# The Jini Model

**Topics in This Chapter**

- The goals of Jini
- What Jini is; what Jini isn’t
- Basic concepts:  discovery, lookup, leasing, remote events, transactions
- Downloadable proxies as the key to service delivery

# Chapter 3

Now that you've seen the benefits Java can bring to distributed computing, it's time to see what Jini brings to Java. The Jini vision is simply this: You can walk up with any Jini-enabled device—be it a digital camera, a new printer, a PDA, or a cell phone—plug it into a TCP/IP network, and automatically see and use the variety of other Jini-enabled devices in your vicinity. Any resource available on the network is available to your Jini-enabled device, as if it were directly attached to it, or the device had been explicitly programmed to use it. And adding a new device to this "network community" is as simple as plugging it in.

Likewise, new software services can be added or moved without extensive configuration, and without having to tweak the clients of those services.

In this chapter, you'll learn about what Jini is: both in terms of the "center of gravity" of its design—what aspects the Jini developers considered crucial to focus on—and in terms of the new concepts Jini introduces.

## Jini Design Center

In this section, I'll talk about Jini's "design center." This is the set of areas that the Jini designers felt were the most important to focus on.

## *Simplicity*

Bill Joy, one of the inspirations and champions of Jini, once stated, "Large successful systems start out as small successful systems." This philosophy is very much a part of the Jini motivation. Essentially, if you already know Java, you almost know Jini. Jini is built using the fundamental Java concepts—especially as they relate to the distributed computing issues discussed in the last chapter—and adds only a thin veneer to allow services on the network to work with each other more easily.

This is important: Jini is, at its heart, about how services connect to one another—not about what those services are or what they do or how they work. In fact, Jini services can even be written in a language other than Java; the only requirement is that there exists, somewhere on the network, a bit of code that *is* written in Java that can participate in the mechanisms Jini uses to find other Jini devices and services.

I've been using the terms "devices" and "services" pretty interchangeably so far. But from the Jini perspective, everything—even a device such as a scanner or printer or telephone—is really a service. To use an object-oriented metaphor, everything in the world, even hardware devices, can be understood in terms of the interfaces they present to the world. These interfaces are the services they offer, so Jini uses the term "service" explicitly to refer to some entity on the network that can be used by other Jini participants. The services these entities offer may be implemented (here's the OO terminology again) by some hardware device or combination of devices, or some pure software component or combination of components.

## *Reliability*

I've said that Jini provides the infrastructure that allows these services to find and use one another on a network. But what does this really mean? Is Jini simply a name server like the Internet's Domain Name Service (DNS) or the Lightweight Directory Access Protocol (LDAP) within an organization? As it turns out, Jini does have similarities to a name server; it even provides a service for finding other services in a community (though this service is actually much richer than a traditional name service, as you will see). But there are two essential differences between what Jini does and what simple name servers do:

- Jini supports serendipitous interactions among services and users of those services. That is, services can appear and disappear on a network in a very lightweight way. Interested parties can be automatically notified when the set of available services changes. Jini allows services to come and go without requiring any static configuration or administration. In this way, Jini supports what might be called "spontaneous networking"—services close to one another form a community automatically, with no need for explicit user involvement. This means that you don't have to edit configuration files, shut down or restart name servers, configure gateways, or anything else to use a Jini service—you literally just plug it in and Jini does the rest. Furthermore, every device or service that connects to a Jini community carries with it all the code necessary for it to be used by any other participant in the community.

- Communities of Jini services are largely self-healing. This is a key property built into Jini from the ground up: Jini doesn't make the assumption that networks are perfect, or that software never fails. Given time, the system will repair damage to itself. Jini also supports redundant infrastructure in a very natural way, to reduce the possibility that services will be unavailable if key machines crash.

Taken together, these properties make Jini virtually unique among commercial-grade distributed systems infrastructures. These properties ensure that a Jini community will be virtually administration-free. Spontaneous networking means that the configuration of the network can be changed without involving systems administrators. And the ability for a service to carry with it the code needed to use it means that there is no need for driver or software installation to use a service (other than installing the core Jini software itself, of course). Furthermore, the self-healing nature of Jini also reduces administrative load, and user headaches. A cooperating group of Jini services will be resilient to changes in network topology, service loss, and network partitions in a clean way. Jini services are able to cope with network failures. Perhaps they will not be able to fully do their jobs (even the telephone network may report errors to the user at times), but at least they will work predictably in the face of failures, and will recover by themselves over time.

## *Scalability*

I've said that groups of Jini services join together in cooperating sets. In Jini, these groups of services are called *communities*; all services in a community are aware of each other and able to use each other.[1]

So Jini services band together to form communities. But how large are these communities? What did the Jini designers envision as the "target" size for a community? This is an important question. If Jini's design favors very large groups—say, a group composed of every Jini-enabled device in the continental United States—it will have very different performance and interaction characteristics than a design that favors very small groups. The key issue here is *scalability*—how Jini is designed to accommodate varying numbers of services, from the very large to the very small.

Jini addresses scalability through *federation.* Federation is the ability for Jini communities to be linked together, or federated, into larger groups. Ideally, the size for a single Jini community is about the size of a workgroup—that is, the number of printers, PDAs, cell phones, scanners, and other devices and network services needed by a group of 10 to 100 people. The reason for this workgroup focus is that, most often, people tend to collaborate with those they work closely with. Jini makes it easy to bring together this group of people into a community, and makes their resources shareable.

Even if you're part of a workgroup community, you may occasionally need access to resources "further" away (in the network sense)—for example, that fast, new color laser printer up in marketing. Jini supports access to services in other communities via federating them together into larger units. Specifically, the Jini lookup service—the entity responsible for keeping track of all the services in a community—is *itself* a Jini service. The lookup service for a given community can register itself in other communities, essentially offering itself up as a resource for users and services there. (As you'll see later, Jini actually bootstraps itself: Many of the core Jini features are themselves Jini services that can be shared and used by other services using the normal Jini mechanisms.)

The topology of these communities is very lightweight. When you install the Jini software, the system will, by itself, create communities that form along network boundaries. So, for example, if your engineering and marketing departments are on different networks, each will form a unique Jini community. If you want to federate these communities, it's trivial to do a tiny bit of administration to ensure that the lookup services for each community are known to the other.

1. If you peruse some older Jini technical documentation, you may see the word *djinn* used to refer to a community of Java services. Rather than using an obscure word (the original Arabic word from which "genie" is derived), I'll stick to the more meaningful "community." Often you may see the word "federation" used to describe a Jini community, and some wags, particularly those with a bent toward Star Trek, may refer to a Jini community as a "collective," reminiscent of the Borg.

**Core Note: Jini and administration**

*Jini is designed to work well in an administration-free setting. But of course you can apply a bit of hand-holding to tailor the system for your particular circumstances.*

*Federating communities along organizational boundaries is a prime example of this sort of hand-holding: Jini knows enough by itself to form communities along network lines, but has no idea what your company's organizational structure is. So you have to tell Jini that information yourself.*

## *Device Agnosticism*

This was alluded to in the previous section, but is important enough that it deserves restating: Jini is agnostic with regard to devices. What does this mean? Essentially it means that Jini is designed to support a wide variety of entities that can participate in a Jini community. These "entities" may be devices or software or some combination of both; in fact, it's generally impossible for the user of one of these things to know which it is. This is one of the key contributions of Jini. To use something you don't have to know—and indeed, don't even care—whether that something is hardware or software. You only have to understand the interface it presents.

If an actual hardware device is connected to the network, Jini is flexible about how much computational power it needs to have. Jini can work with full-blown desktop or server computers, capable of running multiple JVMs and connecting with each other at gigabit speeds. It can also work with such devices as PDAs and cell phones that may have only limited Java capabilities—say, a Java 2 Micro Edition (J2ME) implementation with a limited set of class libraries.

In fact, Jini is actually able to accommodate devices that are so simple they may have no computation on them at all. As you'll see in the next chapter, Jini can accommodate devices with the computational intelligence of—literally—a light switch. The only requirement is that some other, perhaps shared, computational resource that can participate in the Jini community-building protocols on behalf of that device must exist.

Furthermore, and this may be somewhat surprising, Jini doesn't even require that the device or service be written in or understand Java! Again, all that is required is that some Java-speaking device be willing to act as a proxy on behalf of the Java-challenged device or service. You'll see how this works in the next chapter.

## What Jini Is Not

Now that I've talked a bit about what Jini is, I should say a few words about what Jini is not.

### *Jini Is Not a Name Server*

As mentioned previously, Jini is not just a name server. Some of what Jini does—like keeping track of the services known within a community—looks like a name server, and even uses the Jini lookup facilities, which provide functionality similar to (but not quite the same as) a name server. But Jini is much more. It is a paradigm for building distributed systems that support spontaneous appearance and disappearance in a community, and the ability to self-heal when things go wrong.

### *Jini Is Not JavaBeans*

JavaBeans provides a way for software components—called beans—to find each other, use services provided by other beans, introspect each other, and so forth. But JavaBeans has a very different design center from Jini. Beans is largely intended for use within a single address space. The mechanisms used for communication between beans are based on direct method invocation, not remote protocols. The beans model, flexible as it is, is also far less dynamic than Jini. When a new bean appears on your system, the current beans in your application don't suddenly know about it and start using it. You—the designer of the system—have to explicitly link the bean into your application and "wire it up" to the other beans. JavaBeans is intended largely for design-time, and to a lesser degree for run-time use in a single address space. Jini is all about run-time use across address spaces. (This isn't to say that Jini and JavaBeans are incompatible systems, however. Jini can leverage JavaBeans in some nice ways. Later, you'll see how to use the JavaBeans event model from Jini, and how to attach beans to Jini services.)

### *Jini Is Not Enterprise JavaBeans*

Likewise, Jini is not Enterprise JavaBeans (EJB). EJB has, on the surface, some characteristics of Jini. It provides the notion of services on the network. Enterprise beans can, and usually do, live in different address spaces. But again, the design center of EJB is quite different than that of Jini. EJB is designed to hook together legacy enterprise systems, covered by Java wrappers, to form the back-end business logic of enterprise applications. It is designed to support easy construction of this logic, and leverages the transaction, messaging, and database services already on the enterprise network. As such, EJB is largely used to configure relatively static pathways between enterprise software components. As long as the logic of the system doesn't change, there's probably little need to reorganize the connections between the beans. Again, defining how these connections will take place happens mostly at design time. In contrast, Jini is about dynamic, run-time discovery of services and run-time connectivity between them. (This is not to say that Jini and EJB can't interact. For example, an enterprise JavaBean could be "Jini-ized" so that it could discover and join a Jini community of services.)

## Jini Is Not RMI

Jini is not the same thing as Java RMI. While Jini *uses* the RMI semantics extensively, particularly its rules for mobile code, Jini is a set of services and conventions built atop these semantics. As such, services that speak Jini can enjoy the full benefits of Jini's spontaneous networking and self-healing abilities. While it would be possible (although a lot of work) to build these abilities into a generic RMI application, they are not a part of RMI itself, and generic RMI applications do not see these advantages.

## Jini Is Not a Distributed Operating System

Finally, Jini is not a distributed operating system. In some ways, it is much larger than a distributed operating system because pieces of it must run atop some platform that provides a JVM at a minimum; in other ways, it is much smaller than a distributed operating system—the facilities offered and the concepts used by Jini are very limited. Jini only has the notion of services, and the facilities for finding those services. True distributed operating systems provide all the services of traditional operating systems (file access, CPU scheduling, user logins), but do this over a connected group of machines. Jini allows much simpler devices to participate in Jini communities than would devices that rely on running a copy of a full-blown distributed operating system.

## What Jini Is

So far in this chapter I've spent some time talking about the design center in Jini, as well as what Jini is not. Now it's time to say exactly what Jini is. I've said that conceptual simplicity was one of the key design goals of Jini, and now you'll get to see what that means in practice.

Broadly speaking, there are three main concepts that form the foundation for Jini. These work in concert to provide the ability for Jini services to spontaneously interconnect with each other, without cumbersome administration. The short sections below introduce these concepts briefly; I'll talk about them in more detail later in the chapter.

### *Services Carry the Code Needed to Use Them*

Applications use services through objects called *proxies*, which provide all of the code needed to connect to a particular service. You can think of proxies as being similar to device drivers in that they allow an application program to interact with a service while shielding it from the details of that service.

But unlike device drivers, which are typically installed by some systems administrator and have to be in place before the device can be used, Jini proxies are carried by the services themselves, and are dynamically downloaded by clients when they wish to use a service. In essence, Jini services can "teach" clients how to use them by extending the capabilities of clients on-the-fly, using all the benefits of mobile code discussed in the last chapter. Applications don't have to know the implementation details of these proxies, and they don't have to be "compiled in"—or even known—when the applications are written.

These proxy objects typically communicate over the network with the back-end portion of the service using whatever protocols the back end was written to understand. Importantly, though, clients are shielded from having to know or care *how* a proxy does its communication with the back end.

### *A "Meta Service" Provides Access to All Other Services*

The ability to dynamically download proxy code isn't enough, though. Simple dynamic downloading says nothing about how you know *what* services are available to you in the first place. Jini provides an interesting solution to this "service discovery" problem. It uses a special service that keeps track of all the other services in the community. This service, called the *lookup service*, is essentially a meta-service that provides access to all other services.

When a service wishes to make itself available, it “publishes” its proxy by storing it in a lookup service. Clients can then connect to the lookup service and ask it what services are available. The lookup service is the indispensible bit of infrastructure in a Jini community. It keeps track of the proxies of the services in the community, makes them available (and searchable) by clients, and can inform interested parties when new services appear or when services leave the community.

And yet, while the lookup service plays a very special role in Jini communities, it is still just a service, with all the characteristics of any other Jini service (including the ability to be deployed redundantly, the fact that lookup services are largely administration free, and so on).

## *A Bootstrapping Process is Used to Find Proxies for the Lookup Service*

As I mentioned, the essential tasks of service discovery are performed by a Jini service, the lookup service. This begs the question: If applications use services by downloading their proxies from the lookup service, and yet the lookup service is *itself* a service, how do applications get the proxies for the lookup service? This is the bootstrapping problem—lookup services are the things that make other services available, but there has to be a way for lookup services themselves to be made available to applications.

You could imagine a couple of solutions to this problem, including providing some fixed “root” lookup service that provides proxies for all other lookup services in the known universe. But this solution isn’t in keeping with the Jini spirit, since it depends on a centralized node (and a single point of failure). Another solution might be to simply pre-compile the proxy code that’s used for talking to lookup services into every application. But this approach would negate the chief advantage of using downloadable proxies: that applications don’t have to know ahead of time how a particular service is implemented. Ideally you’d like to have potentially many different implementations of the lookup service tuned for different environments, and thus each with their own proxy implementations for talking to the service’s back end.

Jini’s solution is simple and yet powerful. Jini uses a process called *discovery* by which lookup services, and the applications that wish to use them, periodically send out messages on the network. The end result of these messages is that an application will automatically find any lookup service in its vicinity, and will have an IP address and port number from which to download the proxy from the lookup service directly. This "out of band" protocol for discovering lookup services is how Jini services "self assemble" into communities. (There are other discovery protocols that allow other types of assembly; I'll talk about these later in this chapter.)

## Fleshing Out The Key Ideas

Lookup and discovery are the key notions that Jini uses to allow an important Java feature—mobile code—to be used to allow applications to find and use services they may never have heard of before. Together, these technologies form the core infrastructure of Jini. If you only understand one key fact about how Jini works, this is the one to remember.

But while these features form the key to Jini, they're really only the most basic substrate on top of which other Jini concepts are layered. For example, the high-level descriptions above say nothing about the *particulars* of how these features are used. For example, they don't say anything about how lookup services come and go, or how clients actually use a lookup service to find services. In this section, I'll talk in more detail about the details of dynamic proxies, discovery, and lookup.

And, of course, these features aren't all there is to Jini. Layered atop these essential mechanisms are several other important concepts that affect the way you build Jini applications in practice. These include aspects of the Jini programming model such as remote events and transactions, as well as Jini's strategy for self-healing, called leasing. In the rest of this section, I'll flesh out the most fundamental concepts in Jini. In the next section, I'll talk about other important "supporting technologies" that you also need to understand to use Jini effectively.

### Downloadable Proxies

Much of the power of Jini comes from its ability to effectively leverage a feature central to Java—the ability to download Java bytecodes from the network and execute them securely. In Jini, services are always accessed via an object provided by the service itself, called a proxy. This proxy is downloaded to the client—code and all—at the time the client wishes to use the service. The client then makes calls on it, just as it would any other object, to control and use the service.

*This idea of downloadable service proxies is the key idea that gives Jini its ability to use services and devices without doing any explicit driver or software installation.* Services publish the code that can be used to access them. A printer, for instance, publishes a proxy that understands how to control that particular printer.

A scanner publishes a proxy that knows how to talk to that particular scanner. An application that uses these services downloads their proxies and uses them without needing any understanding of how the proxies are implemented or how (or even whether) they talk to any back-end device or process.

In some ways, Jini proxies are analogous to Java applets: applets provide a zero-administration way to acquire and use an application; Jini proxies provide a zero-administration way to acquire and use the "glue logic" for communicating with any arbitrary service. But whereas applets are typically designed for "human consumption"—meaning that they usually appear in a Web page when a user asks for them, and come with a graphical interface—Jini proxies are designed to be found, downloaded, and used programmatically. They are essentially secure, network-aware, on-demand device drivers.

The particulars of how a proxy interacts with its service are completely up to the creator of the service/proxy pair. There are a number of common scenarios, though.

- The proxy can sometimes actually perform the service itself. In this case, the object that is sent to clients implements all of the service's functionality itself (and therefore, calling it a "proxy" is really unfair). This strategy is used when the service is implemented purely in software, and there are no external resources that need to be used. An example might be a language translation service that is completely implemented as in Java code and has no need to talk to any external processes to do its job.
- The proxy can be an RMI stub for talking to some remote service. If you've already got an RMI-based service running on the network, the stub for the service is fine for use as a Jini proxy. This is a particuarly easy and minimal way to bring an RMI-based service into the Jini world. In this case, the proxy is a minimal, automatically-generated bit of code that has only the "intelligence" necessary to speak RMI.
- The proxy can be a "smart" proxy that can use any private protocol necessary to communicate with the service. Using a "smart" proxy (as opposed to an automatically generated RMI stub) allows more processing to be embedded in the client. For example, a smart proxy may cache data in the client. It may be able to "fail over" to multiple back-end services. And it fully hides the details of the communications protocol from the client. Smart proxies can be used to provide a Jini interface to legacy (non-Java) services that speak sockets or CORBA or some other

protocol, and they can provide access to hardware devices that have their own communications protocols.

Proxies are the key to Jini's ability for clients to use services they may never have heard of before, and without any administration or driver installation. Even better, the client has no idea how the proxy is implemented. All the user of a printer service has to know is that the proxy implements the `Printer` interface—not the particulars of how it will talk to any specific printer.

## *Using Lookup Services*

As mentioned earlier in the quick run-through of the main ideas in Jini, services store their proxies with lookup services, where clients can then find them.

You can think of Jini lookup services as being like name servers, since they keep track of all of the services that have joined a Jini community. But unlike a traditional name server, which provides a mapping from strings to stored objects, Jini lookup services support a much richer set of semantics. For one thing, the lookup service understands the Java type system. So you can search for proxies that implement particular Java interfaces, and the lookup service will return to you proxies that implement that interface. You can even search by looking for superclasses and superinterfaces of proxies.

How the internals of the lookup service are actually implemented is hidden from you. It may be built using a simple hashtable that gets saved to disk periodically, or it may be a high-powered directory service with lightning-fast lookups and logged writes to persistent storage. All you as the user of a lookup service knows is that the proxy for the lookup service implements the `ServiceRegistrar` interface, which is common to all lookup service implementations. You do not know the details of how it communicates with the lookup service itself—it may be using Java RMI, vanilla sockets, or smoke signals for all you know. This permits multiple, wildly varying implementations of the lookup service, with all of the details hidden behind the object you get that implements the lookup interfaces.

### Publishing a Service

The most important thing a Jini service has to do is publish its proxy with the lookup service so that clients may use it. Abstractly, you can think of the lookup service as storing a set of "service items," each of which contains the proxy for a service, a unique ID for the service, and a set of descriptive “attributes” associated with the service. In Figure 3–1, you see a lookup service that’s holding three service items.

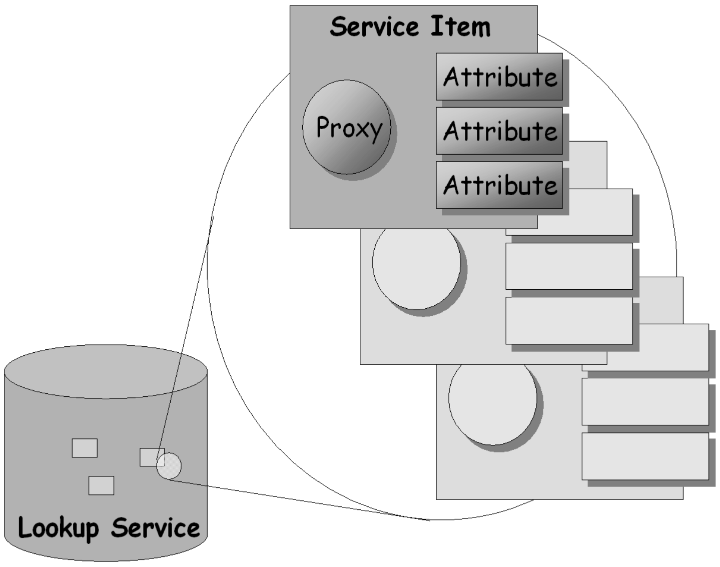

***Figure 3–1*** *Lookup services maintain lists of service items*

If you are publishing a service (say, a printer) that you wish to make available to anyone who wants to use it, you *join* all the lookup services you find from the discovery process. The `ServiceRegistrar` interface has a method called `register()` that lets you join a lookup service. Figure 3–2 shows a service joining a community by registering its proxy with a lookup service.

**Core Tip: Join every lookup service for your community**

*Note that any given community (that is, each unique group name on a given network) may have any number of lookup services supporting it. For fault-tolerance purposes, many redundant lookup services may be active.*

*Typically, a service that needs to join a community will join all the lookup services that support that community, so that if one lookup service fails, others can stand in for it. Unless you have some special need, or need to explicitly limit the scope of visibility of your service, you should usually join all of the lookup services that you discover.*

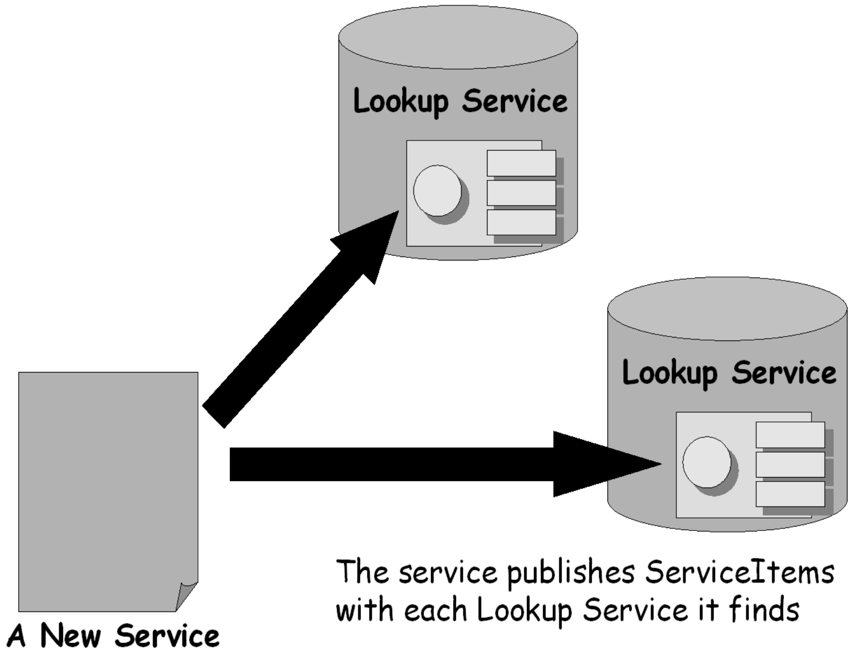
Lookup Service
Lookup Service
A New Service
The service publishes ServiceItems with each Lookup Service it finds

***Figure 3–2*** *A service joins a Jini community*

You invoke the `register()` method by passing in a service item object as an argument. You fill in the attributes with objects that may describe your service. There are some standardized attributes provided by Jini for this purpose—service name, service location, comments, and so on. If your company sells a number of Jini services, you may even have your own standardized set of attributes that reveal extra information about the services.

Jini calls the process of publishing a proxy the *join protocol*. This isn't a protocol in the network sense, but rather a series of steps that services should take to ensure that they're well-behaved with respect to the other services in a community. Chapter 8 goes into quite a bit of detail on the join protocol and other aspects of how services interact with lookup services to publish their proxies.

## Finding a Service

Once a service has joined a community by publishing its proxy, clients can use the service. Typically, clients will search the lookup services they have found for services that implement particular interfaces that they know how to use.

The lookup service provides a number of ways to search—you can search based on the type of the proxy, you can search for the service's unique ID if you happen to know it, or you can search based on the descriptive attributes associated with the service's proxy.

This process of finding the services you want to use is the core functionality of the lookup service and so, not surprisingly, the `ServiceRegistrar` interface defines a method called `lookup()` that does exactly this. By using this method, clients can retrieve the proxies needed to use the services they desire.

Chapters 9 and 10 go into detail on how clients interact with lookup services, but Figure 3–3 illustrates the high-level view of the process. Here you see a client application downloading the proxy for a particular service from a lookup service. Once the proxy has been downloaded, the client uses it as a "front end" to communicate directly with the service's "back end," which is typically a long-lived process or a hardware device connected to the network.

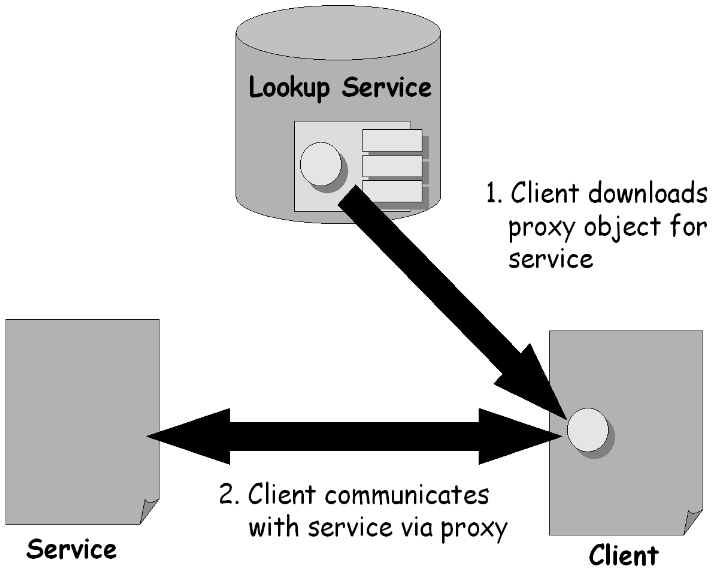
Lookup Service
1. Client downloads proxy object for service
2. Client communicates with service via proxy
Service
Client

***Figure 3–3*** *Using a Jini service*

Say you are writing the software for a digital camera and you want to be able to print from the camera directly to a printer when you detect that one is nearby. In this case, the most likely scenario is that the camera does a search of all the lookup services in its community, looking for proxies that implement the `Printer` interface. The lookup services may return one or more proxies that implement this interface, and the camera may choose to present them, possibly annotated with name, location, or comment information from their attributes, to the user on the LCD display of the camera. When the user prints a picture to a particular printer, the `Printer` methods on the front-end proxy object are invoked, and output is sent to the printer.

## Common Interfaces, Divergent Implementations

You may be asking yourself, “How did the camera know to look for service items for the `Printer` interface?” Clearly, applications need to have at least *some* understanding of the semantics of the interfaces they are calling. You, as the writer of the digital camera software, may not know *how* the `Printer` interface works, but you have at least a basic idea of *what* it does—it causes things to be sent to the printer. If you were to encounter an interface with some unintelligible name, say the `Fleezle` interface, you would have no idea what it did, or what to pass as arguments to its methods, or what to do with its return values. In such a case, it’s impossible to really use this unknown interface programmatically, without some end-user involvement to tell you what to do. For this reason, most Jini services will be written to implement well-known interfaces, and to expect the other services in a community to implement well-known interfaces. This is the only way they can know *how* to programmatically interact with services they encounter.

Sun, along with its partners and the wider Jini community of users, is working to define a set of common interfaces for printers, scanners, cellular telephones, storage services, and other common network devices and services. Development of these is ongoing at the time of this writing. If at all possible, writers of new services should use standardized interfaces wherever appropriate to ensure that Jini services can take advantage of each other.

It is, of course, possible to use unknown interfaces if the user can tell you what to do with them. For this reason, it’s a good idea to create user interfaces (in addition to programmatic interfaces) to your services. Such interfaces can be displayed to a user and may allow them to interact “manually” with a service, even if a client doesn’t know how to use it programmatically. Thus, a user might recognize the `Fleezle` service from its user interface, and control it directly from the camera, even if the camera’s software could not.

## *Discovery*

Before a Jini-aware application—either a service or client—can take advantage of Jini services, it must first find one or more Jini communities. The way it does this is by finding the lookup services that keep track of the shared resources of that community—this process of finding the available lookup services is called *discovery.*

There is not necessarily a one-to-one mapping of communities to lookup services. Each lookup service on a network may provide service for one or more communities, and each community may have one or more lookup services that support it, depending on how the lookup services were started.

### The Discovery Protocols

There is not just a single discovery protocol—Jini supports several useful protocols in different situations.

- The Multicast Request Protocol is used when an application or service first becomes active, and needs to find the "nearby" lookup services that may be active.
- The Multicast Announcement Protocol is used by lookup services to announce their presence. When a new lookup service that is part of an existing community starts up, any interested parties will be informed via the Multicast Announcement Protocol.
- The Unicast Discovery Protocol is used when an application or service already knows the particular lookup service it wishes to talk to. The Unicast Discovery Protocol is used to talk directly to a lookup service, which may not be a part of the local network, when the name of the lookup service is known. Lookup services are named using a URL syntax with `jini` as the protocol specifier (`jini://turbodog.parc.xerox.com` specifies the lookup service running on the host `turbodog.parc.xerox.com` on the default port, for example). Unicast lookup is used when there is a need to create static connections between services and lookup services, such as when explicitly federating lookup services together.

The end result of the discovery process is that proxies for the discovered lookup services are returned to the application doing the discovery. As mentioned earlier, these proxies will all implement the `ServiceRegistrar` interface, and allow services to publish proxies, and clients to find them.

## Supporting Multiple Communities

In Jini, communities can have names. In the Jini APIs, these names are called “groups,” and during the discovery process, a service or application can specify the groups it wishes to find. The discovery protocols will return all of the lookup services it can find that are members of those groups. Lookup services can be members of multiple groups. (And, even though other types of services can register with any number of lookup services which have membership in any number of groups, you typically don’t think of those services as being members of those groups. The Jini team thinks of “group membership” as being a trait of lookup services only, whereas other services simply join lookup services that are members of those groups.)

You can, in most cases, think of “groups” and “communities” as being the same things—groups are simply the names used to specify and represent communities. The most important difference to note between communities and groups is that because of network separation, different communities may have the same group name—the “public” group at Xerox PARC does not refer to the same “public” group at Javasoft, for example. Even though the names are the same, these names can refer to different actual communities depending on where they are. Put a different way, group names are not globally unique, nor are they necessarily globally accessible. But, for most purposes, you can think of groups and communities as being interchangeable.

How does a service know which community to join? In most cases, services will simply look for the “default” group, which is named by the empty string and—by convention—is treated as a “public” community, and then use the multicast protocols to connect to any and all lookup services they can find. The multicast protocols are designed to ensure that the discovery process will only reach lookup services running on the local network, to keep from blasting the entire Internet with discovery protocol packets.

In some cases, a service may need to join a non-default community. For example, a product development lab may test out new services in an “experimental” community that happens to exist on the same subnet as the “production” community, which might be the default. These two communities can share resources (if services join both communities), and even share lookup services (if the lookup services are members of both communities). To join the non-default experimental community, services would use the group name “experimental” to find the lookup services that are members of that community.

In other cases, a service may need to join a non-local community that is “out of range” of multicast discovery. Such services can be explicitly configured with a set of lookup services that they will connect to via unicast discovery. Since unicast discovery doesn’t have the same range restrictions as multicast discovery, services can use this feature to join communities no matter where they are. The ability to use the unicast discovery protocol allows Jini to be flexible in creating its community structure.

While Chapter 6 goes into quite a bit of detail on the various discovery protocols, using them in practice is thankfully easy. Most applications will use a set of convenient utility classes that come with Jini and will never have to worry about interacting with discovery directly.

## Supporting Technology

While the use of discovery and lookup to deliver service proxies is the central notion in Jini, this isn’t enough in itself to really support rich, robust, distributed systems. So on top of this basic substrate, Jini layers some more technologies that are used to provide features like reliability and the ability to detect changes in services.

In this last section of the chapter, I’ll talk about these supporting technologies and introduce several new concepts. You’ll need to understand the ideas here—and the programming libraries that support them—to build real Jini software.

There are three particularly important ideas that are layered atop the basic Jini facilities:

- Jini’s approach to self-healing is through a technique called *leasing*. Essentially, Jini never grants access to resources in perpetuity, but requires that resource holders continually “renew” their leases on resources.
- Jini provides a flexible *remote event* system to allow distributed applications to communicate with one another.
- Jini supports a lightweight model for *distributed transactions* to ensure that computations can reach a “safe” state.

The rest of this chapter provides an overview of these three important concepts in the Jini programming model.

### *Leasing*

All the talk in this chapter about discovery and lookup covers the aspects of Jini that allow communities of applications to form spontaneously and exchange code. But I haven’t said anything yet about how to ensure that these communities are stable, self-healing, and resilient in the face of (inevitable) network failures, machine crashes, and software errors.

Consider an example: Suppose our digital camera service joins a community by registering itself with a lookup service; presumably this happens whenever the camera is attached to a computer that is itself on the network. The camera publishes the fact that it is available for use, and all is well. That is, all is well until the user unceremoniously yanks the camera out of its cradle without turning it off first. What happens here? To the other members of the community, this may look like a classic partial failure situation—they may not be able to tell if the remote host to which the camera is connected has gone down, if it's simply slow to answer, if it's not answering network traffic because of a change in its configuration, if the camera's software has crashed, or even if the camera has been smashed with a hammer. But, regardless of how it was disconnected, the fact is that it has not had a chance to unregister itself before disconnecting because of its abrupt "termination."

The result of the user disconnecting the camera without shutting down properly—a completely understandable and common occurrence—is that, without some special facilities, a "stale" registration will linger in the lookup service for the community. Services that wish to use the camera will see it registered but will not be able to use it. But even more severe are the problems that the lookup service itself faces—if service registrations are never properly cleaned up, it will accrete registrations and slowly bog down under the weight of stale data.

This accumulation of state is a serious problem in long-lived distributed systems. You simply cannot ensure that services will never crash or become disconnected before they've had a chance to deregister themselves.

In the case I've described, the camera holds a *resource* in the lookup service—it is asking the lookup service to use some of its (possibly scant or expensive) storage and computation to maintain the camera's registration. If the Jini infrastructure used a traditional approach to *resource reservation*, the registration would simply stay active until it was canceled, or until some human administrator went through the logs and cleaned out the stale services.

Obviously, this solution violates everything that you want from Jini. First, it doesn't ensure that the system self-heals: Partial failures aren't recognized and cleaned up, and services that hold resources on behalf of others may grow without bound. Second, and perhaps even worse, it requires explicit human intervention to administer the system.

## Time-Based Resource Reservation

To get around these problems, Jini uses a technique called *leasing*. Leasing is based on the idea that, rather than granting access to a resource for an unlimited amount of time, the resource is “loaned” to some consumer for a fixed period of time. Jini leases require demonstrable proof-of-interest on the part of the resource consumer to continue to hold onto the resource.

Jini leases work much like leases in “real life.” Jini leases may be denied by the grantor of the lease. They can be renewed by the holder. Leases will expire at a predetermined date unless they are renewed. They can be canceled early (and, unlike in real life, Jini imposes no penalty on early lease cancellation). Finally, leases can be negotiated, but, as in real life, the grantor has the final word on the terms of the lease that is offered.

Leases provide a consistent means to free unused or unneeded resources throughout Jini: If a service goes away, either intentionally or unintentionally, without cleaning up after itself, its leases will eventually expire and the service will be forgotten. Leasing is used extensively by the lookup service and in other aspects of Jini, so it’s important to understand how leases work.

One of the great aspects of leases is that they make it very hard to screw up the entire system: The system acts conservatively, so if you forget to do lease management, or a programming bug causes you to never renew your leases, your unreliable code simply drops out of the community without causing widespread damage to others. From the perspective of a community, buggy programs that forget to renew their leases look exactly the same as network errors and machine crashes—all the community sees is that the service’s lease has expired and that it is no longer available.

The second great aspect of leasing is that it makes the persistent storage used by the members of a Jini community virtually maintenance-free. A systems administrator will never have to crawl through logs, trying to determine which services are active, which are inactive, and which have left stale data scattered throughout the system. Given a bit of time, the community will identify unused resources and free them, like antibodies attacking a virus. Certainly it would be a great day if we could erase all traces of unwanted applications, unused drivers, and obsolete OS upgrades from our PCs so easily.

Leasing is a rich topic in Jini, and there are many aspects of leases that don’t warrant discussion in this (already long) chapter. I’ll discuss leasing in detail in Chapters 12 and 13, though.

## *Remote Events*

Jini services, like many software components in a system, whether distributed or local, occasionally need to be notified when some interesting change happens in the world. For example, in the local programming model, a software component may need to be notified when the user clicks a mouse, or when the user closes a window.

These are examples of *asynchronous notifications*. They are messages sent directly to a software component, and they are handled outside the normal flow of control of the component. That is, rather than continually polling to see whether some interesting change has occurred, a method on the component will be “automatically” called when the change occurs. The asynchronous nature of these notifications can often simplify programming, since you don’t have to insert code to periodically check the state of some external entity.

Jini, like most of Java, uses *events* to do asynchronous notifications. An event is an object that contains information about some external state change that a software component may be interested in. For example, in AWT, a `MouseEvent` is sent whenever the mouse changes state—whenever it moves, or a mouse button is pressed or released. Events are injected into the system by an *event generator* that is watching for state changes. In AWT, there is a thread called the `AWTEventGenerator` that performs this service. In Java, once an event is introduced into the system, it is “sent” to the interested parties that want to hear about it. In this regard, Jini’s event model works exactly the same as the standard event model used by JavaBeans and the Java Foundation Classes—all of these models support events and asynchronously call methods on listeners when events arrive.

## How Jini Uses Events

There are many cases when a Jini application may need to receive a notification of a change in the state of the world. Earlier, I gave the example of a digital camera that wants to be able to use any printers available in its Jini community. I said that this camera would contact all of the lookup services it could find, and then search for services that implement the `Printer` interface. This example makes a grievous assumption that I completely glossed over: It assumes that the printers will be connected to the network and available for use *before* the camera. What if the inverse is true? In this case, there are no printers available when the camera first connects, although printers may come on line later. Certainly, you’d still like to be able to print, regardless of the order in which you plug in your devices.

The answer is that the camera needs to be notified when any services that it might be able to use appear in a community. It’s easy to imagine the user interface to such a camera. The “Print” button on the LCD is grayed out. You plug a printer into the network, and suddenly the Print button comes alive! The camera has just received a notification that a printer is now active on the network.

This is only one example of how events are used in Jini. Obviously, the lookup service will generate events to interested parties when services appear, disappear, or change. But other Jini entities may generate or consume events as well. The `Printer` service may let other services listen for events that denote special printer conditions, such as `OutOfPaper` or `PrinterJammed`. Thus, events don't just have to go between the existing Jini infrastructure and services; they can fly among services themselves.

## Remote vs. Local Events

But the Jini event model has some differences vs. the "normal" Java event model. Why this distinction? Aren't the Java event models good enough? As it turns out, the Java models are fine for what they were designed for: delivering asynchronous notifications within a single Java VM. The distributed computing world is a very different place, though, and calls for a slightly different event model to fully accommodate the kinds of programs that will run there.

There are a number of very important differences between events that are intended to be delivered locally and events that are intended for distributed delivery.

- In the local case, it's much easier to cause events to be delivered in the order in which they were generated. This is because local event delivery schemes can use a centralized queue that acts as a "choke point," forcing the events into a serial order. Distributed systems, because they lack this centralized event manager, and because of the issues of transporting the events over the network, cannot make this guarantee without serious performance penalties. (Java's AWT event dispatch mechanisms actually do *not* provide this guarantee of deterministic ordering for local events—but in general, local event delivery schemes can implement such an ordering relatively easily.)
- In the local case, an event that is sent will always be delivered, barring catastrophic failure (such as a crash of the entire application). Stand-alone systems are not susceptible to the kinds of partial failures that distributed systems are. Partial failures in a distributed system, whether a machine crash or a network partition, can cause events to go undelivered.
- The cost of sending a local event is typically small compared to the work that may be done to handle the event. Typically, "sending" an event is simply a method call to tell the interested party that the event has arrived. The computation that the recipient does as a result will likely dwarf the time required to send the event. In the remote case, the situation is reversed.

Delivery of the event may require orders of magnitude more time than the local case, and dominate the time spent in handling the event. The performance differences mean that distributed systems are likely to be architected to generate as few events as possible.

- Finally, in the local case, if a component has asked to be a recipient of events, the sender knows that it can safely deliver the event to that recipient. The remote case is far more complicated. The remote recipient of an event may be temporarily disconnected from the network, in which case the sender should probably keep trying to send. The recipient may have crashed, in which case the sender may wish to discard the event. Or, the recipient may be “inactive” for a period of time and not able to process the event. With this range of possibilities it’s often impossible for a sender to know what the right response is.

As you can see, there are many more *policy* decisions to be made in the remote case—many more “knobs” that are available for turning. Does the order of delivery of events matter? Should events be delivered at all costs? If the recipient cannot be contacted, should the event be dropped? Stored until asked for? Resent?

There are no single answers to these questions. Each application must make its own decisions about what makes sense for it. An on-line banking application will almost certainly require very conservative answers to these questions—events should arrive, in order, and at all costs. If a recipient is down, the events should be logged until they can be resent. Other applications may have far less stringent requirements. An on-line game may be able to tolerate a few missed or badly sequenced events and still be fine.

Jini takes an interesting approach to solving this dilemma of how to express so many application constraints: It supports none of them directly, but provides generic mechanisms by which applications can extend the event processing behavior of the system. I’ll talk about this ability in detail in Chapter 16, but the basic idea is that you can write new services that can receive and process *any* type of event from *any* Jini service, even if the new service doesn’t understand the particular events types it’s dealing with. These new services can implement particular policies that may be needed by applications.

The good news for application writers, though, is that for the most part, programming with Jini events looks very much like programming with “normal” events. Most of the time, you only need to understand the remote programming model insofar as you need to understand remote exception types and RMI. Only if you’re planning on using (or writing) a new service to process events do you typically have to understand all the ins and outs of the Jini remote event model.

## *Distributed Transactions*

The last major concept in the Jini programming model is *distributed transactions*. Unfortunately, this concept is one of the most difficult to master in Jini, but luckily many applications will never have to use transactions.

I've talked a lot about the need for reliability and robustness in distributed systems, and the evils that partial failures can cause. Recall that partial failures are when one stage of a computation fails, or when one component that's needed in a computation fails. If *every* stage that participated in a computation failed, then recovery would be easy—you would know that none of them had successfully completed, and you could simply retry the entire computation later. And, obviously, if every stage succeeded, then you would have no need to recover at all.

But the worst scenario, from the programmer's perspective, is when only a subset of the work that has to be done completes. Let's look at a classic example from database programming that exists in the local case. Bank databases need to be reliable, because millions or billions of dollars are moved through the databases each day. Many of the monetary exchanges are in the form of transfers between two accounts: The money is extracted from one account and added to another. Think about how you would program this transfer. Likely, you would write a bit of code to decrement the total in account A, and then write a bit of code to increment the total in account B by the same amount.

This is all simple and obvious, until you think about ways the transfer can fail. What if you crash just after you've decremented the total in account A? Account A may be out millions of dollars, but the money hasn't gone to B! It's simply lost in the ether, the victim of an accounting error caused by a program that wasn't resilient to partial failures.

Distributed systems compound the problems of partial failures: Each stage of your computation may involve contacting components that live somewhere on the network. These components may crash, or the network may become unstable, at some point midway through the computation. What do you do now? You need to either keep trying to contact the unreachable components (and you'd better succeed before you, yourself, crash), or you need to contact the successful components and tell them to undo whatever changes they've made. And, of course, there's always the chance that only a subset of the undo orders would succeed.

### Ensuring Data Integrity

This is where transactions come to the rescue. Transactions are a way to group a series of related operations so that there can be only two possible outcomes: Either all of the operations succeed, or all of the operations fail. In either case, the system moves to a known state in which it is relatively easy to do the right thing—either move on if the transaction succeeds, or try again later if the transaction failed.

Transactions provide what are often called the “ACID” properties to data manipulations, so called because of the initials of the four individual properties.

- Atomicity. Either all of the operations grouped under a transaction succeed, or they all fail: They execute as if they were a single atomic operation.
- Consistency. After the transaction completes, the system should be in a consistent, understandable state. The notion of “consistency” is something that may only be discernible by the human users of the system—therefore, transactions are merely a way to help ensure consistency, not guarantees of consistency itself.
- Isolation. Transactions don’t affect one another until they complete. That is, the effects of a transaction that is in the middle of executing will appear “invisible” to other computations outside of that transaction. This property ensures that computations won’t be based on bogus data that may change if a transaction fails.
- Durability. Once a transaction completes successfully—meaning that all of its changes have been made permanent—then these changes must not be lost due to any subsequent failure or crash. The results of the transaction must be at least as persistent as the entities that use them.

Transactions work by coordinating all of the cooperating parties through a centralized entity. Essentially, this pushes the burden of getting all the parties to agree or all disagree, and tracking their states, into a piece of software called a *transaction manager*. Because the transaction manager is centralized with respect to the participants, it can have a “privileged” viewpoint, from which it watches all of the participants.

In the local cases, such as the bank database of our example, this centralized transaction manager is typically the database itself. Commercial databases have extensive facilities to ensure that sets of operations in a transaction either all complete or fail together. The database keeps logging information so that it can know whether a transaction has completed or failed if it crashes in the middle of executing a transaction.

Distributed databases—or any system that involves components running on multiple machines—are a little more complicated, though. In the local case, the crash of the database means that all of the operations in the transaction that were in progress also stopped at that point. The database can keep logs that are accurate, because all of the operations in a transaction happen locally to the database (they happen "within" the database itself).

In distributed databases, however, operations may be executed on remote machines that the database has no direct control over; the database may not be able to determine if an operation running on a remote machine has successfully completed or failed. All this may sound complicated but, as I said at the start of this section, the good news is that most Jini programmers are completely shielded from this complexity. I'll talk more about the nitty-gritty details of how transactions work under the covers in Chapter 18 but, for now, let's look at the programmer's view of how transactions work.

## Transactions in Jini

Using transactions in practice is pretty easy, for three reasons. First, chances are you won't have to do a lot with transactions. Of the current Jini services, only the JavaSpaces storage service actively makes much use of transactions. So, if you never have to see a transaction, dealing with them is quite easy!

Second, even when you do have to deal with transactions, you really don't have to understand any of the mechanics of how the underlying distributed transaction protocols actually work to use them effectively as a *client*. (If you are *providing* an operation that can work in a transaction, then you have to do more work, though.)

Finally, when you do need to use transactions, the end-programmer transaction APIs are quite simple: You just create a transaction, pass it to all the operations you want to be grouped, and then tell it to try to execute all the operations atomically, which will either succeed or fail.

Programmatically, the first step in using transactions is to use the Jini `TransactionFactory` object to create a `Transaction` object to "hold" the grouped operations. Every service that knows how to participate in transactions will accept a `Transaction` parameter to its "transactable" methods. This is how you group an operation into a transaction: You simply pass in the `Transaction` object to one or more transactable methods to cause them to be grouped.

Once you've collected together the operations, you call `commmit()` on the `Transaction` object to call the Jini transaction system to attempt to execute all of these operations. The `commit()` call will either succeed, or an exception will be raised if one of the participants couldn't complete the operation. If this is the case, Jini will ensure that all of the other participants undo their operations.

In most cases, this simple explanation represents all you have to know about transactions. There are more complicated cases in the programming model (such as nested transactions), but these are rarely used. And, of course, if you wish to build a “transactable” service (meaning one that can participate in, rather than just originate, transactions), then you have to understand the transaction model deeply. But for most cases, using transactions is reasonably simple.

Because transactions are likely to be one of the aspects of Jini that you use the least, I’ll save the in-depth discussion of them until near the end of this book.

## What’s Next?

I’ve introduced the most fundamental concepts in Jini—the notion that services carry the code needed to use them, the lookup process to finding and using that code, and discovery as a means to “jumpstart” a Jini community. I’ve also talked about three essential concepts in the Jini programming model that are built atop these basic ideas—leasing as a way to support self-healing, remote events for notifications, and distributed transactions for “safe” computing. I’ve talked about how all of these work conceptually, but haven’t provided many details yet on how to actually program using these ideas.

In the chapters in Part II, I’ll talk about each of these notions in greater detail. You’ll see how to program against the Jini discovery and lookup APIs, see how to write downloadable proxy code for your services, and write services that make effective use of leasing, transactions, and remote events. You’ll also learn how to use the various utility services that come with Jini.

Before Part II, however, there is one more chapter that provides a bit of introductory material before you jump into building real Jini services. If you’re eager to get going, you may want to skip ahead. But if you do, I’d recommend revisiting Chapter 4, which has details on some common strategies for actually building and deploying “real world” Jini services later.

On to Chapter 4!