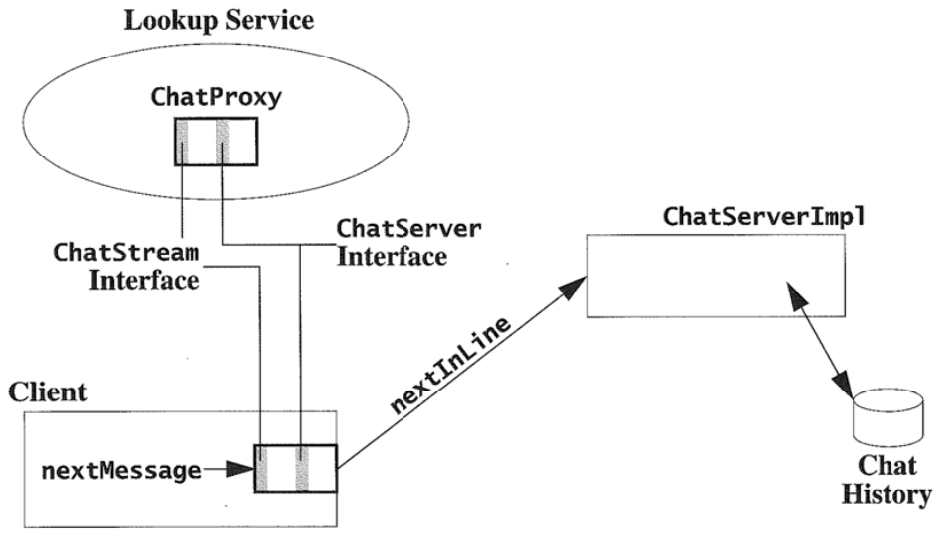
These are very different situations, but the client has no possible way to distinguish between the two cases. If the current position in the stream for each client was stored at the server, the next call to `nextMessage` by the client could return either message 29 (in the first case, in which the server never got the original,

failed request) or message 30 (in the second case, in which the server thought it had returned message 29 but it didn't get to the client).

The `nextMessage` method of `MessageStream` is documented to be *idempotent*, that is, it can be re-invoked after an error to get the same result that would have come had there been no error. For `FortuneStream` idempotency was easy—the fortune was picked at random, so the next message will be equally random, no matter which of the failure situations actually happened.

But for `ChatStream`, this is not good enough. If the proxy was designed naïvely, an utterance might be skipped, and the utterance skipped could be the most important one of the discussion. If a call to `nextMessage` throws an exception because of a communication failure, the next time the client invokes `nextMessage` it should get the same message from the list that it would have gotten on the previous call had there been no failure. Suppose, for example, that we used the same strategy for a `ChatStream` proxy that we did for the `FortuneStreamImpl` proxy—an RMI stub. Then, after getting message number 28 from the server, a network exception is thrown when trying to get message number 29.

So the proxy object registered with lookup services for a `ChatStream` cannot be a simple RMI stub. It must contain enough state to help the service return the right message even in the face of a network failure. To accomplish this, the proxy object will implement the `ChatStream` interface for the client to use, but the server will have an implementation-specific interface that the proxy uses to tell the server which message should be next. It will look like this:



The proxy will
cessfully retriev
interface. That
the ChatStrea
maintain the id
The ChatS
inherits nextMe
ods of its own:

```
package c

public i
    publ

    publ
    publ
}
```

Like all the cc
method lets pe
is the name of
what the subje
These last tw
used to look u
When a m

```
public c
    priv
    priv

    pub

}

pub

pub

pub
```

The proxy will use its internal stored state (the number of the last message successfully retrieved) as an argument to the `nextInLine` method of the `ChatServer` interface. That method is hidden from the client, and different implementations of the `ChatStream` service are welcome to use a different mechanism so long as they maintain the idempotency of `nextMessage`.

The `ChatStream` interface—the public service interface that the clients use—inherits `nextMessage` from the `MessageStream` interfaces, and adds a few methods of its own:

```
package chat;

public interface ChatStream extends MessageStream {
    public void add(String speaker, String[] message)
        throws RemoteException;
    public String getSubject() throws RemoteException;
    public String[] getSpeakers() throws RemoteException;
}
```

Like all the code in this example this class is part of the `chat` package. The `add` method lets people add new messages to the discussion. The `speaker` parameter is the name of the speaker; `message` is what they say. You can ask a `ChatStream` what the subject of the chat is, and for the names of the people who have spoken. These last two things are also stored as attributes of the service so they can be used to look up streams.

When a message is read, it will be a `ChatMessage` object:

```
public class ChatMessage implements Serializable {
    private String speaker;
    private String[] content;

    public ChatMessage(String speaker, String[] content) {
        this.speaker = speaker;
        this.content = content;
    }

    public String getSpeaker() { return speaker; }

    public String[] getContent() { return content; }

    public String toString() {
        StringBuffer buf = new StringBuffer(speaker);
        buf.append(": ");
```

```
            for (int i = 0; i < content.length; i++)
                buf.append(content[i]).append('\n');
            buf.setLength(buf.length() - 1); // strip newline
            return buf.toString();
        }
    }
```

ChatMessage has methods to pick out the pieces of the message—its speaker and the content—and its toString method prints out a reasonable default representation of the message.

When looking for a ChatStream, a user might want to choose the subject, so we define a ChatSubject attribute type:

```
public class ChatSubject extends AbstractEntry
    implements ServiceControlled
{
    public String subject;

    public ChatSubject() { }

    public ChatSubject(String subject) {
        this.subject = subject;
    }
}
```

A ChatStream service should mark itself as being on a certain subject—the same subject that getSubject would return. A user might also want to search for chats that had particular speakers, so a stream should also mark itself with a ChatSpeaker attribute for each speaker:

```
public class ChatSpeaker extends AbstractEntry
    implements ServiceControlled
{
    public String speaker;

    public ChatSpeaker() { }

    public ChatSpeaker(String speaker) {
        this.speaker = speaker;
    }
}
```

(Remember that we have chosen to use string-based attributes to simplify the examples in this text. Fields in attributes can be any serializable type, so when you design your own attributes, don't use the string-based nature of our examples with a requirement of attributes in general. Use the types you need, not just strings.)

3.3.1 "Service" versus "Server"

At this point it is important to discuss the difference between the word "service" and the word "server." A *service* is a logical notion that has at least one object—the object registered in the lookup service. It usually has other parts as well. Often at least one of those parts will be a *server*—a process running on a machine in the network.

Our fortune service is made up of a proxy object (the RMI stub), a fortune server (the `FortuneStreamImpl` object running on some host), and the underlying storage. A service may use one or more servers to provide its service. In both the fortune and chat examples, each service uses exactly one remote object, which in turn uses an underlying store. Other services might talk to no remote servers (doing all computation locally in the proxy) or several (combining the information from more than one server).

3.3.2 Creating the Service

As we stated before, the chat service's proxy (which runs on the client) needs to hold some state so that it can tell the server which message was last returned successfully. The communication between the proxy and the server must include this information. The `nextMessage` method has no way to impart that data, so the proxy will need a different way to talk to the server in order to pass it along. For this purpose the implementation of our service adds an internal, package-accessible interface:

```
interface ChatServer extends Remote {
    ChatMessage nextInLine(int lastIndex)
        throws EOFException, RemoteException;
    void add(String speaker, String[] msg)
        throws RemoteException;
    String getSubject() throws RemoteException;
    String[] getSpeakers() throws RemoteException;
}
```

The proxy will use the `nextInLine` method to get the message following the last successful one, which it represents by index. The message is returned to the client by the proxy's `nextMessage` method, and the new index is remembered for the

next invocation. The other methods do not require any different treatment from those in the ChatStream interface, and so they are declared identically.

The proxy implementation is pretty simple: The proxy object contains an RMI reference to the server that implements ChatServer and the index of the last successfully returned message:

```
class ChatProxy implements ChatStream, Serializable {
    private final ChatServer server;
    private int lastIndex = -1;
    private transient String subject;

    ChatProxy(ChatServer server) {
        this.server = server;
    }

    public synchronized Object nextMessage()
        throws RemoteException, EOFException
    {
        ChatMessage msg = server.nextInLine(lastIndex);
        lastIndex++;
        return msg;
    }

    public void add(String speaker, String[] msg)
        throws RemoteException
    {
        server.add(speaker, msg);
    }

    public synchronized String getSubject()
        throws RemoteException
    {
        if (subject == null)
            subject = server.getSubject();
        return subject;
    }

    public String[] getSpeakers() throws RemoteException {
        return server.getSpeakers();
    }
}
```

When the client invokes `nextMessage`, the proxy invokes the remote server's `nextInLine` method, passing in the `lastIndex` field. If `nextInLine` returns successfully, it increments its notion of the last message index and then returns the message. If instead `nextInLine` throws an exception, the code following the invocation will not be executed, leaving the value of `lastIndex` unchanged. So in our example, even if a network failure happens after the request reaches the server, the client will get an exception and so the next invocation of `nextMessage` by the client will cause a `nextInLine` to be sent that gets the same message again.[2]

The proxy's `add` and `getSpeakers` methods simply forward the request along to the remote server. The proxy's `getSubject` method uses the fact that the subject of a single `ChatStream` never changes—once the proxy gets the subject it can be remembered to avoid a round trip to the server to get it again. Here again the proxy adds value.

3.3.3 The Chat Server

Now let us look at the server side. Our chat server implementation is decidedly simple to keep the example focused on the Jini service. We will allow an administrator to create a new chat service, which means creating a remotely accessible `ChatServerImpl` object that implements the `ChatServer` interface. This object registers a `ChatProxy` object with the lookup service, giving it the appropriate `ChatSubject` attribute and (initially) no `ChatSpeaker` attributes. The `ChatProxy` object contains a reference to its `ChatServerImpl` object.

The `ChatServerImpl` object will be *activatable,* that is, it will use the RMI activation mechanism to ensure that it is always available, even if the system it is running on crashes and reboots. The fortune service you saw before lives only as long as its virtual machine. Should the machine on which it runs die, it will die too. This may be acceptable for some services, but not others. Many Jini services will need to be activatable, or use some other mechanism to outlast reboots.

This service will be activatable, but this is not the place for a full tutorial on writing activatable services. We will give an overview, point out the places in the code where activation is visible, and provide the full code in Appendix B.

Activation works by having an *activation system* that starts virtual machines for remotely accessible objects when needed. Each activatable object is part of an *activation group*—remotely accessible objects that are part of the same group will

[2] Note that the proxy's implementation of `nextMessage` is synchronized. This ensures that two threads in the same virtual machine invoking `nextMessage` at the same time on the same proxy object will not both use or modify `lastIndex` inconsistently.

always be activated in the same virtual machine, while objects that are in different groups will always be in different virtual machines.

An activatable object is created by registering it with the activation system, telling the system which group the object belongs to, providing a storage key that can be used by the object when it is activated to find its persistent state, and optionally a "keep active" flag. This registration returns a remote reference to a newly available remote object. The reference can be sent around the network like any other remote reference.

If the "keep active" flag is `true`, the activation system will always keep the object active when it can. For example, when a system is rebooted, the activation system will activate each "keep active" object. If the flag is `false`, the activation system will wait until it gets the first message for the object and then activate it. In our example we will set the "keep active" flag to be `true` so the active service can register with the lookup service and maintain its lease. Otherwise the service would be inactive, unable to renew its leases, and so would never be found by anyone looking for a chat stream.

Activation of an object is done via its *activation constructor*—a constructor with the following signature:

```
public ActivatableClass(ActivationID id,
                        MarshalledObject state)
{
    // ...
}
```

During activation the activation system first either creates a virtual machine to manage the group, or finds the existing virtual machine that is already doing so. It then has that virtual machine create a new local object of the correct class using its activation constructor.

An activatable class must extend `java.rmi.activation.Activatable`—in which case the activation constructor must invoke `super(id)`—or invoke the static method `java.rmi.activation.ActivatableObject.exportObject`. Either of these actions lets the activation system know that the object is ready to receive incoming messages.

Once the activation constructor returns, the activation system will tell clients of the remote object to talk directly to the running server object. This means that at most the first message from a client to an activatable object requires talking to the activation system (unless there is an intervening server crash). All subsequent requests go directly to the running service.

In our example we will provide a `ChatServerImpl` class that provides a `ChatStream` service by registration with the activation system. You create a new server with the following command:

```
java [java-options] chat.ChatServerAdmin directory subject
     [groups|lookup-url classpath codebase policy-file]
```

`ChatServerAdmin` is a class that creates an activatable `ChatServerImpl` object for the server. The *`java-options`* typically include the security policy file used during creation. The *`directory`* will define an activation group. If the directory does not exist it will be created; a new activation group will also be created and its information written into a file in that directory. If the directory does exist and contains such a file, that information will be used to place the new chat stream into the same activation group. A typical chat stream will not significantly occupy a single virtual machine, so grouping multiple activatable `ChatServerImpl` objects for different subjects into the same virtual machine will keep overall overhead low.

If you want to create a new activation group for the stream, you must give the last four parameters: the *`groups`* or *`lookup-url`* to specify the lookup services you want the chat registered with, and the *`classpath`*, *`codebase`*, and *`policy-file`* for the activated virtual machine. The classpath will be the one for the running server, the codebase will be where clients will download the remote parts of the service from, and the policy file will be the one used by the running server. This is different from the policy file provided in the *`java-options`*, which is the policy file used only during creation. The *`policy-file`* parameter defines the policy file that will be used by the activated virtual machine.

So a typical invocation to create a new chat stream in a new group would look like this:

```
java -Djava.security.policy=/policies/creation
     chat.ChatServerAdmin /files/chats/technical "Cats" ""
     /jars/chat.jar http://server/chat-dl.jar
     /policies/runtime
```

This invocation would create the `/files/chats/technical` directory (if necessary), create a new activation group, store the group information in it, and put the storage for the `"Cats"` chat in that directory. The service would register with the public group, `""`. The server would run using classes from `/jars/chat.jar`, clients would download code from the codebase `http://server/chat-dl.jar`, and the server's security policy file would be `/policies/runtime`. The subsequent command

```
java -Djava.security.policy=/policies/creation
     chat.ChatServerAdmin /files/chats/technical "Dogs"
```

would create a "Dogs" chat stream in the same activation group as the stream for the subject "Cats", and therefore with the same lookup group, classpath, codebase, and security policy because these are defined by the activation group—all objects sharing an activation group will, by virtue of sharing a single virtual machine, have the same lookup registration, classpath, codebase, and security policy.

Let us look at ChatServerAdmin.main:

```
public static void main(String[] args) throws Exception
{
    if (args.length != 2 && args.length != 6) {
        usage();            // print usage message
        System.exit(1);
    }

    File dir = new File(args[0]);
    String subject = args[1];

    ActivationGroupID group = null;
    if (args.length == 2)
        group = getGroup(dir);
    else {
        String[] groups = ParseUtil.parseGroups(args[2]);
        String lookupURL =
            (args[2].indexOf(':') > 0 ? args[2] : null);
        String classpath = args[3];
        String codebase = args[4];
        String policy = args[5];
        group = createGroup(dir, groups, lookupURL,
            classpath, codebase, policy);
    }

    File data = new File(dir, subject);
    MarshalledObject state = new MarshalledObject(data);
    ActivationDesc desc =
        new ActivationDesc(group, "chat.ChatServerImpl",
                           null, state, true);
    Remote newObj = Activatable.register(desc);
    ChatServer server = (ChatServer)newObj;
```

```
        String s = server.getSubject(); // force server up
        System.out.println("server created for " + s);
    }
```

The `main` method first figures out whether it is using an existing group or creating a new group, and gets the group accordingly. It then creates a `MarshalledObject` that contains the directory and subject; this `MarshalledObject` will be the one that is passed in to the activation constructor when each stream is activated, allowing it to recover its state, as you will see shortly.[3] With the group and startup information in hand, we can tell the activation system to register this new object. The `true` in the registration call is the "keep active" flag. We then invoke the `getSubject` method to force the chat stream to be active for the first time. Until this first call, the chat stream object will be inactive. Once `getSubject` forces the server to be active, it will start its discovery and registration.

This process of creation and subsequent activating is shown in Figure 3–1. When `main` invokes `createGroup`, the activation system remembers the group setup options. After `register`, the activation system has a record of a new object in that activation group. When `main` invokes `getSubject` on the newly registered stream, the activation system (1) starts up a new virtual machine using the settings given when the group was created; and then (2) tells the virtual machine (via a piece of its own code running in it) to create a new `ChatStreamImpl` object using its activation constructor, passing the persistent state `MarshalledObject` given to it when the object was registered. When the constructor invokes `exportObject`, the activation system views the object as ready for incoming messages. In the future, when the activation system starts up it will start up the object in the same way, but without requiring any method invocation to get things going.

The figure shows all this work being handled internally by the client's `ChatServerImpl` stub. A stub for an activatable object contains a direct reference to the remote service. When the stub is first used, it sets this reference by asking the activation system for a direct reference to the remote server. The activation system either activates the service to get a direct reference and then returns it or, if it is already active, simply returns the direct reference. The actual messages are sent directly to the service. Once the stub has a direct reference, it sends all future messages directly to the remote server without contacting the activation system.

The `createGroup` method creates the activation group, setting up the command line that will start the virtual machine to use the correct classpath, codebase,

[3] A `java.rmi.MarshalledObject` stores an object in the same way as it would be marshalled to be passed as an argument in an RMI method call. Its `get` method returns the unmarshalled object. The activation system uses a `MarshalledObject` for the persistence parameter because it does not use the object—it just holds on to it and passes it back—so it has no need to download any required code for the persistence parameter.

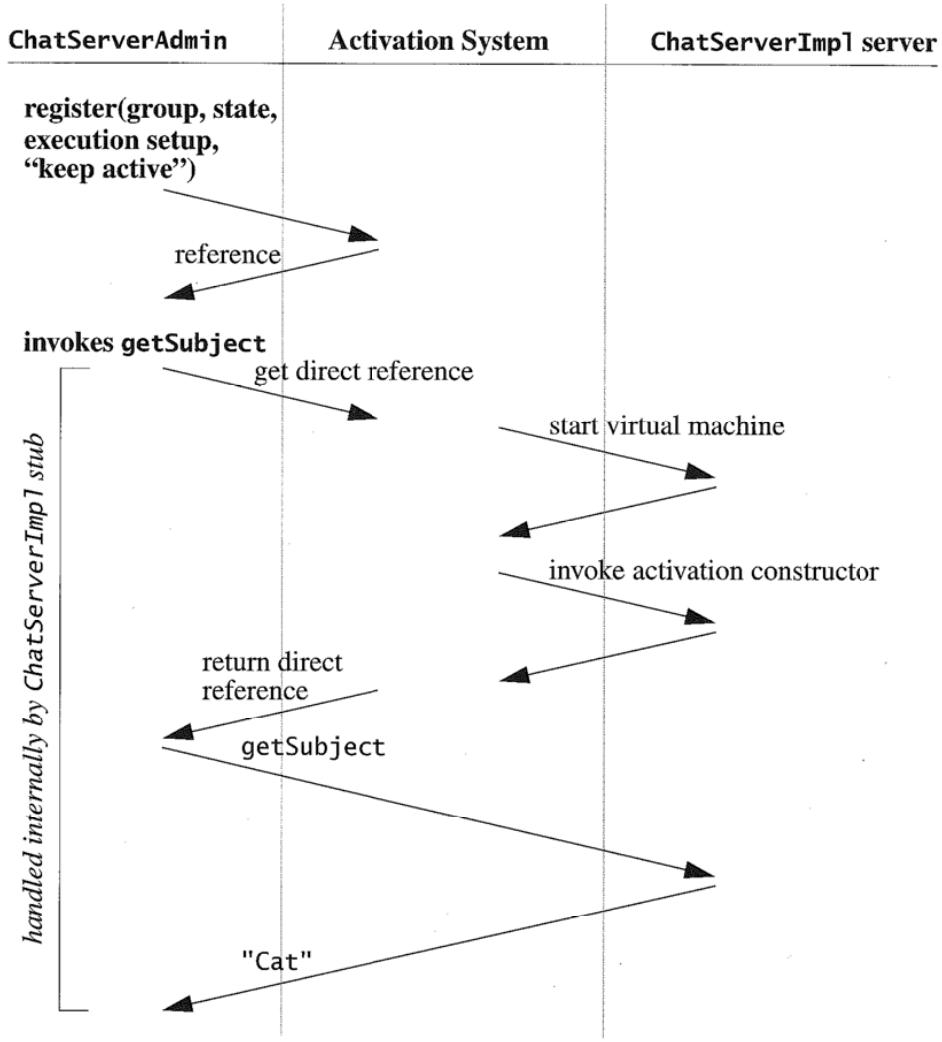


FIGURE 3–1: ***Registration and Activation in ChatAdmin***

and policy file. It then serializes the group descriptor into a file so that future creations that want to share it can find it, adding the lookup groups and URL to the file for the server to use. The `getGroup` method finds an existing group by opening up the directory's group description file and returning the deserialized `ActivationGroupID`. The details of this activation and file work are in the full code in Appendix B.

When ChatServerAdmin.main invokes getSubject or when the activation system restarts, the ChatServerImpl class's activation constructor gets invoked to create the local object in the activated virtual machine:

```
public ChatServerImpl(ActivationID actID,
                      MarshalledObject state)
    throws IOException, ClassNotFoundException
{
    File dir = (File) state.get();
    store = new ChatStore(dir);
    ChatProxy proxy = new ChatProxy(this);

    LookupLocator[] locators = null;
    if (lookupURL != null) {
        LookupLocator loc = new LookupLocator(lookupURL);
        locators = new LookupLocator[] { loc };
    }
    joinMgr = new JoinManager(proxy, getAttrs(), groups,
        locators, store, renewer);
    Activatable.exportObject(this, actID, 0);
}
```

The activation constructor uses the state object stored by ChatServerAdmin to find the directory in which the chat record is stored and to find its record within the directory (by the subject name).

The ChatStore object manages the server's persistent storage. When the server is first activated, the Jini service ID has not yet been assigned, so we want to know when the ID gets assigned. The JoinManager constructor allows us to provide a com.sun.jini.lookup.ServiceIDListener object that will be told when the identifier is assigned. The ChatStore class is an inner class of ChatServerImpl that implements this interface, adding the ID to the persistent store for future use. The relevant part of ChatStore looks like this:

```
class ChatStore extends LogHandler
    implements ServiceIDListener
{
    //...
    public void serviceIDNotify(ServiceID serviceID) {
        try {
            log.update(serviceID);
        } catch (IOException e) {
            unexpectedException(e);
```

```
            }
            ChatServerImpl.this.serviceID = serviceID;
        }
    }
```

The `serviceIDNotify` method is invoked by the join manager when the service ID is first allocated. Our implementation stores it in the file system for future use. The `log` field and the `LogHandler` interface are part of a "reliable log" subsystem from the `com.sun.jini.reliableLog` package in the release of the Jini technology; the details are left for the full source in Appendix B.

3.3.4 Implementing `nextInLine`

The `nextInLine` method of the chat server takes the incoming message number, looks up the message associated with it, and returns it:

```
public synchronized ChatMessage nextInLine(int index) {
    try {
        int nextIndex = index + 1;
        while (nextIndex >= messages.size())
            wait();
        return (ChatMessage)messages.get(nextIndex);
    } catch (InterruptedException e) {
        unexpectedException(e);
        return null; // keeps the compiler happy
    }
}
```

If the next message isn't available yet, `nextInLine` waits until someone has put one in using `add`:

```
public synchronized void add(String speaker, String[] lines)
{
    ChatMessage msg = new ChatMessage(speaker, lines);
    store.add(msg);
    addSpeaker(speaker);
    messages.add(msg);
    notifyAll();
}

private synchronized void addSpeaker(String speaker) {
    if (speakers.contains(speaker))
```

```
            return;
        speakers.add(speaker);
        Entry speakerAttr = new ChatSpeaker(speaker);
        attrs.add(speakerAttr);
        joinMgr.addAttributes(new Entry[] { speakerAttr });
    }
```

When a new message is added, we create the `ChatMessage` object for the message and then store it in the log. We then add the speaker (`addSpeaker` ignores already known speakers), add the message to our in-memory list of messages, and notify any waiting `nextInLine` method that there is a new message to return.

If the speaker is a new one, `addSpeaker` creates a new `ChatSpeaker` attribute object and stamps it on itself by using the join manager's `addAttributes` method. The join manager will add this attribute to all current and future lookup service registrations.

We have not shown the `store.add` method because it consists only of file-system and data structure management, not Jini service implementation. The full code in Appendix B, of course, shows its implementation.

3.3.5 Notes on Improving `ChatServerImpl`

As shown `ChatServerImpl` works, but it does not scale to large systems well. Each client uses up a thread in the server virtual machine when `nextInLine` blocks waiting for a future message. If there are hundreds of observers of a discussion, the number of threads blocked in the server will also be hundreds as each client waits for its invocation of `nextInLine` to return. There are many possible solutions to this problem. The most interesting is to rewrite the proxy/server interaction to use event notification as described in the distributed event specification. The design would look something like this:

- The `nextInLine` method takes a `RemoteEventListener` object. When `nextInLine` has no message to return, it returns an event registration instead of a message.
- When a new message is added, all registered listeners are notified.
- A proxy that gets an event registration will renew the registration's lease until it receives notification from the server that a new message is available. It will then resume asking for the `nextInLine` until it is blocked again.

We leave an actual implementation of this as an exercise to the reader, as well as other things that could be done to improve the service, such as:

- Making add idempotent.
- Handling the results of system crashes that result in partial creation of the service. The activation constructor should detect such corrupt data and unregister itself.
- A way to mark a chat as being completed so that people can see a record of it without adding to it. This might require adding a new method or two in `ChatStream`.
- Administrative interfaces to allow users and administrators to add their own attributes to the service and to configure a running service as to which lookup groups and lookup URLs it will join. As examples, see the interface `net.jini.admin.JoinAdmin`.

Other improvements could be made as well. You might find it useful to get the existing source compiled and running, and then try adding one or more improvements to it to get a better feel for Jini service implementation.

3.3.6 The Clients

When a chat stream service is created, we will have a service that can be used anywhere in the network that can reach the relevant lookup services. The generic `StreamReader` client can read a chat discussion stream from the beginning. A more specialized client would let users add messages to the chat stream. The generic client has more limited functionality but can work across a broader array of services. A specialized chat client uses the extended features of a `ChatStream`. Both use the same service in different ways.

As an example of a specialized client, here is a `Chatter` client that will use a command line to provide access to a `ChatStream`:

```
package chatter;

public class Chatter extends StreamReader {
    public static void main(String[] args) throws Exception
    {
        String[] fullargs = new String[args.length + 3];
        fullargs[0] = "-c";
        fullargs[1] = String.valueOf(Integer.MAX_VALUE);
        System.arraycopy(args, 0, fullargs, 2, args.length);
        fullargs[fullargs.length - 1] = "chat.ChatStream";
        Chatter chatter = new Chatter(fullargs);
        chatter.execute();
```

```
    }

    private Chatter(String[] args) {
        super(args);
    }

    public void readStream(MessageStream msgStream)
        throws RemoteException
    {
        ChatStream stream = (ChatStream)msgStream;
        new ChatterThread(stream).start();
        super.readStream(stream);
    }

    public void printMessage(int msgNum, Object msg) {
        if (!(msg instanceof ChatMessage))
            super.printMessage(msgNum, msg);
        else {
            ChatMessage cmsg = (ChatMessage)msg;
            System.out.println(cmsg.getSpeaker() + ":");
            String[] lines = cmsg.getContent();
            for (int i = 0; i < lines.length; i++) {
                System.out.print("    ");
                System.out.println(lines[i]);
            }
        }
    }
}
```

All the client code in this section is in the `chatter` package. `Chatter` extends `StreamReader` (the generic client described in Section 2) to force an effectively infinite count of messages to read, and to require that the stream found be at least a `ChatStream`, not simply a `MessageStream`. It overrides `readStream` so that when the stream is found, a new thread will be created to read the user's input. The `printMessage` method is overridden to take advantage of the knowledge that the message object is a `ChatMessage`.

`ChatterThread` uses the stream's `add` method when the user types something:

```
class ChatterThread extends Thread {
    private ChatStream stream;
```

```
ChatterThread(ChatStream stream) {
    this.stream = stream;
}

public void run() {
    BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));
    String user = System.getProperty("user.name");
    List msg = new ArrayList();
    String[] msgArray = new String[0];
    for (;;) {
        try {
            String line = in.readLine();
            if (line == null)
                System.exit(0);

            boolean more = line.endsWith("\\");
            if (more) {     // strip trailing backslash
                int stripped = line.length() - 1;
                line = line.substring(0, stripped);
            }
            msg.add(line);
            if (!more) {
                msgArray = (String[])
                    msg.toArray(new String[msg.size()]);
                stream.add(user, msgArray);
                msg.clear();
            }
        } catch (RemoteException e) {
            System.out.println("RemoteException:retry");
            for (;;) {
                try {
                    Thread.sleep(1000);
                    stream.add(user, msgArray);
                    msg.clear();
                    break;
                } catch (RemoteException re) {
                    continue;       // try again
                } catch (InterruptedException ie) {
                    System.exit(1);
                }
```

```
                }
            } catch (IOException e) {
                System.exit(1);
            }
        }
    }
}
```

The `run` method will be invoked by the virtual machine when the thread is started. It reads lines from the user to build up messages and uses `add` to add each message to the chat. Lines that end in \ (backslash) mean that the message continues on the next line. When the user types a line that doesn't end in backslash that line is put together with any preceding lines to create the message. The value defined in the `user.name` property (provided by the virtual machine) will be user's name in the chat. If `add` throws a `RemoteException` we retry adding the message until we succeed or until the user kills the application.

When the end of input has been reached, `readLine` returns `null`, and this thread will invoke `System.exit` to bring down the entire virtual machine, including the thread that is reading other speakers' messages.

4 The Rest of This Book

A good question is never answered.
It is not a bolt to be tightened into place but a seed to be planted
and to bear more seed toward the hope of greening the landscape of idea.
—John Ciardi

By now you should have an overview of how the Jini technology works and what it takes to write a client and service. The rest of this book contains the specification of the Jini architecture. Each subpart of the specification is prefaced by a short paragraph describing where it fits into the architecture. After the specification you will find a glossary that defines terms used in the specifications. Appendix A is a reprint of "A Note on Distributed Computing," whose thinking undergirds the Jini architecture. You can follow the Jini architecture and related technical discussions at `http://jini.org`. Appendix B contains the full code for the examples.

Each specification has a two-letter code. For example, the Jini Architecture Specification has the code "AR." This provides a common name for each part of the specification (for example AR.2.1) no matter what order the parts are placed in. For example, in this book we have placed the parts in a reasonable reading order. In another book it might be best to publish only relevant parts of the specification, or publish the parts in a different order. The common names let you talk with others about specification sections using the same section names no matter where each of you read the work. The two letter codes are shown at the beginning of each specification part, in the section and figure numbers within that part, and on the black thumb tabs at the edge of the right-hand pages.

This book is the first in a series that will come "...from the source"— from those who design, implement, and document the Jini system. These books will all be written either by the originators of the work in question or by people who work closely with them to document the designs and technologies. Other good books and web sites will, we expect, also follow from other sources. We hope that the Jini system and its designs prove useful to you both as user and as developer. At our series' web site `http://java.sun.com/docs/books/jini/` you will find

related resources including a downloadable version of the source in the series' books (including this book's source), errata, and other series-related information.

THE JINI ARCHITECTURE SPECIFICATION defines the top-level view of the Jini architecture, its components, and the systems on which the Jini architecture is layered. This will give you a high-level view of the architecture that will be filled out in the following specifications.

AR

The Jini Architecture Specification

AR.1 Introduction

THIS document describes the high-level architecture of a Jini software system, defines the different components that make up the system, characterizes the use of those components, discusses some of the component interactions, and gives an example. This document identifies those parts of the system that are necessary infrastructure, those that are part of the programming model, and those that are optional services that can live within the system.

AR.1.1 Goals of the System

A Jini system is a distributed system based on the idea of federating groups of users and the resources required by those users. The overall goal is to turn the network into a flexible, easily administered tool with which resources can be found by human and computational clients. Resources can be implemented as either hardware devices, software programs, or a combination of the two. The focus of the system is to make the network a more dynamic entity that better reflects the dynamic nature of the workgroup by enabling the ability to add and delete services flexibly.

A Jini system consists of the following parts:

- A set of components that provides an infrastructure for federating services in a distributed system

- A programming model that supports and encourages the production of reliable distributed services
- Services that can be made part of a federated Jini system and that offer functionality to any other member of the federation

Although these pieces are separable and distinct, they are interrelated, which can blur the distinction in practice. The components that make up the Jini technology infrastructure make use of the Jini programming model; services that reside within the infrastructure also use that model; and the programming model is well supported by components in the infrastructure.

The end goals of the system span a number of different audiences; these goals include the following:

- Enabling users to share services and resources over a network
- Providing users easy access to resources anywhere on the network while allowing the network location of the user to change
- Simplifying the task of building, maintaining, and altering a network of devices, software, and users

The Jini system extends the Java application environment from a single virtual machine to a network of machines. The Java application environment provides a good computing platform for distributed computing because both code and data can move from machine to machine. The environment has built-in security that allows the confidence to run code downloaded from another machine. Strong typing in the Java application environment enables identifying the class of an object to be run on a virtual machine even when the object did not originate on that machine. The result is a system in which the network supports a fluid configuration of objects that can move from place to place as needed and can call any part of the network to perform operations.

The Jini architecture exploits these characteristics of the Java application environment to simplify the construction of a distributed system. The Jini architecture adds mechanisms that allow fluidity of all components in a distributed system, extending the easy movement of objects to the entire networked system.

The Jini technology infrastructure provides mechanisms for devices, services, and users to join and detach from a network. Joining and leaving a Jini system are easy and natural, often automatic, occurrences. Jini systems are far more dynamic than is currently possible in networked groups where configuring a network is a centralized function done by hand.

AR.1.2 Environmental Assumptions

The Jini system federates computers and computing devices into what appears to the user as a single system. It relies on the existence of a network of reasonable speed connecting those computers and devices. Some devices require much higher bandwidth and others can do with much less—displays and printers are examples of extreme points. We assume that the latency of the network is reasonable.

We assume that each Jini technology-enabled device has some memory and processing power. Devices without processing power or memory may be connected to a Jini system, but those devices are controlled by another piece of hardware and/or software, called a *proxy*, that presents the device to the Jini system and itself contains both processing power and memory. The architecture for devices not equipped with a Java virtual machine (JVM) is discussed more fully in a separate document.

The Jini system is Java technology centered. The Jini architecture gains much of its simplicity from assuming that the Java programming language is the implementation language for components. The ability to dynamically download and run code is central to a number of the features of the Jini architecture. However, the Java technology-centered nature of the Jini architecture depends on the Java application environment rather than on the Java programming language. Any programming language can be supported by a Jini system if it has a compiler that produces compliant bytecodes for the Java programming language.

AR.2 System Overview

AR.2.1 Key Concepts

THE purpose of the Jini architecture is to *federate* groups of devices and software components into a single, dynamic distributed system. The resulting federation provides the simplicity of access, ease of administration, and support for sharing that are provided by a large monolithic system while retaining the flexibility, uniform response, and control provided by a personal computer or workstation.

The architecture of a single Jini system is targeted to the workgroup. Members of the federation are assumed to agree on basic notions of trust, administration, identification, and policy. It is possible to federate Jini systems themselves for larger organizations.

AR.2.1.1 Services

The most important concept within the Jini architecture is that of a *service*. A service is an entity that can be used by a person, a program, or another service. A service may be a computation, storage, a communication channel to another user, a software filter, a hardware device, or another user. Two examples of services are printing a document and translating from one word-processor format to some other.

Members of a Jini system federate to share access to services. A Jini system should not be thought of as sets of clients and servers, users and programs, or even programs and files. Instead, a Jini system consists of services that can be collected together for the performance of a particular task. Services may make use of other services, and a client of one service may itself be a service with clients of its own. The dynamic nature of a Jini system enables services to be added or withdrawn from a federation at any time according to demand, need, or the changing requirements of the workgroup using the system.

Jini systems provide mechanisms for service construction, lookup, communication, and use in a distributed system. Examples of services include: devices such