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:

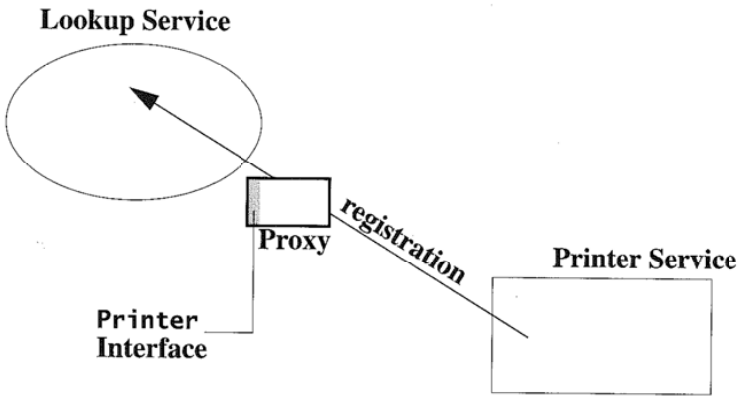
- **Network plug-and-work:** You should be able to plug a service into the network and have it be visible and available to those who want to use it. Plugging something into a network should be all or almost all you need to do to deploy the service.
- **Erase the hardware/software distinction:** You want 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

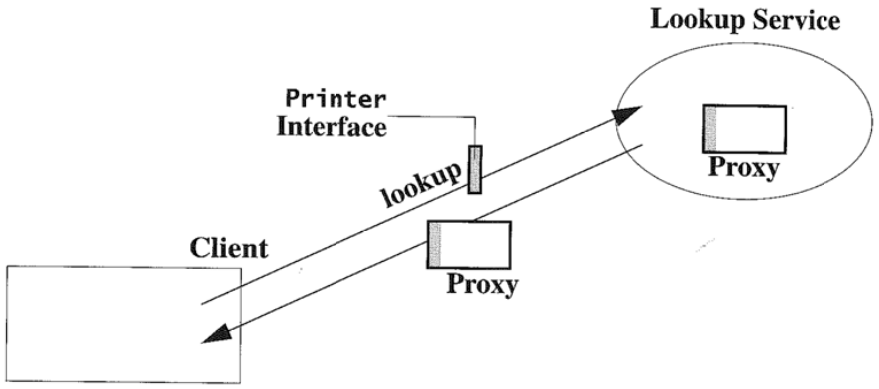
Each Jini system is built around one or more *lookup* services. The lookup service is where services advertise their availability so that you can find them. There may be one or more lookup services running in a network.

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.

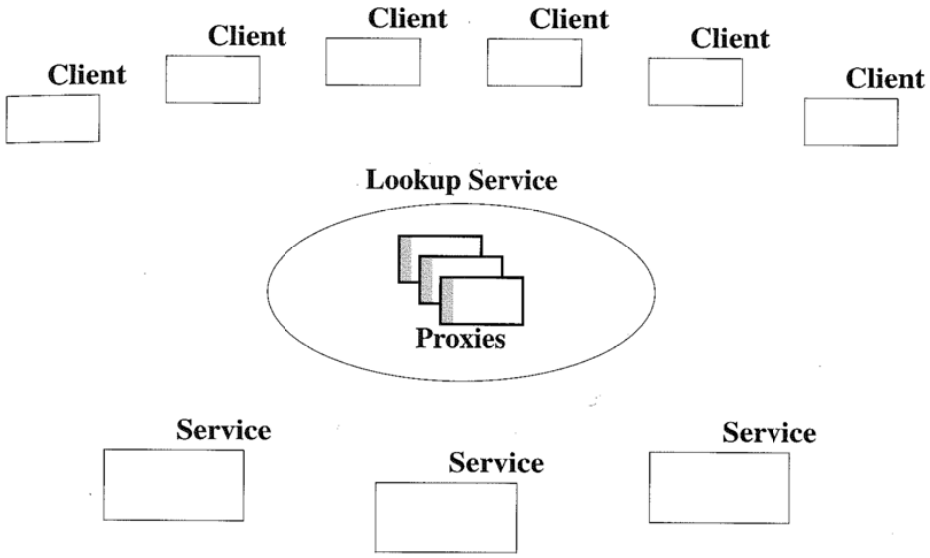
This encapsulation allows the `Printer` interface to be designed as a good client API without requiring it to be a good network protocol for talking to a remote printer. The `Printer` interface should be designed at the abstraction level appropriate for client code. Each proxy object that implements 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) {
```